**CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DA PARAÍBA**
**COORDENAÇÃO DO CURSO DE TECNOLOGIA EM TELEMÁTICA**

## APOSTILA DE

# ESTRUTURA DE DADOS

**PROF. CÂNDIDO EGYPTO**

**JOÃO PESSOA / PB**
**JULHO / 2003**

# SUMÁRIO

# 1 – INTRODUÇÃO

**O que é uma Estrutura de Dados (ED)?**

- Tipos de Dados
- Estruturas de Dados e
- Tipos Abstratos de Dados

Embora estes termos sejam parecidos, eles têm significados diferentes. Em linguagens de programação, o **tipo de dados** de uma variável define o conjunto de valores que a variável pode assumir. Por exemplo, uma variável do tipo *lógico* pode assumir o valor *verdadeiro* ou *falso*.

Uma declaração de variável em uma linguagem como *C* ou *Pascal* especifica:

1. O conjunto de valores que pode assumir.
2. O conjunto de operações que podemos efetuar.
3. A quantidade de bytes que deve ser reservada para ela.
4. Como o dado representado por esses bytes deve ser interpretado (por exemplo, uma cadeia de bits pode ser interpretada como um inteiro ou real...).

Então, tipos de dados podem ser vistos como métodos para interpretar o conteúdo da memória do computador.

Mas podemos ver o conceito de **Tipo de Dados** de uma outra perspectiva: não em termos do que um **computador** pode fazer (interpretar os bits...) mas em termos do que os **usuários** desejam fazer (somar dois inteiros...)

Este conceito de **Tipo de Dado** divorciado do hardware é chamado **Tipo Abstrato de Dado - TAD.**

**Estrutura de Dados** é um método particular de se implementar um **TAD**.

A **implementação** de um **TAD** escolhe uma **ED** para representá-lo. Cada **ED** é construída dos tipos primitivos (inteiro, real, char,...) ou dos tipos compostos (array, registro,...) de uma linguagem de programação.

Não importa que tipo de dados estaremos trabalhando, a primeira operação a ser efetuada em um TAD é a **criação**. Depois, podemos realizar **inclusões** e **remoções** de dados. A operação que varre todos os dados armazenados num TAD é o **percurso**, podendo também ser realizada uma **busca** por algum valor dentro da estrutura.

**Exemplos de TAD:**

**Lineares:**

- Listas Ordenadas
- Pilhas
- Filas
- Deques

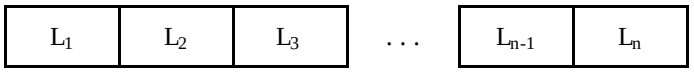
**Não Lineares:**

- Árvores
- Grafos

# 2 – LISTAS

São estruturas formadas por um conjunto de dados de forma a preservar a relação de ordem linear entre eles. Uma lista é composta por nós, os quais podem conter, cada um deles, um dado primitivo ou composto.

**Representação:**




**Sendo:**

$L_1$ → 1º elemento da lista
$L_2$ → Sucessor de $L_1$
$L_{n-1}$ → Antecessor de $L_{n-1}$
$L_n$ → Último elemento da lista.

**Exemplos de Lista:**

- Lista Telefônica
- Lista de clientes de uma agência bancária
- Lista de setores de disco a serem acessados por um sistema operacional
- Lista de pacotes a serem transmitidos em um nó de uma rede de computação de pacotes.

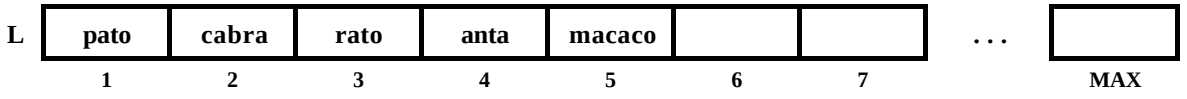
**Operações Realizadas com Listas:**

- Criar uma lista vazia
- Verificar se uma lista está vazia
- Obter o tamanho da uma lista
- Obter/modificar o valor do elemento de uma determinada posição na lista
- Obter a posição de elemento cujo valor é dado
- Inserir um novo elemento após (ou antes) de uma determinada posição na lista
- Remover um elemento de uma determinada posição na lista
- Exibir os elementos de uma lista
- Concatenar duas listas

**FORMAS DE REPRESENTAÇÃO:**

**a) Seqüencial:**

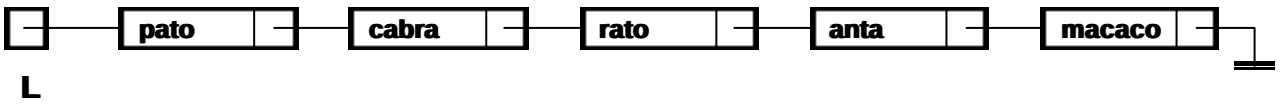
Explora a sequencialidade da memória do computador, de tal forma que os nós de uma lista sejam armazenados em endereços sequenciais, ou igualmente distanciados um do outro. Pode ser representado por um vetor na memória principal ou um arquivo seqüencial em disco.




**b) Encadeada:**

Esta estrutura é tida como uma seqüência de elementos encadeados por ponteiros, ou seja, cada elemento deve conter, além do dado propriamente dito, uma referência para o próximo elemento da lista.

Ex: L = pato, cabra, rato, anta, macaco



## 2.1 – LISTA SEQÜENCIAL

Uma lista representada de forma seqüencial é um conjunto de registros onde estão estabelecidas regras de precedência entre seus elementos. O sucessor de um elemento ocupa posição física subsequente.

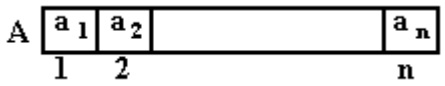
A implementação de operações pode ser feita utilizando *array* e *registro*, associando o elemento a(i) com o índice i (mapeamento seqüencial).

**CARACTERÍSTICAS**

- os elementos na lista estão armazenados fisicamente em posições consecutivas;
- a inserção de um elemento na posição a(i) causa o deslocamento a direita do elemento de a(i) ao último;
- a eliminação do elemento a(i) requer o deslocamento à esquerda do a(i+1) ao último;

Mas, absolutamente, uma lista seqüencial ou é vazia ou pode ser escrita como (a(1), a(2), a(3), ... a(n)) onde a(i) são átomos de um mesmo conjunto A.

Além disso, a(1) é o primeiro elemento, a(i) precede a(i+1), e a(n) é o último elemento.




Assim as propriedades estruturadas da lista permitem responder a questões tais como:

- se uma lista está vazia
- se uma lista está cheia
- quantos elementos existem na lista
- qual é o elemento de uma determinada posição
- qual a posição de um determinado elemento
- inserir um elemento na lista
- eliminar um elemento da lista

**Conseqüência:**
As quatro primeiras operações são feitas em tempo constante. As demais, porém, requererão mais cuidados.

**Vantagem:**

- acesso direto indexado a qualquer elemento da lista
- tempo constante para acessar o elemento *i* - dependerá somente do índice.

**Desvantagem:**

- movimentação quando eliminado/inserido elemento
- tamanho máximo pré-estimado

**Quando usar:**

- listas pequenas
- inserção/remoção no fim da lista
- tamanho máximo bem definido

Vamos tentar evitar as desvantagens anteriores ao usar endereços não consecutivos (Lista Encadeada).

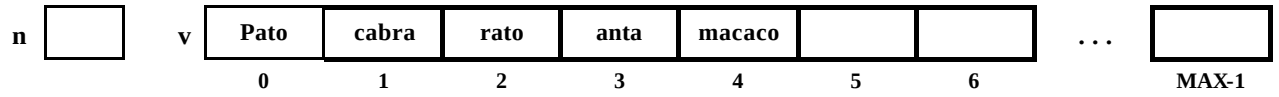
**OPERAÇÕES BÁSICAS**

A seguir, apresentaremos a estrutura de dados e as operações básicas sobre listas, implementadas na linguagem C.

**Definição da ED:**

```
#define MAX _________     /* tamanho máximo da lista */

typedef ______ telem;    /* tipo base dos elementos da lista */
typedef struct {
           telem v[MAX]; /* vetor que contém a lista */
           int n;        /* posição do último elemento da lista */
       } tlista;         /* tipo lista */
```

**tlista**




**Operações simples utilizando lista seqüencial:**

**1) Criar uma lista vazia**

```
void criar (tlista *L) {
   L->n = 0;
}
```

**2) Verificar se uma lista está vazia**

```
int vazia (tlista L) {
   return (L.n == 0);
}
```

**3) Verificar se uma lista está cheia**

```
int cheia (tlista L) {
   return (L.n == MAX);
}
```

**4) Obter o tamanho de uma lista**

```
int tamanho (tlista L) {
   return (L.n);
}
```

**5) Obter o i-ésimo elemento de uma lista**

```
int elemento (tlista L, int pos, telem *dado) {

   /* O parâmetro dado irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   if ( (pos > L.n) || (pos <= 0) ) return (0);

   *dado = L.v[pos-1];
   return (1);
}
```

**6) Pesquisar um dado elemento, retornando a sua posição**

```
int posicao (tlista L, telem dado) {

  /* Retorna a posição do elemento ou 0 caso não seja encontrado */

  int i;

  for (i=1; i<=L.n; i++)
     if (L.v[i-1] == dado)
        return (i);

  return (0);
}
```

**7) Inserção de um elemento em uma determinada posição**

Requer o deslocamento à direita dos elementos v(i+1)...v(n)

```
int inserir (tlista *L, int pos, telem dado) {

  /* Retorna 0 se a posição for inválida ou se a lista estiver cheia */
  /* Caso contrário, retorna 1 */

  int i;

  if ( (L->n == MAX)) || (pos > L->n + 1) )  return (0);

  for (i=L->n; i>=pos; i--)
     L->v[i] = L->v[i-1];

  L->v[pos-1] = dado;
  (L->n)++;
  return (1);
}
```

**8) Remoção do elemento de uma determinada posição**

Requer o deslocamento à esquerda dos elementos v(p+1)...v(n)

```
int remover (tlista *L, int pos, telem *dado) {

  /* O parâmetro dado irá receber o elemento encontrado */
  /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

  int i;

  if ( (pos > L->n) || (pos <= 0) ) return (0);

  *dado = L->v[pos-1];

  for (i=pos; i<=(L->n)-1; i++)
     L->v[i-1] = L->v[i];

  (L->n)--;
  return (1);
}
```

## PRÁTICA DE LABORATÓRIO

01. Faça um programa que:
    a) crie uma lista L;
    b) exiba o seguinte menu de opções:

```
EDITOR DE LISTAS

1 – EXIBIR LISTA
2 – INSERIR
3 – REMOVER
4 – EXIBIR ELEMENTO
5 – EXIBIR POSIÇÃO
6 – ESVAZIAR
ESC – SAIR

DIGITE SUA OPÇÃO:
```

    c) leia a opção do usuário;
    d) execute a opção escolhida pelo usuário;
    e) implemente a estrutura de dados LISTA em uma biblioteca chamada L_SEQ (com implementação seqüencial e usando o **inteiro** como tipo base), contendo apenas as operações básicas de listas (citadas anteriormente);
    f) na opção de exibir lista, devem ser exibidos o tamanho da lista e os seus elementos;
    g) na opção de inserção, deve ser lido o valor do elemento a ser inserido e a posição onde será efetuada a inserção;
    h) na opção de remoção, deve ser lido a posição do elemento a ser removido;
    i) na opção de exibir elemento, deve ser lido a posição do elemento;
    j) na opção de exibir posição, deve ser lido o valor do elemento;
    k) as operações de exibir e esvaziar a lista devem estar inseridas no programa principal;
    l) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

## EXERCÍCIOS – LISTA SEQUENCIAL

01. Inclua, na biblioteca L_SEQ, as funções abaixo especificadas (obs: não faça uso das funções já implementadas).
    a) inserir um dado elemento na primeira posição de uma lista;
    b) inserir um dado elemento na última posição de uma lista;
    c) modificar um elemento de uma lista, sendo dado a sua posição e o novo valor;
    d) remover o primeiro elemento de uma lista;
    e) remover o último elemento de uma lista;
    f) remover um elemento dado o seu valor.

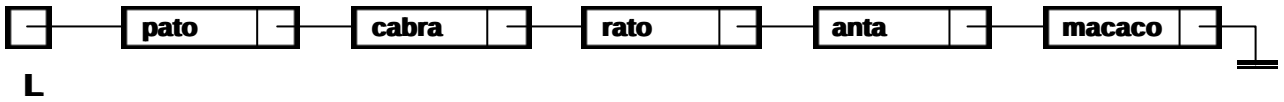
02. Modifique o seu programa EDITOR DE LISTAS, adicionando todas as operações relacionadas na questão anterior.

03. Escreva um programa que, utilizando a biblioteca L_SEQ, faça o seguinte:
    a) crie quatro listas (L1, L2, L3 e L4);
    b) insira sequenciamente, na lista L1, 10 números inteiros obtidos de forma randômica (entre 0 e 99);
    c) idem para a lista L2;
    d) concatene as listas L1 e L2, armazenando o resultado na lista L3;
    e) armazene na lista L4 os elementos da lista L3 (na ordem inversa);
    f) exiba as listas L1, L2, L3 e L4.

## 2.2 – LISTA ENCADEADA

Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.

Ex: L = pato, cabra, rato, anta, macaco




Cada registro é: 


Há duas alternativas para implementação de operações de listas encadeadas: utilizando **arrays estáticos** ou **variáveis dinâmicas.** Entretanto, não há vantagem na implementação da lista encadeada com array, uma vez que permanece o problema do tamanho máximo da lista predefinido e necessita de mais espaço para armazenar os endereços dos elementos sucessores. Portanto, vamos utilizar variáveis dinâmicas como forma de implementação para as nossas listas encadeadas.

Antes, é importante explicarmos o conceito e utilização de variáveis dinâmicas e de ponteiros (apontadores).

**Variáveis Dinâmicas e Ponteiros**

As linguagens de programação modernas tornaram possível explicitar não apenas o acesso aos dados, mas também aos endereços desses dados.

Isso significa que ficou possível a utilização de ponteiros explicitamente implicando que uma distinção notacional deve existir entre os dados e as referências (endereços) desses dados.

A linguagem C utiliza a seguinte notação para a manipulação de ponteiros:

```
tp *P
```

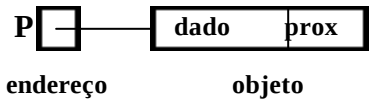
Essa declaração expressa que a variável **P** é um ponteiro para uma variável do tipo **tp**.

Valores para ponteiros são gerados quando dados correspondentes a seus tipos são alocados/desalocados dinamicamente.

Em C, a função **malloc** é utilizada para alocar um novo espaço de memória e a função **free** é utilizada para liberar um espaço alocado. Estão disponíveis na biblioteca <alloc.h>.

Portanto, deixa-se a cargo do programa (via linguagem de programação), e não do programador, prover e devolver espaço para inserções e eliminações em tempo de execução.

Um ponteiro como **P** pode assumir o conjunto de valores que correspondem a endereços reais de memória. Por exemplo, sendo **tp** um registro contendo os campos **dado** e **prox**, e sendo **P** do tipo **tp**, podemos ter:




onde o conteúdo de P corresponderia ao endereço do objeto. Esses endereços serão as ligações das listas encadeadas dinâmicas.

Para designar ponteiro, objeto e campos, a notação utilizada é:

```
ponteiro: P
objeto:  *P
campos: (*P).dado  ou  P->dado
        (*P).prox  ou  P->prox
```

**Endereço nulo (terra)**

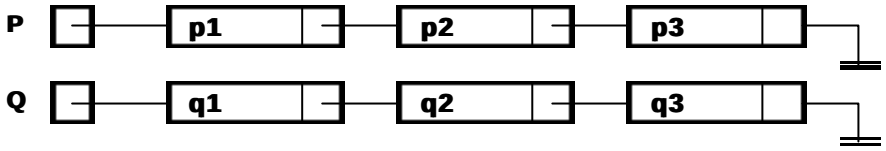
A linguagem C utiliza o valor **NULL**, disponível em <alloc.h>, para denotar o endereço nulo. Podemos utilizá-lo para atribuições e testes, como nos exemplos abaixo:

```
1) L = NULL;
2) if (P = NULL) ...
```

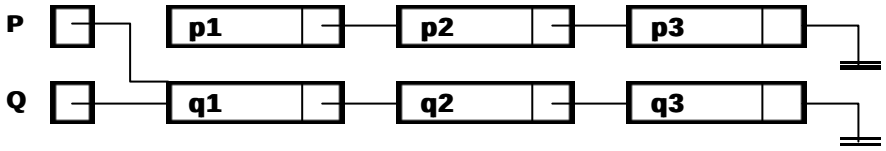
**Ponteiro x Objeto Apontado**

Nos exemplos abaixo, é ilustrada a diferença entre as operações de atribuição entre ponteiros (por exemplo, **P = Q**) e a atribuição entre o conteúdo das variáveis apontadas pelos ponteiros (isto é: ***P = *Q**).

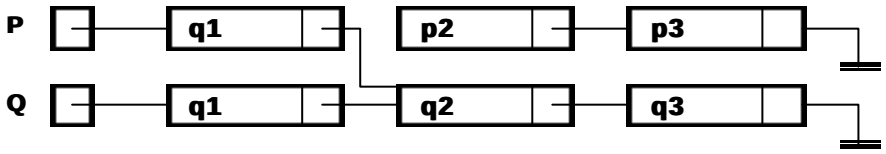
Dada a situação abaixo, chamada de (a):




Dada a situação (a), após a atribuição **P = Q** temos a representação abaixo (b).




Dada a situação (a), após a atribuição ***P = *Q** temos a representação abaixo (c), onde o conteúdo é atribuído. (Lembre-se que ***P = *Q** equivale às atribuições **P->dado = Q->dado** e **P->prox = Q->prox**)




**Operações Básicas com Ponteiros**

**1) Declaração de Variável**

```
tp *P;
```


A variável do tipo apontador P é alocada na memória, com valor indefinido.

**2) Criação de um registro**

```
p = (tp *) malloc(sizeof (tp));
```


a) aloca uma nova variável do tipo **tp** (registro)
b) atribui o endereço da nova variável ao ponteiro **P**

**3) Atribuição de conteúdo ao registro:**

```
P->dado = valor;
```

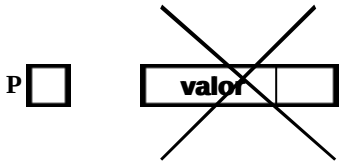

**4) Atribuição do valor nulo ao campo ponteiro do registro**

```
P->prox = NULL;
```


**5) Liberação de um registro**

```
free(P);
```

Libera o espaço apontado por P.
É conveniente atribuir NULL à P para evitar uma utilização indevida ao endereço apontado por P.




**DEFINIÇÃO DA ED**

```
#include <alloc.h>

typedef ____ telem;             /* tipo base da lista */

typedef struct no {
              telem dado;       /* campo da informação */
              struct no* prox; /* campo do ponteiro para o próximo nó */
} tno;                          /* tipo do nó */

typedef tno* tlista;            /* tipo lista */
```

**OPERAÇÕES SOBRE LISTAS ENCADEADAS**

**1) Criação da lista vazia**

```
void criar(tlista *L)
{
   *L = NULL;
}
```

**2) Verificar se a lista está vazia**

```
int vazia(tlista L)
{
   return (L == NULL);
}
```

**3) Obter o tamanho da lista**

```
int tamanho(tlista L)
{
   tlista p = L;
   int n = 0;

   while (p != NULL) {
      p = p->prox;
      n++;
   }

   return n;
}
```

**4) Obter o valor do elemento de uma posição dada**

```
int elemento(tlista L, int pos, telem *elem)
{
   /* O parâmetro elem irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   tlista p = L;
   int n = 1;

   if (L == NULL) return 0;  /* erro: lista vazia */

   while ((p != NULL) && (n < pos)) {
      p = p->prox;
      n++;
   }

   if (p == NULL) return 0;  /* erro: posição inválida */
   *elem = p->dado;
   return 1;
}
```

**5) Obter a posição de elemento cujo valor é dado**

```
int posicao(tlista L, telem valor)
{
   /* Retorna a posição do elemento ou 0 caso não seja encontrado */

   if (L != NULL)) {

      tlista p = L;
      int n = 1;

      while (p != NULL) {
         if (p->dado == valor) return n;
         p = p->prox;
         n++;
      }

   }

   return 0;
}
```

**6) Inserir um elemento na lista, dado a sua posição**

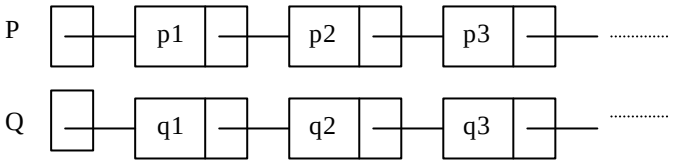
```
int inserir(tlista *L, int pos, telem valor)
{
   /* Retorna 0 se a posição for inválida ou se a lista estiver cheia */
   /* Caso contrário, retorna 1 */

   tlista p, novo;
   int n;

   /* inserção em lista vazia */

   if (*L == NULL)) {

      if (pos != 1) return 0;  /* erro: posição inválida */

      novo = (tlista) malloc(sizeof(tno));
      if (novo == NULL) return 0;  /* erro: memória insuficiente */

      novo->dado = valor;
      novo->prox = NULL;
      *L = novo;

      return 1;
   }

   /* inserção na primeira posição em lista não vazia */

   if (pos == 1) {

      novo = (tlista) malloc(sizeof(tno));
      if (novo == NULL) return 0;  /* erro: mem¢ria insuficiente */

      novo->dado = valor;
      novo->prox = *L;
      *L = novo;

      return 1;
   }

   /* inserção após a primeira posição em lista não vazia */

   p = *L;
   n = 1;

   while ((n < pos-1) && (p != NULL)) {
      p = p->prox;
      n++;
   }

   if (p == NULL) return 0; /* erro: posição inválida */

   novo = (tlista) malloc(sizeof(tno));
   if (novo == NULL) return 0;  /* erro: memória insuficiente */

   novo->dado = valor;
   novo->prox = p->prox;
   p->prox = novo;

   return 1;
}
```

**7) Remover um elemento de uma determinada posição**

```
int remover(tlista *L, int pos, telem *elem)
{
   /* O parâmetro elem irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   tlista a, p;
   int n;

   if (vazia(*L)) return 0;  /* erro: lista vazia */

   p = *L;
   n = 1;

   while ((n<=pos-1) && (p!=NULL)) {
      a = p;
      p = p->prox;
      n++;
   }

   if (p == NULL) return 0;  /* erro: posição inválida */

   *elem = p->dado;

   if (pos == 1)
      *L = p->prox;
   else
      a->prox = p->prox;

   free(p);

   return(1);
}
```

## EXERCÍCIOS – LISTA ENCADEADA

1) Partindo da situação mostrada no desenho abaixo, e sendo P e Q duas listas encadeadas dinâmicas, explique (com desenhos) o que acontece nas atribuições seguintes (obs: cada item parte da situação inicial mostrada abaixo)




a) `P->prox = Q->prox;`

b) `p = Q;`

c) `p = NULL;`

d) `p = P->prox;`

e) `P->dado = (P->prox)->dado`

2) Implemente a estrutura de dados LISTA em uma biblioteca chamada L_DIN (com implementação encadeada). Esta biblioteca deve seguir o mesmo padrão da biblioteca L_SEQ desenvolvida no exercício anterior, ou seja, contendo a mesma interface disponibilizada (definição de tipos, constantes e funções).

3) Modifique o seu programa Editor de Listas (do exercício anterior) para que o mesmo utilize a biblioteca L_DIN em vez da L_SEQ (como foi feito anteriormente). Execute e teste o programa (ele deve funcionar normalmente).

4) Uma maneira usual de se representar um conjunto é pela lista de seus elementos. Supondo esta representação, escreva procedimentos para as operações usuais de conjunto: união, interseção, diferença e pertinência.

# 3 – LISTAS ORDENADAS

São listas lineares onde os elementos estão ordenados segundo um critério pré-estabelecido. Na realidade, as listas ordenadas diferem das listas genéricas pelo fato de que cada novo elemento a ser inserido ocupará uma posição específica, obedecendo à ordenação dos valores já existentes.

## 3.1 – LISTA ORDENADA SEQÜENCIAL

### DEFINIÇÃO DA ED

```
#define MAX _______                 /* tamanho máximo da lista */

typedef _____ telem;                /* tipo base dos elementos da lista */

typedef struct {
                  telem v[MAX];     /* vetor que contém a lista */
                  int n;            /* posição do último elemento da lista */
} tlistaord;                        /* tipo lista ordenada */
```

### OPERAÇÕES BÁSICAS

**1) Criar uma lista vazia**

```
void criar (tlistaord *L)
{
   L->n = 0;
}
```

**2) Verificar se uma lista está vazia**

```
int vazia (tlistaord L)
{
   return (L.n == 0);
}
```

**3) Verificar se uma lista está cheia**

```
int cheia (tlistaord L)
{
   return (L.n == MAX);
}
```

**4) Obter o tamanho de uma lista**

```
int tamanho (tlistaord L)
{
   return (L.n);
}
```

**5) Obter o i-ésimo elemento de uma lista**

```
int elemento (tlistaord L, int pos, telem *dado)
{
   /* O parâmetro dado irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   if ( (pos > L.n) || (pos <= 0) ) return (0);
   *dado = L.v[pos-1];
   return (1);
}
```

**6) Pesquisar um dado elemento, retornando a sua posição**

```
int posicao (tlistaord L, telem dado)
{
   /* Retorna a posição do elemento ou 0 caso não seja encontrado */

   int i;

   for (i=1; i<=L.n; i++)
      if (L.v[i-1] == dado)
         return (i);

   return (0);
}
```

**7) Inserção de um elemento dado**

```
int inserir (tlistaord *L, telem dado)
{
   /* Retorna 0 se a posição for inválida ou se a lista estiver cheia */
   /* Caso contrário, retorna 1 */

   int i, pos;

   if (L->n == MAX) return (0); /* erro: lista cheia */

   for (i=0; i<L->n; i++) {
      if (L->v[i] == dado) return (0); /* erro: dado já existente */
      if (L->v[i] > dado) break;
   }

   pos = i;

   for (i=L->n; i>pos; i--)
      L->v[i] = L->v[i-1];

   L->v[i] = dado;
   (L->n)++;

   return (1);
}
```

**8) Remoção do elemento de uma determinada posição**

```
int remover (tlistaord *L, int pos, telem *dado)
{
   /* O parâmetro dado irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   int i;

   if ( (pos > L->n)) || (pos <= 0) ) return (0); /* erro: posição inválida */

   *dado = L->v[pos-1];

   for (i=pos; i<=(L->n)-1; i++)
      L->v[i-1] = L->v[i];

   (L->n)--;

   return (1);
}
```

## 3.2 – LISTA ORDENADA ENCADEADA

### DEFINIÇÃO DA ED

```
#include <alloc.h>

typedef ____ telem;          /* tipo base da lista */

typedef struct no {
            telem dado;      /* campo da informação */
            struct no* prox; /* campo do ponteiro para o próximo nó */
} tno;                       /* tipo do nó */

typedef tno* tlistaord; /* tipo lista */
```

### OPERAÇÕES BÁSICAS

#### 1) Criação da lista vazia

```
void criar(tlistaord *L)
{
   *L = NULL;
}
```

#### 2) Verificar se a lista está vazia

```
int vazia(tlistaord L)
{
   return (L == NULL);
}
```

#### 3) Obter o tamanho da lista

```
int tamanho(tlistaord L)
{
   tlistaord p = L;
   int n = 0;

   while (p != NULL) {
      p = p->prox;
      n++;
   }

   return n;
}
```

#### 4) Obter o valor do elemento de uma posição dada

```
int elemento(tlistaord L, int pos, telem *elem)
{
   /* O parâmetro elem irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   tlistaord p = L;
   int n = 1;

   if (L == NULL) return 0; /* erro: lista vazia */
```

```
   while ((p != NULL) && (n < pos)) {
      p = p->prox;
      n++;
   }

   if (p == NULL) return 0; /* erro: posição inválida */

   *elem = p->dado;

   return 1;
}
```

**5) Obter a posição de elemento cujo valor é dado**

```
int posicao(tlistaord L, telem valor)
{
   /* Retorna a posição do elemento ou 0 caso não seja encontrado */

   if (L != NULL) {

      tlistaord p = L;
      int n = 1;

      while (p != NULL) {

         if (p->dado == valor) return n;

         p = p->prox;
         n++;
      }

   }

   return 0;
}
```

**6) Inserir um dado elemento na lista**

```
int inserir(tlistaord *L, telem valor)
{
   /* Retorna 0 se a posição for inválida ou se a lista estiver cheia */
   /* Caso contrário, retorna 1 */

   tlistaord atual, ant, novo;
   int n;

   novo = (tlistaord) malloc(sizeof(tno));

   if (novo == NULL) return 0; /* erro: memória insuficiente */

   novo->dado = valor;
   novo->prox = NULL;
   ant = NULL;
   atual = *L;

   while (atual != NULL && valor >= atual->dado) {

      if (atual->dado == valor) return 0; /* erro: valor já existente */

      ant = atual;
      atual = atual->prox;
   }
```

```
   if (ant == NULL) {
      novo->prox = *L;
      *L = novo;
   }
   else {
      ant->prox = novo;
      novo->prox = atual;
   }

   return 1;
}
```

**7) Remover um elemento de uma determinada posição**

```
int remover(tlistaord *L, int pos, telem *elem)
{
   /* O parâmetro elem irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   tlistaord a, p;
   int n;

   if (vazia(*L)) return 0; /* erro: lista vazia */

   p = *L;
   n = 1;

   while ((n<=pos-1) && (p!=NULL)) {
      a = p;
      p = p->prox;
      n++;
   }

   if (p == NULL) return 0; /* erro: posição inválida */

   *elem = p->dado;

   if (pos == 1)
      *L = p->prox;
   else
      a->prox = p->prox;

   free(p);

   return(1);
}
```

# EXERCÍCIOS – LISTAS ORDENADAS

01. Implemente a TAD Lista Ordenada com representação seqüencial em uma biblioteca chamada LORD_SEQ (usando como base o tipo string), contendo apenas a estrutura de dados e as operações básicas de listas ordenadas (descritas anteriormente).

02. Implemente a TAD Lista Ordenada com representação encadeada em uma biblioteca chamada LORD_ENC (usando como base o tipo string), contendo apenas a estrutura de dados e as operações básicas de listas ordenadas (descritas anteriormente).

03. Faça um programa que, utilizando qualquer uma das bibliotecas criadas nos itens 01 e 02:
    a) crie uma lista ordenada L;
    b) exiba o seguinte menu de opções:

```
EDITOR DE LISTAS ORDENADAS

   1 – EXIBIR LISTA
   2 – INSERIR
   3 – REMOVER
   4 – EXIBIR ELEMENTO
   5 – EXIBIR POSIÇÃO
   6 – ESVAZIAR
ESC – SAIR

DIGITE SUA OPÇÃO:
```

    c) leia a opção do usuário;
    d) execute a opção escolhida pelo usuário;
    e) na opção de exibir lista, devem ser exibidos o tamanho da lista e os seus elementos;
    f) na opção de inserção, deve ser lido o valor do elemento a ser inserido;
    g) na opção de remoção, deve ser lido a posição do elemento a ser removido;
    h) na opção de exibir elemento, deve ser lido a posição do elemento;
    i) na opção de exibir posição, deve ser lido o valor do elemento;
    j) as operações de exibir e esvaziar a lista devem estar inseridas no programa principal;
    k) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

04. Faça um programa que leia (via teclado) o nome e o sexo de várias pessoas. Se a pessoa for do sexo masculino, insira o seu nome em uma lista chamada MACHOS. Caso contrário, insira o nome numa lista chamada FEMEAS.

    Ambas as listas devem ser criadas e têm como tipo base o string.

    Escolha uma condição de parada para a entrada de dados a seu gosto.

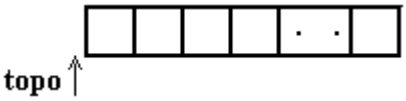
    Ao final, devem ser exibidas as duas listas criadas.

    OBS: utilize qualquer uma das bibliotecas criadas nos itens 1 e 2.

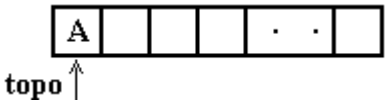
# 4 – PILHAS

**Pilhas** são listas onde a inserção de um novo item ou a remoção de um item já existente se dá em uma única extremidade, no topo.

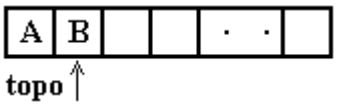
**Pilha vazia**



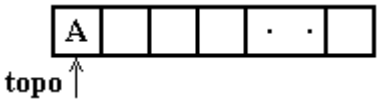

**Insere (A)**



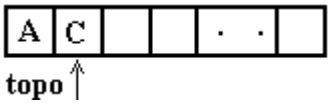

**Insere (B)**



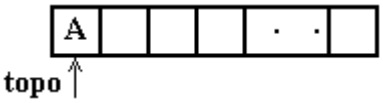

**Retira (B)**



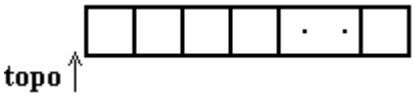

**Insere (C)**




**Retira (C)**




**Retira (A)**




**Definição:**

Dada uma pilha P = ( a(1), a(2), ..., a(n) ), dizemos que a(1) é o elemento da base da pilha; a(n) é o elemento topo da pilha; e a(i+1) está acima de a(i).

Pilhas são também conhecidas como listas **LIFO** (last in first out).

**Operações Associadas:**

1. Criar uma pilha P vazia
2. Testar se P está vazia
3. Obter o elemento do topo da pilha (sem eliminar)
4. Inserir um novo elemento no topo de P (empilhar)
5. Remover o elemento do topo de P (desempilhar)

**Implementação de Pilhas**

Como lista **Seqüencial** ou **Encadeada**?

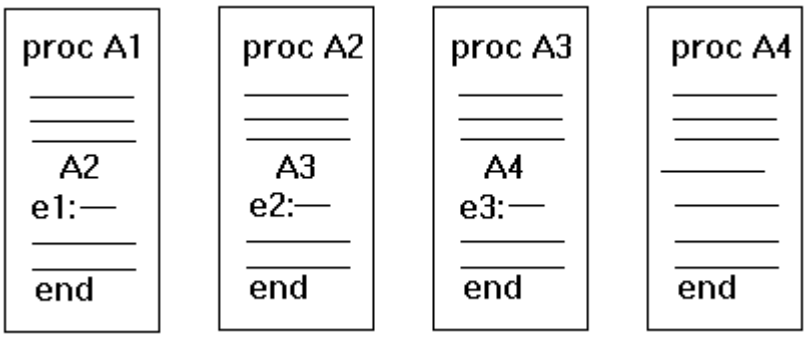
No caso geral de listas ordenadas, a maior vantagem da alocação encadeada sobre a seqüencial - se a memória não for problema - é a eliminação de deslocamentos na inserção ou eliminação dos elementos. No caso das pilhas, essas operações de deslocamento não ocorrem.

Portanto, podemos dizer que a alocação sequencial é mais vantajosa na maioria das vezes.

**Exemplo do Uso de Pilhas**

**Chamadas de procedimentos**

Suponha a seguinte situação:




Quando o procedimento A1 é executado, ele efetua uma chamada a A2, que deve carregar consigo o endereço de retorno e1. Ao término de A2, o processamento deve retornar ao A1, no devido endereço. Situação idêntica ocorre em A2 e A3.

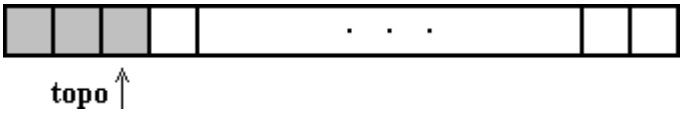
Assim, quando um procedimento termina, é o seu endereço de retorno que deve ser consultado. Portanto, há uma lista **implícita** de endereços (e0, e1, e2, e3) que deve ser manipulada como uma **pilha** pelo sistema, onde e0 é o endereço de retorno de A1.

No caso de processamento recursivo - por exemplo uma chamada a A2 dentro de A4 - o gerenciamento da lista como uma **pilha** resolve automaticamente a obtenção dos endereços de retorno na ordem apropriada (e0, e1, e2, e3, e4).

## 4.1 – ALOCAÇÃO SEQÜENCIAL DE PILHAS

**Definição da Estrutura de Dados:**

```
#define MAX 10

typedef int telem;

typedef struct {
             telem v[MAX];
             int topo;
        } tpilha;
```



**OPERAÇÕES:**

**1) Criar uma pilha vazia**

```
void criar (tpilha *p)
{
   p->topo = -1;
}
```

**2) Testar se a pilha está vazia**

```
int vazia (tpilha p)
{
   return (p.topo == -1);
}
```

**3) Obter o elemento do topo da pilha (sem eliminar)**

```
int elemtopo (tpilha p, telem *valor)
{
   if (vazia(p)) return 0;
   *valor = p.v[p.topo];
   return 1;
}
```

**4) Inserir um novo elemento no topo da pilha (empilhar)**

```
int push (tpilha *p, telem valor)
{
   if (p->topo == MAX-1) return 0;
   p->v[>topo] = valor;
   return 1;
}
```

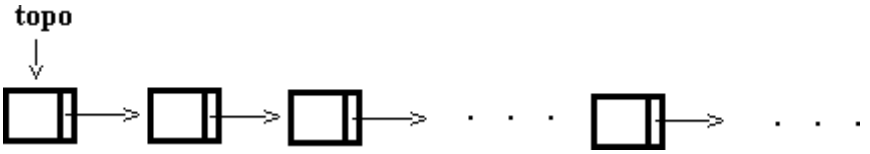
**5) Remove o elemento do topo da pilha (desempilhar), retornando o elemento removido**

```
int pop (tpilha *p, telem *valor)
{
   if (vazia(*p)) return 0;
   *valor = p->v[p->topo--];
   return 1;
}
```

## 3.2 – ALOCAÇÃO DINÂMICA ENCADEADA DE PILHAS

**Definição da Estrutura de Dados:**

```
#include <alloc.h>
typedef int telem;
typedef struct no{
                   telem dado;
                   struct no* prox;
        } tno;
typedef tno* tpilha;
```



**OPERAÇÕES:**

**1) Criar uma pilha vazia**

```
void criar (tpilha *p)
{
   *p = NULL;
}
```

**2) Testar se a pilha está vazia**

```
int vazia (tpilha p)
{
   return (p == NULL);
}
```

**3) Obter o elemento do topo da pilha (sem eliminar)**

```
int elemtopo (tpilha p, telem *elem)
{
   if (vazia(p)) return 0;  /* erro: pilha vazia */

   *elem = p->dado;
   return 1;
}
```

**4) Inserir um novo elemento no topo da pilha (empilhar)**

```
int push (tpilha *p, telem valor)
{
   tpilha novo;

   novo = (tno *) malloc(sizeof(tno));

   if (novo == NULL) return 0;  /* erro: mem¢ria insuficiente */

   novo->dado = valor;
   novo->prox = *p;
   *p = novo;
   return 1;
}
```

**5) Remove o elemento do topo da pilha (desempilhar), retornando o elemento removido**

```
int pop (tpilha *p, telem *valor)
{
   tpilha aux;

   if (vazia(*p)) return 0;  /* erro: lista vazia */

   aux = *p;
   *valor = (*p)->dado;
   *p = aux->prox;
   free(aux);
   return 1;
}
```

# EXERCÍCIOS – PILHAS

1. Implemente a TAD **Pilha** com **representação seqüencial** em uma biblioteca chamada **P_SEQ** (usando como tipo base o **char**), contendo apenas a estrutura de dados e as operações básicas de pilhas (descritas anteriormente).

2. Implemente a TAD **Pilha** com **representação encadeada** em uma biblioteca chamada **P_ENC** (usando como tipo base o **char**), contendo apenas a estrutura de dados e as operações básicas de pilhas (descritas anteriormente).

3. Faça um programa que, utilizando qualquer uma das bibliotecas criadas nos itens 1 e 2 :
   a) crie uma pilha P;
   b) exiba o seguinte menu de opções:

   **EDITOR DE PILHA**

   1 – EMPILHAR
   2 – DESEMPILHAR
   3 – EXIBIR ELEMENTO DO TOPO
   4 – EXIBIR A PILHA
   5 – ESVAZIAR A PILHA

   DIGITE SUA OPÇÃO:

   c) leia a opção do usuário;
   d) execute a opção escolhida pelo usuário;
   e) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

2. Escreva um programa que receba uma linha de texto e use uma pilha para exibir a linha invertida.

3. Escreva um programa que use uma pilha para determinar se uma string é um palíndromo (isto é, tem a mesma leitura no sentido normal e no sentido inverso). O programa deve ignorar espaços em branco, pontuação e caracteres especiais.

# 5 – FILAS

É uma lista linear em que a inserção é feita numa extremidade e a eliminação na outra. Conhecida com estrutura FIFO (*First In, First Out*).

```
  ( a1, a2  , ... , an )
```
eliminações no início — inserções no final

**Exemplos:**

- Escalonamento de "Jobs": fila de processos aguardando os recursos do sistema operacional.
- Fila de pacotes a serem transmitidos numa rede de comutação de pacotes.
- Simulação: fila de caixa em banco.

**Operações associadas:**

1. **Criar** - cria uma fila vazia
2. **Vazia** - testa se um fila está vazia
3. **Primeiro** - obtém o elemento do início de uma fila
4. **Inserir** - insere um elemento no fim de uma fila
5. **Remover** - remove o elemento do início de uma fila, retornando o elemento removido.

**Implementação de Filas**

Como lista **Seqüencial** ou **Encadeada** ?

Pelas suas características, as filas têm as eliminações feitas no seu início e as inserções feitas no seu final. A implementação **encadeada dinâmica** torna mais simples as operações (usando uma lista de duas cabeças). Já a implementação **seqüencial** é um pouco mais complexa (teremos que usar o conceito de fila circular), mas pode ser usada quando há previsão do tamanho máximo da fila.

## 5.1 – IMPLEMENTAÇÃO SEQÜENCIAL DE FILA

**Definição da Estrutura de Dados:**

Devido a sua estrutura, será necessária a utilização de dois campos que armazenarão os índices do início e do final da fila e um vetor de elementos (onde serão armazenados os dados) com tamanho pré-estabelecido.

```
#define MAX ______      /* tamanho máximo da fila */

typedef _____ telem;    /* tipo base dos elementos da fila */

typedef struct{

         telem v[MAX];
         int inicio;    /* posição do primeiro elemento */
         int final;     /* posição do último elemento  */

      } tfila;
```

**OPERAÇÕES COM FILAS:**

**1. Criar** - cria uma fila vazia

```
void criar (tfila *F)
{
   F->inicio = 0;
   F->final = -1;
}
```

**2. Vazia** - testa se uma fila está vazia

```
int vazia (tfila F)
{
   return (F.inicio > F.final);
}
```

**3. Primeiro** - obtém o elemento do início da fila

```
int primeiro (tfila F, telem *dado)
{
   if (vazia(F)) return 0;  /* Erro: fila vazia */

   *dado = F.v[F.inicio];

   return (1);
}
```

**5. Insere** - insere um elemento no fim de uma fila

```
int inserir (tfila *F, telem valor)
{
   if (F->final == MAX-1) return 0;

   (F->final)++;
   F->v[F->final] = valor;

   return(1);
}
```

**6. Remove** - remove o elemento do início de uma fila, retornando o elemento removido

```
int remover (tfila *F, telem *valor)
{
   if (vazia(*F)) return 0;  /* Erro: fila vazia */

   primeiro(*F,valor);
   (F->inicio)++;

   return(1);
}
```

**Problema na implementação seqüencial**

O que acontece com a fila considerando a seguinte seqüência de operações sobre um fila:

**I E I E I E I E I ...**

(I - inserção e E - eliminação)

Note que a fila vai se deslocando da esquerda para a direita do vetor. Chegará a condição de "overflow" (cheia), porém estando vazia, ou seja, sem nenhum elemento.

**Alternativa:**

No algoritmo de remoção, após a atualização de **inicio**, verificar se a fila ficou vazia. Se este for o caso, reinicializar **inicio = 0** e **final = -1**

Portanto, ficaria:

```
int remover (tfila *F, telem *valor) {

   if (vazia(*F)) return 0;  /* Erro: fila vazia */

   primeiro(*F,valor);
   (F->inicio)++;

   if (vazia(*F)) {
      F->inicio = 0;
      F->final = -1;
   }

   return(1);
}
```

**O que aconteceria se a seqüência fosse:**

**I I E I E I E I E I ...**

A lista estaria com no máximo dois elementos, mas ainda ocorreria overflow com a lista quase vazia.

**Alternativa:**

Forçar **final** a usar o espaço liberado por **inicio** (**Fila Circular**)

**FILA CIRCULAR**

Para permitir a reutilização das posições já ocupadas, usa-se o conceito de "Fila Circular". Precisamos de um novo componente para indicar quantos elementos existem na fila, no momento.

A estrutura de dados com o novo componente ficaria assim representada:

```
#define MAX ______      /* tamanho máximo da fila */

typedef _____ telem;    /* tipo base dos elementos da fila */

typedef struct{
         telem v[MAX];
         int inicio;    /* posição do primeiro elemento */
         int final;     /* posição do último elemento  */
         int tam;       /* número de elementos da fila */
      } tfila;
```

As operações são agora executadas assim:

**1. Criar** - cria uma fila vazia

```
void criar (tfila *F)
{
   F->inicio = 0;
   F->final = -1;
   F->tam = 0;
}
```

**2. Vazia** - testa se uma fila está vazia

```
int vazia (tfila F)
{
   return (F.tam == 0);
}
```

**3. Primeiro** - obtém o elemento do início da fila

```
int primeiro (tfila F, telem *dado)
{
   if (vazia(F)) return 0;  /* Erro: fila vazia */

   *dado = F.v[F.inicio];

   return (1);
}
```

**5. Inserir** - insere um elemento no fim de uma fila

```
int inserir (tfila *F, telem valor)
{
   if (F->tam == MAX) return 0;

   (F->tam)++;
   F->final = (F->final + 1) % MAX;
   F->v[F->final] = valor;

   return(1);
}
```

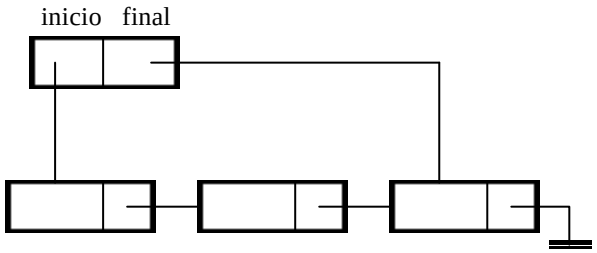
**6. Remover** - remove o elemento do início de uma fila, retornando o elemento removido

```
int remover (tfila *F, telem *valor)
{
   if (vazia(*F)) return 0;  /* Erro: fila vazia */

   primeiro(*F,valor);
   (F->tam)--;
   F->inicio = (F->inicio + 1) % MAX;

   return(1);
}
```

## 5.2 – IMPLEMENTAÇÃO ENCADEADA DE FILA

**Definição da Es trutura de Dados:**

```
#include <alloc.h>

typedef int telem;

typedef struct no {

         telem dado;
         struct no* prox;

      } tno;

typedef struct fila {

         tno* inicio;
         tno* final;

      } tfila;
```




**Operações:**

**1. Criar** - cria uma fila vazia

```
void criar (tfila *F)
{
   F->inicio = F->final = NULL;
}
```

**2. Vazia** - testa se uma fila está vazia

```
int vazia (tfila F)
{
   return (F.inicio == NULL && F.final == NULL);
}
```

**3. Primeiro** - obtém o elemento do início de uma fila

```
int primeiro (tfila F, telem *elem)
{
   if (vazia(F)) return 0;  /* erro: fila vazia */

   *elem = (F.inicio)->dado;

   return 1;
}
```

**4. Inserir** - insere um elemento no fim de uma fila

```
int inserir (tfila *F, telem valor)
{
   tno *novo;

   novo = (tno*) malloc(sizeof(tno));
   if (novo == NULL) return 0;  /* Erro: mem¢ria insuficiente */

   novo->dado = valor;
   novo->prox = NULL;

   if (vazia(*F))
      F->inicio = novo;
   else
      (F->final)->prox = novo;

   F->final = novo;

   return 1;
}
```

**5. Remover** - remove o elemento do início de uma fila, retornando o elemento removido

```
int remover (tfila *F, telem *valor)
{
   tno *aux;

   if (vazia(*F)) return 0;  /* Erro: fila vazia  */

   primeiro(*F,valor);

   if (F->inicio == F->final)
      F->final = NULL;

   aux = F->inicio;
   F->inicio = (F->inicio)->prox;
   free(aux);

   return 1;
}
```

## EXERCÍCIOS – FILAS

1) Implemente o TAD **Fila** com **representação seqüencial** em uma biblioteca chamada **F_SEQ** (usando como tipo base o tipo **inteiro**), contendo apenas a estrutura de dados e as operações básicas de filas (descritas anteriormente).

2) Implemente o TAD **Fila** com **representação encadeada** em uma biblioteca chamada **F_ENC** (usando como tipo base o tipo **inteiro**), contendo apenas a estrutura de dados e as operações básicas de filas (descritas anteriormente).

3) Faça um programa que, utilizando qualquer uma das bibliotecas criadas nos itens 1 e 2 :
   a) crie uma fila F;
   b) exiba o seguinte menu de opções:

> **EDITOR DE FILA**
>
> 1 – INSERIR
> 2 – REMOVER
> 3 – EXIBIR PRIMEIRO ELEMENTO
> 4 – EXIBIR A FILA
> 5 – ESVAZIAR A FILA
>
> DIGITE SUA OPÇÃO:

   c) leia a opção do usuário;
   d) execute a opção escolhida pelo usuário;
   e) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

4) Faça um programa que simule o tráfego de veículos por um semáforo. Considere o seguinte:
   a) O semáforo controla duas pistas que se cruzam, e cada uma possui direção única.
   b) O semáforo possui apenas as luzes verde e vermelha, levando 15 segundos para cada mudança de luz.
   c) A pista por onde o carro vai (1 ou 2) será escolhida aleatoriamente.
   d) Cada carro chegará ao semáforo num intervalo (aleatório) de 1 a 30 segundos após outro.
   e) Quando o semáforo estiver verde, cada carro da fila leva um tempo de 3 segundos após a saída do carro da frente para dar partida.

   Faça outras considerações que achar necessário.

   O programa deve apresentar como saída o movimento no cruzamento a cada instante, mostrando tanto as filas dos carros parados no semáforo, como também os carros que ainda irão chegar ao cruzamento.

   Sugestões:
   - Inicialmente, gere todos os possíveis carros que irão chegar ao semáforo, colocando-os em 2 filas: uma para a pista 1 e outra para a pista 2.
   - Os carros podem ser identificados pelo seu tempo previsto de chegada ao cruzamento.
   - Use filas também para os carros que estão parados no semáforo.
   - Apresente na tela o máximo de controles possíveis (contadores, flags, etc) que você estiver usando, para facilitar o acompanhamento da simulação.

# 6 – ÁRVORES

Nos capítulos anteriores estudamos a organização dos dados de uma maneira linear, onde a propriedade básica é a relação seqüencial mantida entre seus elementos.
Agora, estudaremos uma outra forma de organizar os dados, chamada genericamente de "não-linear". Essa forma permite que outros tipos de relação entre dados possam ser representados, como por exemplo, hierarquia e composição. Um dos exemplos mais significativos de estruturas não-lineares é a árvore.

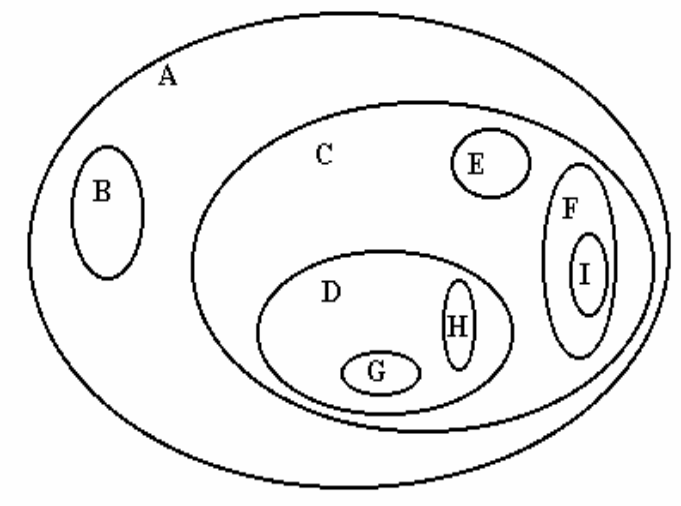
## Representação Gráfica

As três formas mais comuns de representação gráfica de uma árvore são:

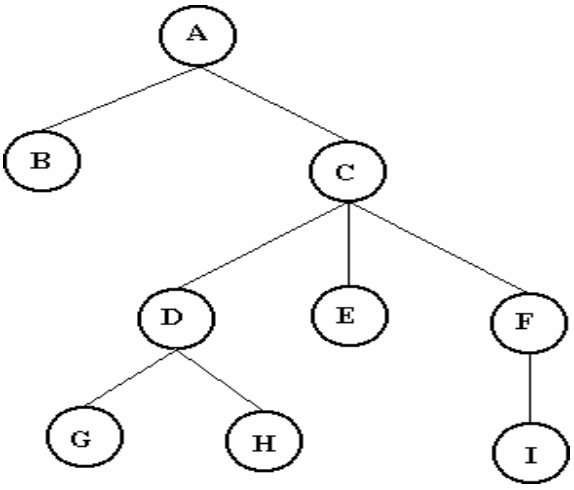
**1) Representação por parênteses aninhados**

```
(  A (B) ( C (D (G) (H)) (E) (F (I)) )  )
```

**2) Diagrama de inclusão**




**3) Representação hierárquica**



**Definição:**

Uma árvore T é um conjunto finito de elementos denominados nós tais que:

- T = $\phi$ , e a árvore é dita vazia, ou
- existe um nó especial r, chamado raiz de T; os restantes constituem um único conjunto vazio ou são divididos em m ≥ 1 conjuntos disjuntos não vazios que são as subárvores de r, cada qual, por sua vez, uma árvore.

**Notação:**

Se v é um nó de T, então a notação Tv indica a subárvore de T com raiz em v.

**Subárvore:**

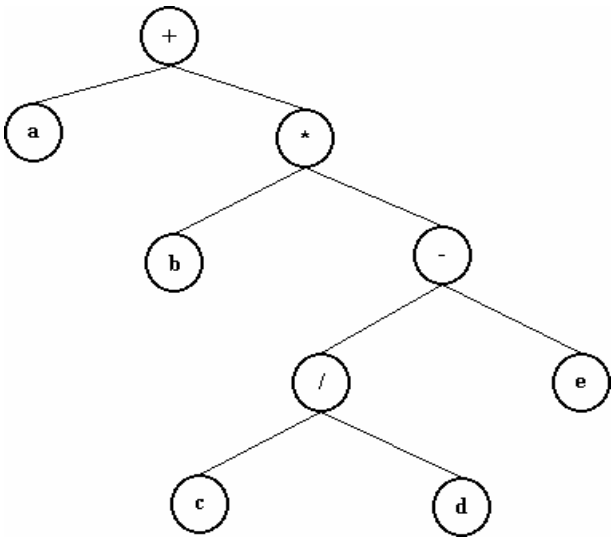
Seja a árvore acima T = {A, B, C, ...}

A árvore T possui duas subárvores: Tb e Tc, onde
Tb = { B } e Tc = {C, D, ...}

A subárvore Tc possui 3 subárvores: Td, Tf e Te, onde
```
Td = {D, G, H}
Tf = {F, I}
Te = {E}
```

As subárvores Tb, Te, Tg, Th, Ti possuem apenas o nó raiz e nenhuma subárvore.

**Exemplo:** representação da expressão aritmética: (a + (b * ( (c / d) - e) ) )




**Nós filhos, pais, tios, irmãos e avô**

Seja v o nó raiz da subárvore Tv de T.

Os nós raízes w1, w2, ... wj das subárvores de Tv são chamados filhos de v.

v é chamado pai de w1, w2, ... wj.

Os nós w1, w2, ...wj são irmãos.

Se z é filho de w1 então w2 é tio de z e v é avô de z.

**Grau de saída, descendente e ancestral**

O número de filhos de um nó é chamado grau de saída desse nó.

Se x pertence à subárvore Tv, então, x é descendente de v e v é ancestral, ou antecessor, de x.

**Nó folha e nó interior**

Um nó que não possui descendentes próprios é chamado de nó folha, ou seja, um nó folha é aquele com grau nulo.

Um nó que não é folha (isto é, possui grau diferente de zero) é chamado nó interior ou nó interno.

**Grau de uma árvore**

O grau de uma árvore é o máximo entre os graus de seus nós.

**Floresta**

Uma floresta é um conjunto de zero ou mais árvores.

**Caminho, comprimento do caminho**

Uma seqüência de nós distintos v1, v2, ..., vk, tal que existe sempre entre nós consecutivos (isto é, entre v1 e v2, entre v2 e v3, ... , v(k-1) e vk) a relação "é filho de"ou "é pai de" é denominada um caminho na árvore.

Diz-se que v1 alcança vk e que vk é alcançado por v1.

Um caminho de vk vértices é obtido pela seqüência de k-1 pares. O valor k é o comprimento do caminho.

**Nível (ou profundidade) e altura de um nó**

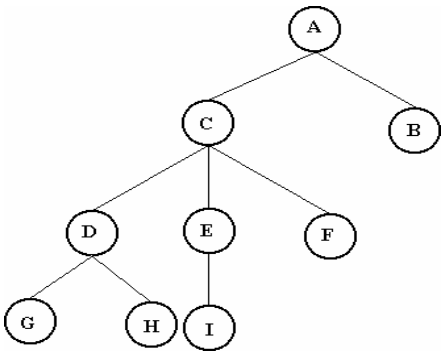
O nível ou profundidade, de um nó v é o número de nós do caminho da raiz até o nó v. O nível da raiz é, portanto, 1.

A altura de um nó v é o número de nós no maior caminho de v até um de seus descendentes. As folhas têm altura 1.

**Nível da raiz (profundidade) e altura de uma árvore**

A altura de uma árvore T é igual ao máximo nível de seus nós. Representa-se a altura de T por h(T) e a altura da subárvore de raiz v por h(v).

**Árvore Ordenada**




Uma árvore ordenada é aquela na qual os filhos de cada nó estão ordenados. Assume-se ordenação da esquerda para a direita. Desse modo a árvore do primeiro exemplo é ordenada, mas, a árvore acima não.

**Árvores Isomórfas**

Duas árvores não ordenadas são isomórfas quando puderem se tornar coincidentes através de uma permutação na ordem das subárvores de seus nós.

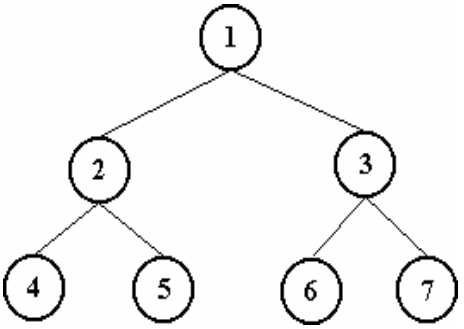
Duas árvores ordenadas são isomórfas quando forem coincidentes segundo a ordenação existente entre seus nós.

**Árvore Cheia**

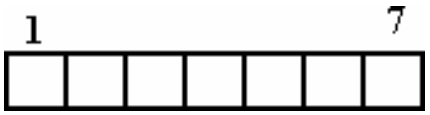
Árvore com número máximo de nós

Uma árvore de grau d tem número máximo de nós se cada nó, com exceção das folhas, tem grau d.

**Árvore cheia de grau 2: implementação seqüencial.**




Array com 7 posições:



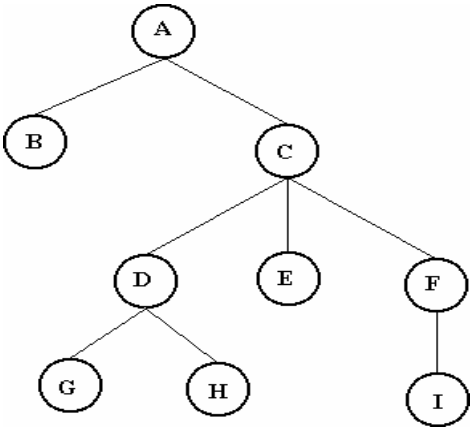

**Armazenamento por nível:**

| posição do nó | posição dos filhos do nó |
|---|---|
| 1 | 2,3 |
| 2 | 4,5 |
| 3 | 6,7 |
| i | (2i,2i+1) |

Dentre as árvores, as binárias são, sem dúvida, as mais comumente utilizadas nas aplicações em computação.

**EXERCÍCIOS**

1. Para a árvore abaixo:

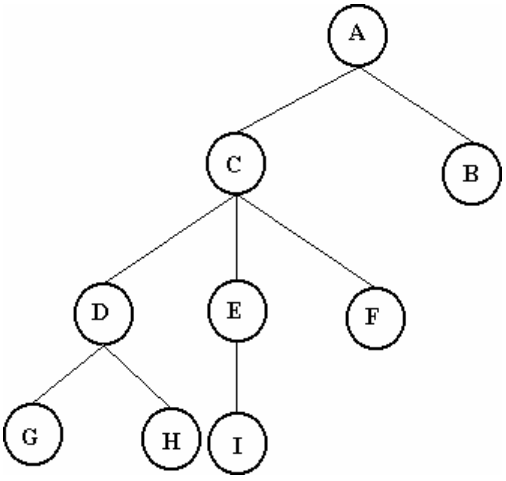


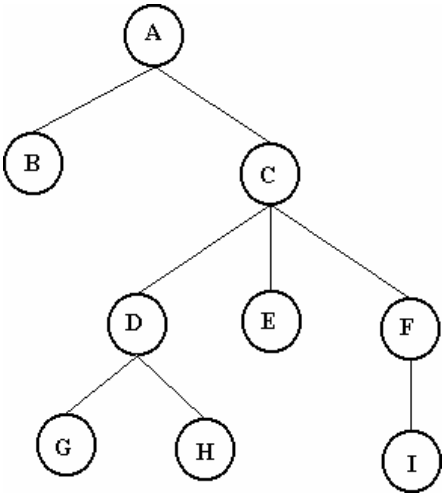

a) Quantas subárvores ela contém?
b) Quais os nós folhas?
c) Qual o grau de cada nó?
d) Qual o grau da árvore?
e) Liste os ancestrais dos nós B, G e I.
f) Liste os descendentes do nó D.
g) Dê o nível e altura do vértice F.
h) Dê o nível e a altura do vértice A.
i) Qual a altura da árvore ?

2. Dada uma árvore cujo grau da raiz é d, como transformá-la em uma floresta com d árvores?

3. Dada uma floresta com d árvores como transformá-la em u ma árvore cujo nó raiz tem grau d?

4. Explique porque a duas árvores abaixo são isomorfas.




6. Para uma árvore de grau d com número máximo de nós, diga:
- Qual o grau dos nós internos da árvore?
- Qual o grau dos nós folhas?
- Quantos nós tem a árvore se o grau d e a altura é h?
- Qual a altura da árvore se o grau é d e o número de nós é n?

7. Para uma árvore cheia de grau d:
- Se um nó estiver armazenado na posição i de um array, em que posições estarão seus d filhos?
- Considerando esta alocação seqüencial, quais as conseqüências de inserções e eliminações na árvore?

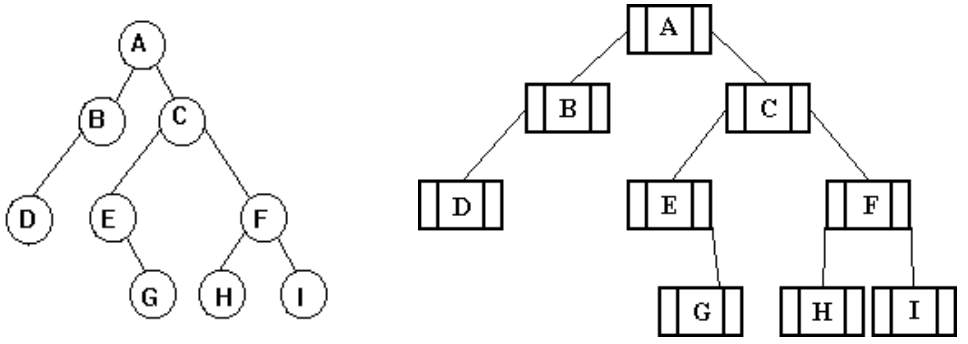
# 7 – ÁRVORE BINÁRIA

Uma Árvore Binária T é um conjunto finito de elementos denominados nós, tal que:

- se T = ϕ, a árvore é dita vazia ou
- existe um nó especial r, chamado raiz de T; os restantes podem ser divididos em dois subconjuntos disjuntos, Te e Td, que são as subárvores esquerda e direita de r, respectivamente e as quais, por sua vez, também são árvores binárias.

**Definição da Estrutura de Dados:**

```
typedef int telem;
typedef struct no{
         struct no* esq;
         telem info;
         struct no* dir;
      } tno;
typedef tno* tarvbin;
```




**Operações associadas ao TAD árvore binária padrão:**

1) Criar uma árvore vazia
2) Verificar se árvore está vazia ou não
3) Buscar um elemento na árvore
4) Inserir um nó raiz
5) Inserir um filho à direita de um nó
6) Inserir um filho à esquerda de um nó
7) Esvaziar uma árvore
8) Exibir a árvore

**1) Criar uma árvore vazia**

Define uma árvore vazia e deve ser utilizado antes de qualquer outro.

```
void criar(tarvbin *T)
{
   *T = NULL;
}
```

**2) Verifica se árvore vazia ou não**

Retorna 1 se árvore estiver vazia, 0 caso contrário.

```
int vazia(tarvbin T)
{
   return (T == NULL);
}
```

**3) Buscar um elemento na árvore**

Busca um elemento na árvore, retornando o seu endereço, caso o encontre. Se o elemento não for encontrado, retorna o endereço nulo (NULL).

```
tarvbin busca(tarvbin T, telem dado)
{
   tarvbin achou;

   if (T == NULL) return NULL;

   if (T->info == dado) return T;

   achou = busca(T->esq, dado);
   if (achou == NULL) achou = busca(T->dir, dado);
   return achou;
}
```

**4) Inserir um nó raiz**

Insere um nó raiz numa árvore vazia. Retorna 1 se a inserção for bem sucedida, ou 0 caso contrário.

```
int ins_raiz(tarvbin *T, telem dado)
{
   tarvbin novo;

   if (*T != NULL) return 0;  /* erro: já existe raiz */

   novo = (tno*) malloc(sizeof(tno));
   if (novo == NULL) return 0;  /* erro: memória insuficiente */

   novo->info = dado;
   novo->esq = novo->dir = NULL;
   *T = novo;
   return 1;
}
```

**5) Inserir um filho à direita de um dado nó**

```
int ins_dir(tarvbin T, telem pai, telem filho)
{
   tarvbin f, p, novo;

   /* verifica se o elemento já não existe */
   f = busca(T,filho);
   if (f != NULL) return 0;  /* erro: dado já existente */

   /* busca o endereço do pai e verifica se já não possui filho direito */
   p = busca(T,pai);
   if (p == NULL) return 0;  /* erro: pai não encontrado */
   if (p->dir != NULL) return 0;  /* erro: já existe filho direito */

   novo = (tno*) malloc(sizeof(tno));
   if (novo == NULL) return 0;  /* erro: memória insuficiente */

   novo->info = filho;
   novo->esq = novo->dir = NULL;
   p->dir = novo;
   return 1;
}
```

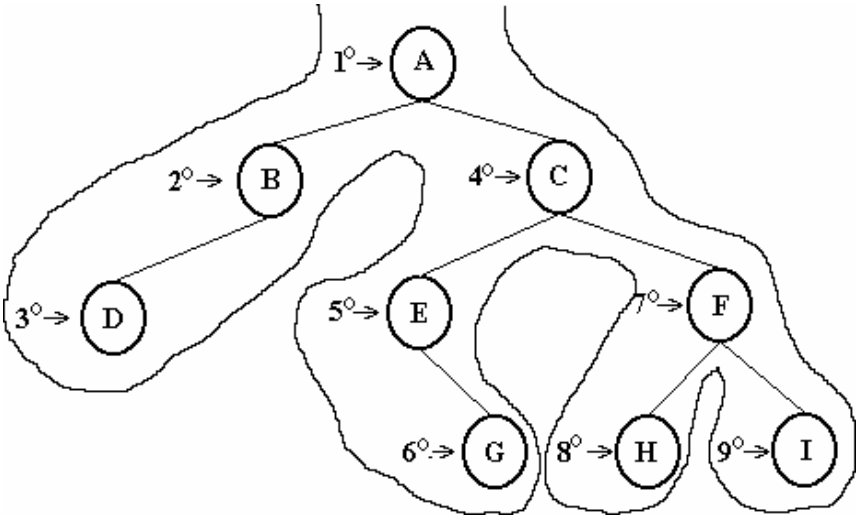
**A implementação da operação 6 fica como exercício para o aluno. A implementação das operações 7 e 8 serão mostradas mais adiante.**

## PERCURSO

Percorrer uma árvore visitando cada nó uma única vez gera uma seqüência linear de nós, e então passa a ter sentido falar em sucessor e predecessor de um nó segundo um determinado percurso. Há três maneiras recursivas de se percorrer árvores binárias.

**Travessia em Pré-Ordem**

6. se árvore vazia; fim
7. visitar o nó raiz
8. percorrer em pré-ordem a subárvore esquerda
9. percorrer em pré-ordem a subárvore direita

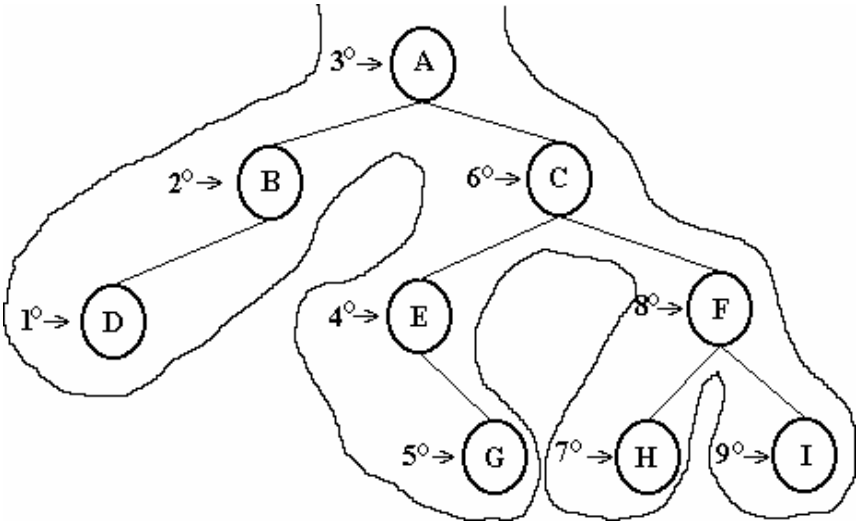



ABDCEGFHI

- visita o nó quando passar a sua esquerda
- notação pré-fix

**Travessia em In-Ordem**

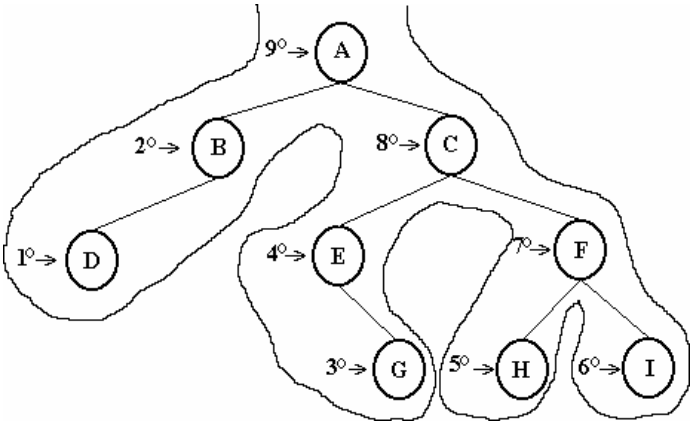
1. se árvore vazia, fim
2. percorrer em in-ordem a subárvore esquerda
3. visitar o nó raiz
4. percorrer em in-ordem a subárvore direita




- DBAEGCHFI
- visita o nó quando passar embaixo do nó
- notação in-fix

**Travessia em Pós-Ordem**

1. se árvore vazia, fim
2. percorrer em Pós-Ordem a subárvore esquerda
3. percorrer em Pós-Ordem a subárvore direita
4. visitar o nó raiz




- DBGEHIFCA
- visita o nó quando passar a sua direita
- notação pós-fix

**7) Esvaziar uma árvore**

Desaloca todo o espaço de memória da árvore e retorna a árvore ao estado equivalente ao define, isto é, nula. Utiliza o algoritmo de **Pós-Ordem** para percurso.

```
void esvaziar(tarvbin *T)
{
   if (*T == NULL) return;

   esvaziar(&(*T)->esq);
   esvaziar(&(*T)->dir);
   free(*T);
   *T = NULL;
}
```

**8) Exibir a árvore**

Um procedimento recursivo para exibir a árvore, usando um percurso pré-ordem, poderia ser o seguinte:

```
void exibir(tarvbin T, int col, int lin, int desloc)
{
   // col e lin são as coordenadas da tela onde a árvore irá iniciar,
   // ou seja, a posição da raiz, e desloc representa o deslocamento na tela
   // (em colunas) de um nó em relação ao nó anterior.

   if (T == NULL) return;  /* condição de parada do procedimento recursivo */

   gotoxy(col,lin);
   printf("%d",T->info);

   if (T->esq != NULL)
      exibir(T->esq,col-desloc,lin+2,desloc/2+1);

   if (T->dir != NULL)
      exibir(T->dir,col+desloc,lin+2,desloc/2+1);
}
```

## Implementação de uma árvore qualquer

Como seria a implementação de uma árvore de grau qualquer?

Numa árvore qualquer, cada nó pode conter um número diferente de subárvores. Utilizando alocação encadeada dinâmica, uma solução imediata seria fazer com que o tipo de dado dos nós tenha tantas referências (ponteiros) quanto o maior número de subárvores que um nó possa ter (grau da árvore).

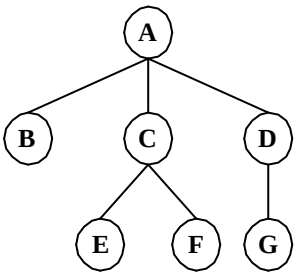
Com esta organização há, geralmente, uma grande quantidade de referências não usadas naqueles nós de grau muito menor do que o máximo previsto.

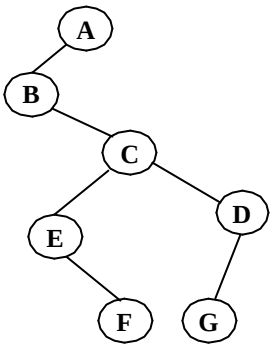
A solução alternativa mais econômica para este caso é transformar a árvore a ser representada em uma árvore binária.

## Conversão em árvore binária

Seja T uma árvore qualquer. T é convertida em uma árvore binária B(T) da seguinte maneira:

- B(T) possui um nó B(v) para cada nó v de T.
- As raízes de T e B(T) coincidem.
- O filho esquerdo de um nó B(v) em B(T) corresponde ao primeiro filho de v em T, caso exista. Se não existir, a subárvore esquerda de B(v) é vazia.
- O filho direito de um nó B(v) em B(T) corresponde ao irmão de v em T, localizado imediatamente a sua direita, caso exista. Se não existir, a subárvore direita de B(v) é vazia.



## EXERCÍCIOS

1. Implemente, em um arquivo chamado ARVBIN.LIB, o tipo abstrato de dados Arvore Binária, com alocação encadeada dinâmica e tendo como base o tipo inteiro. A biblioteca deve conter, além da estrutura de dados do TAD, as seguintes operações:

    a) Criar uma árvore vazia

    b) Verificar se árvore está vazia ou não

    c) Inserir um nó raiz

    d) Inserir um filho à direita de um nó

    e) Inserir um filho à esquerda de um nó

    f) Obter o filho esquerdo de um nó

    g) Obter o filho direito de um nó

    h) Buscar um elemento na árvore

    i) Buscar o pai de um elemento

    j) Excluir um nó (e todos os seus nós descendentes)

    k) Esvaziar uma árvore

    l) Exibir a árvore

2. Faça um programa que, utilizando a biblioteca ARVBIN.LIB, faça o seguinte:

    a) Crie uma árvore binária T.

    b) Exiba o seguinte menu de opções:

```
EDITOR DE ÁRVORE BINÁRIA

1 – INSERIR RAIZ
2 – INSERIR FILHO ESQUERDO
3 – INSERIR FILHO DIREITO
4 – REMOVER UM NÓ
5 – EXIBIR O PAI DE UM NÓ
6 – EXIBIR OS FILHOS DE UM NÓ
7 – EXIBIR OS ASCENDENTES DE UM NÓ
8 – EXIBIR OS DESCENDENTES DE UM NÓ
9 – ESVAZIAR A ÁRVORE
0 – EXIBIR A ÁRVORE

DIGITE SUA OPÇÃO:
```

    c) Leia a opção do usuário.

    d) Execute a opção escolhida pelo usuário.

    e) Após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

# 8 – PESQUISA DE DADOS

Iremos agora abordar métodos para pesquisar grandes quantidades de dados para encontrar determinada informação. Conforme veremos, certos métodos de organizar dados tornam o processo de busca mais eficiente. Como a operação de busca é uma tarefa muito comum em computação, o conhecimento desses métodos é de grande importância para a formação de um bom programador.

Antes de explicarmos os métodos, vamos definir alguns termos. Uma tabela ou arquivo é um grupo de elementos, cada uma dos quais chamado registro. Existe uma chave associada a cada registro, usada para diferenciar os registros entre si.

Para todo arquivo, existe pelo menos um conjunto exclusivo de chaves (possivelmente mais). Este tipo de chave é chamado de chave primária.

Entretanto, como todo campo de um registro pode servir como chave em uma determinada aplicação, nem sempre as chaves precisam ser exclusivas. Esse tipo de chave é chamado de chave secundária.

Quanto ao modo de organização da tabela ou do arquivo, ele pode ser um vetor de registros, uma lista encadeada, uma árvore, etc. Como diferentes técnicas de busca podem ser adequadas a diferentes organizações de tabelas, uma tabela é freqüentemente elaborada com uma técnica de busca em mente.

A tabela pode ficar totalmente contida na memória interna, totalmente na memória externa, ou pode ser dividida entre ambas. É evidente que são necessárias diferentes técnicas de pesquisa sob essas diferentes premissas. Neste momento, concentraremos nossos estudos nas técnicas apropriadas para busca em memória interna.

## 8.1 – PESQUISA SEQUENCIAL

Este é o método mais simples de pesquisa e consiste em uma varredura serial da tabela, durante a qual o argumento de pesquisa é comparado com a chave de cada registro até ser encontrada uma que seja igual, ou ser atingido o final da tabela, caso a chave procurada não se encontre na tabela.

A seguir é apresentado, na linguagem C, o algoritmo de pesquisa seqüencial em uma tabela não ordenada. O desempenho deste algoritmo é bastante modesto, já que o número médio de comparações para a localização de uma chave arbitrária em uma busca de sucesso é dado por (n+1)/2, considerando que todas as entradas possuem a mesma probabilidade de serem solicitadas. No caso de um busca sem êxito serão necessárias n comparações.

Adotaremos para a nossa implementação uma função que recebe como parâmetro um vetor de elementos inteiros (v), a quantidade de elementos do vetor (n) e a chave a ser pesquisada (k), retornando o endereço (índice) do elemento encontrado, ou –1 caso contrário.

```
int busca_sequencial (int v[], int n, int k)
{
  int i;
  for (i=0; i<n; i++)
    if (k == v[I]) return i;
  return –1;
}
```

A eliminação de um registro em uma tabela armazenada como um vetor desordenado é implementada substituindo-se o registro a ser eliminado pelo último registro no vetor e reduzindo-se o tamanho da tabela de 1.

### Usando sentinela (dummy)

Um método de busca ainda mais eficiente requer a inserção de chave de argumento no final do vetor antes de iniciar a operação de busca, garantindo assim que a chave seja encontrada. Com isto, eliminamos o teste para verificar se a varredura chegou ao final do vetor, já que na pior das hipóteses, a chave será encontrada na última posição, o que caracterizará que a busca não teve sucesso.

Também será necessário alocar uma posição a mais no vetor, justamente para armazenar a sentinela.

```
int busca_sequencial_com_sentinela (int v[], int n, int k)
{
  int i;
  k[n] = k;
  for (i=0; k != v[i]; i++);
  if (i < n)
    return i;
  else
    return –1;
}
```

### Reordenando a lista para obter mais eficiência

Existem várias aplicações onde nem todas as entradas possuem a mesma probabilidade de serem solicitadas. Em certas situações, um registro que foi ultimamente acessado pode ter maior probabilidade de um novo acesso do que os demais. Sendo assim, seria útil ter um algoritmo que reordenasse continuamente a tabela de modo que os registros mais acessados fossem deslocados para o início, enquanto os acessados com menos freqüência fossem deslocados para o final.

Existem dois métodos de busca que fazem essa operação: o método *mover-para-frente* e a *transposição*.

No método ***mover-para-frente***, sempre que uma pesquisa obtiver êxito, o registro recuperado será removido de sua atual posição da lista e colocado no início da lista. Obviamente esse método só será eficiente se aplicado sobre uma tabela organizada como uma lista encadeada.

O contra-argumento do método *mover-para-frente* é que uma única recuperação não implica necessariamente que o registro será recuperado freqüentemente; posiciona-lo no início da tabela reduzirá a eficiência da busca de todos os outros registros que o precediam anteriormente. Se avançarmos um registro somente uma posição sempre que ele for recuperado, garantiremos que ele avançará para o início da lista apenas se recuperado com freqüência.

Nesse método, chamado de ***transposição***, um registro recuperado com sucesso é trocado pelo registro imediatamente anterior.

Outra vantagem deste método em relação ao método *mover-para-frente*, é que ele pode ser aplicado com eficiência sobre tabelas armazenadas em vetores e sobre tabelas armazenadas em listas encadeadas.

## 8.2 – PESQUISA EM TABELA ORDENADA

Se a tabela estiver armazenada em ordem crescente ou decrescente de chaves de registros, várias técnicas poderão ser empregadas para aumentar a eficiência da operação de busca.

Uma vantagem evidente ao pesquisar uma tabela ordenada em comparação a uma tabela não-ordenada é no caso em que a chave do argumento está ausente da tabela. No caso de um arquivo não-ordenado, são necessárias *n* comparações para detectar esse fato. No caso de uma tabela ordenada, presumindo-se que as chaves dos argumentos estejam uniformemente distribuídas pela faixa de chaves na tabela, só serão necessárias *n/2* comparações (em média). Isso acontece porque sabemos que determinada chave está ausente numa tabela armazenada em ordem crescente de chaves assim que encontramos uma chave maior que o argumento. Na realidade, essa média é de *n* comparações, considerando que a cada elemento acessado são feitas duas comparações de chave.

A busca seqüencial em tabela ordenada pode ser assim implementada:

```
int busca_sequencial_ordenada (int v[], int n, int k)
{
  int i;
  for (i=0; (i<n)&&(k>=v[I]); i++)
    if (k == v[I])
       return i;
  return –1;
}
```

Devido à simplicidade e à eficiência do processamento seqüencial sobre tabelas ordenadas, talvez compense classificar um arquivo antes de pesquisar chaves dentro dele. Isto se verifica principalmente nas situações onde a maior parte das transações é de busca a registros, com pouquíssimas inserções e remoções na tabela.

## 8.3 – PESQUISA BINÁRIA

A pesquisa binária é um método que pode ser aplicado a tabelas ordenadas, armazenadas em dispositivos de acesso direto.

O método consiste na comparação do argumento de pesquisa (k) com a chave localizada no meio da tabela. Se for igual, a pesquisa termina com sucesso. Se k for maior, o processo é repetido para a metade superior da tabela e se for menor, para a metade inferior.

Assim, a cada comparação, a área de pesquisa é reduzida à metade do número de elementos. O número máximo de comparações será, portanto, igual a aproximadamente $log_2 n$ para a localização de uma entrada ou para a constatação de que ela não está presente na tabela. Na realidade, é $2*log_2 n$, porque, em C, fazemos duas comparações de chave de cada vez por meio da repetição.

Apesar de que a pesquisa binária pode ser mais bem definida recursivamente, é possível que a sobrecarga associada à recursividade a torne inadequada para uso em algumas situações. A seguir, é apresentado o algoritmo de pesquisa binária, numa versão não-recursiva.

```
int busca_binária (int v[], int n, int k)
{
  int low=0, hi=n-1, mid;
  while (low <= hi){
    mid = (low + hi) / 2;
    if (k == v[mid])
      return mid;
    if (k < v[mid])
      hi = mid – 1;
    else
      low = mid + 1;
  }
  return –1;
}
```

## 8.4 – AVALIAÇÃO DOS MÉTODOS DE BUSCA APRESENTADOS

Um dos principais critérios na avaliação dos métodos de busca é o número médio de comparações, que é obtido calculando-se o número médio de comparações efetuadas com algum elemento do vetor pesquisado.

A tabela a seguir compara os três métodos de busca discutidos anteriormente (a variável N, que aparece nas fórmulas, refere-se à quantidade de itens do vetor pesquisado).

| Critério Analisado | SEQUENCIAL | SEQ. ORDENADA | BINÁRIA |
|---|---|---|---|
| Número Médio de Comparações para Pesquisa Bem Sucedida | N/2 | N | $2*LOG_2N$ |
| Número Médio de Comparações para Pesquisa Incerta | N-(N*P/2) | N | $2*LOG_2N$ |
| Número Máximo de Comparações para Pesquisa Sem Sucesso | N | N | $2*((LOG_2N)+1)$ |

Onde: P = Probabilidade da pesquisa ser bem sucedida

## 8.5 – CONCLUSÕES

- A busca binária possui desempenho melhor que os outros dois métodos.
- Para o pior caso (da busca sem sucesso), os métodos seqüencial e seqüencial ordenado possuem, praticamente, o mesmo desempenho.
- A busca binária e a seqüencial ordenada requerem um esforço adicional para classificar o vetor de pesquisa.
- Para vetores pequenos, os três métodos assemelham-se; à medida que N cresce, as diferenças no número médio de comparações cresce assustadoramente (veja tabela a seguir).

| NÚMERO MÉDIO DE COMPARAÇÕES PARA BUSCA BEM SUCEDIDA | | | |
|---|---|---|---|
| **TAMANHO DO VETOR ( N )** | **PESQ. BINÁRIA $2 * LOG_2N$** | **PESQ. SEQ. ORDENADA N / 2** | **PESQ. SEQUENCIAL N** |
| 2 | 2 | 1 | 2 |
| 4 | 4 | 2 | 4 |
| 8 | 6 | 4 | 8 |
| 16 | 8 | 8 | 16 |
| 32 | 10 | 16 | 32 |
| 64 | 12 | 32 | 64 |
| 1024 | 20 | 512 | 1024 |

## EXERCÍCIOS:

1. Qual a vantagem da utilização de um dummy (sentinela) em um algoritmo de busca seqüencial?

2. Em algumas situações, a busca seqüencial se torna mais eficiente utilizando-se as técnicas de "mover-para-frente" e de "transposição". Que situações seriam essas e qual a diferença entre estas duas técnicas?

3. Faça um programa que gere, aleatoriamente, um vetor V de 10 elementos inteiros, leia um valor inteiro K e, usando a pesquisa seqüencial, verifique se existe algum elemento de V que seja igual a K.

4. Escreva uma nova versão para o método da busca seqüencial, acrescentando a ele o método da transposição.

5. Faça um programa que gere um vetor V contendo os 10 primeiros números inteiros múltiplos de 5, leia um valor inteiro K e, usando a pesquisa binária, verifique se existe algum elemento de V que seja igual a K.

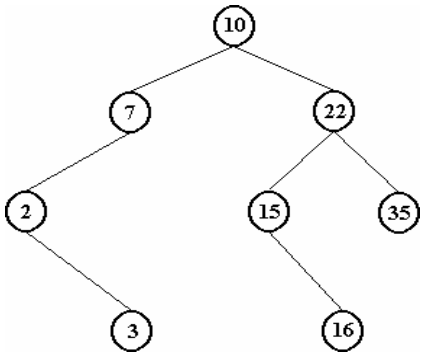
6. Escreva uma versão recursiva para o método da pesquisa binária.

7. Faça uma análise comparativa, do ponto de vista de desempenho, dos 3 métodos de pesquisa apresentados.

# 9 – ÁRVORE BINÁRIA DE PESQUISA

Uma **Árvore Binária de Pesquisa T (ABP)** ou **Árvore Binária de Busca** é tal que T = $\phi$ e a árvore é dita vazia ou seu nó raiz contém uma chave e:

1. Todas as chaves da subárvore esquerda são menores que a chave da raiz.
2. Todas as chaves da subárvore direita são maiores que a chave raiz.
3. As subárvores direita e esquerda são também Árvores Binárias de Busca.




Num algoritmo de busca a medida de eficiência é dada pelo número de comparações necessárias para se localizar uma chave, ou descobrir que ela não existe.

Numa lista linear com n chaves, temos que, no pior caso, faremos n comparações. O número de comparações cresce linearmente em função do número de chaves.

**BUSCA BINÁRIA**

Pesquisa realizada se informação está arma zenada de forma ordenada e em seqüência (em um array).

Qual a eficiência do algoritmo de busca binária?

Cada comparação na busca binária reduz o número de possíveis candidatos por uma fator de 2. Sendo assim, o número máximo de comparações da chave é aproximadamente $\mathbf{log_2 n}$.

Com busca binária obtemos a melhor eficiência possível em array, mas, ficamos limitados à representação seqüencial (deslocamentos, previsão de memória, etc).

Podemos utilizar a Árvore Binária de Pesquisa e obteremos o mesmo desempenho anterior (desde que a altura seja mínima). É a característica de altura mínima que garante que estamos tomando a chave do meio da porção pesquisada.

Para garantir a performance ótima temos que garantir que a árvore seja balanceada durante a construção e que o balanceamento seja mantido em inserções e eliminações (árvore AVL).

**ESTRUTURA DE DADOS**

```
typedef _____ telem;    /* tipo base dos elementos da árvore */

typedef struct no {
            struct no *esq;
            telem info;
            struct no *dir;
} tno;                  /* nó da árvore */

typedef tno *tabp;      /* ponteiro para a raiz da árvore */
```

**OPERAÇÕES BÁSICAS:**

1. Criação
2. Testar se está vazia
3. Busca
4. Inserção
5. Remoção
6. Exibição
7. Esvaziar

**1. Criação**

A criação de uma ABP se dá de forma idêntica a de uma árvore binária padrão, ou seja, incializando a variável que aponta para a raiz da árvore com o valor nulo.

```
void criar (tabp *T)
{
  *T = NULL;
}
```

**2. Testar se está vazia**

Também esta operação não tem nenhuma diferença ao que foi visto na árvore binária comum. A condição para a ABP estar vazia é se a variável que aponta para a raiz contém o valor nulo.

```
int vazia (tabp T)
{
  return (T == NULL);
}
```

**3. Busca**

Função que procura um símbolo em uma árvore binária de pesquisa. Se o símbolo procurado estiver presente na árvore, a função retorna o endereço do nó que o contém. Caso contrário, o valor retornado é o endereço nulo. Torna-se mais eficiente se implementada com percurso pré-ordem.

```
tabp busca(tabp T, telem dado)
{
   if (T == NULL) return NULL;

   if (T->info == dado) return T;

   if (T->info > dado)
      return busca(T->esq, dado);
   else
      return busca(T->dir, dado);
}
```

**4. Inserção**

Os novos elementos inseridos em uma árvore entram sempre na condição de folhas. O processo de inserção desenvolve-se da seguinte forma:

a) Se a árvore for vazia, o símbolo é instalado na raiz e a função retorna o valor 1.

b) Caso contrário, é instalado na sub-árvore da esquerda, se for menor que o símbolo da raiz, ou da direita, se for maior.

c) Se for igual ao símbolo da raiz, o elemento não será inserido na árvore e a função retornará o valor 0.

```
int inserir (tabp *T, telem item)
{
  int ok;
  if (*T == NULL) {
    *T = (tno *) malloc(sizeof(tno));
    if (*T == NULL) return 0;
    (*T)->esq = NULL;
    (*T)->dir = NULL;
    (*T)->info = item;
    return 1;
  }
  if ((*T)->info < item)
    ok = inserir (&((*T)->dir), item);
  else
    if ((*T)->info > item)
      ok = inserir (&((*T)->esq), item);
    else
      ok = 0;
  return ok;
}
```

**5. Exclusão**

Para a remoção de um nó de uma árvore deve-se levar em consideração que seus filhos devem continuar na árvore e esta deverá continuar ordenada (menor à esquerda, maior à direita). E então caímos em três possibilidades que devem ser tratadas separadamente, a fim de manter a estrutura da árvore binária. Sendo elas:

a) Quando o nó a ser excluído não contenha filhos:
   O nó simplesmente é removido.
b) Quando o nó a ser excluído contenha somente um dos filhos:
   O pai do nó a ser excluído passa a apontar para este filho e o nó é removido.
c) Quando o nó a ser excluído contenha os dois filhos:
   Busca-se o maior elemento da sub-árvore da esquerda (a partir da raiz da sub-árvore esquerda caminha-se sempre para a direita até achar um nó cujo filho à direita é nulo).
   Transfere-se a informação deste nó para o nó a ser removido e remove-se este novo nó, que cairá no caso (a) ou (b).

Vamos deixar a cargo do aluno, como exercício, a implementação desta rotina.

**6. Exibição**

Para exibir todos os símbolos da árvore de forma a se preservar a ordenação, deve-se utilizar o percurso in-ordem.

```
void exibir (tabp T)
{
  if (T != NULL) {
    exibir (T->esq);
    printf ("%d ",T->info);
    exibir (T->dir);
  }
}
```

**7. Esvaziar**

Para remover todos os elementos da uma ABP, deve-se utilizar o percurso pós-ordem. Esta operação é implementada da mesma forma que na árvore binária padrão.

```
void esvaziar(tabp *T)
{
   if (*T == NULL) return;
   esvaziar(&(*T)->esq);
   esvaziar(&(*T)->dir);
   free(*T);
   *T = NULL;
}
```

## EXERCÍCIO

1. Implemente, em um arquivo chamado ABP.LIB, o tipo abstrato de dados ABP (árvore binária de pesquisa), com alocação encadeada dinâmica e tendo como base o tipo inteiro. A biblioteca deve conter, além da estrutura de dados do TAD, as seguintes operações:

    a) Criar uma árvore binária de pesquisa vazia

    b) Verificar se uma árvore está vazia ou não

    c) Inserir um novo elemento

    d) Remover um elemento, dado o seu valor

    e) Verificar se a uma árvore contém um dado valor, retornando o endereço do nó, caso encontrado, ou nil, caso contrário.

    f) Exibir todos os valores de uma árvore

    g) Esvaziar uma árvore

2. Faça um programa que, utilizando a biblioteca ABP.LIB, faça o seguinte:

    a) Crie uma árvore binária de pesquisa T.

    b) Exiba o seguinte menu de opções:

```
EDITOR DE ÁRVORE BINÁRIA DE PESQUISA

1 – INSERIR
2 – REMOVER
3 – PESQUISAR
4 – EXIBIR A ÁRVORE
5 – ESVAZIAR A ÁRVORE

DIGITE SUA OPÇÃO:
```

    c) Leia a opção do usuário.

    d) Execute a opção escolhida pelo usuário.

    e) Após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

# 10 – ÁRVORE AVL

## ÁRVORE BALANCEADA

Sucessivas inserções e remoções de nós em uma árvore binária de pesquisa fazem com que existam diferenças entre os níveis das suas folhas, o que acarreta grandes diferenças de performance no acesso a seus nós.

No pior caso, se os dados já estão ordenados, a árvore será uma lista, perdendo toda a eficiência da busca. Nesses casos, diz-se que a árvore ficou degenerada.

Uma árvore é dita balanceada quando, para qualquer nó, as suas sub-árvores à esquerda e à direita possuem a mesma altura. Isso equivale a dizer que todos os ponteiros nulos estão no mesmo nível, ou seja, que a árvore está completa. Isso nem sempre é possível, por conta do número de nós da árvore.

Um critério mais simplificado seria considerar uma árvore balanceada se o número máximo de nós da sub-árvore esquerda diferencie no máximo 1 da sub-árvore direita.

## BALANCEAMENTO ESTÁTICO

Consiste em construir uma nova versão de uma árvore binária de pesquisa, reorganizando-a. Essa reorganização possui duas etapas:

1. Percurso in-ordem sobre a árvore, para gerar um vetor ordenado com o conteúdo de todos os seus nós.

2. Criação de uma ABP a partir desse vetor, da seguinte forma:
   a) Identifica-se o nó médio do vetor, que passa a ser considerado a raiz da ABP que está sendo criada.
   b) Cada uma das metades do vetor é tratada analogamente, de modo que cada nó intermediário será a raiz de uma sub-árvore.

## BALANCEAMENTO DINÂMICO: AVL

Em 1962, dois matemáticos russos (Adelson-Velskii e Landis) definiram uma nova estrutura para balanceamento de árvores ABP e deram o nome de AVL.

Esta nova estrutura permite o balanceamento dinâmico da árvore e com boa performance.

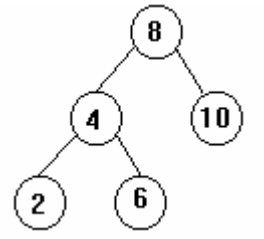
**Fator de Balanceamento (FB) de um nó**

É a altura da subárvore direita do nó menos a altura da subárvore esquerda do nó.

**Inserção AVL:**

O que pode acontecer quando um novo nó é inserido numa árvore balanceada?

Na árvore abaixo:




- Nós 9 ou 11 podem ser inseridos sem balanceamento. Subárvore com raiz 10 passa a ter uma subárvore e subárvore com raiz 8 vai ficar melhor balanceada.

- Inserção dos nós 1, 3, 5 ou 7 requerem que a árvore seja rebalanceada.

Seja T uma árvore AVL na qual serão feitas inserções de novos nós. Para que a árvore se mantenha AVL (ABP e balanceada), é preciso efetuar operações de rebalanceamento quando necessário.

A idéia consiste em verificar, após cada inclusão, se algum nó ficou desbalanceado, isto é, se a diferença de altura entre as 2 sub-árvores ficou maior do que 1. Em caso afirmativo, aplicam-se as transformações apropriadas para rebalanceá-lo.

Torna-se então necessário que a estrutura dos nós da árvore contenha mais um campo (**bal**) além dos já existentes (ESQ, INFO e DIR). Este novo campo armazenará o Fator de Balanceamento (FB) de cada nó.

Veja abaixo como será a estrutura de dados para uma árvore AVL:

```
typedef _____ telem;

typedef struct no {
        struct no *esq;
        telem info;
        int bal;
        struct no *dir;
        } tno;

typedef tno *tavl;
```

Serão utilizadas 4 transformações, indicadas nas figuras dos casos abaixo. Observe que as subárvores T1, T2, T3 e T4 podem ser vazias ou não.

O ponteiro T aponta para o nó raiz p, que é o nó raiz da transformação. Observe que a ordem dos nós no percurso in-ordem é preservada em todas as rotações, e portanto é possível aplicar qualquer número de rotações sobre uma árvore para torná-la balanceada, sem alterar a ordem dos nós. Ou seja, as transformações preservam a árvore como sendo ABP.

Vamos analisar o que pode ocorrer durante o processo de inserção em uma árvore AVL - e aí vai ficar claro o contexto em que estas transformações são aplicadas.

Suponha que um nó q foi inserido em uma árvore AVL T. Se, após a inserção todos os nós de T permanecem balanceados, a árvore continua AVL. Caso contrário, seja p o nó mais próximo às folhas de T que ficou desbalanceado (ou seja, o ancestral mais jovem que ficou desbalanceado). Observe que não há ambigüidade na escolha de p, pois qualquer sub-árvore de T que se tornou desbalanceada após a inclusão de q deve necessariamente conter p. Logo, p se encontra no caminho entre q e a raiz de T, e sua escolha é única.

Sejam hE(p) e hD(p) as alturas das sub-árvores esquerda e direita de p, respectivamente. Naturalmente, |hE(p)-hD(p)| > 1. Então, conclui-se que |hE(p)-hD(p)| = 2, pois T era uma árvore AVL e a inserção de um único nó não pode aumentar a altura de qualquer sub-árvore em mais do que 1. As 4 situações possíveis a serem analisadas são:

**Caso 1: hE(p) > hD(p)**
Seja q o nó que foi inserido na sub-árvore esquerda de p. Além disso, sabe-se que p possui um filho esquerdo u diferente de q (caso contrário, p não teria ficado desbalanceado). Finalmente, por esse mesmo motivo, sabe-se que hE(u) é diferente de hD(u). Neste caso, são duas situações possíveis:

**Caso 1.1: hE(u) > hD(u)**
Esta situação está ilustrada na fig 1(a), sendo q um nó pertencente a T1. Observe que h(T1) - h(T2) = 1 e h(T2) = h(T3).

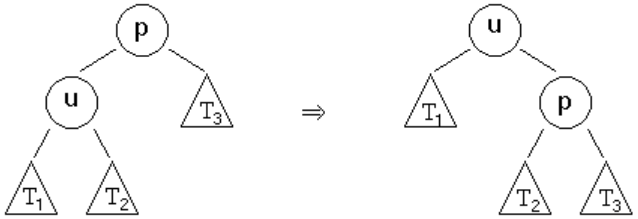



fig.1: rotação para a direita

```
void rot_dir (tavl *T)
{
  tavl u;
  u = (*T)->esq;
  (*T)->esq = u->dir;
  u->dir = *T;
  (*T)->bal = 0;
  *T = u;
}
```

Conseqüentemente, uma rotação direita da árvore com raiz p transforma a sub-árvore considerada na árvore da fig.1(b), o que restabelece o balanceamento da sub-árvore em p e, conseqüentemente, da sub-árvore T.

**Caso 1.2: hD(u) > hE(u)**
Neste caso, u possui um filho direito v, e a situação é ilustrada na fig.2(a), sendo que o modulo de h(T2)- h(T3) é menor que 1 e o máximo de h(T2) e h(T3) = h(T1)=h(T4).

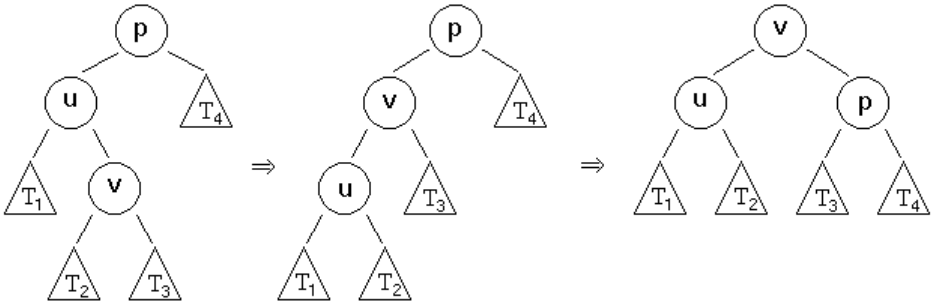



fig.2: rotação para a esquerda e em seguida para a direita

```
void rot_esq_dir (tavl *T)
{
  tavl u,v;
  u = (*T)->esq;
  v = u->dir;
  u->dir = v->esq;
  v->esq = u;
  (*T)->esq = v->dir;
  v->dir = *T;
  if (v->bal == -1)
    (*T)->bal = 1;
  else
    (*T)->bal = 0;
  if (v->bal == 1)
    u->bal = -1;
  else
    u->bal = 0;
  *T = v;
}
```

Aplica-se então a rotação esquerda_direita à raiz p. A nova configuração da árvore aparece em 2(b), e o balanceamento é restabelecido.

**Caso 2: hD(p) > hE(p)**
Ou seja, q foi inserido na subárvore direita de p. Dessa vez, sabe-se que p possui um filho direito u diferente de q (caso contrário, p não teria ficado desbalanceado). Segue que hE(u) é diferente de hD(u), e as duas situações possíveis são:

**Caso 2.1: hD(u) > hE(u)**
Este caso corresponde ao da fig.3(a), sendo q pertencente a T3. As relações de altura são h(T3) - h(T2) = 1 e h(T2) = h(T1).

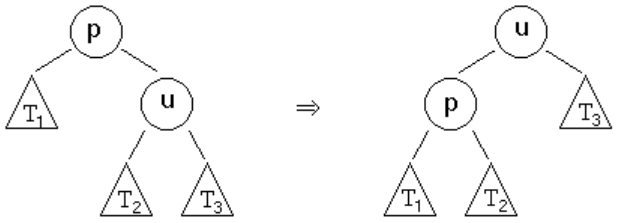



fig.3: rotação para a esquerda

```
void rot_esq (tavl *T)
{
  tavl u;
  u = (*T)->dir;
  (*T)->dir = u->esq;
  u->esq = *T;
  (*T)->bal = 0;
  *T = u;
}
```

Isso significa que a rotação esquerda torna a árvore novamente AVL (fig 3(b)).

**Caso 2.2: hE(u) > hD(u)**
Então, u possui o filho esquerdo v (fig.4(a)). Observe que as alturas da sub-árvores T1, T2, T3 e T4 satisfazem as mesmas relações do caso 1.2. Dessa forma, a aplicação da rotação direita-esquerda torna a árvore novamente AVL (fig.4(b)), ao balancear a sub-árvore com raiz p.

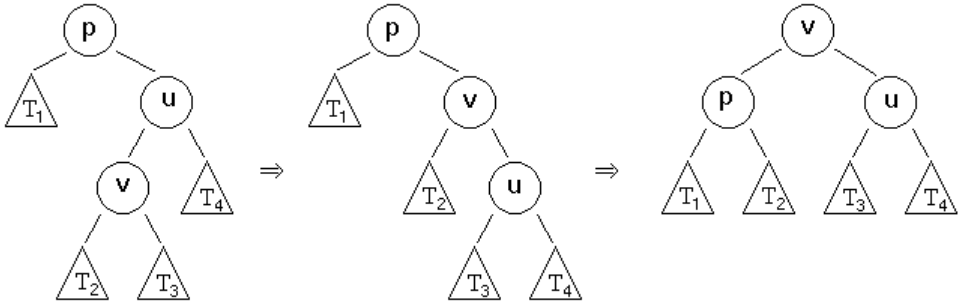



fig.4: rotação para a direita e em seguida para a esquerda

```
void rot_dir_esq (tavl *T)
{
  tavl u,v;
  u = (*T)->dir;
  v = u->esq;
  u->esq = v->dir;
  v->dir = u;

  (*T)->dir = v->esq;
  v->esq = *T;
  if (v->bal == 1)
    (*T)->bal = -1;
  else
    (*T)->bal = 0;
  if (v->bal == -1)
    u->bal = 1;
  else
    u->bal = 0;
  *T = v;
}
```

É fácil concluir que o rebalanceamento de p implica também no rebalanceamento de seu nó pai, pois a transformação aplicada diminui em 1 a altura da sub-árvore com raiz em p. Isso assegura o rebalanceamento de todos os nós ancestrais de p, portanto uma única transformação resulta no rebalanceamento de T.

## Implementação do procedimento de inserção

Seja T uma árvore AVL, e x a chave a ser incluída em algum novo nó q. O processo pode ser descrito do seguinte modo: Primeiro, busca a chave x. Se ela já está na árvore, nada precisa ser feito. Caso contrário, a busca determina a posição de inserção da nova chave, e a inserção é feita. A seguir, verifica se a inserção desbalanceou algum nó. Em caso negativo, a inserção termina. Caso contrário, é feito o rebalanceamento de T, através de uma das operações de rotação vistas, dependendo do caso.

Para verificar se algum nó de T ficou desbalanceado, vamos utilizar o campo **bal** existente na estrutura de cada nó. Esse campo armazena, para cada nó v, o valor hD(v) - hE(v). O nó está balanceado se $-1 \leq$ v->bal $\leq 1$. O problema é como atualizar este campo de forma eficiente, tendo em vista a inclusão de q. Se q for inserido na subárvore esquerda de v, e esta inclusão ocasionar um aumento na altura desta subárvore, subtrai-se 1 de v->bal. Se esse valor ficar igual a -2, então v ficou desbalanceado. Analogamente, se q for inserido na subárvore direita de v e provocar um aumento na sua altura, adiciona-se 1 a v->bal, e o nó ficará desbalanceado se o valor resultante for 2.

Para completar o processo, precisamos identificar os casos em que a inclusão de q provoca um aumento na altura h(v) da subárvore Tv. Inicialmente, observamos que a inserção do nó q acarreta obrigatoriamente uma alteração na subárvore esquerda ou direita de seu nó pai w. O campo balanço permite avaliar se esta alteração pode ou não se propagar aos outros nós v que estão no caminho entre w e a raiz da árvore.

Suponha que o nó q foi inserido na subárvore esquerda de v. A análise se inicia com v = w e prossegue em seus nós ancestrais, de forma recursiva. O processo é encerrado quando se acha a raiz de uma sub-árvore Tv. que não foi modificada. Três situações distintas podem ocorrer:

**Caso A: v->bal=1 antes da inclusão.**
Neste caso, v↑.bal passa a ser 0, e a altura da subárvore de raiz v não foi modificada. Conseqüentemente, os balanços dos nós no caminho entre v e a raiz não foram alterados.

**Caso B: v->bal=0 antes da inclusão.**
Neste caso, v->bal passa a ser -1, e a altura da subárvore de raiz v foi modificada. Conseqüentemente, os nós no caminho de v até a raiz podem ter seus balanços modificados, e devem ser analisados. Se v é a raiz de T, a análise se encerra, pois nenhum nó ficou desbalanceado. Caso contrário, deve-se repetir o processo substituindo v pelo seu pai.

**Caso C: v->bal=-1 antes da inclusão.**
Neste caso, v->bal passa a ser -2, e o nó ficou desbalanceado. A rotação correta deve ser aplicada. Qualquer rotação implica em que a subárvore resultante tenha a mesma altura da subárvore antes da inclusão. As alturas dos ancestrais de v não precisam ser avaliadas.

Para uma inserção na subárvore direita de v, casos simétricos devem ser considerados.

A rotina de busca e inserção é dada a seguir. A chamada externa é ins_avl (&T.item), onde:

- T é o endereço da raiz da árvore;
- item é a chave a ser inserida;
- O retorno será 1 se a inserção tiver sucesso ou 0 caso contrário e servirá como um auxiliar para propagar a verificação do FB.

Foram criadas duas rotinas auxiliares, **caso1** e **caso2**, para facilitar a implementação.

```
void caso1(tavl *T)
{
  /* item foi inserido à esquerda de T e causa desbalanceamento FB(T)=-2 */

  tavl u;
  u = (*T)->esq;

  if (u->bal == -1)
    rot_dir(&(*T));      /* Caso 1.1 sinais iguais e negativo */
  else
    rot_esq_dir(&(*T));  /* Caso 1.2 sinais diferentes */

  (*T)->bal = 0;
}
```

```
void caso2(tavl *T)
{
  /* item foi inserido à direita de T e causa desbalanceamento FB(T)=2 */
  tavl u;
  u = (*T)->dir;
  if (u->bal == 1)
    rot_esq(&(*T));      /* Caso 2.1 sinais iguais e positivo */
  else
    rot_dir_esq(&(*T));  /* Caso 2.2 sinais diferentes */
  (*T)->bal = 0;
}

int ins_avl(tavl *T, telem item)
{
  int ok;

  if (*T == NULL) {
    *T = (tno *) malloc(sizeof(tno));
    if (*T == NULL) return 0;
    (*T)->esq = NULL;
    (*T)->dir = NULL;
    (*T)->info = item;
    (*T)->bal = 0;
    return 1;
  }

  if ((*T)->info > item) {      /* recursividade à esquerda */
    ok = ins_avl (&((*T)->esq), item);
    if (ok)
      switch ((*T)->bal) {       /* próxima raiz a se verificar o FB */
        case 1 : (*T)->bal = 0;  /* era mais alto à direita, fica com FB=0 */
                 ok = 0;        /* interrompe propagação de balanceamento */
                 break;
        case 0 : (*T)->bal = -1; /* ficou com a esquerda maior e propaga FB */
                 break;
        case -1: caso1(&(*T));   /* FB(p)=-2 */
                 ok = 0;        /* nao propaga mais */
                 break;
      }
  }

  else
    if ((*T)->info < item) {      /* recursividade à direita */
      ok = ins_avl (&((*T)->dir), item);
      if (ok)

      switch ((*T)->bal) {        /* próxima raiz a se verificar o FB */
        case -1: (*T)->bal = 0;   /* era mais alto à esquerda, fica com FB=0 */
                 ok = 0;          /* interrompe propagação de balanceamento */
                 break;
        case 0 : (*T)->bal = 1;   /* ficou com a direita maior e propaga FB */
                 break;
        case 1 : caso2(&(*T));    /* FB(p)=2 */
                 ok = 0;          /* nao propaga mais */
                 break;
      }
    }

    else
      ok = 0;

  return ok;
}
```

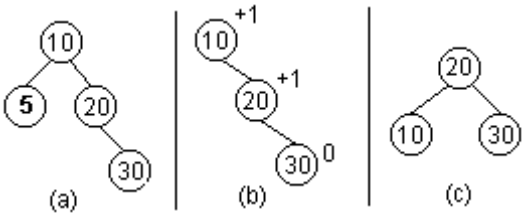
## Remoção em Árvore AVL

A retirada de um valor de uma Árvore Binária de Busca é mais complexa do que a inserção, porque, enquanto qualquer inserção sempre ocorre numa folha da árvore, uma retirada envolve pelo menos dois casos distintos: os nós com dois filhos e os demais nós. Nas árvores AVL essa dificuldade permanece.

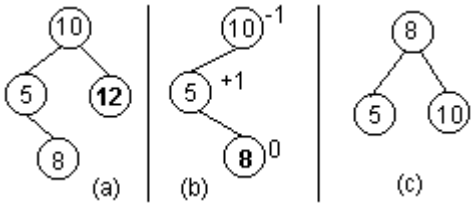
**Método manual de remoção de árvore AVL**

Inicialmente, faz-se a retirada do nó, usando o algoritmo de busca e retirada de uma ABP. Se não desbalanceou, o processo está encerrado. Se desbalanceou a árvore, isto é, se um ou mais nós ficou com |FB(nó)| > 1, raciocina-se em termos de inserção, perguntando: se o desbalanceamento ocorresse devido a uma inserção, que nó teria sido inserido para causar tal desequilíbrio? Identificado o nó, simula-se sua inserção e faz-se a rotação necessária.

**Exemplo 1**: dada a árvore abaixo(a), retirando o 5 resulta uma árvore desbalanceada no nó 10(b). A partir daí, raciocina-se como se estivéssemos inserindo: que nó inserido teria causado esse desequilíbrio? o 30. Aplicando os conhecimentos de inserção em árvore AVL, constata-se que uma rotação simples à esquerda resolve o problema. O resultado está na AVL (c).

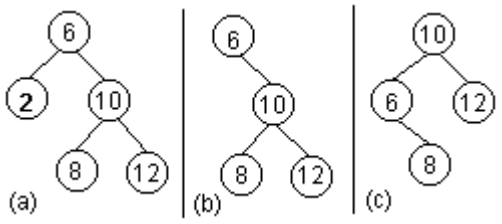



**Exemplo 2**: Retirando a folha 12 de (a), na figura abaixo, desbalanceia a raiz (b); supondo-se a inserção recente de 8, corrige-se o desequilíbrio mediante uma rotação dupla à esquerda (c).




**Caso especial**:
Seja a árvore AVL (a), no desenho abaixo. A retirada da folha 2 desbalanceia a raiz 6 (b). Todavia, essa configuração jamais poderia resultar de uma seqüência de inserções, pois, se ela fosse 8, 12 ou 12, 8, a primeira dessas inclusões provocaria rotação.

**Solução**: escolhe-se arbitrariamente um desses dois nós, despreza-se o outro (mantendo-o na árvore, obviamente), e simula-se a sua inserção. Escolhemos o 12, que exige uma operação mais simples: rotação simples à esquerda. O resultado é a árvore AVL (c).




Se escolhêssemos o 8, desprezando o 12, a rotação exigida seria dupla à direita, mas chegaríamos, também, a um resultado satisfatório (experimente).

Infelizmente, há situações mais complexas, onde o próprio processo de balanceamento devido a retirada de um nó de uma subárvore, por reduzir a altura desta, pode provocar um novo desequilíbrio na árvore a qual ela pertence.

Então, não resta outra saída senão reaplicar o método para a árvore que desbalanceou. E novo desequilíbrio pode ser provocado mais acima, na estrutura, exigindo novo balanceamento. E assim por diante, até que toda a árvore volte a ser uma AVL.

## EXERCÍCIO

1. Implemente, em um arquivo chamado AVL.LIB, o tipo abstrato de dados Árvore AVL, com alocação encadeada dinâmica e tendo como base o tipo inteiro. A biblioteca deve conter, além da estrutura de dados do TAD, as seguintes operações:

    a) Criar uma árvore AVL vazia

    b) Verificar se uma árvore está vazia ou não

    c) Inserir um novo elemento

    d) Remover um elemento, dado o seu valor

    e) Verificar se a uma árvore contém um dado valor, retornando o endereço do nó, caso encontrado, ou nil, caso contrário.

    f) Exibir todos os valores de uma árvore

    g) Esvaziar uma árvore

2. Faça um programa que, utilizando a biblioteca AVL.LIB, faça o seguinte:

    a) Crie uma árvore AVL T.

    b) Exiba o seguinte menu de opções:

```
EDITOR DE ÁRVORE AVL

1 – INSERIR
2 – REMOVER
3 – PESQUISAR
4 – EXIBIR A ÁRVORE
5 – ESVAZIAR A ÁRVORE

DIGITE SUA OPÇÃO:
```

    c) Leia a opção do usuário.

    d) Execute a opção escolhida pelo usuário.

    e) Após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

# 11 – INDEXAÇÃO

Para redução do tempo de acesso a um registro o mais indicado é o uso de índice para orientar a busca. Por este motivo, um dos importantes componentes de um arquivo é o diretório que abriga uma coleção de um ou mais índices do arquivo.

Um índice é um mapeamento que associa a cada chave, uma referência ao registro que a contém. Assim, dada uma chave de pesquisa, o índice fornece imediatamente a localização do registro correspondente. Resumindo, índice é um par (chave, endereço).

Quando trabalhamos com chave primária, seus valores terão sempre valores únicos e não nulos de forma a identificar unicamente cada registro. Porém, é comum a existência de valores chaves (não chaves primárias, mas simplesmente chaves) que se repetem, como por exemplo, num arquivo de funcionários, a chave função certamente terá repetições. Mesmo com repetições de valores de chaves o uso de índices não será eliminado, será apenas manipulado de maneira mais complexa.

Um arquivo que utiliza a estrutura de índices é composto, basicamente, por duas partes: um arquivo de dados e uma estrutura de índice associada. A cada inserção de um novo registro no arquivo de dados, é inserido no índice uma referência a ele. Para acessar um determinado registro, para consulta ou alteração, é feita a pesquisa na estrutura de índice para que sua localização no disco seja encontrada e, em seguida, é feito o acesso direto para leitura ou gravação.

**Arquivo de Dados**

| | CHAVE | REGISTRO |
|---|---|---|
| 0 | João | ... |
| 1 | Maria | ... |
| 2 | Antonio | ... |
| 3 | Rosa | ... |
| 4 | Carlos | ... |
| 5 | Beto | ... |
| 6 | Francisco | ... |

**Estrutura de Índice**

| CHAVE | ENDEREÇO |
|---|---|
| Antonio | 2 |
| Beto | 5 |
| Carlos | 4 |
| Francisco | 6 |
| João | 0 |
| Maria | 1 |
| Rosa | 3 |

O objetivo é melhorar o desempenho no processo de inclusão de novos registros, pesquisa do arquivo e atualização do arquivo.

Não havendo movimentação dentro do arquivo, a eficiência na inclusão pode ser facilmente obtida se os registros forem inseridos sempre no final do arquivo. Porém, a pesquisa será extremamente lenta, uma vez que os registros não estão armazenados ordenados e a busca será realizada através de uma leitura seqüencial dos registros. Com o objetivo de garantir rapidez em situações como esta, a utilização de uma estrutura de índice reduziria bastante o número de acessos a disco já que a chave procurada seria armazenada nesta estrutura que seria mantida temporariamente na memória principal.

Considerando a existência de índice, o processo de atualização será rápido. Fornecida a chave do registro do registro a ser alterado, seu acesso será feito rapidamente, bem como a gravação das novas informações.

No caso de remoção, ela poderá ser feita simplesmente eliminando sua referência ao índice, fazendo apenas a remoção lógica. A exclusão física ocorrerá apenas no processo de reorganização do arquivo, quando uma cópia dele será gerada, contendo apenas os registros ainda referenciados pelo índice. A implementação de um sistema que utiliza estruturas de índice, permitindo exclusões lógicas e físicas, pode também ser feita acrescentado-se um campo lógico em cada nodo da árvore. Assim, ao invés de remover o nodo da árvore quando o registro for excluído, simplesmente será feita a atualização deste campo para indicar a exclusão. A estrutura de índice deverá ser mantida em disco para que o sistema reconheça os registros excluídos.

A velocidade, a complexidade e o uso eficiente da memória variam muito de acordo com a estrutura interna implementada de um índice. Como já foi visto anteriormente, um índice pode ser uma simples tabela, mas pode se apresentar também em uma das formas de árvore e, especialmente, se apresentada na forma de uma árvore de busca binária para arquivos de porte médio aliará a simplicidade da implementação a um desempenho satisfatório. Para arquivos maiores, outros tipos de implementações baseados em árvores poderão ser mais indicados.

Quando a implementação é feita utilizando-se índices baseados em árvores, todos os procedimentos de inclusão, exclusão, alteração e busca ao índice são similares aos estudados em árvores.

As implementações de estruturas de índices podem ser desenvolvidas de duas formas: a estrutura é recriada toda vez que o arquivo de dados é aberto ou a estrutura também é mantida em disco de modo a tornar o processo de carga mais rápido.

## EXERCÍCIOS:

**1.** Considerando o arquivo de dados abaixo, desenhe uma estrutura de índice associada ao mesmo.

**Arquivo de Dados**

| | CHAVE | REGISTRO |
|---|---|---|
| 0 | 3487 | ... |
| 1 | 2166 | ... |
| 2 | 9842 | ... |
| 3 | 3660 | ... |
| 4 | 5045 | ... |
| 5 | 7124 | ... |
| 6 | 3535 | ... |

**Estrutura de Índice**

**2.** Utilizando a técnica de indexação, faça um programa na linguagem C que crie e manipule um arquivo (binário) de clientes, cujo registro possui a seguinte estrutura:

```
typedef struct {
   int cod;   // código do cliente
   char nome[31]; // nome do cliente
   char fone[10]; // telefone do cliente
} treg;
```

O programa deve exibir um menu com as seguintes operações: (1) incluir um novo cliente; (2) alterar os dados de um cliente (exceto o seu código); (3) excluir um cliente; (4) listar os dados de um cliente, dado o seu código; (5) listar todos os clientes (por ordem de código); (6) excluir todos os registros do arquivo.

Deve ser criado um arquivo de índice (indexado pelo campo cod) organizado como uma ABP, sendo este arquivo de índice armazenado em memória interna e recriado a cada abertura do arquivo de dados.

O programa deve, antes do seu encerramento, reorganizar o arquivo de dados com base nos registros referenciados pelo índice. Obs: mantenha a ordenação física do arquivo original.

# 12 – HASHING

Nas estruturas de dados vistas até agora, a busca por uma informação armazenada era feita com base na comparação de chaves, ou seja, dada uma chave usava-se os mecanismos de acesso para encontrar a mesma chave armazenada.

Veremos agora um novo método de busca por chaves que tem como maior vantagem o fato de que, na média dos casos, é possível encontrar a chave com apenas uma operação de leitura.

A idéia geral do método é gerar, a partir da chave procurada, o endereço da entrada de uma tabela onde se encontra a informação associada à chave. Este método de transformação de chaves é conhecido na literatura com hashing (espalhamento) e a tabela é chamada tabela de dispersão.

A construção da tabela também é feita com base em hashing, ou seja, a partir da chave é gerado um endereço de uma entrada que deverá ser ocupada na tabela. Como pode acontecer de que o mesmo endereço seja gerado a partir de mais de uma chave, são necessários mecanismos para tratar estas situações chamadas de colisões.

## Funções de Transformação de Chaves

Seja uma tabela de dispersão com *M* entradas. Uma função de transformação deve gerar para cada elemento de um conjunto de chaves um valor entre *0* e *M-1* correspondente a um endereço na tabela.

Como a transformação da chave é uma operação aritmética, é preciso, no caso da chave ser composta de caracteres, gerar um valor numérico equivalente através da soma dos valores na tabela ASCII dos caracteres que compõe a chave.

A seguir, *K* é usado para gerar o endereço para a tabela através de uma função de transformação *h(K)*. É esperado que *h(K)* seja uma função que produza um baixo número de colisões ao mesmo tempo em que tenha um bom grau de "espalhamento", ou seja que os endereços sejam uniformemente gerados. Além disso, é importante que a função seja simples de calcular.

Embora o conhecimento do conjunto de chaves possa ajudar bastante na escolha da função, vários autores indicam que uma boa função quando não se tem esse conhecimento é aquela que calcula o resto da divisão de *K* por *M* (**método da divisão**):

$$h(K) = K \% M$$

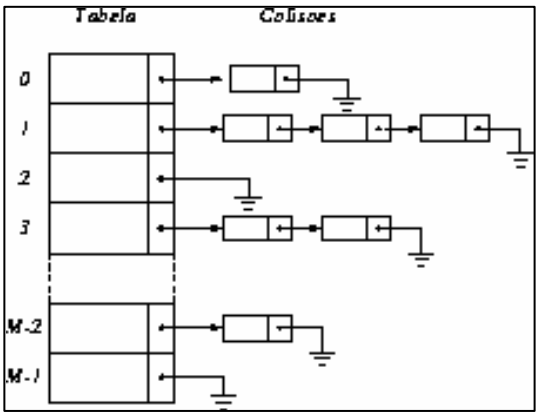
## TRATAMENTO DE COLISÕES

Como já mencionamos é muito comum o caso em que um mesmo endereço seja gerado para mais de uma chave. Isto ocorre basicamente porque em geral o número N de chaves possíveis é muito maior que o número de entradas disponíveis M e também porque não se pode garantir que as funções hashing em geral possuem um bom potencial de espalhamento.

As técnicas de tratamento de colisão são basicamente de dois tipos: encadeamento e endereçamento aberto.

## Encadeamento

Neste tipo tratamento de colisões (figura ao lado), as entradas da tabela de dispersão possuem um apontador para uma lista encadeada cujos elementos armazenam as chaves que geram o endereço da entrada (e eventualmente a informação associada).




Também é possível implementar tratamento de hashing por encadeamento sem usar alocação dinâmica de memória. Para isso, acrescenta-se a cada entrada da tabela um campo que indica a próxima entrada a ser pesquisada se a entrada não contiver a chave que deveria conter e que foi deslocada por causa de uma colisão. Em alguns casos estas entradas extras se encontram na área separada da tabela chamada área de *overflow*.

## Endereçamento Aberto (ReHash)

Para evitar o uso de recursos adicionais na construção da estrutura de dados, a estratégia de endereçamento aberto simplesmente aplica outra função de transformação de chave quando houver uma colisão causada pela função original *h(K)*.

Assim, se há uma colisão causada por *h(k)* usa-se uma outra função *rh(h(K))* que determina outra entrada onde a chave deve ser inserida. Se houver outra colisão usa-se *rh(rh(h(K)))* e assim por diante. Se não é encontrada nenhuma posição vazia é porque a tabela já está cheia e não podem ser incluídos novos elementos. A busca da chave segue a mesma estratégia.

Uma das alternativas mais usadas para a geração das entradas alternativas é chamada **hashing linear**, onde a posição na tabela é dada por:

***rh(i) = ( i + 1 ) % M***

Suponha por exemplo que desejamos inserir, na ordem em que aparecem de cima para baixo, as seguintes chaves em uma tabela com M=7:

| **chave** | **K = ord(chave)** | **$i_1$ = h(K)** | **$i_2$ = rh($i_1$)** | **$i_3$ = rh($i_2$)** | **$i_4$ = rh($i_3$)** |
|---|---|---|---|---|---|
| C | 67 | 4 | - | - | - |
| H | 72 | 2 | - | - | - |
| A | 65 | 2 | 3 | - | - |
| V | 86 | 2 | 3 | 4 | 5 |
| E | 69 | 6 | - | - | - |
| S | 83 | 6 | 0 | - | - |

A figura a seguir mostra a distribuição destas chaves na tabela de dispersão.

***Tabela***

| | |
|---|---|
| 0 | S |
| 1 | |
| 2 | H |
| 3 | A |
| 4 | C |
| 5 | V |
| 6 | E |

Contudo, há um problema gerado com o espalhamento linear: quando há um conjunto com muitas entradas contíguas preenchidas, este apresenta mais chances de promover uma inserção que outro com entradas mais dispersas. Este fenômeno em que duas chaves espalhadas em dois valores diferentes competem entre si em sucessivos reespalhamentos é chamado de agrupamento primário.

Existem vários métodos que resolvem o problema do agrupamento primário. Contudo, eles não eliminam um outro fenômeno, conhecido como agrupamento secundário, no qual chaves diferentes que espalham o mesmo valor seguem o mesmo percurso de reespalhamento. Uma maneira de eliminar todo esse agrupamento é o espalhamento duplo, que detalharemos a seguir.

## Espalhamento Duplo

O espalhamento duplo aparece como solução aos problemas do agrupamento primário e agrupamento secundário. Para usá-lo precisamos de duas funções, isso permite que os resultados da função hash se aproximem de permutações aleatórias. Veja a seguir as funções hash definidas para atender aos requisitos do espalhamento duplo.

Para o primeiro cálculo:

***h(K) = K % M***

Caso haja colisão, inicialmente calculamos h2(K), que pode ser definida como:

$$h_2(K) = 1 + ( K \% (M-1) )$$

Em seguida calculamos a função rehash como sendo:

$$rh(i,K) = ( i + h_2(K) ) \% M$$

Veja o seguinte exemplo: suponha uma tabela com 10 entradas (M = 10), todas inicialmente vazias. A inserção do valor 35 será feita na posição 5, pois:

***h(35) = 35 % 10 = 5***

A inserção, em seguida, do valor 65 será calculada da seguinte maneira:

***h(65) = 65 % 10 = 5 (colisão)***

$$h_2(65) = 1 + ( 65 \% (10-1) ) = 3$$

***rh(5,65) = (5 + 3) % 10 = 8***

Portanto, 65 será inserido na posição 8 da tabela.

## EXERCÍCIOS:

**1.** Ilustre a organização final de uma tabela hash após a inserção das seguintes chaves: 35, 99, 27, 18, 65, 45. Considere a tabela com tamanho 6 (posições 0 a 5), o *método da divisão* como função hash e tratamento de colisão por *encadeamento*.

**2.** Idem à questão anterior, porém o tratamento de colisão será por *endereçamento aberto*, usando a técnica de *hashing linear*.

**3.** Idem à questão anterior, porém usando a técnica de *hashing duplo*.

**4.** Utilizando hashing, faça um programa na linguagem C que crie e manipule um arquivo de clientes, cujo registro possui a seguinte estrutura:

```
typedef struct {
   int cod;   // código do cliente
   char nome[31]; // nome do cliente
   char fone[10]; // telefone do cliente
} treg;
```

O programa deve realizar as seguintes operações: (1) incluir um novo cliente; (2) alterar os dados de um cliente (exceto o seu código); (3) excluir um cliente; (4) listar os dados de um cliente, dado o seu código (5) listar todo o arquivo (seqüencialmente, mostrando a ordem de entrada de cada registro); (6) excluir todos os registros do arquivo.

O arquivo deve conter 20 entradas (insira inicialmente 20 registros vazios no arquivo). Utilize o *método da divisão* como função hash e tratamento de colisão por *endereçamento aberto*, usando a técnica de *hashing duplo*.

**5.** Faça uma nova versão do programa do item anterior, porém utilizando tratamento de colisão *por encadeamento*, utilizando o próprio arquivo como "área de overflow".

# 13 – ÁRVORE-B

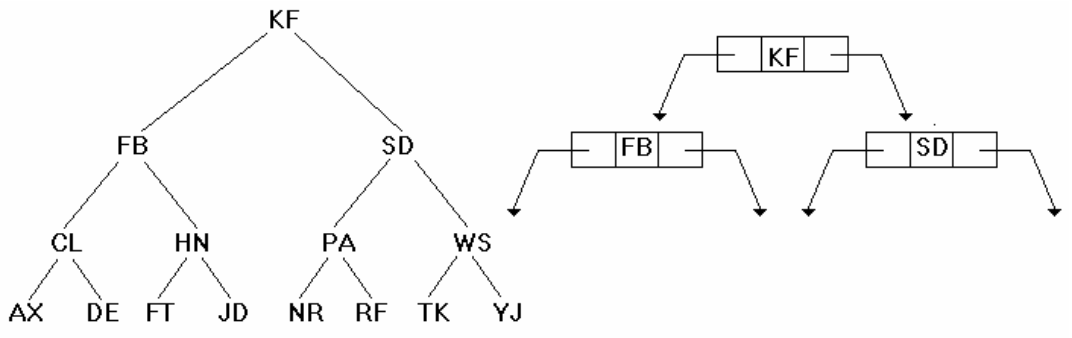
Bayer and McGreight, em 1972, publicaram o artigo: *"Organization and Maintenance of Large Ordered Indexes"*. Em 1979, o uso de árvores-B já era praticamente o padrão adotado em sistemas de arquivos de propósito geral para a manutenção de índices para bases de dados.

O problema fundamental associado à manutenção de um índice em disco é que o acesso é muito lento, o que coloca problemas para uma manutenção eficiente. O melhor acesso a um índice ordenado, até agora, foi dado pela Pesquisa Binária, porém:

- Pesquisa binária requer muitos acessos. 15 itens podem requerer 4 acessos, 1000 itens podem requerer até 11 acessos. Esses números são muito altos.
- Pode ficar muito caro manter um índice ordenado de forma a permitir busca binária. É necessário um método no qual a inserção e a eliminação de registros tenha apenas efeitos locais, isto é, não exija a reorganização total do índice.

## Solução através de árvores binárias de pesquisa (ABP)

Os registros são mantidos num arquivo, e ponteiros (**esq** e **dir**) indicam onde estão os registros filhos. Esta estrutura pode ser mantida em memória secundária: os ponteiros para os filhos dariam o número dos registros correspondentes aos filhos, i.e., a sua localização no arquivo.




Raiz = 9

| | key | filho esq. | filho dir. |
|---|---|---|---|
| 0 | FB | 10 | 8 |
| 1 | JD | | |
| 2 | RF | | |
| 3 | SD | 6 | 15 |
| 4 | AX | | |
| 5 | YJ | | |
| 6 | PA | 11 | 2 |
| 7 | FT | | |

| | key | filho esq. | filho dir. |
|---|---|---|---|
| 8 | HN | 7 | 1 |
| 9 | KF | 0 | 3 |
| 10 | CL | 4 | 12 |
| 11 | NR | | |
| 12 | DE | | |
| 13 | WS | 14 | 5 |
| 14 | TK | | |

- Vantagem: a ordem lógica dos registros não está associada à ordem física no arquivo.
- O arquivo físico do índice não precisa mais ser mantido ordenado, uma vez que a seqüência física dos registros no arquivo é irrelevante: o que interessa é poder recuperar a estrutura lógica da árvore, o que é feito através dos campos **esq** e **dir**.

Se acrescentarmos uma nova chave ao arquivo, por exemplo, LV, é necessário apenas saber aonde inserir esta chave na árvore, de modo a mantê-la como ABP. A busca pelo registro é necessária, mas a reorganização do arquivo não é.

## Solução por árvores AVL

A eficiência do uso de árvores binárias de pesquisa exige que estas sejam mantidas balanceadas. Isso implica no uso de árvores AVL, e dos algoritmos associados para inserção e eliminação de registros.

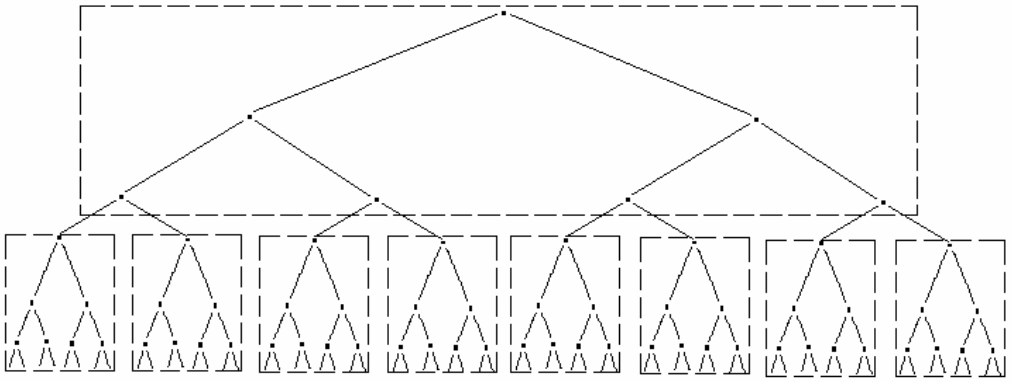
Para uma árvore binária completamente balanceada, o número máximo de comparações para localizar uma chave em uma árvore com N chaves é igual à altura da árvore, dada por log2 (N+1). Para uma árvore AVL, esse número é 1.44*log2 (N+2). Portanto, dadas 1.000.000 de chaves, em uma árvore completamente balanceada uma busca iria percorrer até 20 níveis da árvore. Para uma árvore AVL, a busca poderia percorrer até 28 níveis.

Entretanto, se as chaves estão em memória secundária, qualquer procedimento que exija mais do que 5 ou 6 acessos para localizar uma chave é altamente indesejável: 20 ou 28 *seeks* são igualmente inaceitáveis. Assim, árvores balanceadas são uma boa alternativa se considerarmos o problema da ordenação, pois não requerem a ordenação do índice e sua reorganização sempre que houver nova inserção. Por outro lado, as soluções vistas até agora não resolvem o problema no número excessivo de acessos a disco.

## Solução por Árvores Binárias Paginadas (*Paged Binary Trees*)

Ou seja, o velho problema está de volta: a busca por uma posição específica do disco é muito lenta. Por outro lado, uma vez encontrada a posição, pode-se ler uma grande quantidade registros seqüencialmente a um custo relativamente pequeno.

Esta combinação de busca (*seek*) lenta e transferência rápida sugere a noção de **página**: em um sistema "paginado", você não incorre no custo de fazer um "*seek*" para recuperar apenas alguns bytes: ao invés disso, uma vez realizado um *seek*, que consome um tempo considerável, toda os registros em uma mesma "página" do arquivo são lidos. Esta página pode conter um número bastante grande de registros, e se por acaso o próximo registro a ser recuperado estiver na mesma página, você economizou um acesso a disco.



A divisão de uma árvore binária em páginas é ilustrada na figura acima. Nessa árvore de 9 páginas, quaisquer dos 63 registros podem ser acessados em, no máximo, 2 acessos. Se a árvore é estendida com um nível de paginação adicional, adicionamos 64 novas páginas, e poderemos encontrar qualquer um das 511 chaves armazenadas com apenas 3 *seeks* (quantos seeks seriam necessários em uma busca binária?)

Se cada página dessa árvore ocupar 8KB, permitindo a armazenagem de 511 pares chave-referência; e cada página contiver uma árvore completa perfeitamente balanceada, então a árvore toda pode armazenar um total de 134.217.727 chaves, sendo que qualquer delas pode ser acessada em, no máximo, 3 acessos ao disco.

Pior caso para: árvore binária completa, perfeitamente balanceada: $\log_2 (N+1)$ versão em páginas: $\log_k+1 (N+1)$ em que N é o número total de chaves, e k é o número de chaves armazenadas em uma página. Note que, para árvores binárias, k = 2. Observe o efeito logarítmico do tamanho da página para k = 2 e k = 511:

- árvore binária: log2 (134.217.727) = 27 acessos
- versão em páginas: log511+1 (134.217.727) = 3 acessos

Preços a pagar:

- maior tempo na transmissão de grandes quantidades de dados, e, mais sério,
- a necessidade de manutenção da organização da árvore.

## Problema da construção *Top-Down* de árvores paginadas

Construir uma árvore paginada é relativamente simples se temos todo o conjunto de chaves antes de iniciar a construção, pois sabemos que temos de começar com a chave do meio para garantir que o conjunto de chaves seja dividido de forma balanceada.

Entretanto, a situação se complica se estamos recebendo as chaves em uma seqüência aleatória e construindo a árvore a medida em que as chaves chegam. Provavelmente a árvore ficará desbalanceada. E então, o que fazer? Não dá para rotacionar páginas como rotacionamos chaves individuais!

## Construção *Bottom-Up*: Árvores-B

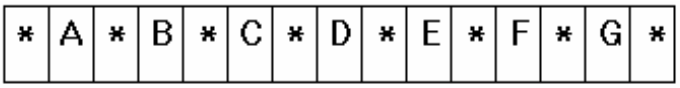
Bayer e McCreight, no artigo mencionado anteriormente, propuseram que as árvores fossem construídas de baixo para cima. Desta forma, as chaves raiz da árvore emergem naturalmente. Uma idéia elegante e poderosa.

Numa árvore-B:

- Cada página é formada por uma seqüência ordenada de chaves e um conjunto de ponteiros.
- Não existe uma árvore explícita dentro de uma página (ou nó).
- O número de ponteiros em um nó excede o número de chaves em 1.
- O número máximo de ponteiros que podem ser armazenados em um nó é a **ordem** da árvore.
- O número máximo de ponteiros é igual ao número máximo de descendentes de um nó. Exemplo: uma árvore-B de ordem 8 possui nós com, no máximo, 7 chaves e 8 filhos.
- Os nós folha não possuem filhos, e seus ponteiros são nulos.

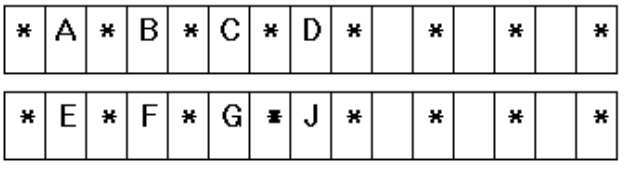
## Inserção em Árvore-B

Seja a seguinte página inicial de uma árvore-B de ordem 8, que armazena 7 chaves.



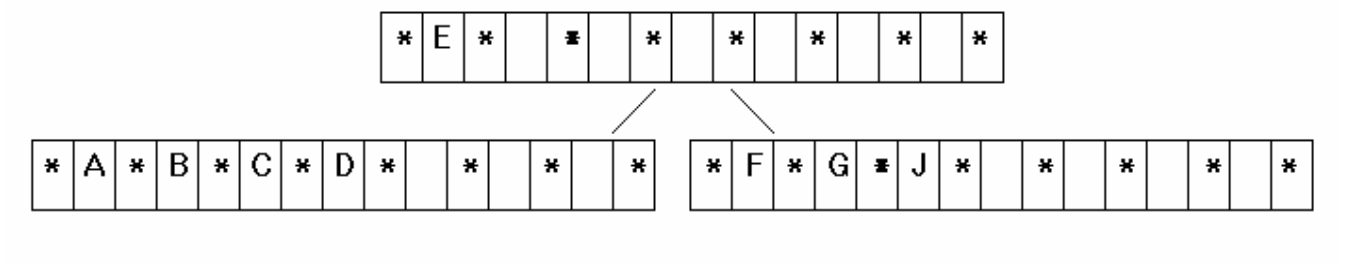

Observe que, em uma situação real, além das chaves e ponteiros armazena-se outras informações associadas às chaves, como uma referência para a posição do registro associado à chave em um arquivo de dados. Esta folha que, coincidentemente, é também a raiz da árvore, está cheia. Como inserir uma nova chave, digamos J?

Subdividimos (***split***) o nó folha em dois nós folhas, distribuindo as chaves igualmente entre os nós.




Temos agora duas folhas: precisamos criar uma nova raiz. Fazemos isso "promovendo" (***promote***), ou "subindo" uma das chaves que estão nos limites de separação das folhas.

Nesse caso, "promovemos" a chave E para a raiz:



**Em resumo:**

Para inserirmos um novo elemento em uma árvore-B, basta localizar o nó folha X onde o novo elemento deva ser inserido.

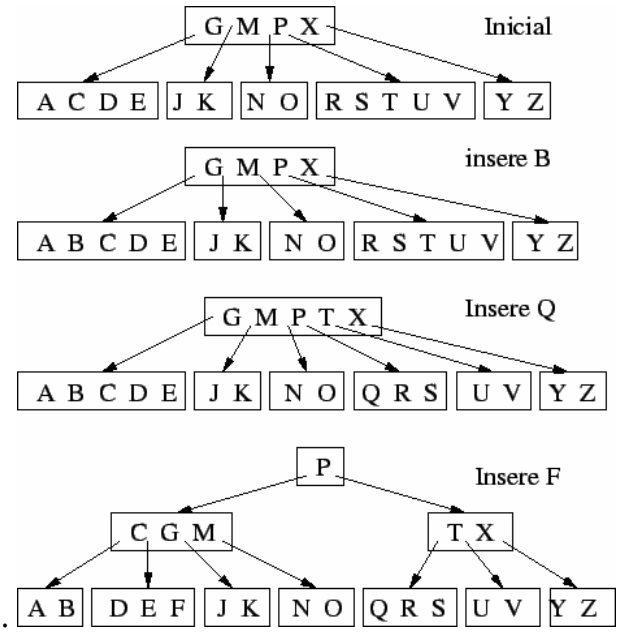
Se o nó X estiver cheio, precisamos realizar uma subdivisão de nós, que consiste em passar o elemento mediano de X para seu pai e subdividir X em dois novos nós com *round* (m/2) - 1 elementos e depois inserir a nova chave.

Se o pai de X também estiver cheio, repetimos recursivamente a subdivisão acima para o pai de X.

No pior caso, teremos que aumentar a altura da árvore-B para podermos inserir o novo elemento.

Note que, diferentemente das árvores binárias, as árvores-B crescem para cima.

Veja, a seguir, um exemplo de inclusão de novos elementos em uma árvore-B de ordem 6:




## Definição e Propriedades da Árvore-B

Segundo vimos, a **ordem** de uma árvore-B é dada pelo número máximo de descendentes que uma página, ou nó, pode possuir. A definição apresentada vincula a ordem de uma árvore-B ao número de descendentes de um nó (isto é, de ponteiros). Deste modo, numa árvore-B de ordem **m**, o número máximo de chaves em uma página é **m-1**.

Para uma árvore-B de ordem **m**:

1. cada página tem no máximo m descendentes e m-1 chaves.
2. cada página, exceto a raiz e as folhas, tem um mínimo de *round* (m/2) filhos.
3. cada página, exceto a raiz, tem um mínimo de *round* (m/2) - 1 chaves.
4. todas as folhas aparecem num mesmo nível (o nível mais baixo).
5. uma página que não é folha e possui k filhos, e contém k-1 chaves.

## Remoção em Árvore-B

O processo de sub-divisão (*split*) de páginas garante a manutenção das propriedades da árvore-B durante a inserção. Essas propriedades precisam ser mantidas, também, durante a eliminação de chaves.

**Caso 1:** eliminação de uma chave em uma página folha, sendo que o número mínimo de chaves na página é respeitado. Solução: a chave é retirada e os registros internos à página são reorganizados.

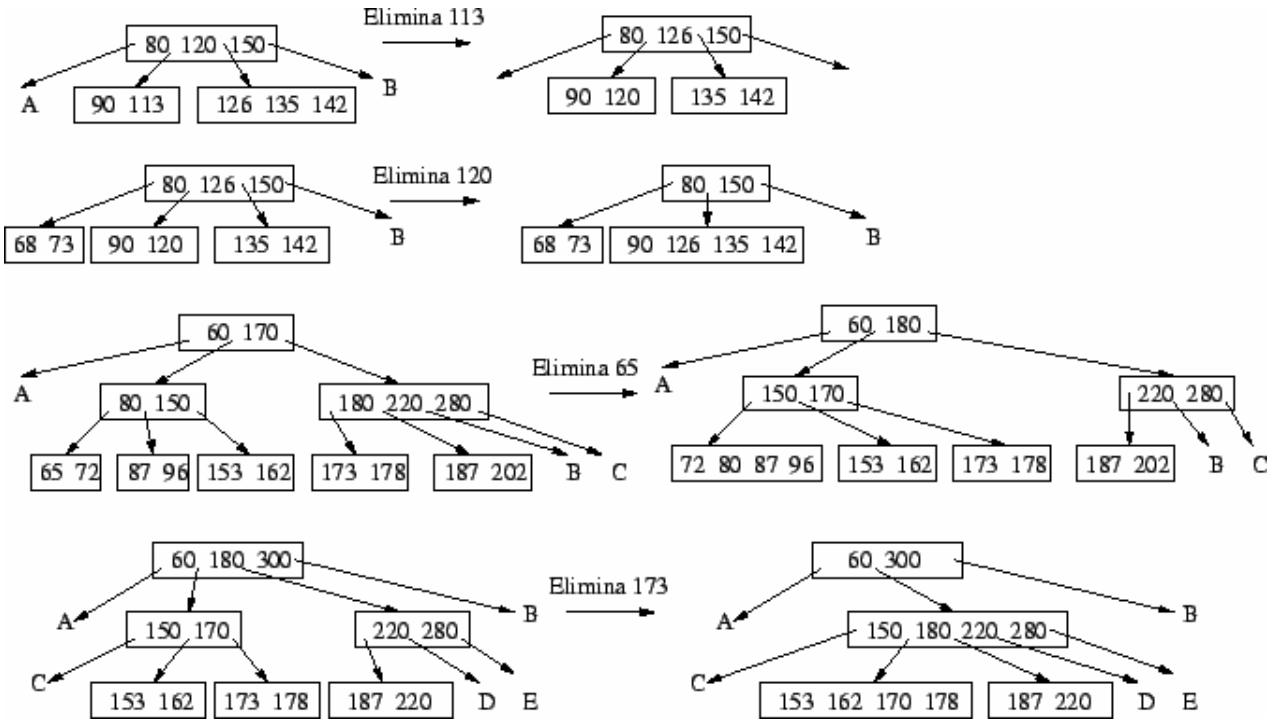
**Caso 2:** eliminação de uma chave que não está numa folha. Solução: sempre eliminamos de páginas folha. Se uma chave deve ser eliminada de uma página que não é folha, trocamos a chave com sua sucessora imediata, que com certeza está numa folha, e então eliminamos a chave da folha.

**Caso 3:** eliminação causa underflow na página. O número mínimo de chaves por página nessa árvore é *round*(m/2)-1. Solução: **redistribuição**. Procura-se uma página irmã (mesmo pai) que contenha mais chaves que o mínimo: se existir redistribui-se as chaves entre essas páginas. A redistribuição pode provocar uma alteração na chave separadora que está no nó pai.

**Caso 4:** ocorre underflow e a redistribuição não pode ser aplicada. Não existem chaves suficientes para dividir entre as duas páginas irmãs. Solução: deve-se utilizar **concatenação**, que consiste na combinação do conteúdo das duas páginas e a chave da página pai para formar uma única página. A concatenação é o inverso do processo de subdivisão. Como conseqüência, pode ocorrer o underflow da página pai.

**Caso 5:** underflow da página pai. Solução: a redistribuição não pode ser aplicada (porquê?). Deve-se utilizar concatenação novamente. Pode levar à diminuição da altura da árvore.

Como exemplo, veja a figura a seguir.




## Redistribuição durante Inserção

Diferentemente da divisão e da concatenação, o efeito da redistribuição é local. Não existe propagação. Outra diferença é que não existe uma regra fixa para o rearranjo das chaves. A eliminação de uma única chave em uma árvore-B não pode provocar o underflow de mais de uma chave. Portanto, a redistribuição pode restabelecer as propriedades da árvore-B movendo apenas uma chave de uma página irmã para a página com problema, ainda que a distribuição de chaves entre as páginas fique muito desigual. A estratégia usual é redistribuir as chaves igualmente entre as páginas.

Exemplo: dada uma árvore-B de ordem 101, os números mínimo e máximo de chaves são, respectivamente, 50 e 100. Se ocorre underflow de uma página, e a página irmã tem 100 chaves, qualquer número de chaves entre 1 e 50 pode ser transferido. Normalmente transfere-se 25, deixando as páginas equilibradas.

A redistribuição não foi comentada quando explicamos o processo de inserção. Entretanto, seria uma opção desejável também na inserção. Ao invés de dividir uma página cheia em duas páginas novas semivazias, pode-se optar por colocar a chave que sobra (ou mais que uma!) em outra página. Essa estratégia resulta em uma melhor utilização do espaço alocado para a árvore.

Depois da divisão de uma página, cada página fica 50% vazia. Portanto, a utilização do espaço, no pior caso, em uma árvore-B que utiliza *splitting*, é de cerca de 50%. Em média, para árvores grandes, foi provado que o índice é de 69%.

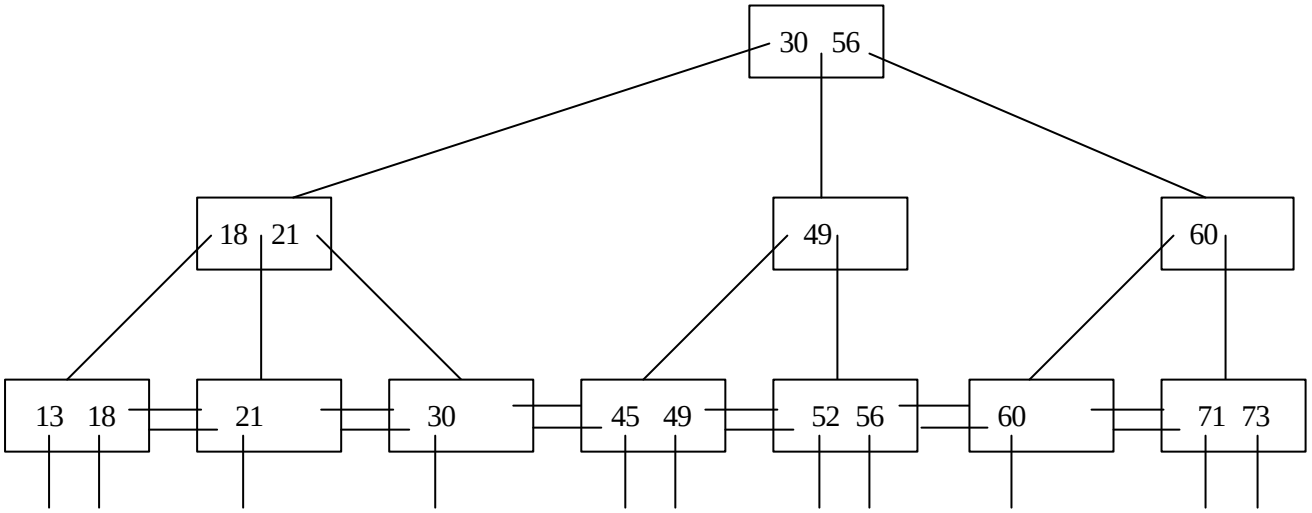
Estudos empíricos indicam que a utilização de redistribuição pode elevar o índice de 67% para 85%. Esses resultados sugerem que qualquer aplicação séria de árvore-B deve utilizar, de fato, redistribuição durante a inserção.

## 13.1 – ÁRVORES-B+

Uma das maiores deficiências da árvore-B é a dificuldade de percorrer as chaves seqüencialmente. Uma variação da estrutura básica da árvore-B é a árvore-$B^+$.

Nesta estrutura, todas as chaves são mantidas em folhas, e algumas chaves são repetidas em nós não-folha para definir caminhos para localizar registros individuais.

As folhas são ligadas através de uma lista duplamente encadeada, de modo a oferecer um caminho seqüencial para percorrer as chaves na árvore. Esta lista é chamada de Conjunto de Seqüência (Sequence Set).




É importante esta separação lógica da árvore-$B^+$ em Conjunto de Índices (Index Set) e Conjunto de Seqüência, pois podemos fazer a maioria das inclusões e exclusões no conjunto de seqüência sem alterar o índice (árvore-B). Quando houver necessidade de inclusão ou exclusão do referido índice, usamos os algoritmos já conhecidos da árvore-B.

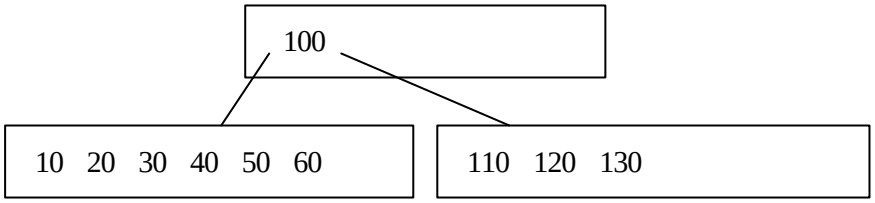
A inserção numa árvore-$B^+$ ocorre praticamente da mesma forma que numa árvore-B, exceto pelo fato de que, quando um nó é dividido, a chave do meio é retida no meio nó esquerdo, além de ser promovida a pai. Quando uma chave é eliminada de uma folha, ela pode ser retida nas não-folhas porque ela ainda é um separador válido entre as chaves nos nós abaixo dela.

A árvore-$B^+$ mantém o baixo custo das operações de pesquisa, inclusão e exclusão e adquire a vantagem de requerer no máximo um acesso para satisfazer a operação da próxima chave. Além disso, para um processamento seqüencial nenhum nodo precisará ser acessado mais de uma vez como acontece no caminhamento *in-order* que precisamos fazer numa árvore-B. Portanto, enquanto a eficiência de localização do registro seguinte em um árvore-B é *O(log n)*, numa árvore-$B^+$ é aumentada para *O(1)*.

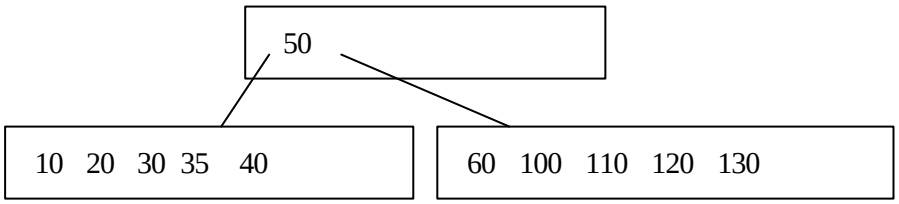
A árvore-B+ é ideal para aplicações que requerem tanto acesso seqüencial quanto aleatório. Por isso e pelas vantagens citadas anteriormente, a árvore-B+ tornou-se bastante popular nos SGBDs comerciais.

## 13.2 – ÁRVORES-B*

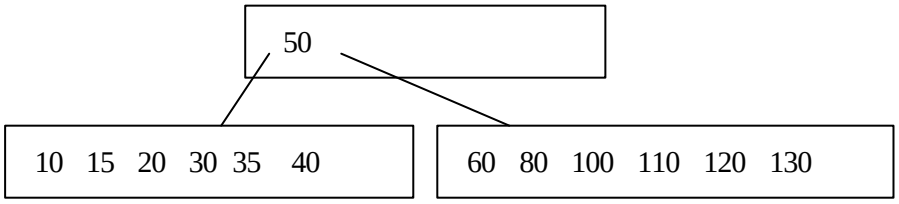
Existem várias formas de aprimorar a utilização do armazenamento de uma árvore-B. Um método é retardar a divisão de um nó no caso de encher. Em substituição, as chaves no nó e um de seus irmãos adjacentes, bem como a chave no pai que separa entre os dois nós, são redistribuídas uniformemente. Esse processo é ilustrado na figura abaixo, numa árvore-B de ordem 7.



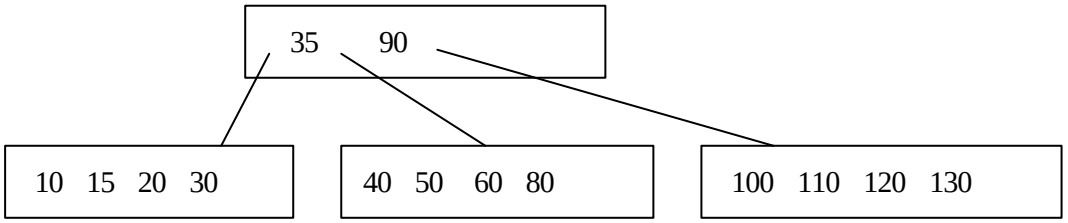

(a) Árvore-B original de ordem 7.




(b) Após a inserção da chave 35 e redistribuição dos irmãos.




(c) Após a inserção das chaves 15 e 80.




(d) Após a inserção da chave 90 e redistribuição de pai e irmãos.

Quando um nó e seu irmão estiverem completos, os dois nós serão divididos em três (two-to-three splitting). Isso assegurará uma utilização mínima do armazenamento de quase 67%, contra 50% de uma árvore-B tradicional. Na prática, a utilização do armazenamento será ainda mais alta. Esse tipo de árvore é chamado de árvore-B*.

Essa técnica pode ser ainda mais ampliada redistribuindo as chaves entre todos os irmãos e o pai de um nó completo. Infelizmente, este método impõe um preço porque exige espaços adicionais dispendiosos durante as inserções, enquanto a utilização do espaço adicional alcançado pela consideração de cada irmão extra torna-se cada vez menor.

## EXERCÍCIOS:

**1.** Por que B-Trees são consideradas geralmente superiores que as árvores binárias de busca para pesquisa externa, e árvores binárias são comumente usadas para pesquisa interna?

**2.** Como uma folha de uma árvore-B difere de um nó interno?

**3.** Suponha que você vai deletar uma chave em uma árvore B, a qual causa um underflow na página. Se pela página irmã do lado direito é necessária concatenação, e pela página esquerda é possível redistribuição, qual opção você escolheria? Por quê?

**4.** Mostre a árvore-B de ordem 3 resultante da inserção do seguinte conjunto de chaves: N S U O A C G J X

**5.** Dada uma árvore B de ordem 128, responda:

a) Qual o número máximo de descendentes por página?

b) Qual o número mínimo de descendentes por página (desconsiderando as folhas e a raiz)?

c) Quantas chaves têm uma página não-folha com 100 descendentes?

d) Quantos descendentes têm uma página folha com 100 chaves?

**6.** Diferencie a árvore B+ da árvore B padrão, enfatizando vantagens e desvantagens.

**7.** Diferencie a árvore B* da árvore B padrão, enfatizando vantagens e desvantagens.

**8.** Implemente em C um "Editor de Árvore B". Este programa deve criar e manipular uma árvore B, de ordem 5, armazenada em disco. O tipo base das chaves será o inteiro. Exiba o seguinte menu de opções:

```
EDITOR DE ÁRVORE B

1 – INSERIR
2 – REMOVER
3 – PESQUISAR
4 – EXIBIR A ÁRVORE
5 – ESVAZIAR A ÁRVORE

DIGITE SUA OPÇÃO:
```

Obs: a organização interna de cada página pode ser implementada como vetor ou ABP, a sua escolha.

# 14 – CLASSIFICAÇÃO DE DADOS

No nosso dia-a-dia nem nos damos conta da importância da classificação das informações na hora de fazermos uma busca. Imagine se você fosse fazer uma pesquisa no catálogo telefônico ou em um dicionário se os dados não seguissem algum tipo de classificação. Daí, notamos a grande importância da classificação para localizarmos uma determinada informação.

Classificação de dados é o processo pelo qual é determinada a ordem que as entradas de uma tabela (os registro de um arquivo) serão apresentadas de forma que obedeça a uma organização previamente determinada por um ou mais campos. O campo pelo qual o arquivo é ordenado é chamado de chave de classificação.

O ambiente de classificação é determinado pelo meio onde estão os dados a serem classificados. Uma classificação é considerada interna se os registros que ela classificar estiverem na memória principal, e externa se alguns dos registros que ela classificar estiverem no armazenamento auxiliar.

A classificação intema apresenta como principais características classificar apenas o que se encontra na memória principal, gastar o mesmo tempo de acesso para buscar qualquer dos endereços e o tempo médio de acesso não é afetado pela seqüência dos dados.

Na classificação externa, os dados são transferidos em blocos para a memória principal para, só então, serem manipulados. Esses dados transferidos num acesso a disco influenciarão na eficiência do processamento e os dados são manipulados bloco a bloco.

É importante notar que os métodos de classificação externa envolvem a aplicação de métodos de classificação interna, tomando a cada vez um subconjunto de dados a classificar. Por isso, nos deteremos ao estudo dos métodos de Classificação Interna.

Há diferentes formas de apresentação do resultado de classificação, as três formas clássicas são: contiguidade física (as entradas são fisicamente rearranjadas, de forma que no final da classificação a ordem lógica é igual a ordem física dos dados), vetor indireto de ordenação – VIO (as entradas são mantidas fisicamente com as mesmas posições originais, sendo a seqüência ordenada determinada por um vetor[VIO] que é gerado durante o processo de classificação e que possui a seqüência dos endereços ordenada pelos valores classificados da tabela associada) e encadeamento (as entradas não sofrem alterações em suas posições físicas, então, é formada uma lista encadeada que inclui todas as entradas da tabela ordenada pelo valor da chave de classificação).

## Métodos de Classificação Interna

Os métodos de classificação interna são divididos em cinco grupos de acordo com a técnica empregada, são eles:

1. Classificação por Inserção (inserção direta e shell)
2. Classificação por Troca (bubblesort e quicksort)
3. Classificação por Seleção (seleção direta e heapsort)
4. Classificação por Distribuição (distribuição de chaves e radixsort)
5. Classificação por Intercalação (mergesort)

## 14.1 – Classificação por Inserção

Na classificação por inserção, a classificação de um conjunto de registros é feita inserindo registros num sub-arquivo classificado anteriormente, ou seja, a inserção de um elemento é feita na posição correta dentro de um sub-arquivo classificado.

## Método da Inserção Direta

Dentre os métodos de inserção, o método da Inserção Direta é o mais simples, porém, é o mais rápido entre os outros métodos considerados básicos – BubbleSort e Seleção Direta, que serão estudados posteriormente. Ele deve ser utilizado para pequenos conjuntos de dados devido a sua baixa eficiência.

A principal característica deste método consiste em ordenarmos nosso arquivo utilizando um sub-arquivo ordenado localizado em seu início, e a cada novo passo, acrescentamos a este sub-arquivo mais um elemento na sua posição correta, até chegar ao último elemento do arquivo gerando um arquivo final ordenado. Como este é um método difícil de ser descrito, faremos a análise do algoritmo através do exemplo.

Iniciando com um arquivo qualquer desordenado que segue:

| 370 | 250 | 210 | 330 | 190 | 230 |
|---|---|---|---|---|---|

Consideraremos o seu primeiro elemento como se ele fosse um sub-arquivo ordenado:

| 370 | 250 | 210 | 330 | 190 | 230 |
|---|---|---|---|---|---|

O passo seguinte é inserir o próximo elemento ao nosso sub-arquivo ordenado, no nosso exemplo o número 250, que é copiado para uma variável auxiliar. Caminharemos pelo o sub-arquivo a partir do último elemento para o primeiro. Assim poderemos encontrar a posição correta da nossa variável auxiliar dentro do sub-arquivo.

Notamos no nosso exemplo que a variável auxiliar, 250, é menor que o último elemento do nosso sub-arquivo ordenado (o nosso sub-arquivo só possui por enquanto um elemento, o número 370). O número 370 é então copiado uma posição para a direita.

| 370 | 370 | 210 | 330 | 190 | 230 |
|---|---|---|---|---|---|

Nossa variável auxiliar com o número 250, é colocada em sua posição correta no sub-arquivo ordenado.

| 250 | 370 | 210 | 330 | 190 | 230 |
|---|---|---|---|---|---|

Vale notar que nosso sub-arquivo já possui dois elementos e está ordenado. Todo esse processo é repetido para o elemento seguinte, o terceiro elemento. O próximo elemento, 210, é copiado na nossa variável auxiliar e será inserido no sub-arquivo ordenado. Então faremos a comparação da variável auxiliar com os elementos de nosso sub-arquivo, sempre a partir do último elemento para o primeiro.

Na primeira comparação notamos que a variável auxiliar é menor que o último elemento de nosso sub-arquivo. Assim, copiamos este elemento para a direita e continuamos com nossas comparações.

| 250 | 370 | 370 | 330 | 190 | 230 |
|---|---|---|---|---|---|

Novamente, a nossa variável auxiliar é menor que o elemento do sub-arquivo que estamos comparando. Por isso ele deve ser copiado para a direita, abrindo espaço para que a variável auxiliar seja colocada em sua posição correta.

| 250 | 250 | 370 | 330 | 190 | 230 |
|---|---|---|---|---|---|

A variável auxiliar com o número 210, é colocada em sua posição correta.

| 210 | 250 | 370 | 330 | 190 | 230 |
|---|---|---|---|---|---|

Veja que nosso sub-arquivo ordenado possui 3 elementos. Continuaremos o processo de ordenação copiando mais uma vez o elemento imediatamente superior, 330, ao nosso sub-arquivo na variável auxiliar. Vamos, então, comparar essa variável auxiliar com os elementos do nosso sub-arquivo a partir do último elemento.

Observe que nossa variável auxiliar é menor que o elemento que está sendo comparado no nosso sub-arquivo. Então ele deve ser copiado para a direita para que continuemos com nossas comparações.

| 210 | 250 | 370 | 370 | 190 | 230 |
|---|---|---|---|---|---|

Nessa comparação notamos que a variável auxiliar é maior que o elemento do sub-arquivo que estamos comparando (250<330). Portanto, encontramos a posição correta para a inserção de nossa variável auxiliar. Vamos inseri-la na sua posição correta, isto quer dizer que vamos copiá-la para o elemento imediatamente superior ao elemento que estava sendo comparado.

| 210 | 250 | 330 | 370 | 190 | 230 |
|---|---|---|---|---|---|

Note que nosso sub-arquivo possui quatro elementos ordenados. Iremos, então, repetir o processo de ordenação, copiando o próximo elemento do arquivo para uma variável auxiliar. Repetiremos as comparações de trás para frente até encontrarmos a posição correta para inserirmos o nosso elemento. Sempre que necessário, copiaremos na sua posição imediatamente à direita os elementos que forem maiores que a variável auxiliar até que se encontre um elemento menor ou até que se chegue ao início do arquivo que indicará a posição procurada. O passo-a-passo é mostrado a seguir.

| 210 | 250 | 330 | 370 | 190 | 230 |
|---|---|---|---|---|---|

Aux = 190

| 210 | 250 | 330 | 370 | 370 | 230 |
|---|---|---|---|---|---|

| 210 | 250 | 330 | 330 | 370 | 230 |
|---|---|---|---|---|---|

| 210 | 250 | 250 | 330 | 370 | 230 |
|---|---|---|---|---|---|

| 210 | 210 | 250 | 330 | 370 | 230 |
|---|---|---|---|---|---|

| 190 | 210 | 250 | 330 | 370 | 230 |
|---|---|---|---|---|---|

Mais uma vez chegamos a um sub-arquivo ordenado. Os passos seguintes mostram a última etapa de ordenação do nosso arquivo.

| 190 | 210 | 250 | 330 | 370 | 230 |
|---|---|---|---|---|---|

Aux = 230

| 190 | 210 | 250 | 330 | 370 | 370 |
|---|---|---|---|---|---|

| 190 | 210 | 250 | 330 | 330 | 370 |
|---|---|---|---|---|---|

| 190 | 210 | 250 | 250 | 330 | 370 |
|---|---|---|---|---|---|

| 190 | 210 | 230 | 250 | 330 | 370 |
|---|---|---|---|---|---|

Chegamos agora ao resultado esperado: nosso arquivo totalmente ordenado.

## Método dos Incrementos Decrescentes (Método de Shell)

Esse método foi proposto por Ronald Shell em 1959. Diferente do Método de Inserção Direta que considera apenas um segmento, nesse método vários segmentos são considerados e feita uma inserção seletiva do elemento em um dos segmentos. Os segmentos são formados pegando-se os elementos que se encontram nas posições múltiplas de um determinado valor chamado incremento. Veja o exemplo a seguir para um incremento igual a 4.

1º Segmento: Array[0], Array[4], Array[8], ...
2º Segmento: Array[1], Array[5], Array[9], ...
3º Segmento: Array[2], Array[6], Array[10], ...
4º Segmento: Array[3], Array[7], Array[11], ...

A cada passo os segmentos são classificados isoladamente por inserção direta e o incremento para o passo subsequente passa a ser menor que no passo anterior, podendo ser a metade do anterior. A cada passo, o número de segmentos passa a ser menor, porém a quantidade de elementos em cada seguimento passa a ser maior. Após o passo onde o incremento foi igual a 1, o vetor estará totalmente ordenado! Nesse caso é feita uma classificação por inserção direta, porém, o arquivo estará quase ordenado.

Dado o exemplo abaixo, vamos seguir a idéia do método Shell.

| 18 | 15 | 12 | 20 | 5 | 10 | 25 | 8 |
|---|---|---|---|---|---|---|---|

Acima está o nosso arquivo que queremos ordenar. Nosso primeiro passo é dividir o arquivo em segmentos definidos pelos elementos do arquivo original que estejam nas posições múltiplas do incremento definido, no nosso caso 4.

| 18 | 15 | 12 | 20 | 5 | 10 | 25 | 8 |
|---|---|---|---|---|---|---|---|
| 1º seg | 2º seg | 3º seg | 4º seg | 1º seg | 2º seg | 3º seg | 4º seg |

Em seguida ordenaremos cada seguimento, utilizando o método da inserção direta. Observe que inicialmente temos muitos segmentos pequenos e que a cada passo teremos menos segmentos, porém, com mais elementos e eles estarão a cada passo mais próximo da ordenação final. Veja abaixo o resultado após o primeiro passo.

| 5 | 10 | 12 | 8 | 18 | 15 | 25 | 20 |
|---|---|---|---|---|---|---|---|
| 1º seg | 2º seg | 3º seg | 4º seg | 1º seg | 2º seg | 3º seg | 4º seg |

Veja que cada segmento se encontra ordenado. O passo seguinte refaz os seguimentos e todo o processo é repetido de forma similar, porém, o incremento passa a ser a metade do valor anterior. Sendo agora o nosso incremento igual a 2.

| 5 | 10 | 12 | 8 | 18 | 15 | 25 | 20 |
|---|---|---|---|---|---|---|---|
| 1º seg | 2º seg | 1º seg | 2º seg | 1º seg | 2º seg | 1º seg | 2º seg |

Após a ordenação dos seguimentos chegamos ao seguinte arquivo:

| 5 | 8 | 12 | 10 | 18 | 15 | 25 | 20 |
|---|---|---|---|---|---|---|---|
| 1º seg | 2º seg | 1º seg | 2º seg | 1º seg | 2º seg | 1º seg | 2º seg |

Mais uma vez, veja que cada segmento se encontra ordenado. O passo seguinte refaz os seguimentos e todo o processo é repetido de forma similar, porém, como o nosso incremento passa a ser igual a 1 será utilizado o método da inserção direta. Vale lembrar que o arquivo está quase ordenado!

| 5 | 8 | 12 | 10 | 18 | 15 | 25 | 20 |
|---|---|---|---|---|---|---|---|

Após a ordenação pela inserção direta chegamos ao seguinte arquivo:

| 5 | 8 | 10 | 12 | 15 | 18 | 20 | 25 |
|---|---|---|---|---|---|---|---|

Chegamos agora ao resultado esperado: nosso arquivo totalmente ordenado.

## 14.2 – Classificação por Troca

Na classificação por troca, a classificação de um conjunto de registros é feita através de comparações entre os elementos e trocas sucessivas desses elementos entre posições no arquivo.

## Método BubbleSort

É certamente o método mais simples, de entendimento e programação mais fáceis. O Bubblesort é um dos mais conhecidos e utilizados métodos de ordenação de arquivos, principalmente por principiantes em programação e professores que procuram desenvolver o raciocínio nas disciplinas de algoritmos e programação.

Entretanto, o Bubblesort não apresenta um desempenho satisfatório se comparado com os demais.

O Bubblesort se baseia em trocas de valores entre posições consecutivas, levando os valores mais altos (ou mais baixos) para o final do arquivo.

Vamos acompanhar o exemplo para colocar o arquivo em ordem crescente de valores.

Dado o exemplo abaixo com o arquivo inicial que desejamos ordenar, vamos seguir a idéia do método Bubblesort.

| 18 | 15 | 20 | 12 |
|---|---|---|---|

Iniciaremos comparando os dois primeiros elementos (o primeiro com o segundo). Como eles estão desordenados, então, trocamos as suas posições.

| 15 | 18 | 20 | 12 |
|---|---|---|---|

Comparamos agora o segundo com o terceiro. Como eles estão ordenados, então, passaremos para a comparação seguinte (o terceiro com o quarto). Como eles estão desordenados, então, trocamos as suas posições.

| 15 | 18 | 12 | 20 |
|---|---|---|---|

Chegamos agora ao final do arquivo! Note que após essa primeira fase de comparações e trocas o maior elemento, 20, está na sua posição correta e final.

Na próxima fase, todo o processo será repetido, porém com a diferença que agora as comparações e trocas não serão mais feitas até o final do arquivo, mais sim até o último número que antecede aquele que chegou a sua posição correta no nosso caso o penúltimo elemento. Vamos mais uma vez iniciar o processo, comparando os dois primeiros elementos do arquivo. Verificamos que como os valores estão ordenados, não será necessária a troca de posições. Continuaremos as comparações e notamos que precisamos trocar o terceiro pelo segundo elemento.

| 15 | 12 | 18 | 20 |
|---|---|---|---|

Note que as comparações e trocas dessa fase chegaram ao final, uma vez que o último elemento já está em sua posição correta e que levamos o segundo maior elemento para a sua posição correta. Vamos mais uma vez iniciar o processo, comparando os dois primeiros elementos do arquivo. Verificamos que como os valores não estão ordenados, será necessária a troca de posições.

| 12 | 15 | 18 | 20 |
|---|---|---|---|

Chegamos agora ao resultado esperado: nosso arquivo totalmente ordenado.

## Método QuickSort

O método QuickSort, também chamado Método por Troca de Partição é o método que apresenta melhor desempenho se comparado com os métodos estudados até o momento. Ele se baseia na idéia “dividir para conquistar”, ou seja, é mais rápido ordenar dois arquivos menores do que um grande.

O particionamento do arquivo é feito através da escolha de um determinado elemento, chamado pivô, e em seguida todos os elementos menores que o pivô ficam a sua esquerda enquanto aqueles maiores ficam do seu lado direito. O processo continua de forma recursiva ordenando-se o sub-arquivo particionado do lado esquerdo e o sub-arquivo particionado do lado direito.

A escolha do pivô pode ser definida usando uma estratégia. Algumas das estratégias mais utilizadas são: escolha do primeiro elemento do arquivo, escolha do elemento mediano do arquivo, escolha do elemento mediano entre o primeiro, o do meio e o último elemento do arquivo, etc.. A estratégia de escolha do pivô adotada influenciará na eficiência do algoritmo.

Dado o exemplo abaixo, vamos seguir a idéia do método Quicksort.

| 10 | 20 | 12 | 5 | 8 | 15 |
|---|---|---|---|---|---|

Acima está o nosso arquivo que queremos ordenar. Nosso primeiro passo é escolher o pivô (através de uma das técnicas), no nosso caso escolheremos o primeiro elemento do arquivo, 10. Seguiremos, colocando um apontador no início(que pesquisa os elementos maiores que o pivô e será chamado de menor) e outro no final (que pesquisa os elementos menores que o pivô e será chamado de maior) do arquivo.

| 10 | 20 | 12 | 5 | 8 | 15 |
|---|---|---|---|---|---|
| menor | | | | | maior |

Incrementaremos menor até acharmos um elemento maior que pivô.

| 10 | 20 | 12 | 5 | 8 | 15 |
|---|---|---|---|---|---|
| | menor | | | | maior |

Incrementaremos maior até acharmos um elemento menor que pivô

| 10 | 20 | 12 | 5 | 8 | 15 |
|---|---|---|---|---|---|
| | menor | | | maior | |

Como a posição para a qual o apontador maior é maior que a posição de menor, então faremos a troca entre os valores.

| 10 | 8 | 12 | 5 | 20 | 15 |
|---|---|---|---|---|---|
| | menor | | | maior | |

Mais uma vez, incrementaremos menor até acharmos um elemento maior que pivô.

| 10 | 8 | 12 | 5 | 20 | 15 |
|---|---|---|---|---|---|
| | | menor | | maior | |

Incrementaremos maior até acharmos um elemento menor que pivô

| 10 | 8 | 12 | 5 | 20 | 15 |
|---|---|---|---|---|---|
| | | menor | maior | | |

Como a posição para a qual o apontador maior é maior que a posição de menor, então faremos a troca entre os valores.

| 10 | 8 | 5 | 12 | 20 | 15 |
|---|---|---|---|---|---|
| | | menor | maior | | |

Mais uma vez, incrementaremos menor até acharmos um elemento maior que pivô.

| 10 | 8 | 5 | 12 | 20 | 15 |
|---|---|---|---|---|---|
| | | | maior menor | | |

Incrementaremos maior até acharmos um elemento menor que pivô

| 10 | 8 | 5 | 12 | 20 | 15 |
|---|---|---|---|---|---|
| | | maior | menor | | |

Como a posição para a qual o apontador maior é menor que a posição de menor, então não faremos a troca entre os valores, mas sim trocaremos o valor de pivô pelo valor de maior.

| 5 | 8 | 10 | 12 | 20 | 15 |
|---|---|---|---|---|---|
| | | maior | menor | | |

Agora, reiniciaremos todo o processo para ordenar o sub-arquivo a esquerda (5,8) e para ordenar o sub-arquivo à direita (12, 20, 15) do nosso pivô (10). Ao final de todas as ordenações dos sub-arquivos gerados, teremos o nosso arquivo totalmente ordenado.

## 14.3 – Classificação por Seleção

Na classificação por seleção, a classificação de um conjunto de registros é feita através de sucessivas seleções do menor elemento de um arquivo ou sub-arquivo, durante o processo o menor valor encontrado é colocado na sua posição correta final no arquivo e o processo é repetido para o sub-arquivo que contém os elementos que ainda não foram selecionados.

## Método Seleção Direta

O metódo de Seleção Direta possui melhor desempenho que o método Bubblesort, porém, só deve ser utilizado em pequenos arquivos. Assim como o Bubblesort, esse método também é bastante usado durante o estudo de algoritmos e desenvolvimento do raciocínio lógico.

A Seleção Direta se baseia na fixação da primeira posição e na busca, por todo o arquivo, do menor elemento quando então é feita a troca dos valores. No final, teremos o menor valor (ou o maior, conforme a comparação) na primeira posição do arquivo.

Este primeiro passo nos garante que o menor elemento fique na primeira posição. Continuamos, assim, a buscar os demais elementos, comparando-os com a segunda posição do arquivo (já desconsiderando a primeira posição, que foi anteriormente ordenada em relação ao arquivo como um todo).

Dado o exemplo abaixo, vamos seguir a idéia do método Seleção Direta.

| 10 | 20 | 12 | 5 | 8 | 15 |
|---|---|---|---|---|---|

Vamos fixar a primeira posição e, utilizando uma variável auxiliar, procurar o menor elemento do arquivo e trocá-lo pelo elemento que ocupava a primeira posição.

| 5 | 20 | 12 | 10 | 8 | 15 |
|---|---|---|---|---|---|

Agora, fixaremos a segunda posição e repetiremos o processo acima.

| 5 | 8 | 12 | 10 | 20 | 15 |
|---|---|---|---|---|---|

Passaremos para a terceira posição e repetiremos o processo. Acompanhe os passos para conclusão do algoritmo.

| 5 | 8 | 10 | 12 | 20 | 15 |
|---|---|---|---|---|---|

| 5 | 8 | 10 | 12 | 20 | 15 |
|---|---|---|---|---|---|

| 5 | 8 | 10 | 12 | 15 | 20 |
|---|---|---|---|---|---|

Chegamos agora ao resultado esperado: nosso arquivo totalmente ordenado.

## Método HeapSort

Possui a mesma eficiência do Quicksort para a maioria dos casos. É dividido em duas etapas:

1. Preparação do monte inicial (heap): organiza o arquivo a ser ordenado em forma de árvore binária (não árvore de busca binária) na qual o valor do nó terá que ser obrigatoriamente maior que seus sucessores;

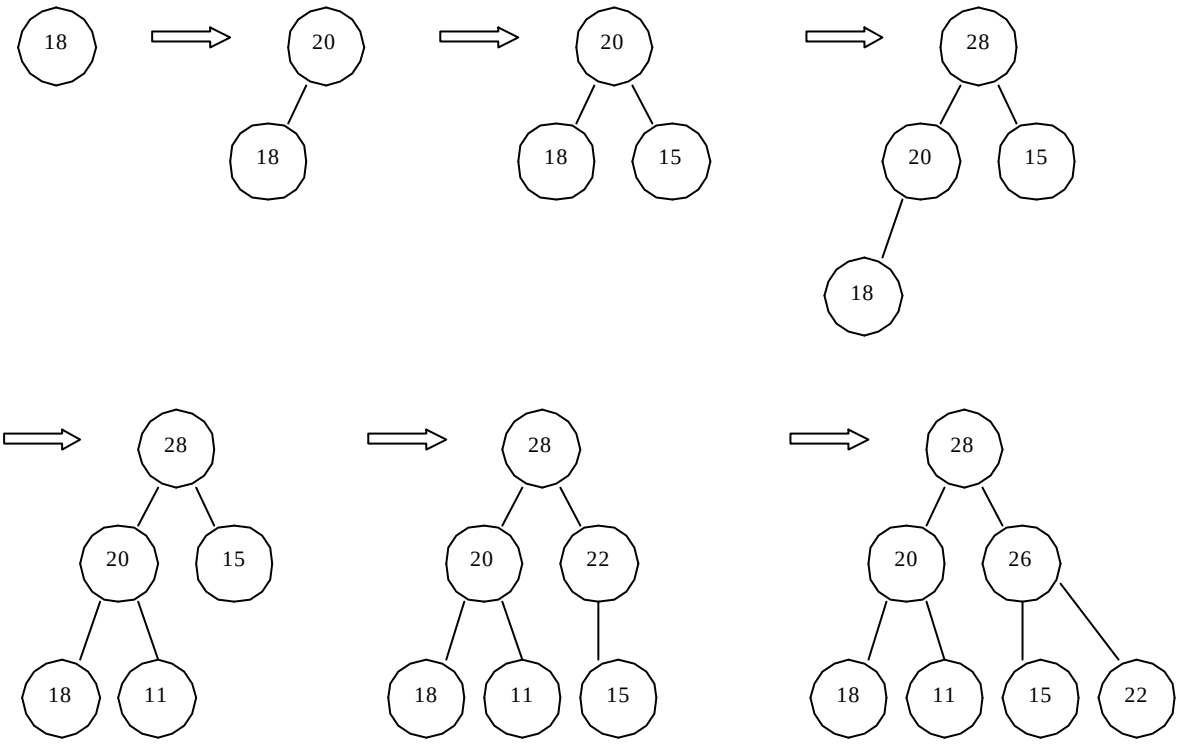
2. Ordenação do heap que gerará o arquivo final ordenado.

A ordenação do heap é feita retirando-se a raiz da árvore, que é o elemento de maior valor do arquivo, colocando-a na posição de maior índice do arquivo e reinserindo o elemento que estava nessa última posição no heap.

Dado o exemplo abaixo, vamos seguir a idéia do método Heapsort.

| 18 | 20 | 15 | 28 | 11 | 22 | 26 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

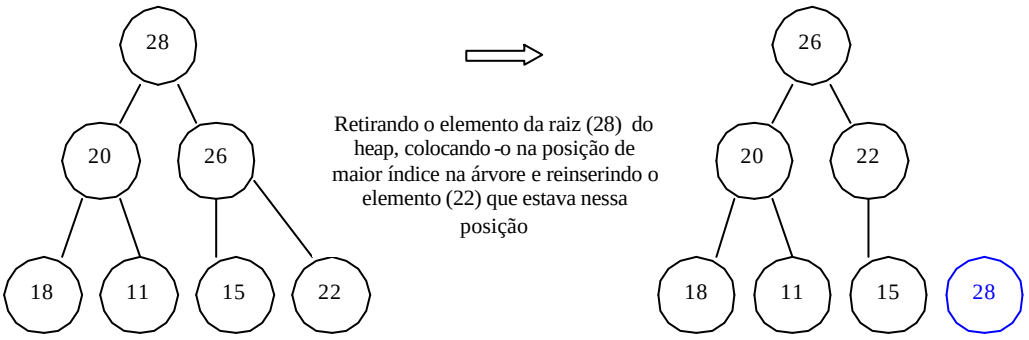
O primeiro passo é a criação do heap (árvore, monte inicial), não esqueça que cada nó tem que ter valor maior que os de seus filhos. Veja a seguir a criação do nosso heap.

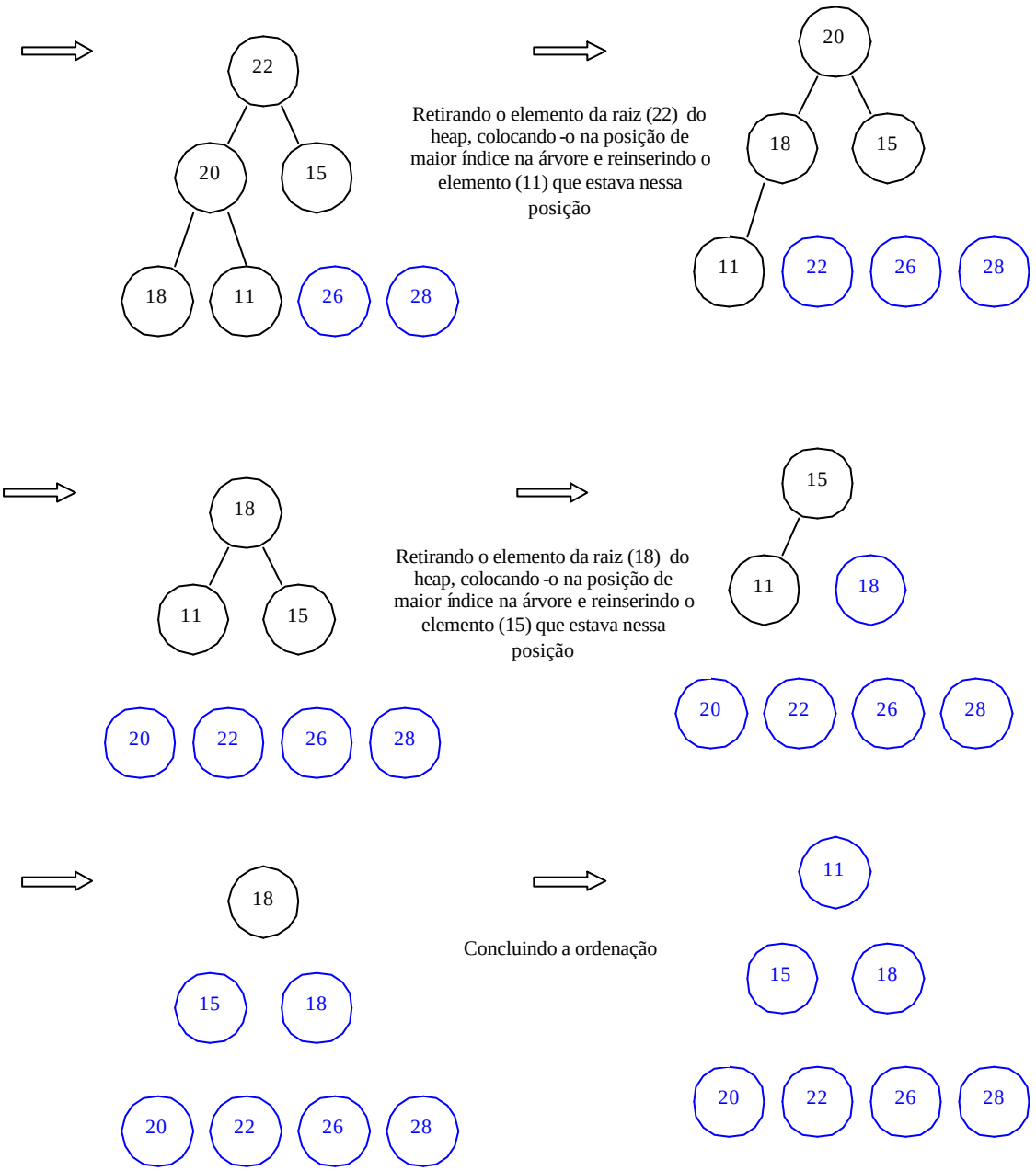



Após a formação do nosso heap, o arquivo estará dessa forma:

| 28 | 20 | 26 | 18 | 11 | 15 | 22 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Os passos de ordenação do heap podem ser acompanhados a seguir.






Após a ordenação teremos o arquivo classificado dessa forma:

| 11 | 15 | 18 | 20 | 22 | 26 | 28 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Chegamos agora ao resultado esperado: nosso arquivo totalmente ordenado.

## 14.4 – Classificação por Distribuição

Na classificação por distribuição encontramos, em geral, um arquivo com números repetidos diversas vezes. Para cada valor repetido no vetor iremos definir em outro vetor o número de repetições do valor, em seguida, recolocamos os valores no arquivo ordenando por uma parte da chave.

A idéia é dividir a chave em partes e para cada uma dessas partes será executada uma etapa de ordenação. O arquivo é ordenado de acordo com a parte da chave correspondente ao passo.

## Classificação por Distribuição de Chave

Dado o exemplo abaixo do arquivo que queremos ordenar, vamos seguir a idéia do método de classificação por Distribuição de Chave.

| 567 | 361 | 852 | 495 | 689 | 752 | 117 | 659 | 427 | 795 | 342 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

O primeiro passo é definir o(s) dígito(s) que será(ão) analisados, no nosso caso iremos analisar o último dígito da chave, ou seja, o dígito das unidades. Em seguida, geraremos a tabela de freqüência do dígito analisado no arquivo a ser ordenado, inicialmente, esta tabela tem todos os seus elementos iguais a zero e depois todo o arquivo original é percorrido pegando a parte do elemento a ser analisada e, a cada ocorrência do valor, a freqüência é acrescida de 1. Veja a seguir a tabela de freqüência gerada.

| Dígitos avaliados | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Freqüência do dígito na parte da chave | 0 | 1 | 3 | 0 | 0 | 2 | 0 | 3 | 0 | 2 |

Isto quer dizer que para o dígito avaliado 2, no nosso caso o último dígito de cada valor de chave, há 3 ocorrências de chaves que possuem o último dígito igual a 2.

Agora, geraremos a tabela de freqüência acumulada, ou seja, a quantidade de ocorrências de chaves com o último dígito sendo igual a qualquer um dos dígitos anteriores. Para entender melhor, verifique, a seguir, a tabela de freqüência acumulada produzida no nosso exemplo.

| Dígitos avaliados | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Freqüência acumulada dos dígitos anteriores (indica o endereço anterior do elemento no arquivo auxiliar) | 0 | 0 | 1 | 4 | 4 | 4 | 6 | 6 | 9 | 9 |

A freqüência acumulada de elementos com último dígito igual a um dos anteriores, quando acrescido de 1, indica, na verdade, a posição no arquivo auxiliar do próximo elemento do arquivo desordenado cujo último dígito seja o avaliado naquele instante.

Para finalizar essa primeira fase, que correspondente a análise do último dígito da chave, utilizaremos um arquivo auxiliar para iniciar a classificação.

Percorrendo o arquivo desordenado inicial, para cada elemento encontrado o colocaremos no arquivo auxiliar na posição seguinte àquela indicada pela tabela de freqüência acumulada analisando-se o dígito avaliado. Ao inserirmos um elemento no arquivo auxiliar, obrigatoriamente, atualizaremos a tabela de freqüência acumulada, somando 1 ao endereço que ela indicava ao elemento de dígito avaliado.

O arquivo auxiliar será criado como segue.

| 361 | 852 | 752 | 342 | 495 | 795 | 567 | 117 | 427 | 689 | 659 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

A tabela de freqüência acumulada produzida ao final da execução da primeira fase do nosso exemplo está logo abaixo.

| Dígitos avaliados | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Freqüência acumulada dos dígitos anteriores (indica o endereço anterior do elemento no arquivo auxiliar) | 0 | ~~0~~<br>1 | ~~1~~<br>~~2~~<br>~~3~~<br>4 | 4 | ~~4~~<br>~~5~~<br>6 | 4 | ~~6~~<br>~~7~~<br>~~8~~<br>9 | 6 | 9 | ~~9~~<br>~~10~~<br>11 |

## Método RadixSort

O metódo RadixSort (Classificação de Raízes) é outro método baseado na distribuição de chaves. Começamos pegando cada dígito menos significativo (dígito das unidades) e posicionando-o em uma das dez filas, dependendo do valor do dígito. Em seguida, restauramos cada fila para o arquivo original, começando pela fila de números com o dígito 0 e terminando com a fila de números com o dígito 9. Quando essas ações tiverem sido executadas para cada dígito, começando com o menos significativo e terminando com o mais significativo, o arquivo estará classificado.

Dado o exemplo abaixo do arquivo que queremos ordenar, vamos seguir a idéia do método RadixSort.

| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Distribuindo cada número em sua fila correspondente ao dígito da unidade, teremos:

| Fila 0 | Fila 1 | Fila 2 | Fila 3 | Fila 4 | Fila 5 | Fila 6 | Fila 7 | Fila 8 | Fila 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | 12<br>92 | 33 | | 25 | 86 | 57<br>37 | 48 | |

Restaurando as filas para o arquivo original, teremos então:

| 12 | 92 | 33 | 25 | 86 | 57 | 37 | 48 |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Repetimos a distribuição dos números nas filas, porém agora se baseando no dígito da dezena.

| Fila 0 | Fila 1 | Fila 2 | Fila 3 | Fila 4 | Fila 5 | Fila 6 | Fila 7 | Fila 8 | Fila 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 12 | 25 | 33<br>37 | 48 | 57 | | | 86 | 92 |

Após restaurar novamente as filas, o arquivo estará classificado.

| 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

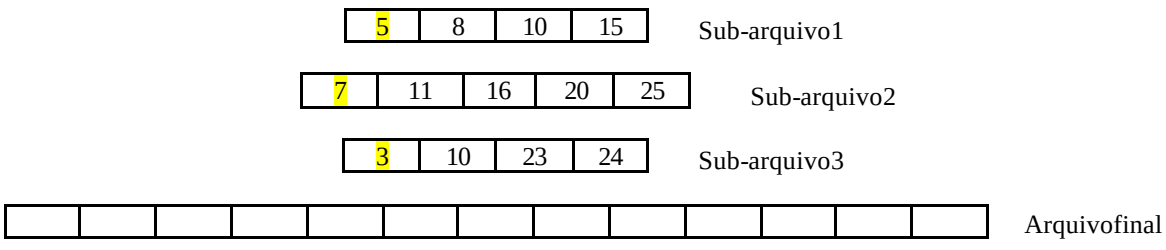
Deixaremos, como exercício, a codificação da rotina que implementa o método RadixSort.

## 14.5 – Classificação por Intercalação

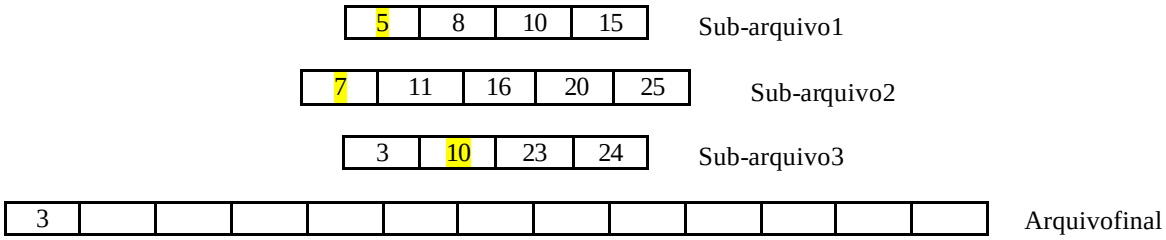
Na classificação por intercalação utilizamos dois ou mais arquivos anteriormente ordenados para gerar um outro arquivo ordenado. É comum a utilização de outros métodos de classificação para ordenação dos sub-arquivos que gerarão o arquivo final ordenado. O processo de intercalação é feito percorrendo-se simultaneamente os sub-arquivos, movendo-se a cada passo o menor entre os elementos da posição analisada dos sub-arquivos para o arquivo resultado final.

É comum dividir o arquivo que queremos ordenar em sub-arquivos ordenados (ou pelo método de Intercalação ou através de qualquer outro) e depois compor o arquivo final através da intercalação simultânea dos sub-arquivos, sendo a cada etapa inserido o menor dos elementos no arquivo que queremos gerar ordenadamente. Vale lembrar que o tamanho do arquivo final é a soma dos tamanhos dos sub-arquivos envolvidos.

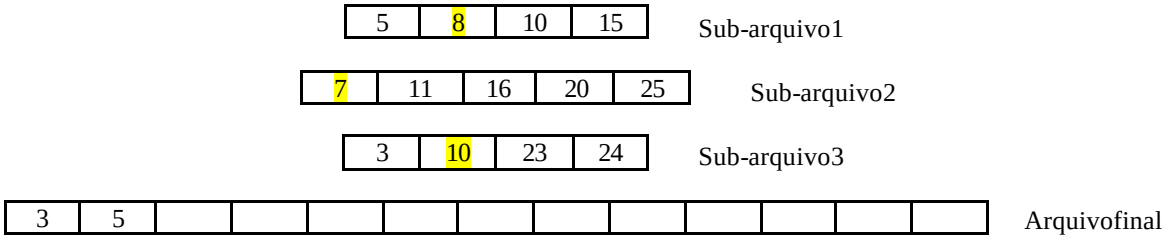
Siga o exemplo abaixo para acompanhar a idéia da classificação por intercalação, utilizando três sub-arquivos para gerar o arquivo final ordenado.

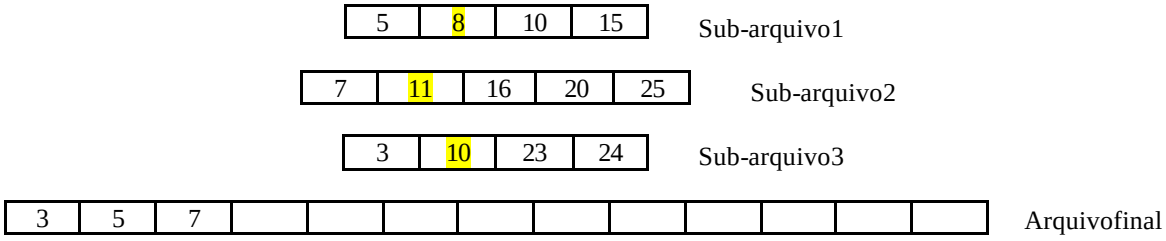


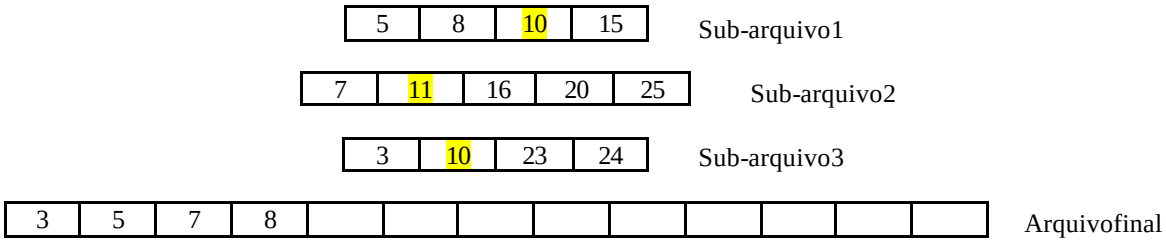
Após o primeiro passo (pega o menor entre os elementos analisados nos sub-arquivos e passa a analisar o próximo apenas no sub-arquivo que possuía o menor entre os elementos) teremos:

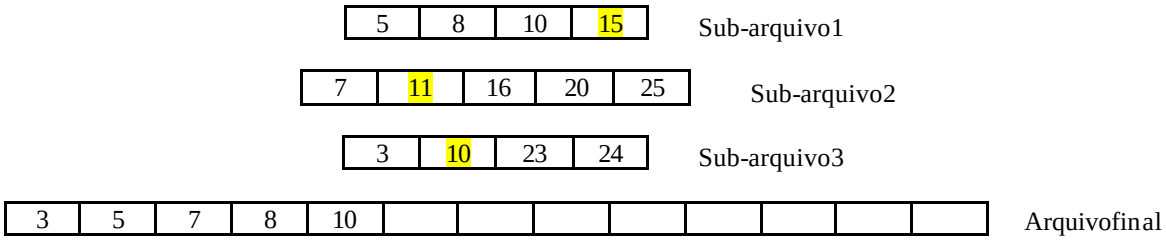


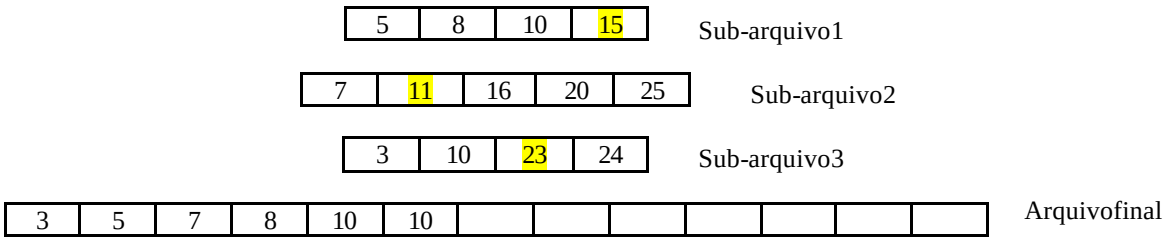

Siga os passos seguintes.

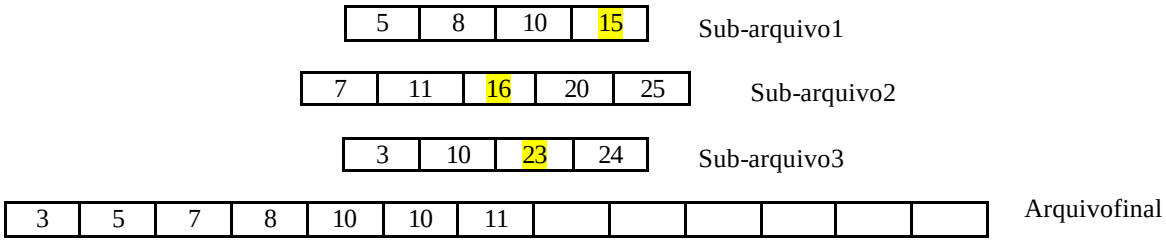


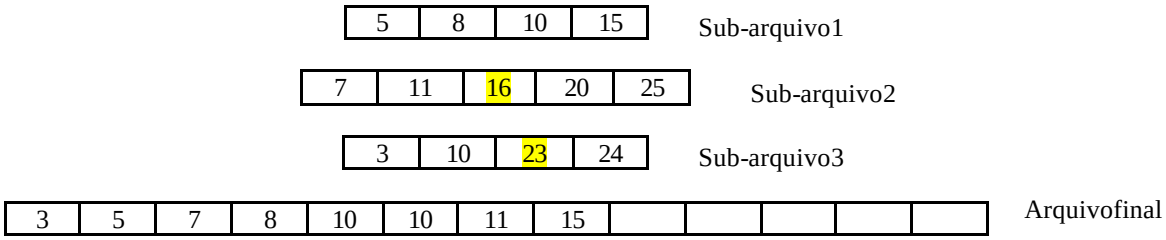



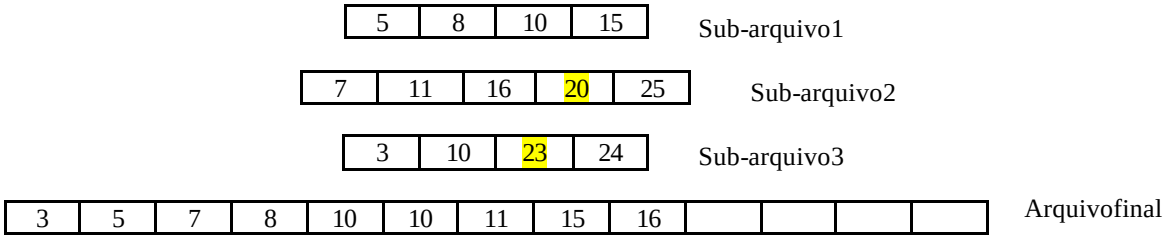



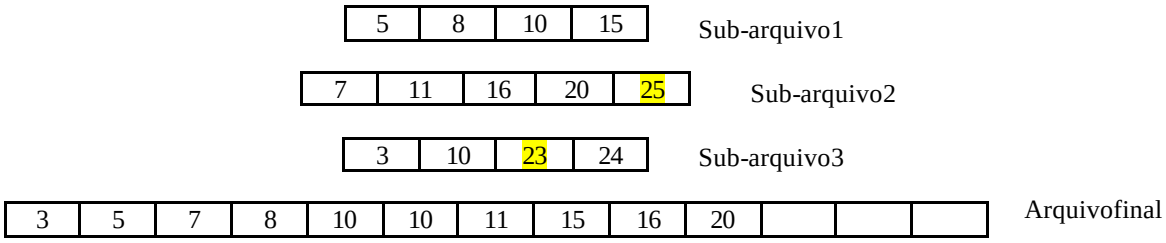



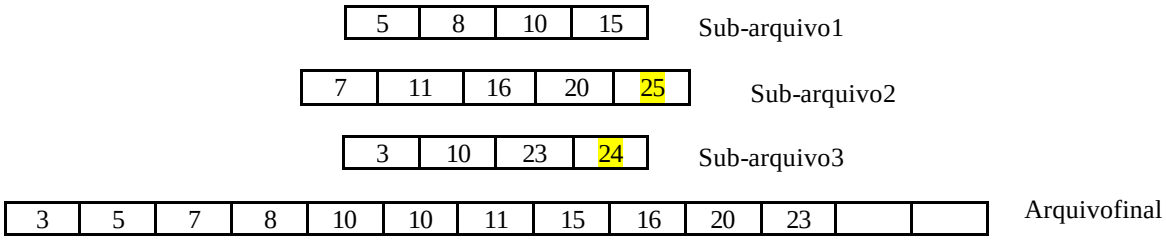



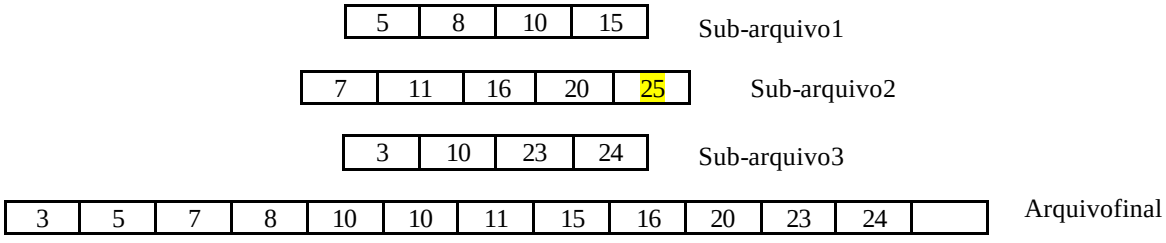

5 8 10 15
Sub-arquivo1
7 11 16 20 25
Sub-arquivo2
3 10 23 24
Sub-arquivo3
3 5 7 8 10 10 11
Arquivofinal

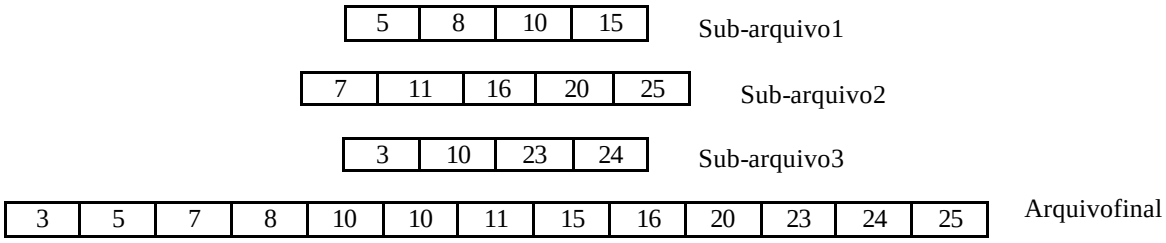

5 8 10 15
Sub-arquivo1
7 11 16 20 25
Sub-arquivo2
3 10 23 24
Sub-arquivo3
3 5 7 8 10 10 11 15
Arquivofinal


5 8 10 15
Sub-arquivo1
7 11 16 20 25
Sub-arquivo2
3 10 23 24
Sub-arquivo3
3 5 7 8 10 10 11 15 16
Arquivofinal


5 8 10 15
Sub-arquivo1
7 11 16 20 25
Sub-arquivo2
3 10 23 24
Sub-arquivo3
3 5 7 8 10 10 11 15 16 20
Arquivofinal


5 8 10 15
Sub-arquivo1
7 11 16 20 25
Sub-arquivo2
3 10 23 24
Sub-arquivo3
3 5 7 8 10 10 11 15 16 20 23
Arquivofinal


5 8 10 15
Sub-arquivo1
7 11 16 20 25
Sub-arquivo2
3 10 23 24
Sub-arquivo3
3 5 7 8 10 10 11 15 16 20 23 24
Arquivofinal

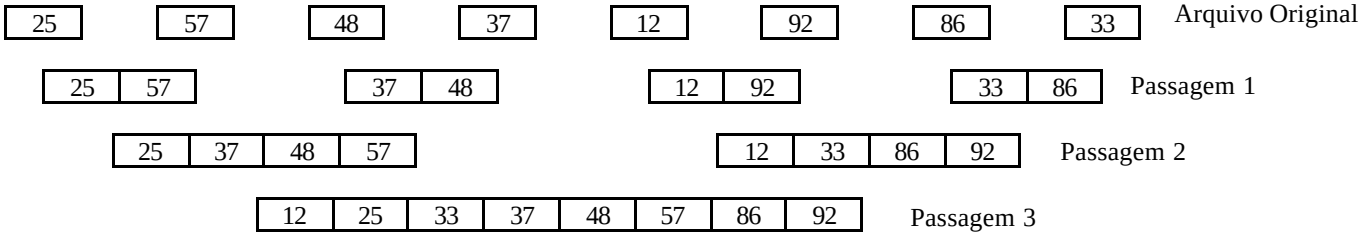



Chegamos agora ao resultado esperado: nosso arquivo totalmente ordenado.

## Método MergeSort

O método MergeSort (intercalação direta) funciona da seguinte maneira: divida o arquivo em n sub-arquivos de tamanho 1 e intercale pares de arquivos adjacentes. Temos, então, aproximadamente n/2 arquivos de tamanho 2. Repita esse processo até restar apenas um arquivo de tamanho n.

Veja o exemplo abaixo:



## EXERCÍCIOS:

**1.** Demonstre graficamente a classificação dos dados 38, 21, 45, 30, 35, 27, usando os seguintes métodos:

a) Bolha

b) Inserção Direta

c) Seleção Direta

d) Shell

e) QuickSort

f) HeapSort

g) MergeSort

h) Distribuição de Chaves

i) RadixSort

**2.** Implemente um programa em C que construa uma tabela comparativa dos métodos de classificação de dados, relacionados a seguir:

- Inserção direta
- Shell
- Bolha
- QuickSort
- Seleção Direta
- HeapSort
- Distribuição de chaves
- MergeSort
- RadixSort

Para cada método, o programa deve calcular:

a) Número médio de comparações

b) Número médio de trocas

c) Média do tempo gasto para a classificação

d) Número de comparações no melhor caso (vetor já classificado)

e) Número de trocas no melhor caso

f) Tempo gasto para o melhor caso

g) Número de comparações no pior caso (vetor classificado inversamente)

h) Número de trocas no pior caso

i) Tempo gasto para o pior caso

Observações:

- Utilize um vetor de 100 elementos inteiros, gerado com valores aleatórios no intervalo de 0 a 999.
- Para o item (a), (b) e (c), obtenha a média da classificação de 20 vetores distintos.