_____ / _____ / _____

Manuseio de vetores Os vetores são elementos fundamentais na programação. Em sentido amplo, quase tudo pode ser entendido como um vetor: a web pode ser um vetor onde cada página é um elemento desse vetor (isto sob o ponto de vista do google, por exemplo), um banco de dados pode ser um vetor (cada registro é um elemento), a memória do computador é um vetor (cada byte é um elemento), e assim por diante.

Assim, vamos definir algumas operações fundamentais em vetores, com a generalidade que nos for possível.

inclusão	adicionar um novo elemento ao vetor
exclusão	eliminar um elemento do vetor
ordenação	mudar a ordem dos elementos do vetor
busca	dado um vetor V e uma chave K, descobrir se K $\in$ V

Ordenação Ordenar um vetor é trocar as posições de seus elementos de maneira a que eles entrem em alguma ordem (crescente ou decrescente).

Em termos matemáticos, seja X um conjunto composto por i elementos entre os quais se pode estabecer uma relação de ordem. Dados X_i e X_j com $i \neq j$, sempre pode-se estabelecer $X_i > X_j$ ou $X_i = X_j$ ou $X_i < X_j$, onde valha sempre a propriedade:

ordem crescente	V[i+i] $\geq$ V[i], $\forall i$
ordem decrescente	V[i+i] $\leq$ V[i], $\forall i$
ordem estritamente crescente	V[i+i] $>$ V[i], $\forall i$
ordem estritamente decrescente	V[i+i] $<$ V[i], $\forall i$

O algoritmo de ordenação é o mais extensivamente estudado (ocupa 80% das quase 600 páginas do volume 2 do "The Art of computer programming" do Knuth) e isso acontece porque:

★ É um algoritmo caro (demorado).
★ É o algoritmo mais usado no universo. (James Martin, sugere que 40% do tempo de todos os computadores da terra é gasto neste algoritmo).

Um algoritmo de ordenação é estável se no caso particular em que se k[i] = k[j], a ordem original dos dados é preservada. É sempre bom que um método de ordenação seja estável.

Veja-se alguns exemplos:

1 4 6 21 60 62	estritamente crescente
1 4 6 6 21 60	crescente
2 2 2 2 2	crescente
1 4 21 19 50 60	desordenado
60 21 6 4 1	estritamente decrescente
60 21 21 6 4 1	decrescente
2 2 2 2 2	decrescente
60 50 19 21 4 1	desordenado
1 2 3 4 5 6	
90 61 15 4 2	
12 16 18 18 90	
12 16 18 17 90	
90 61 15 17 4 2	
17 2 89 7 48 1 90	

A seguir, dois dos algoritmos de ordenação possíveis (Note que são conhecidos mais de 20 algoritmos de ordenação).

Método de seleção direta

Ordenação por seleção direta:

```
 1: Algoritmo SELEÇÃO(V:vetor[1..n]de inteiro)
 2: I,J,COR,INUI : inteiro
 3: I ← 1
 4: enquanto (I ≤ n)
 5:     COR ← V[I]
 6:     INUI ← I
 7:     J ← I + 1
 8:     enquanto (J ≤ n)
 9:         se V[J] < COR
10:             COR ← V[J]
11:             INUI ← J
12:         fim{se}
13:         J ← J + 1
14:     fim{enquanto}
15:     TROCA (I, INUI)
16:     I ← I + 1
17: fim{enquanto}
18: devolva (V)
19: fim algoritmo
```

Bolha

Ordenação por bolha

```
 1: Algoritmo BOLHA (V:vetor[1..n]de inteiro)
 2: inteiro I, J
 3: I ← n
 4: enquanto (I ≥ 2)
 5:     J ← 2
 6:     enquanto (J ≤ I)
 7:         se (V[J-1] > V[J])
 8:             TROCA (J, J-1)
 9:         fim{se}
10:         J← J + 1
11:     fim{enquanto}
12:     I ← I - 1
13: fim{enquanto}
14: devolva V
15: fim algoritmo
```

Inclusão Note-se que a maioria das linguagens não permite a mudança dinâmica do tamanho de um vetor. (Por exemplo, em VG, após definir `V : vetor[1..15] de inteiro` não é possível durante o programa mudar o limite superior de 15 para 16 ou outro número qualquer). Assim, se pretendermos fazer inclusões em um vetor, é necessário definí-lo com espaço livre ao final. Para isso funcionar, temos que ter uma variável de controle (um bom nome para ela é ULTIMO), que contém a última ocorrência válida no vetor.

No exemplo do VG, se quisermos 15 elementos válidos, mas com espaço para futuras inclusões, faremos:

```
V : vetor [1..25] de inteiro
ULTIMO ← 15
```

Se a tabela é mantida desordenada, a inclusão pode e deve ser feita no seu final e o algoritmo é simples:

```
1: função INCLUSAO (V:vetor, K:inteiro) : vetor
2: ULTIMO ← ULTIMO + 1
3: V[ULTIMO] ← K
```

Naturalmente, ao fazer $ULTIMO+1$, há que se ter o cuidado de não ultrapassar o tamanho definido do vetor, sob pena de ganhar um INDEX ERROR.

Se a tabela é mantida ordenada, a inclusão fica um pouco mais complexa, já que:

★ Há que se achar o local exato da inclusão
★ Empurrar todos os elementos à direita desse local, para a casa ao lado
★ Efetuar a inclusão

Acompanhe tudo isso no algoritmo

```
 1: função INCLUSAO (V:vetor, K:inteiro) : vetor
 2: I ← 1
 3: enquanto V[I] < K
 4:     I ← I + 1
 5: fim{enquanto}
 6: J ← ULTIMO
 7: enquanto J ≥ I
 8:     V[J+1] ← V[J]
 9:     J ← J - 1
10: fim{enquanto}
11: V[I] ← K
12: ULTIMO ← ULTIMO + 1
```

Outra alternativa (ruim) é incluir o elemento ao final e reordenar a tabela. Porque esta alternativa é ruim ?

Exclusão Excluir um elemento significa deixar de considerá-lo como válido. Existem duas estratégias aqui: excluir fisicamente o elemento, o que sempre implicará em puxar uma posição os elementos que estavam à direita do excluído (já que não é possível deixar buracos no meio do vetor). Note que este cuidado é necessário mesmo que o vetor esteja desordenado. Eis o algoritmo

```
 1: função EXCLUSAO (V:vetor, K:inteiro) : vetor
 2: I ← 1
 3: enquanto V[I] ≠ K
 4:     I ← I + 1
 5: fim{enquanto}
 6: enquanto I < ULTIMO
 7:     V[I] ← V[I+1]
 8:     I ← J + 1
 9: fim{enquanto}
10: ULTIMO ← ULTIMO - 1
```

A propósito, note-se que o algoritmo pressupõe neste caso que K $\in$ V. Se isto tiver que ser verificado (sempre é bom fazer isso) o algoritmo fica um pouco mais complicado:

```
 1: função EXCLUSAO (V:vetor, K:inteiro) : vetor
 2: I ← 1
 3: enquanto V[I] ≠ K
 4:     I ← I + 1
 5:     se I > tamanho(V)
 6:         ...ERRO: K não existe em V
 7:     fim{se}
 8: fim{enquanto}
 9: enquanto I < ULTIMO
10:     V[I] ← V[I+1]
11:     I ← J + 1
12: fim{enquanto}
13: ULTIMO ← ULTIMO - 1
```

Imaginou-se no caso acima que a ocorrência de K é única. Se houver mais de uma, e todas essas ocorrências precisam ser excluídas, há que se modificar um pouco o algoritmo acima.

Note o custo computacional de fazer uma única exclusão. Para fugir dele é comum usar outra estratégia. Na exclusão de um elemento, ele é excluído apenas LOGICAMENTE, sendo marcado como inválido, embora fisicamente ele continue lá. Isto poderia ser feito colocando um valor inválido (por exemplo, um número negativo) na posição do elemento a ser excluído. Ao fazer isso, deixa-se de precisar puxar todos os seus vizinhos.

Busca A busca é a mais comum das operações. Hoje vamos ver apenas um algoritmo simples e na próxima aula vamos olhar 3 algoritmos que fazem a busca. É também o momento do curso de começar a exercitar um maravilhoso exercício intelectual: *Dados dois (ou mais) algoritmos que fazem exatamente a mesma coisa, qual deles usar ?*.

Não é um exercício inútil. As diferenças de desempenho entre algoritmos equivalente (que fazem a mesma coisa) pode ser de 100 vezes ou mais. Não é incomum este número.

A propósito, vale a pergunta: como se mede o custo de um algoritmo ? Como veremos no segundo ano (no tópico Teoria da Complexidade Algorítmica) em termos de algoritmos é muito verdadeira a frase **tempo é dinheiro**. Só que a medida do tempo é simplificada:

★ Não tem sentido usar minutos ou segundos: cada computador, cada S.O. e cada ambiente são únicos.
★ Usa-se então a unidade "operação computacional" e convenciona-se que ela demora 1 unidade.

Busca Linear Para este algoritmo o conjunto de chaves não é mantido ordenado. O algoritmo básico é:

```
 1: função BUSCAL(CHAVE:inteiro, V:vetor)
 2: ONDE, I : inteiro
 3: ONDE ← -1
 4: I ← 1
 5: enquanto i ≤ tamanho(V)
 6:     se CHAVE = V[I]
 7:         ONDE ← I
 8:         I ← tamanho(V)
 9:     fim{se}
10:     I ← I + 1
11: fim{enquanto}
12: devolva ONDE
```

☞ **Para você fazer** Localize na bibliografia do curso o nome de pelo menos 5 algoritmos (além da seleção e do bolha) que servem para ordenar um vetor. Responda a seguir: