Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

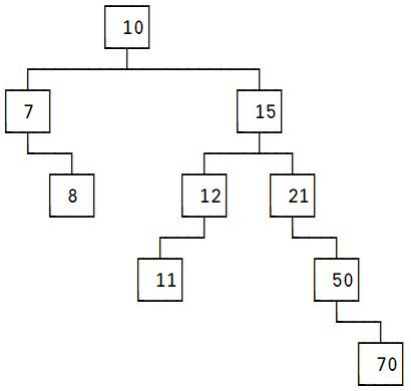
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

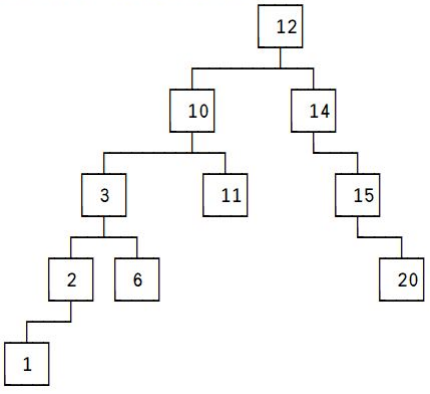
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

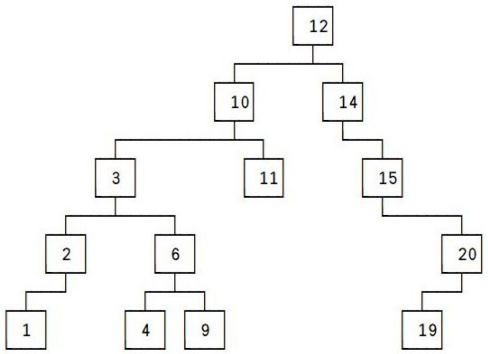
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

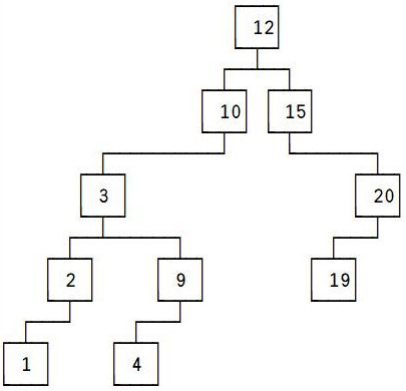
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
20  12   8  14   3  19  11  18   2  13
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
4   7   1
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
12   3   2
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76201 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

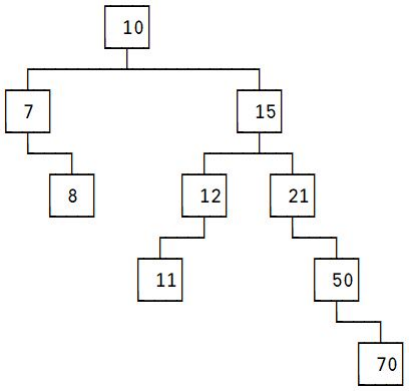
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

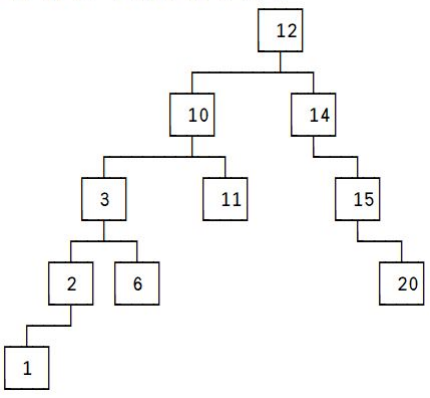
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

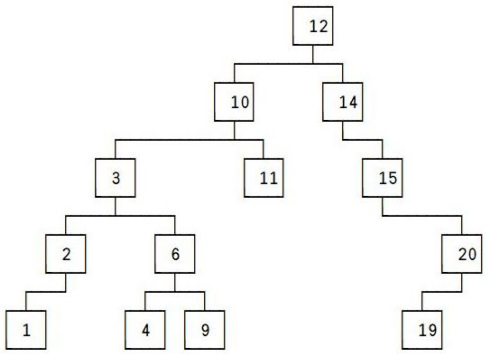
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

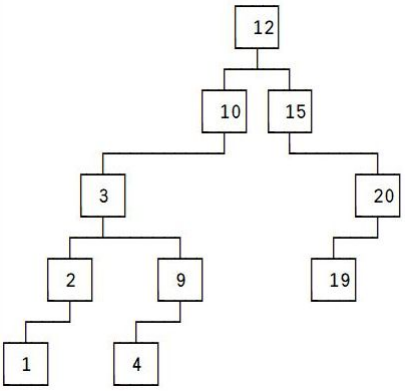
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
20   9  15  10   6   2  13   7   3  17
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
18  14  12
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
6   7  10
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76218 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76225 CASSIO COELHO PAZ
24CC1304 - 3 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

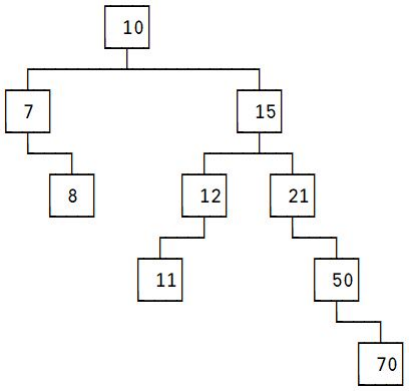
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

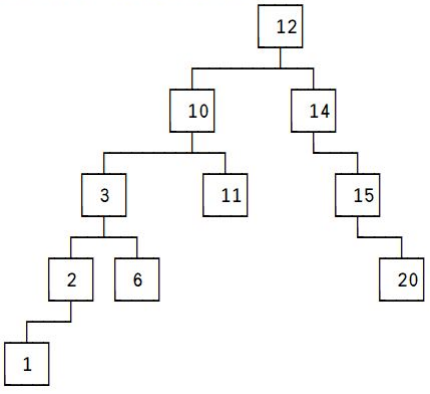
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

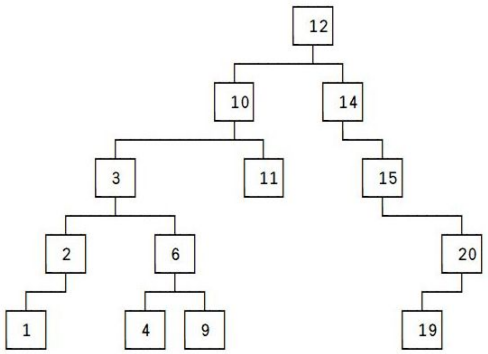
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

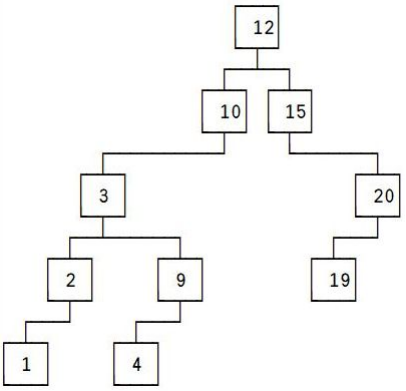
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
6  12  7  20  11  14  4  18  3  19
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
13  10  8
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
12  11  3
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76225 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

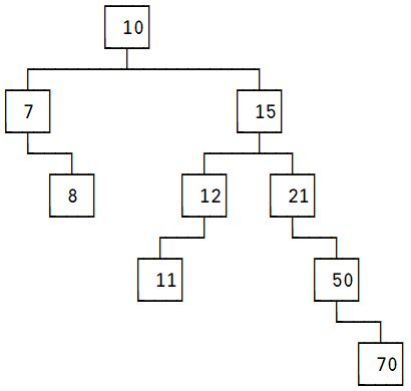
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

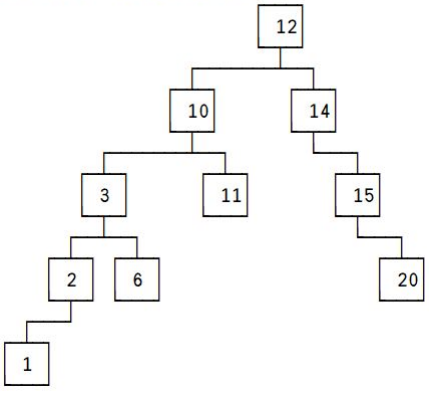
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

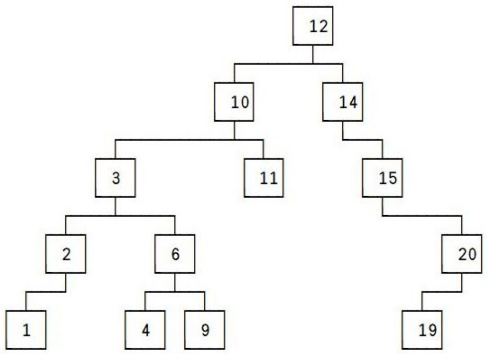
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

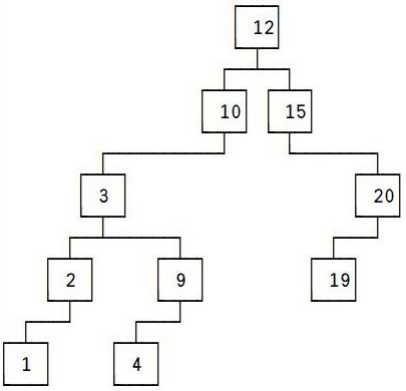
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
4   6  10   5   8  19  16   2  14  13
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
18  11   7
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
5  13  14
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76249 CAUE DAS NEVES MACHADO
24CC1304 - 5 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

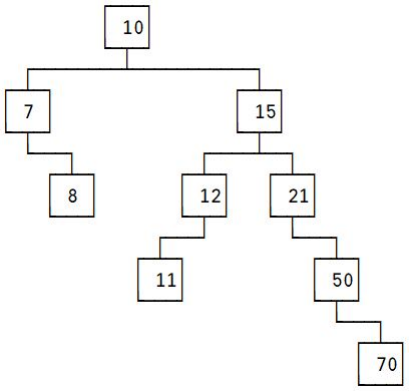
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

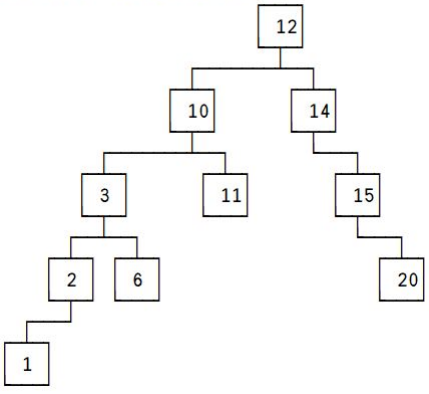
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

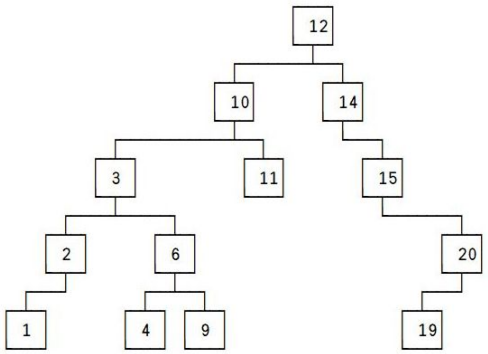
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

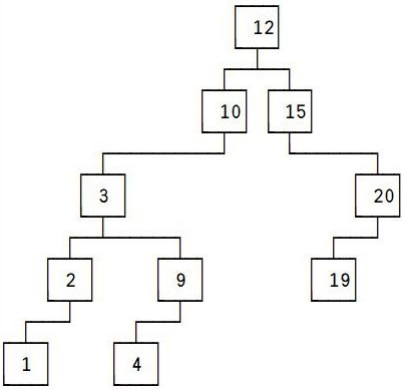
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
20   8  13   6   4  17  16   7  18  14
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
5   3  15
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
7   4  13
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76249 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

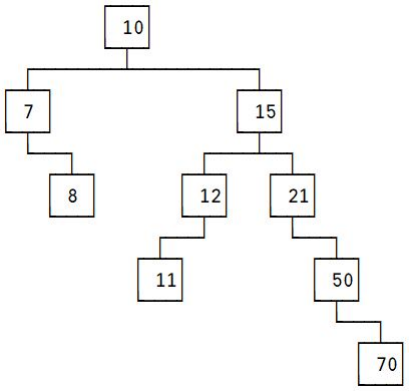
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

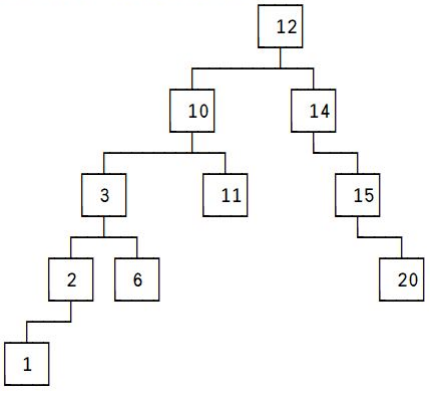
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

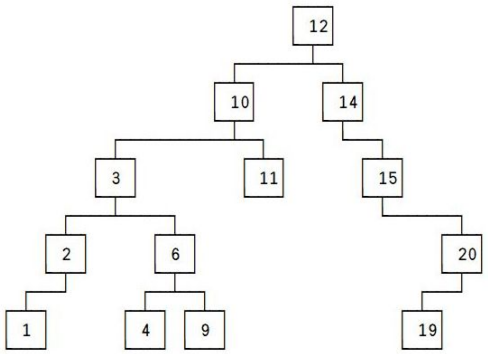
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

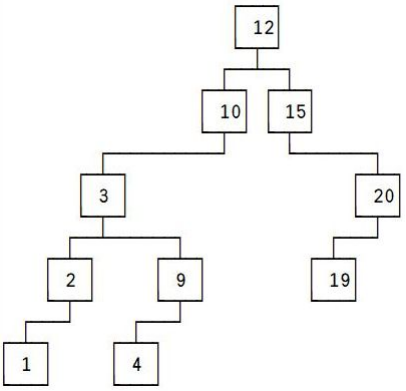
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
6  20  19   9  16  18  14   5   8   3
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
12   7  15
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
19  16   8
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76256 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76263 DAVI HONORIO ROSA
24CC1304 - 7 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

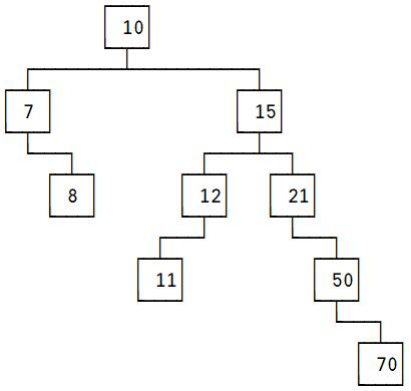
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

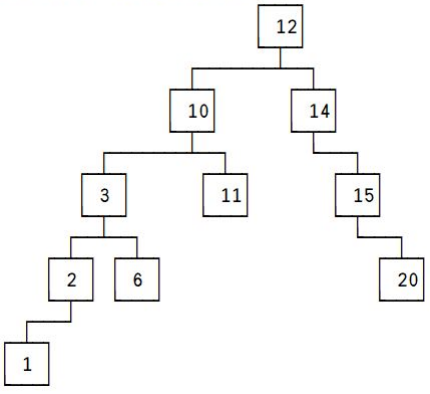
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

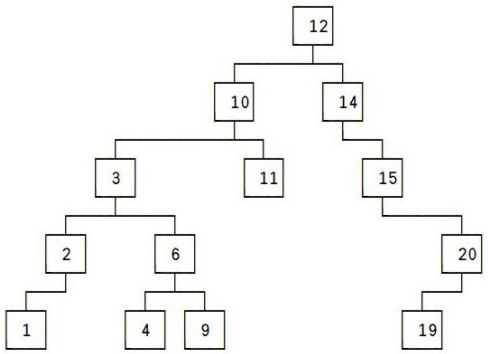
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

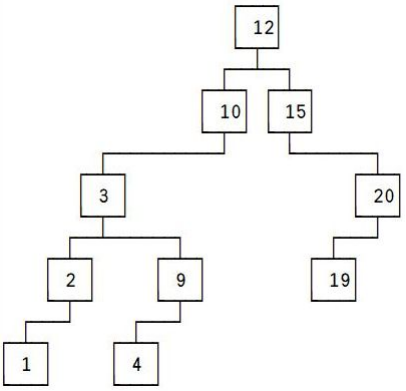
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
19   7   4   5  11  20   8   1  18  14
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
16   3   6
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
18   8  14
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76263 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

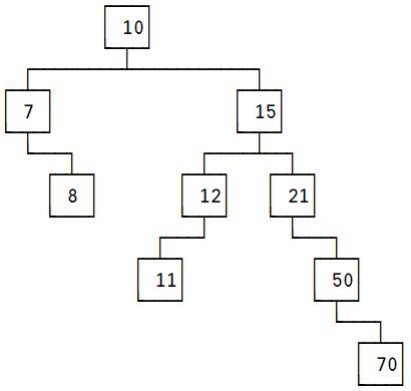
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

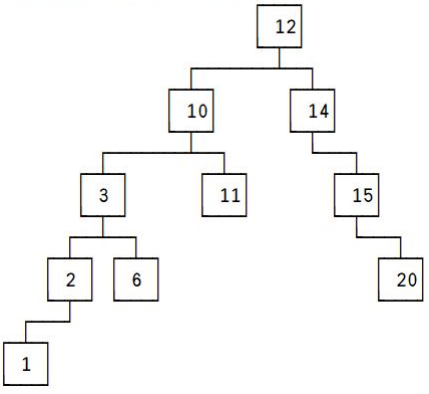
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

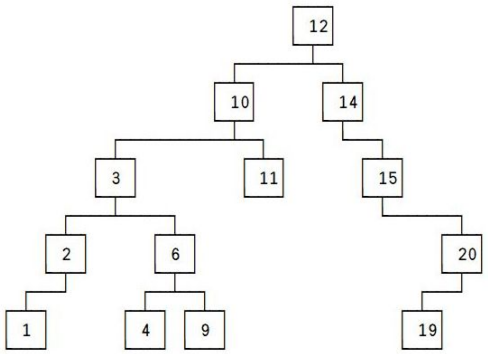
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

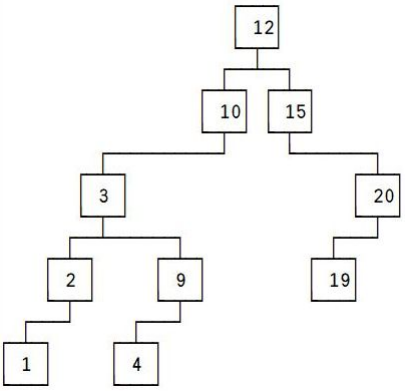
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
12  11   5   1   4   7  10   3  13  15
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
19   8  18
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
3   5  15
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76270 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76287 ENZO KAUAN CARVALHO
24CC1304 - 9 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

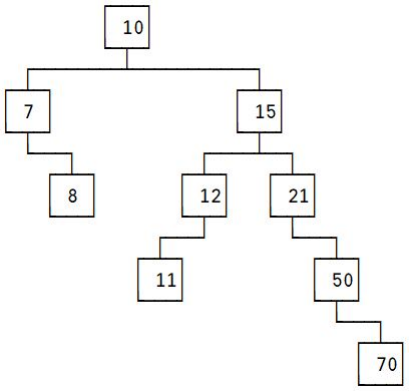
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

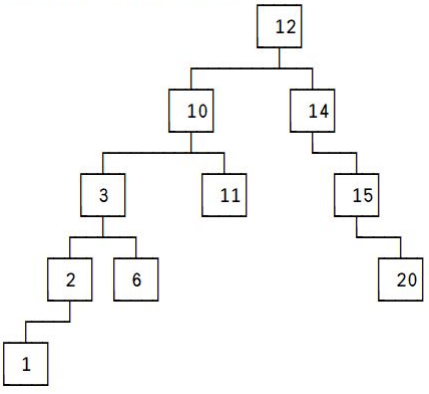
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

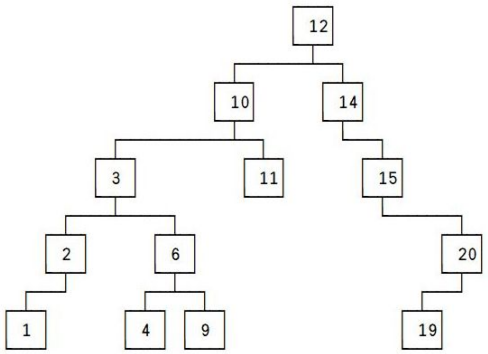
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

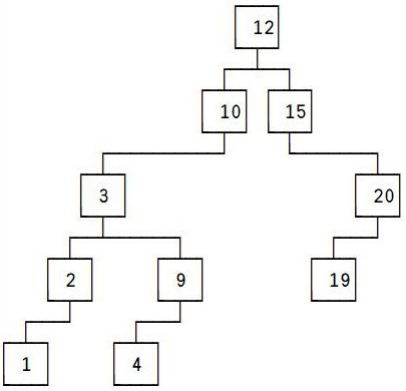
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
19  20  15   1   5   7  18  12   6   4
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
10  13  14
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
4   1  20
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76287 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

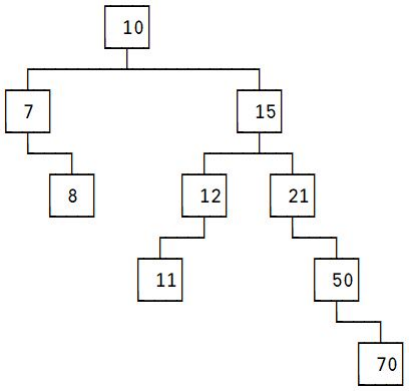
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

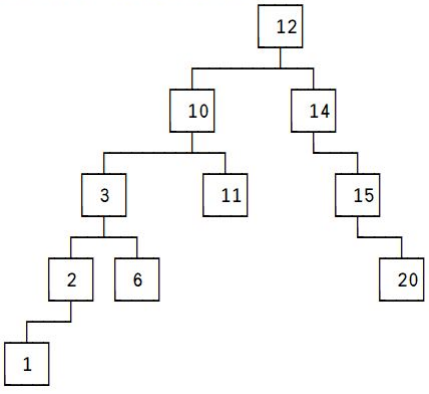
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

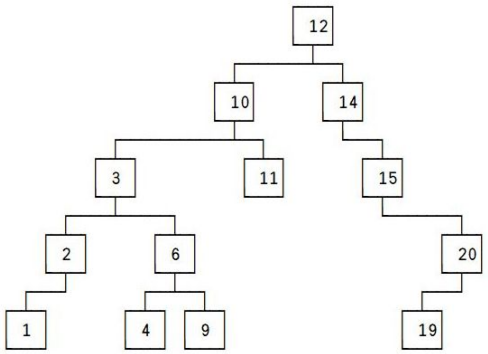
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

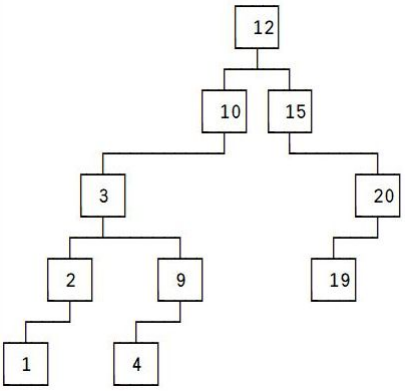
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
6  14  4  15  10  13  11  9  19  16
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
8  5  2
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
14  19  11
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76294 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76306 FELIPE FERNANDES DOS SANTOS
24CC1304 - 11 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

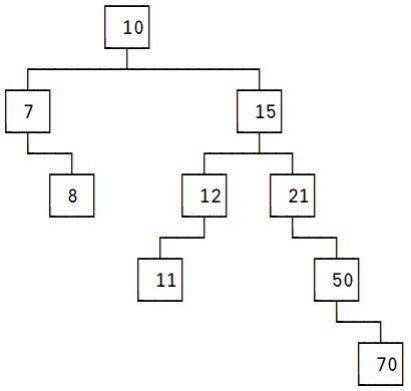
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

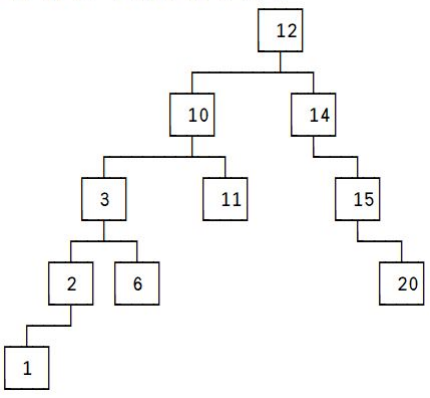
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

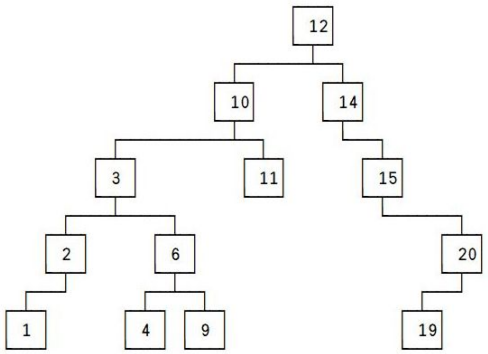
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

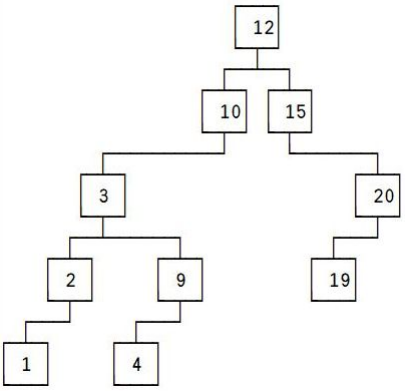
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
8   4  17  20  18  15  19  16   2   9
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
14   5   6
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
16  19  20
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76306 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

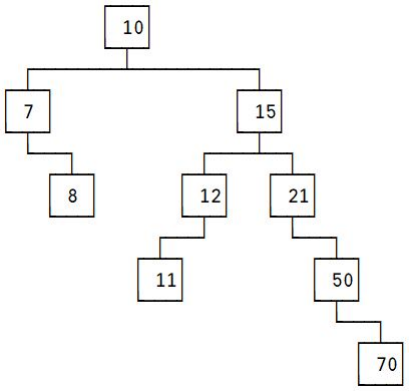
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

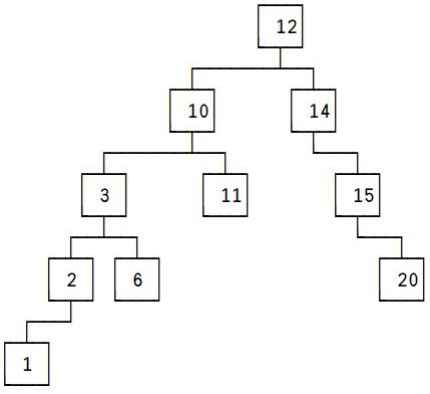
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

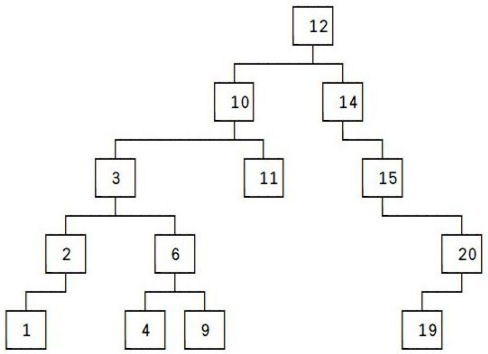
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

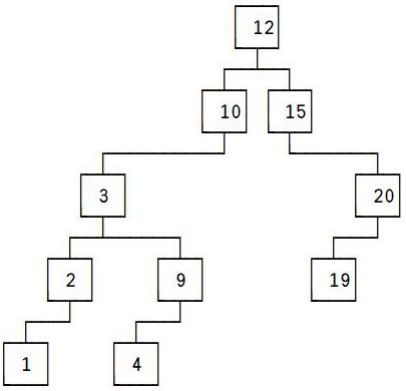
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
17  13   9   5   8   3  18   1   2  20
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
16  15  10
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
20  18   3
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76313 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

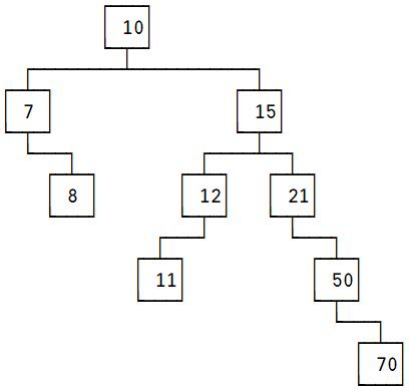
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

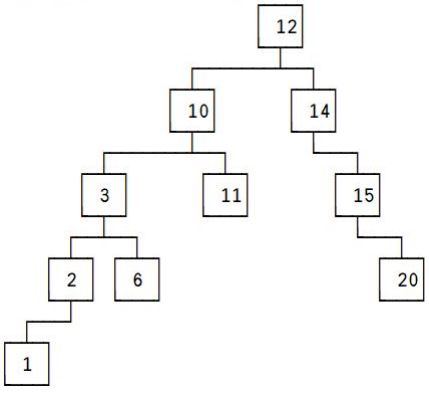
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

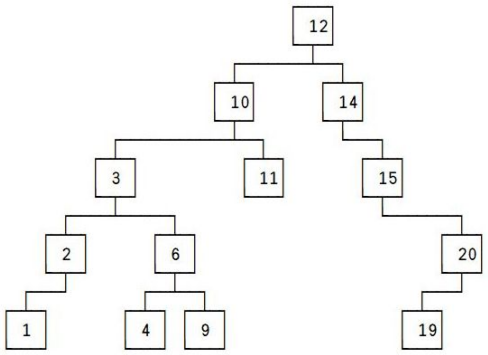
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

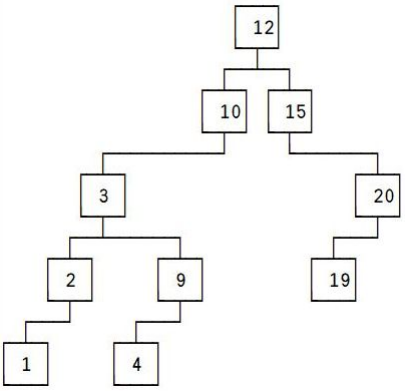
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
9  16  15   2  14   4   5  17   7   8
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
20  10   6
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
7   4   5
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76320 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76337 GUILHERME CAVALER KUSTER
24CC1304 - 14 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

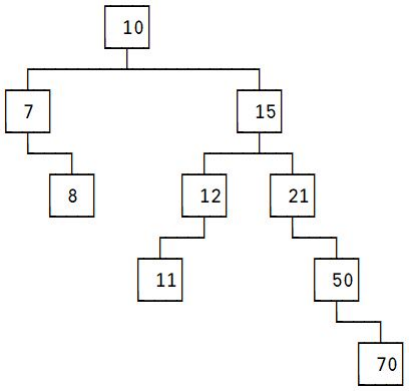
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

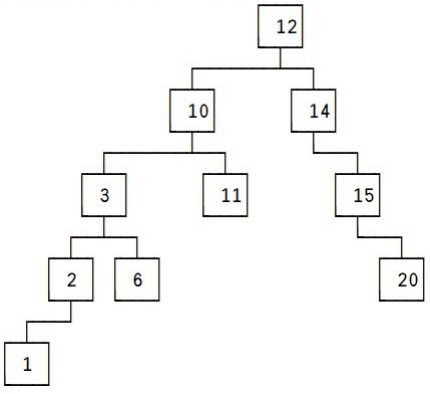
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

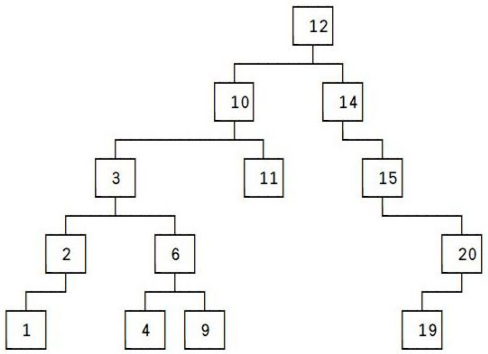
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

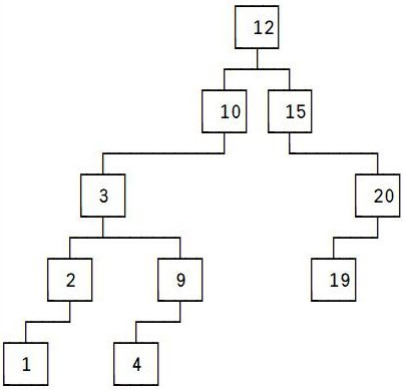
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
8   6   2   9  12  10  15  13  20   7
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
3   4  11
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
9   7  15
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76337 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76344 GUILHERME LUCIANO DE MELO
24CC1304 - 15 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

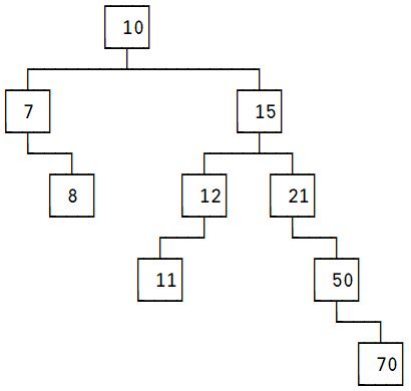
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

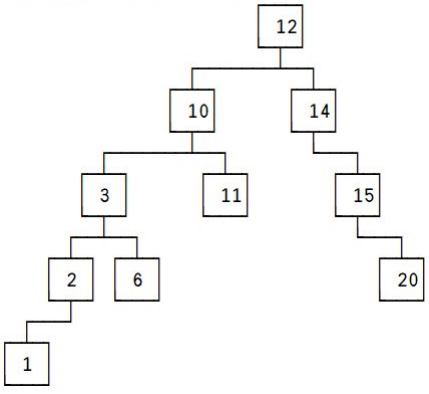
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

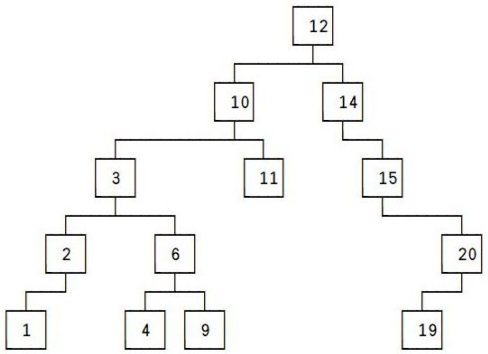
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

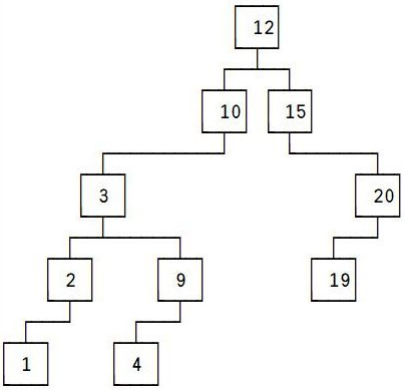
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
20   7  14   4   8   3  11  18  15  17
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
16   1  12
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
18  17  15
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76344 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76351 HENRI KIM PINHEIRO RODRIGUES
24CC1304 - 16 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

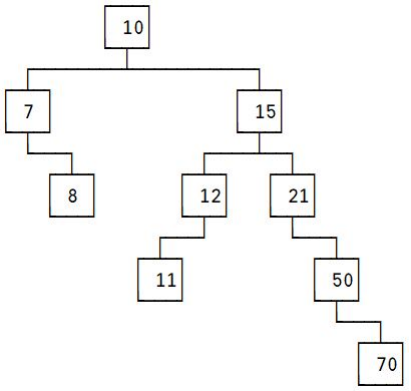
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

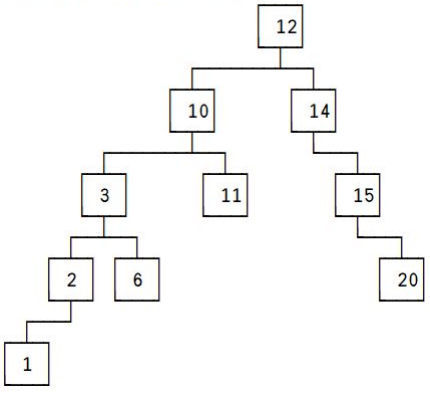
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

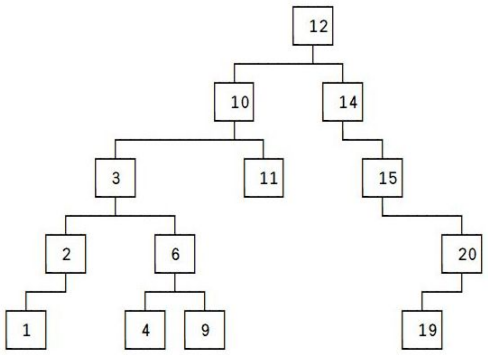
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

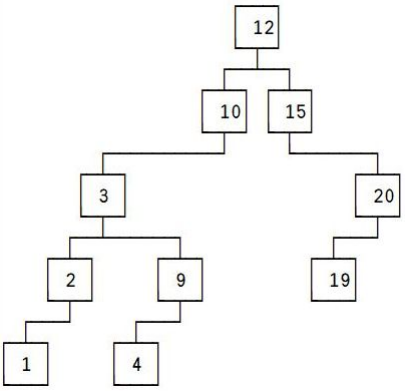
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
11   9  20   3   8   5  19   7  15   2
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
6  10   4
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
9   2   3
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76351 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

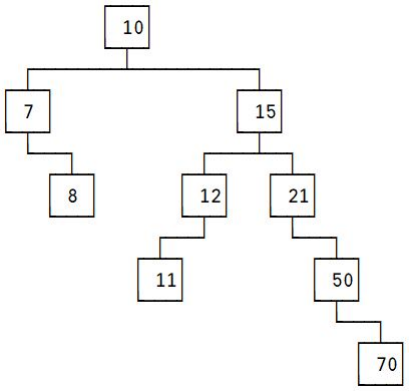
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

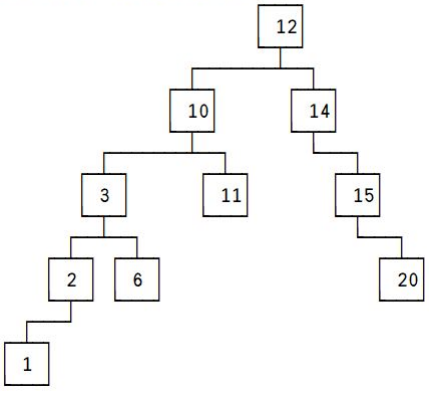
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

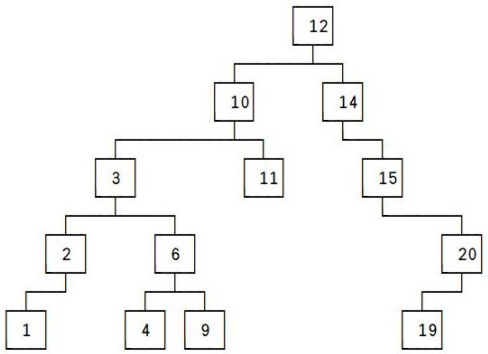
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

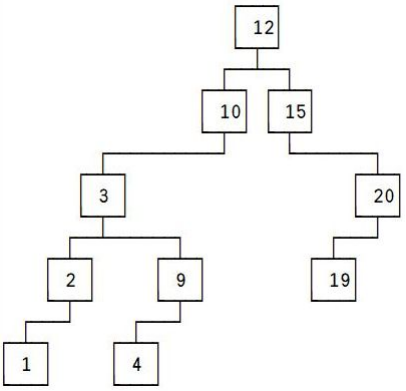
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
12  17   6   8   2  10  19  11  18   7
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
5   9  20
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
8  18  11
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76368 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

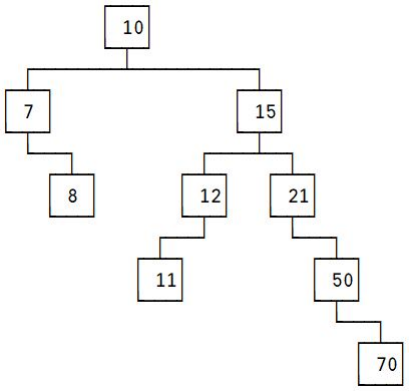
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

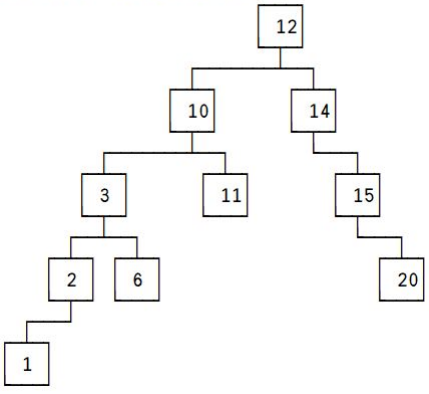
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

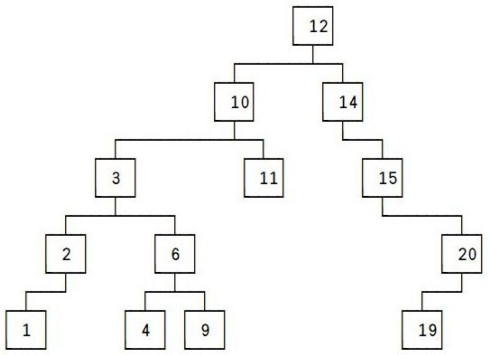
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

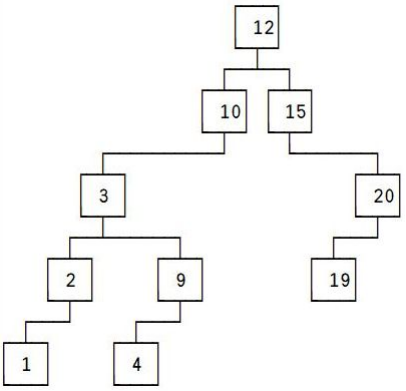
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
2  15  7  14  18  8  4  1  13  3
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
12  19  6
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
7  4  8
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76494 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

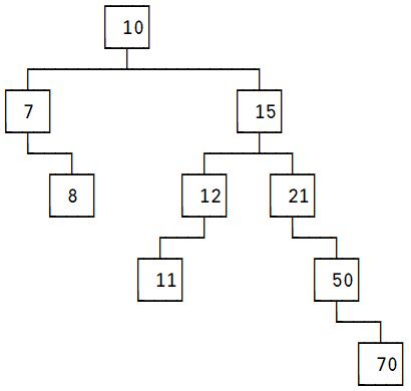
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

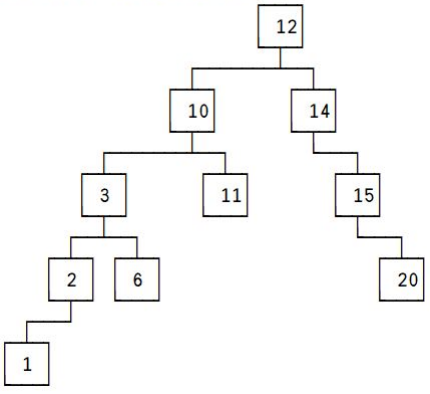
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

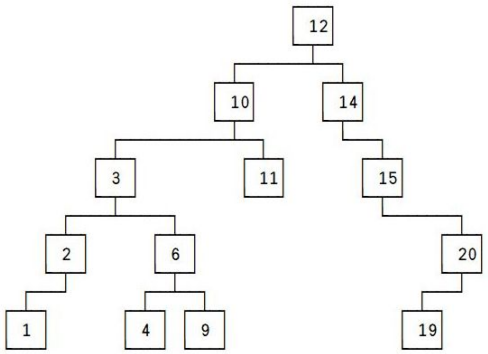
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

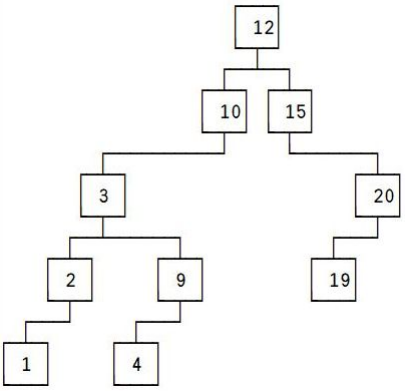
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
6  10  16  13  18  2  7  20  14  5
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
1  19  17
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
10  20  18
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76375 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

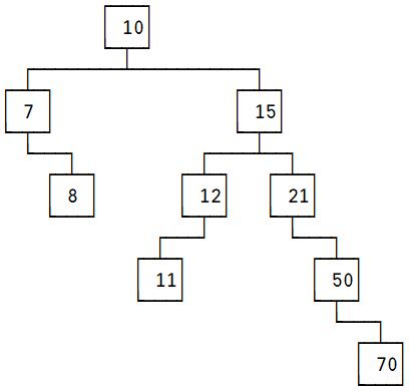
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

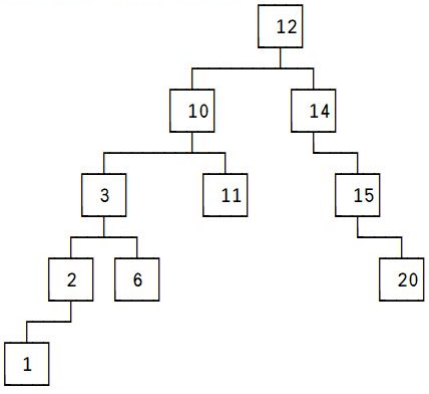
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

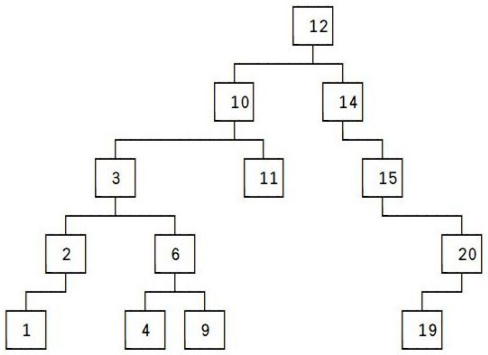
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

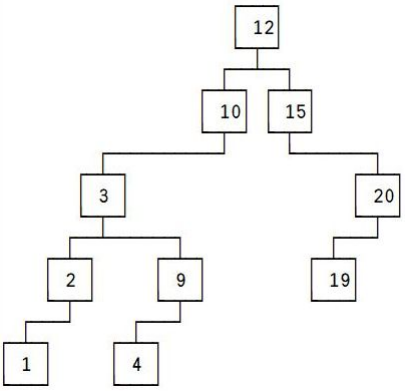
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
16  15  19  17  18   3  13   8  12   7
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
9   5  10
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
7   3  13
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

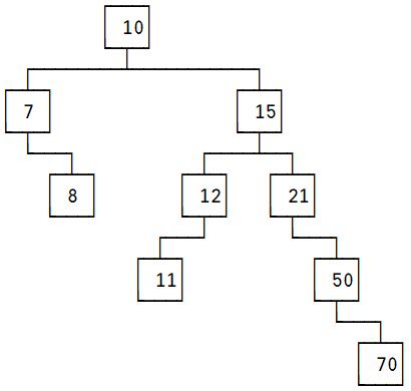
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

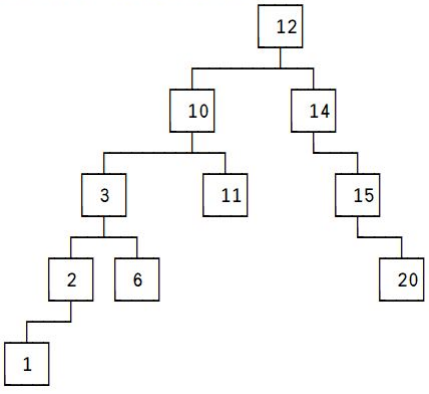
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

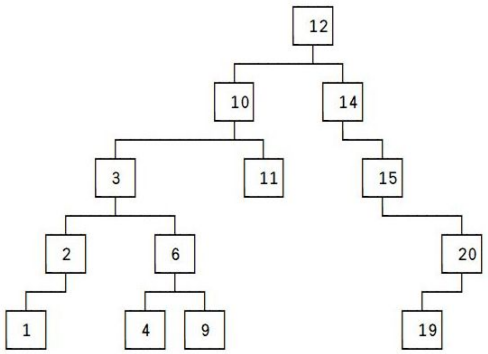
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

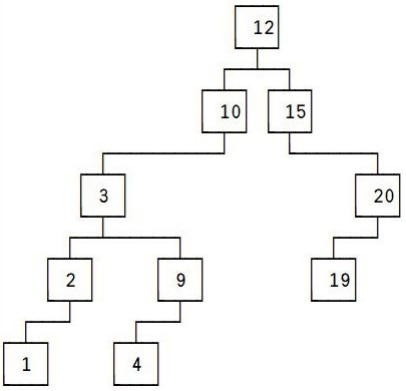
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
8  16  13   2   7   3  10   4  19   5
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
1  15  20
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
3  16  19
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76399 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76401 LUAN TEIXEIRA DIBE
24CC1304 - 22 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

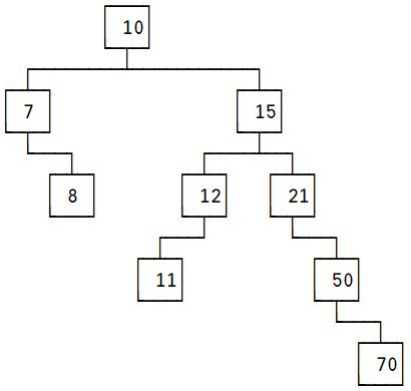
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

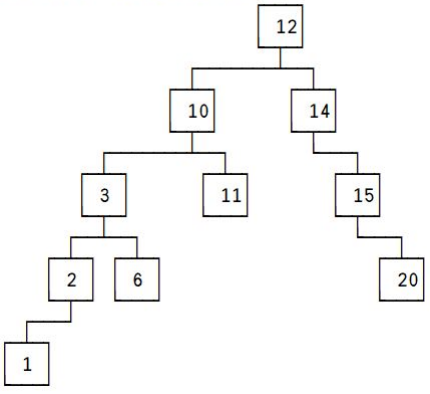
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

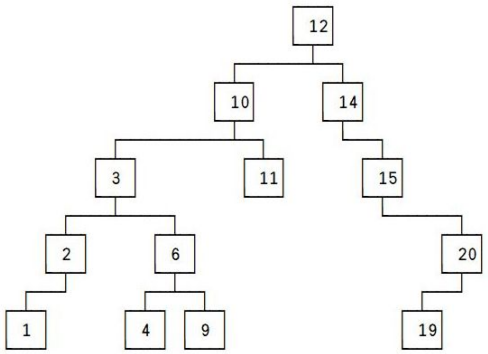
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

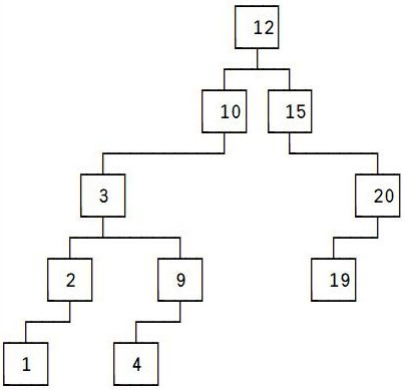
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
17   4   1  11  10   2  20  18   8   6
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
19  14  16
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
20   6  10
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76401 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

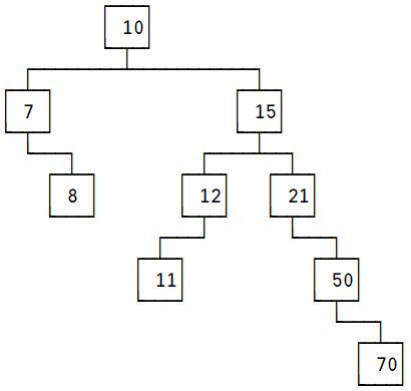
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

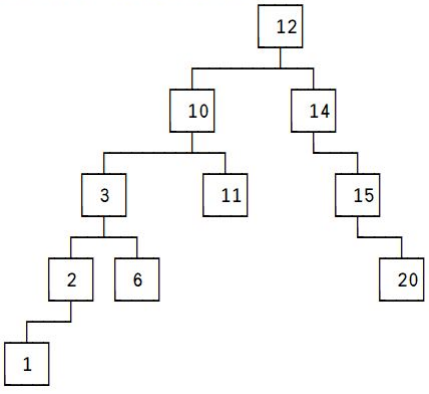
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

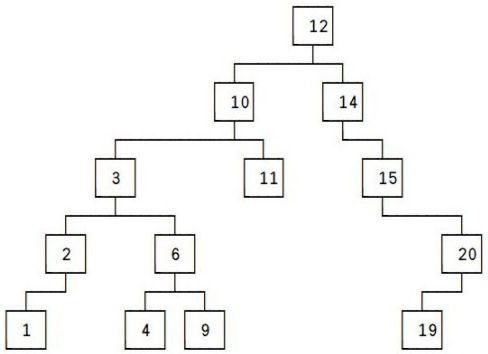
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

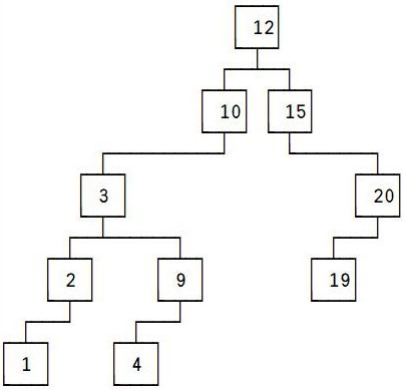
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
14  13   5  15  19  17  10   7  18   9
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
3  20   1
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
7  19  18
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76418 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

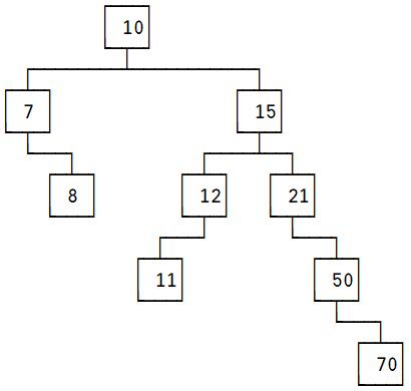
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

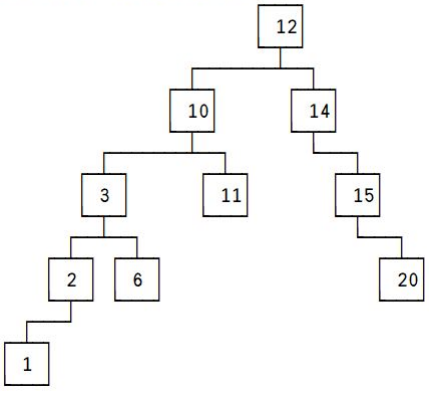
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

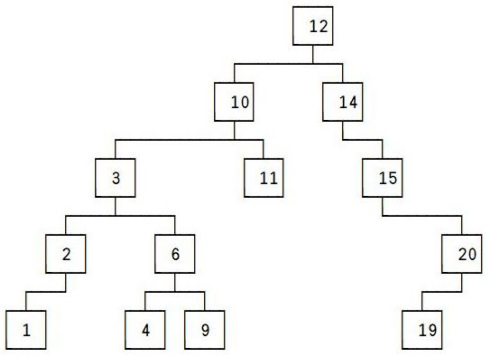
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

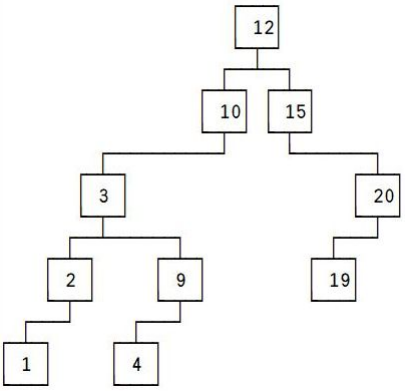
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
9  10   5   3  20   6  15   4  13   7
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
14   2  12
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
13   4   5
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76425 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76432 PAULO GUSTAVO TORRES
24CC1304 - 25 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

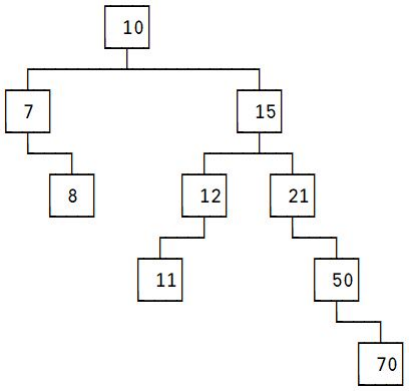
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

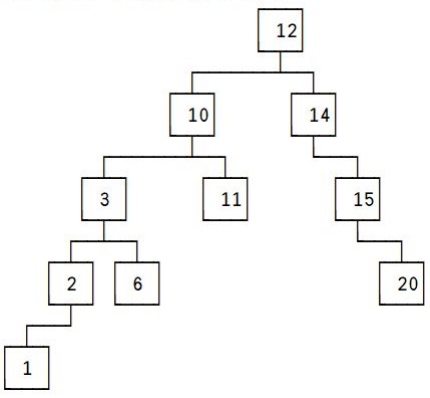
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

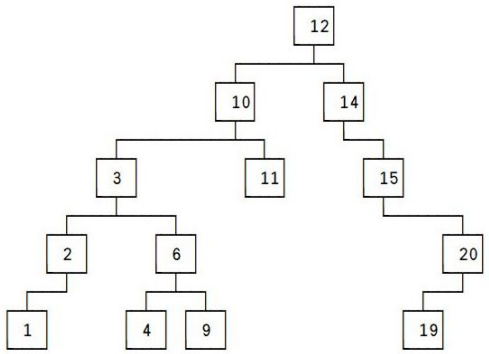
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

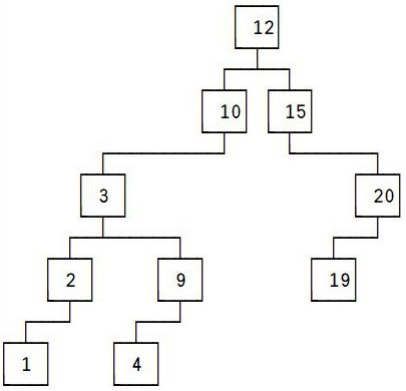
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
19  10  15  16  11   1   9   5   7  17
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
3  14   4
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
10   1  17
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76432 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação 27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76449 PEDRO CAPOTE DA CONCEICAO SOAR
24CC1304 - 26 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

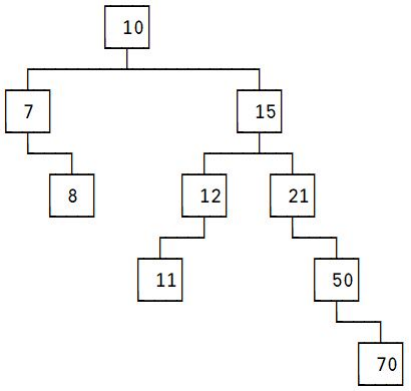
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	$\leftarrow$ *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

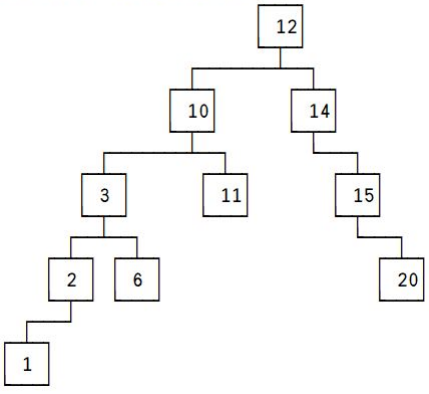
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

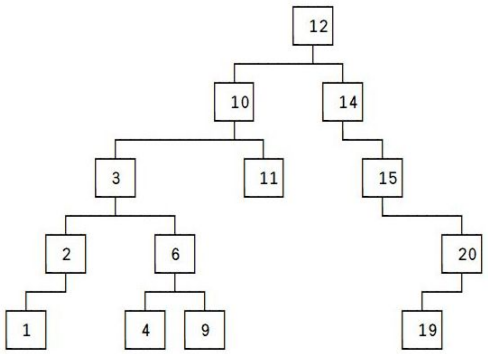
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

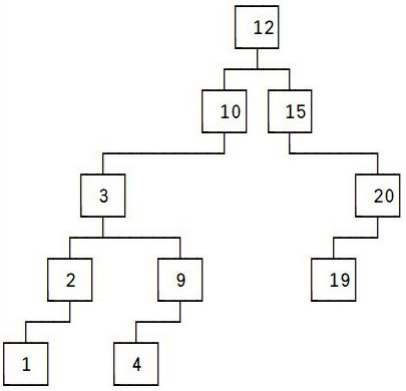
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
17  15   1   2   5  13   7  18   3  19
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
14  16  20
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
18  19  13
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76449 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

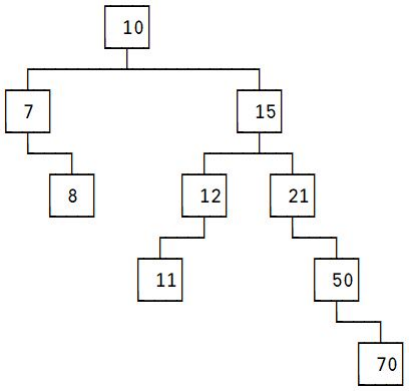
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

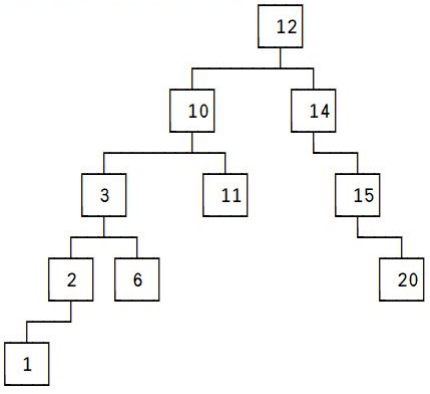
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

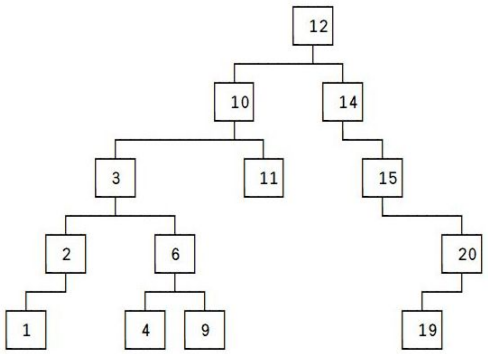
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

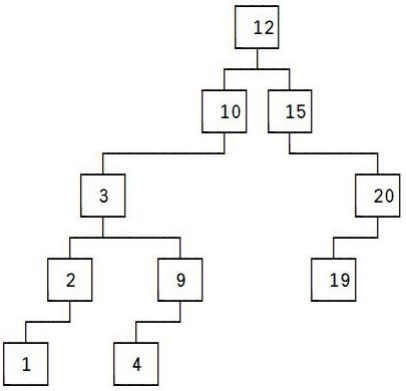
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
17  14  19   8  18   7  13  20   9  12
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
10  11   1
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
13  19  12
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76456 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

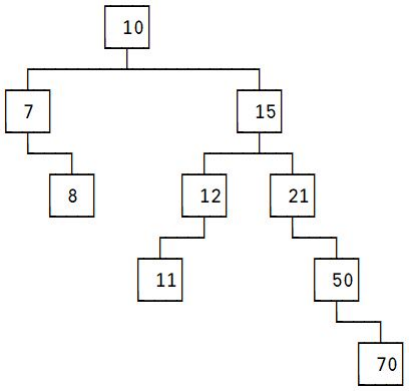
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

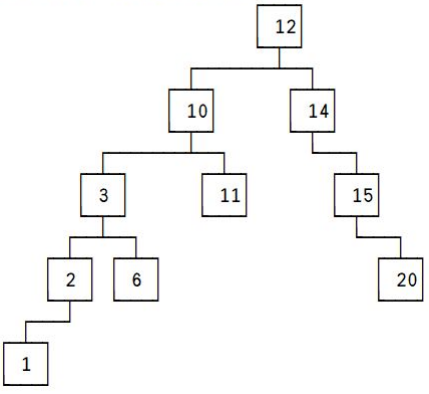
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

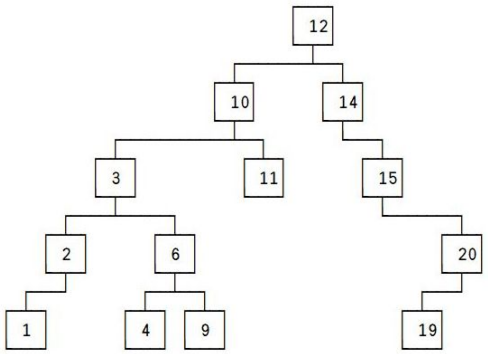
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

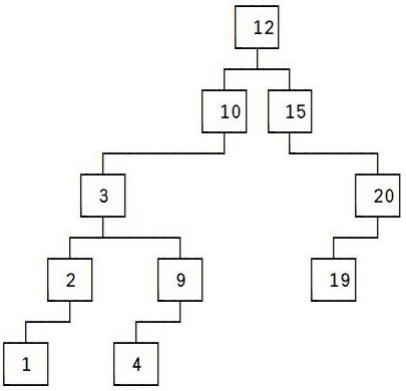
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
5  16  19  11   6   4  14  12   7   9
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
2   1   8
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
7   4  19
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76463 -

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

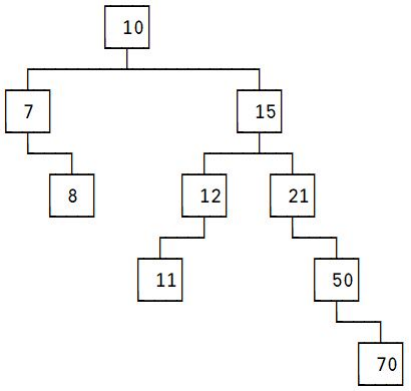
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

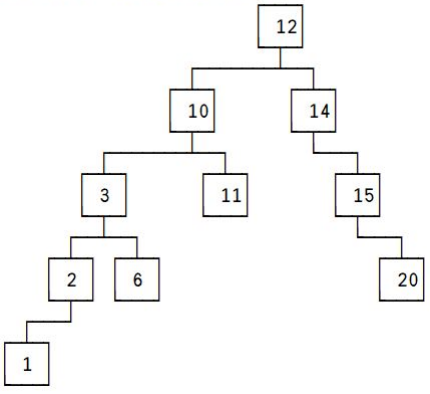
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

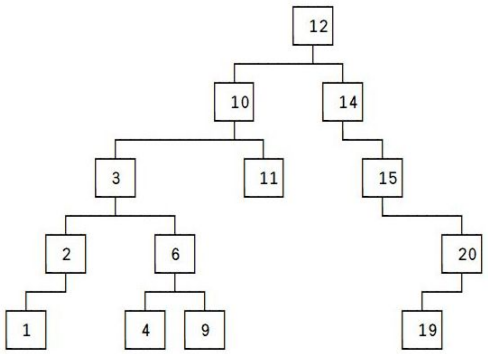
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

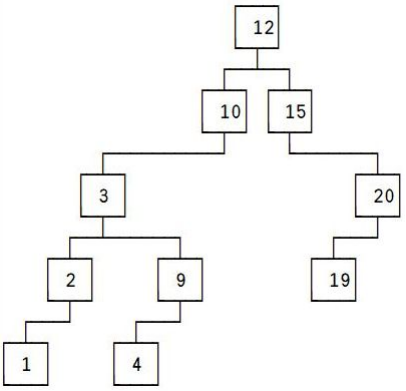
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
9  11   7   8  10  20   4  12  18   1
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
6  19   2
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
10   7   8
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76470 -

CEP-UFPR-UTFPR-PUC/Pr-UP Sistemas de Informação
27/07/2024 - 09:24:18.5
Ciência da Computação Prof Dr P Kantek (pkantek@gmail.com)
Árvores vivxq91a V: 1.03
76487 WILLIAN IELEM PAVOSKI
24CC1304 - 30 _____ / _____ / _____

Árvores

Estruturas de dados muito importantes na Ciência da Computação. Elas permitem implementar relações de hierarquia de maneira muito fácil. Vejamos uma aplicação prática bem simples e importante.

Se você precisar consultar a posição de um elemento em um vetor, talvez o algoritmo mais simples seja percorrer linearmente o vetor. Acompanhe:

```
vetor[100]={...}
k,i=0
enquanto i<100
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

O problema deste algoritmo é a sua ineficência. Suponha que em vez de 100 elementos são 10 milhões.

Melhoria 1 É o uso de uma variável do tipo *sentinela*. Ela sempre sinaliza alguma condição. Neste caso, vamos incluir a chave buscada no final do vetor. Agindo assim, ele sempre será encontrado (*acabei de pôr ele lá...*) e isso elimina um dos testes do ciclo. Acompanhe

```
vetor[101]={...}
vetor[101]=k
i=0
enquanto vetor[i] != k
   i=i+1
fim{enquanto}
se i=101
   não achou
senão
   achou na posição i
fim{se}
```

Melhoria 2 Ordenar o vetor. Agora, a decisão quanto a inexistência pode ser encontrada antes, sem precisar ir até o final do vetor.

```
vetorord[100]={...}
k,i=0
enquanto i<100 E vetorord[i]<=k
   se vetor[i]=k
      achou na posição i
      retorne
   fim{se}
   i=i+1
fim{enquanto}
não achou
```

Convenhamos, é pouco lucro depois de tanto trabalho. Ordenar um vetor é sempre muito *caro*. Os melhores algoritmos de ordenação custam $n.log_2 n$ enquanto os mais caros custam n^2. A propósito o *custo* de um algoritmo é uma função que em geral se refere ao tamanho da entrada n e a quantidade de operações elementares sobre cada elemento da entrada (em número de n, já vimos).

Então, uma vez que já foi gasto o processamento necessário para ordenar o vetor, a busca pode ser muito melhorada, a partir do algoritmo de busca binária. A idéia é quebrar um universo (ordenado) em 2 partes de mesmo tamanho e descobrir se a chave buscada está na primeira ou na segunda metade. Se estiver na primeira, você abandona a segunda e vice-versa. O universo ficou reduzido à metade. Agora, o algoritmo é reaplicado na primeira metade, e agora é eliminado 1/4 do universo. Em poucas instruções (na verdade em $log_2 n$ instruções) a resposta é encontrada.

Antes de continuar, relembremos a tabela de logaritmo de base 2:

n	2	4	8	16	32	64	...	1024
$log_2 n$	1	2	3	4	5	6	...	10

```
k,inicio = 0
fim = tamanho(lista) - 1
enquanto inicio <= fim
  meio = (inicio + fim) // 2
  se lista[meio] = k
    retorne meio
  senão
    se lista[meio] < elemento_buscado:
      inicio = meio + 1  # Busca na metade sup
    senão
      fim = meio - 1 #busca na metade superior
    fim{se}
  fim{se}
fim{enquanto}
return -1  # Elemento não encontrado
```

Como se pode ver, o desempenho é muito bom, mas não nos esquecemos do custo de ordenar/manter o vetor ordenado.

ABP A proposta que vem a seguir, tem o mesmo desempenho da busca linear ($log_2 n$) sem exigir que os dados esteja ordenados. Ou seja, é o melhor de dois mundos. Na verdade, paradoxalmente, o desempenho da árvore binária de busca fica pior quanto mais ordenados estiverem os dados de entrada. O paradoxo é que ela funciona melhor se os dados estiverem bagunçados do que se estiverem ordenados.

O nome oficial desta estrutura é **ABP** - árvore binária de pesquisa e ela:

- é uma árvore, formada por nodos hierárquicos
- de grau=2 (ou binária) o que significa que o número máximo de filhos é 2
- de pesquisa, o que significa que cada nodo tem uma chave. Depois, todas as chaves menores que ela, ficarão nos filhos à ESQUERDA e as chaves maiores do que ela ficarão nos filhos à DIREITA.

Árvore Estrutura hierárquica formada por nodos. Cada nodo tem um pai e pode ter n filhos. Existe um único nodo que não tem pai. É onde a árvore começa, e ele é chamado RAIZ. Tradicionalmente ele é desenhado no alto do espaço, pelo que, na ciência da computação, ao contrário da botânica as árvores crescem para baixo.

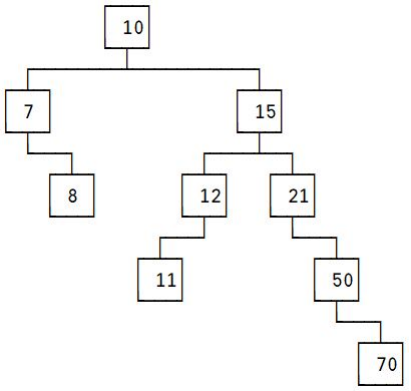
O número máximo de filhos, n é o GRAU da árvore.

Ainda seguindo a comparação botânica, os nodos que não têm filhos, são chamados nodos FOLHA.

Uma das maneiras mais compactas e úteis de descrever uma árvore é usando recursividade. Então, uma árvore é um conjunto de um nodo raiz e n sub-árvores vinculadas à raiz. Para esta definição funcionar, só se precisa aceitar a possibilidade de uma árvore vazia (que é o caso básico da recursividade). Você pode não acreditar mas esta visão simplifica enormemente muitos algoritmos de árvore.

Implementação Árvores podem ser implementadas usando apontadores, quando então serão muito, mas muito eficientes, ou usando cursores, quando então terão menos eficiência (por causa da dupla indireção). Entretanto, é muito mais fácil programar, entender e sobretudo depurar programas de árvores quando eles usam cursores. Daí que vai ser nossa opção aqui. Antes de continuar, pense que se extremo desempenho é requerido, costuma-se desenvolver usando cursores e depois que tudo estiver funcionando, trocar cursor por apontador.

Exemplo de uma ABP



Esta árvore, usando cursores, terá a seguinte tabela de cursores

1	2	4	10	← *raiz*
2	-1	3	7	
3	-1	-1	8	
4	6	5	15	
5	-1	8	21	
6	7	-1	12	
7	-1	-1	11	
8	-1	9	50	
9	-1	-1	70	

Na tabela acima (de cursores) costuma-se combinar que a raiz da árvore aparece sempre na primeira linha da tabela. A ausência de filhos é representada por um valor inválido, chamado TERMINADOR. É um valor que nunca poderá aparecer como índice válido na tabela. No nosso caso, -1.

Altura Para uma dada árvore, a altura é definida como o MAIOR caminho possível entre a raiz e qualquer árvore. No exemplo acima, é 4.

```
função altura(atual)
  se arv[atual][1]=-1 E arv[atual][2]=-1
    retorne 0
  senão
    se altura(arv[atual][1]>altura(arv[atual][2])
      retorne 1+altura(arv[atual][1])
    senão
      retorne 1+altura(arv[atual][2])
    fim{se}
  fim{se}
fim{função}
imprima(altura(1))
```

Depois de fazer `global A = 0`, e chamar altura(0,1) quando a função acabar, em `A` está a altura da árvore.

Caminhamento Ao algoritmo que visita TODOS os nodos de uma árvore (para listar, totalizar, imprimir, contar, etc) dá-se o nome de CAMINHAMENTO. Como não existe uma ordem natural para isto, definem-se muitos caminhamentos: em-ordem, pré-ordem, pós-ordem, em largura, A*, etc. Aqui, novamente a abordagem recursiva é sensacional, senão vejamos

```
função caminhamento-pre(raiz)
  se raiz = -1
    retorne
  fim{se}
  imprima (arvore[raiz][3])
  caminhamento-pre(arvore[raiz][1])
  caminhamento-pre(arvore[raiz][2])
fim{função}

função caminhamento-emordem(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-emordem(arvore[raiz][1])
  imprima (arvore[raiz][3])
  caminhamento-emordem(arvore[raiz][2])
fim{função}

função caminhamento-pos(raiz)
  se raiz = -1
    retorne
  fim{se}
  caminhamento-pos(arvore[raiz][1])
  caminhamento-pos(arvore[raiz][2])
  imprima (arvore[raiz][3])
fim{função}
```

Falta ainda o caminhamento em largura que é obtido usando-se uma FILA como estrutura auxiliar. Acompanhe

```
fila=[]
fila.append(raiz)
enquanto houver fila
   nodo=fila.pop()
   print(arvore[nodo][3])
   se arvore[nodo][1] != -1
      fila.append(arvore[nodo][1])
   fim{se}
   se arvore[nodo][2] != -1
      fila.append(arvore[nodo][2])
   fim{se}
fim{enquanto}
```

Busca A busca é o principal uso de uma ABP. Dada a ABP e uma chave k quer-se descobrir se $k \in ABP$ ou se $k \notin ABP$. Para quem já esqueceu é o mesmo problema que se colocou lá no início desta folha. Construída a ABP (sem que os dados tenham sido ordenados), a busca é

```
função busca(k)
  raiz=1
  pai=-1
  esqdir=-1
  enquanto arvore[raiz][3] != k
    pai=raiz
    se k > arvore[raiz][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    raiz=arvore[raiz][esqdir]
    se raiz = -1
      retorne esqdir,pai,-1 # não achou
    fim{se}
  fim{enquanto}
  retorne esqdir,pai,raiz
fim{função}
```

Apenas para comparar e admirar a beleza dos algoritmos recursivos, eis a mesma coisa, (busca em ABP) na versão recursiva

```
função buscarec(raiz,k)
  se raiz = -1
    retorne -1
  fim{se}
  se arvore[raiz][3]=k
    retorne raiz
  fim{se}
  se k<arvore[raiz][3]
    buscarec(arvore[raiz][1], k)
  senão
    buscarec(arvore[raiz][2], k)
  fim{se}
fim{função}
```

Criação e inclusão Para criar uma ABP usando cursores é muito fácil.

```
global arv [100][3]
global ultimo=0
```

Agora para incluir um elemento na árvore, é só fazer

```
função incluiabp(raiz,k)
  ultimo=ultimo+1
  arv[ultimo][]=-1, -1, k
  onde = raiz
  enquanto onde != -1
    pai = onde
    se k > arvore[onde][3]
      esqdir = 2
    senão
      esqdir = 1
    fim{se}
    onde = arvore[onde][esqdir]
  fim{enquanto}
  arvore[pai][esqdir]=ultimo
  retorne ultimo
fim{função}
```

Exclusão A exclusão é o algoritmo mais complexo da série. Por uma razão simples, já que há que examinar 3 casos distintos:

- Exclusão de uma folha: esta é fácil, basta colocar -1 no apontador para esta folha.
- O nodo a excluir tem 1 filho: colocar o endereço do filho no pai do nodo a excluir.
- O nodo a excluir tem 2 filhos: há que se buscar o sucessor do nodo a excluir. Achando-o, ele é temporariamente excluido, o que sempre é possível já que o sucessor nunca terá 2 filhos, e depois disso, o sucessor ocupa o lugar do nodo original a excluir. A árvore continua ordenada e vida que segue. Acompanhe

```
função excluiabp(raiz, k)
  esqdir,pai,raiz=buscaabp(raiz,k)
  se arvore[raiz][1]=-1 E arvore[raiz][2]=-1
    arvore[pai][esqdir]=-1
    retorne raiz
  senão
    se arvore[raiz][2]== -1
      arvore[pai][esqdir]=arvore[raiz][1]
      retorne raiz
    senão
      se arvore[raiz][1]== -1
        arvore[pai][esqdir] = arvore[raiz][2]
        retorne raiz
      senão
        pai=raiz
        raiz=arvore[raiz][2]
        enquanto raiz != -1
          save=raiz
          raiz=arvore[raiz][1]
        fim{enquanto}
        tempo=excluiabp(arvore[save][3])
        arvore[pai][3]=arvore[tempo][3]
        retorne save
      fim{se}
    fim{se}
  fim{se}
fim{função}
```

Outras árvores

Huffman Usadas no algoritmo ótimo de compressão de caracteres que leva o nome de seu criador.

Tries Muito usados em corretores ortográficos entre outras aplicações.

K-D Para descrição de objetos n-dimensionais e cálculo fácil de métricas sobre eles.

DOM e BOM Uma árvore que descreve uma página internet, para que o browser possa fazer seu trabalho. DOM=Document Object Module e BOM=Browser Object Module.

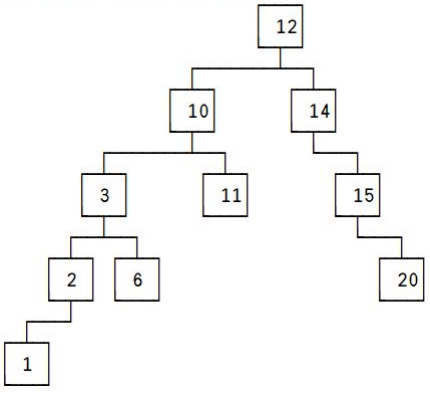
PDF Todo arquivo PDF é constituído ao redor de uma árvore que descreve os componentes do arquivo

ordenação Um dos melhores algoritmos de ordenação sugere construir uma ABP e depois percorrê-la.

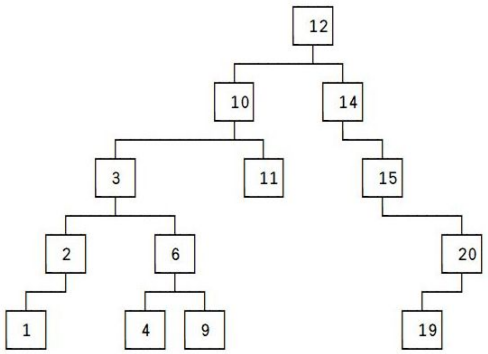
heap Uma estrutura em árvore que é disposta em um vetor sequencial e é base de um excelente método de ordenação.

Um exemplo Seja o mesmo enunciado abaixo para o caso da árvore com 10 valores:

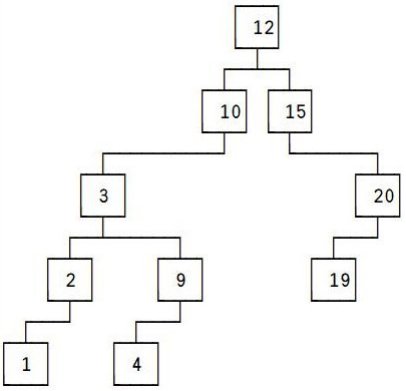
12 10 14 3 11 2 15 1 20 6



Agora, a inclusão de 9, 19, 4 e a árvore fica:



Agora a exclusão de 6, 14, 11 e a árvore fica



As respostas numéricas para este exemplo, são:

```
12 10 14 3 11 15  2  6 20 1
12 10 14 3 11 15  2  6 20 1 4  9 19
12 10 15 3 20  2  9 19  1 4
```

☞ Para você fazer

O exercício pedido é razoavelmente simples: Trata-se de construir uma ABP com os valores abaixo, na ordem em que eles aparecem:

```
5  19  9  20  11  6  12  16  15  14
```

A resposta deve ser dada de 2 maneiras: o desenho da árvore (que pode ser feita em um papel anexo, identificar como **árvore 1**) e o caminhamento EM LARGURA da árvore criada

Depois, você deve incluir os seguintes valores na árvore acima:

```
3  17  18
```

e redesenhar a árvore, (**árvore 2**) agora com os 3 elementos a mais. Também refazer o caminhamento em largura e escrever os nodos:

Finalmente, você deve excluir os valores

```
11  16  6
```

redesenhar a árvore (**árvore 3**), refazendo pela terceira vez o caminhamento em largura, escrevendo os nodos:

304-76487 -