Pesquisa, Ordenação e Técnicas de Armazenamento

Parte I

Métodos de Pesquisa

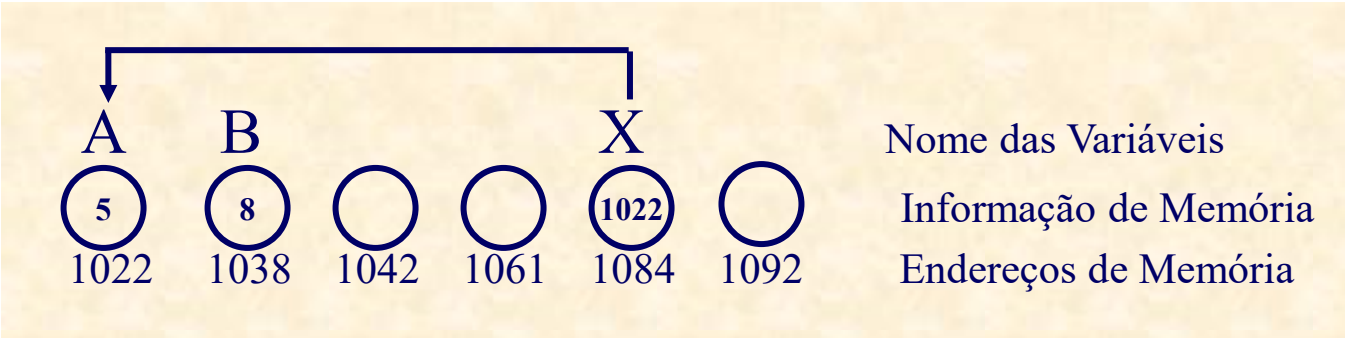
Prof. Dr. Artur Henrique Kronbauer

Agenda

- Revisão
  - Ponteiros
  - Modularização
  - Tipos Abstratos de Dados
  - Vetores
  - Tabelas
  - Recursividade
- Pesquisa Sequencial
- Pesquisa Binária
- Pesquisa Hash
- Árvores de Pesquisa
- Árvores AVL
- Árvores B

Ponteiros

- **Definição**
  - Ponteiros são endereços, isto é, são variáveis que contém um endereço de memória. Se uma variável contém o endereço de outra, então a primeira (o ponteiro) aponta para a segunda.



  - "X" o "ponteiro", aponta para o "inteiro" A.

  - São ferramentas que nos possibilitam manipular endereços de memória e informações contidas nesses endereços.

Ponteiros

- **Operadores**

  - **&** - (E comercial) - fornece o endereço de determinada variável. Atribui o endereço de uma variável para um ponteiro.

    Obs: Não confundir com o operador lógico de operações de baixo nível, de mesmo símbolo.

  - * **-** (Asterisco) – permite acessar o conteúdo de uma variável, cujo endereço é o valor do ponteiro. Devolve o valor endereçado pelo ponteiro.

    Obs: Não confundir com o operador aritmético de multiplicação de mesmo símbolo.

Ponteiros

- **Exemplo 1: Utilização dos operadores & e *.**

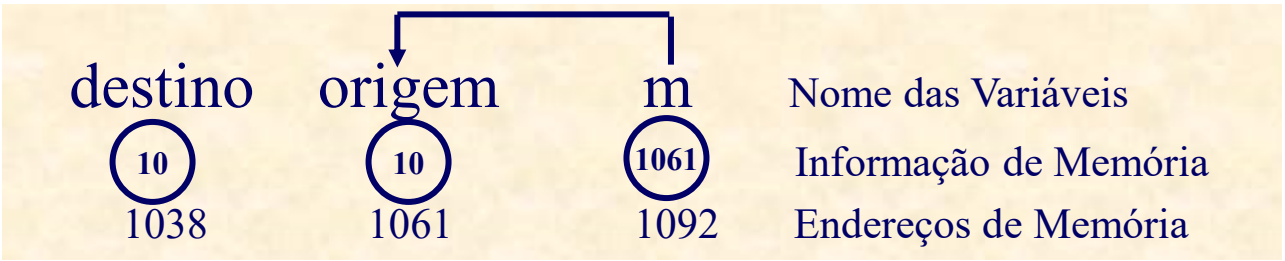
```
#include <stdio.h>;

Main()
{ int destino, origem;
  int *m;
  origem = 10;
  m = &origem;
  destino = *m;
  printf("O resultado é : %i",destino);
}
```

Declaração de um ponteiro.

m obtém o endereço de memória da variável **origem**.

destino recebe a informação contida no endereço apontado por **m**.



ARTUR KRONBAUER

Ponteiros

- **Exemplo 2: Atribuição de ponteiros**

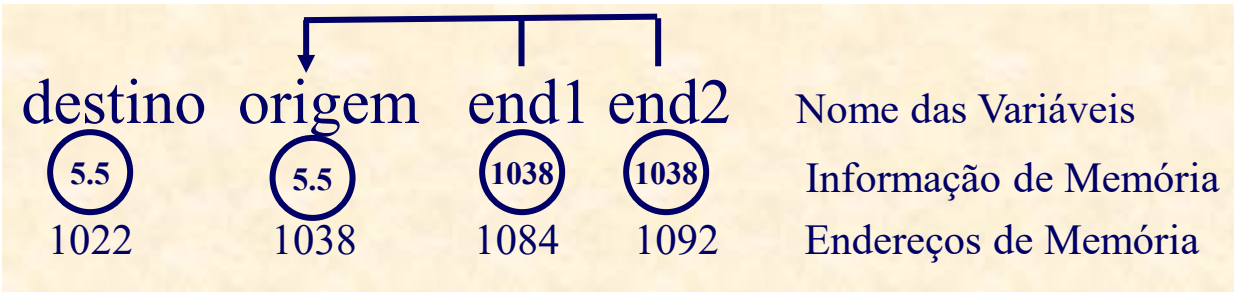
```
#include <stdio.h>;
Main()
{ float destino, origem;
float *end1, *end2;
   origem = 5.5;
   end1 = &origem;
   end2 = end1;
   destino = *end2;
   printf("O resultado é : %f",destino);
}
```

Declaração dos ponteiros.

end1 recebe o endereço de memória da variável **origem**.

destino recebe a informação contida no endereço apontada por **end2.**

end2 recebe a posição de memória da variável **origem** que está guardada em **end1.**

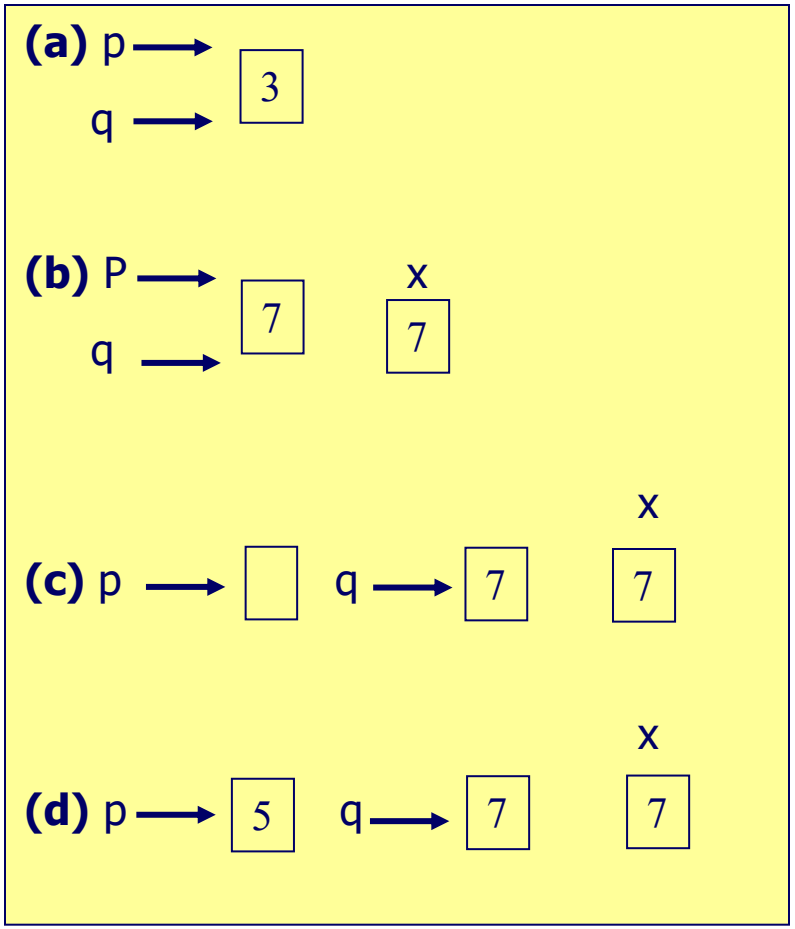


Ponteiros e as Funções malloc e free

- **malloc():** Essa função serve para atribuir a um ponteiro uma determinada região de memória de acordo com o tipo do ponteiro.
  - malloc() está definido na biblioteca **stdlib.h** do C

- **Veja o código a seguir:**

```
        main()
        {   int *p, *q;
            int x;
            p = (int *) malloc (sizeof (int));
            *p = 3;
            q = p;
(a)         printf ("%d  %d \n", *p, *q);
            x = 7;
            *q = x;
(b)         printf ("%d  %d \n", *p, *q);
(c)         p = (int *) malloc (sizeof (int));
(d)         *p = 5;
            printf ("%d %d \n", *p, *q);
            getchar();
        }
```

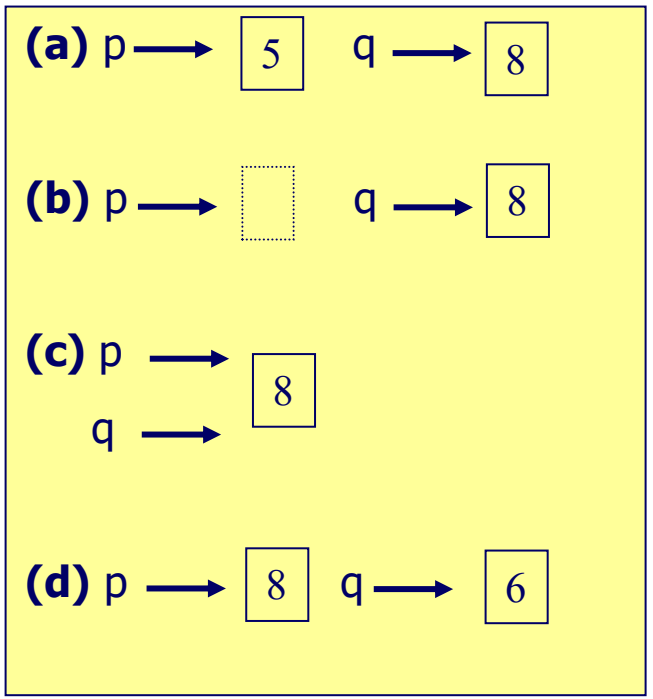


Ponteiros e as Funções malloc e free

- **Free:** Essa função serve para liberar para o sistema operacional uma determinada região de memória alocada por um ponteiro.

- **Veja o Código a seguir:**

```
        main()
        {  int *p, *q;
           int x;
           p = (int *) malloc (sizeof (int));
           *p = 5;
           q = (int *) malloc (sizeof (int));
(a)        *q = 8;
(b)        free(p);
(c)        p = q;
           q = (int *) malloc (sizeof (int));
(d)        *q = 6;
           printf ("%d  %d \n", *p, *q);
           getchar();
        }
```



Modularização

- Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Geralmente as funções são blocos de códigos com alguma funcionalidade específica.

```
tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
    corpo_da_função
}
```

- O comando return ao ser executado encerra a execução de uma função e retorna o valor indicado:

```
return valor_de_retorno;
```

- Protótipos são declarações de funções que são definidas antes da função main() indicando quais serão as funções utilizadas no programa. São indispensáveis para que o compilador tome conhecimento do formato das funções existentes:

```
tipo_de_retorno nome_da_função (declaração_de_parâmetros);
```

Modularização

- Funções com o tipo de retorno **void**, indica que a função não possui retorno, ou seja, pode ser considerada como um procedimento.

- Declarações **Locais** e **Globais**:

  – Variáveis e estruturas declaradas dentro de uma função são consideradas **locais** e são vistas apenas dentro da função. Por outro lado, variáveis e estruturas declaradas fora das funções são vistas em todo o programa e são chamadas de **globais**.

Modularização

- Passagem de Parâmetros:
  - As variáveis listadas no cabeçalho de uma função são chamadas de parâmetros. Os parâmetros podem ser **por valor** ou **por referência**.
  - **Por valor**: a função recebe o valor, o armazena na variável indicada e toda a manipulação sobre a variável só acarreta mudanças validas na variável que recebe o valor, ou seja, só acarreta mudanças na variável dentro da função corrente.
  - **Por referência**: a função recebe o valor, o armazena na variável indicada e toda a manipulação sobre a variável acarreta mudanças também na variável usada para chamar a função. Para isso, é necessário chamar a função usando o operador & ao lado da variável de chamada e usar o operador * na declaração da função ao lado da variável que recebe a informação.

Modularização – Exemplo 1

```
#include <stdio.h>
 float sqr (float);

float sq;

main ()
{    float num;
     printf ("Entre com um numero: ");
     scanf ("%f",&num);
     sq=sqr(num);
     printf ("\n\nO numero original e: %f\n",num);
     printf ("O seu quadrado vale: %f\n",sq);
}

float sqr(float numero)
{    numero=numero*numero;
     return numero;
}
```

Declaração do protótipo da função utilizada no programa

Declaração de uma variável global

Declaração de uma variável local

Chamada da função passando parâmetro

Tipo de retorno

Parâmetro de entrada

Definição da função

Retorno da função

Modularização – Exemplo 2

```
# include<stdio.h>

void leDados(float *, float *);
float calcula(float, float);
float n1, n2, r;

main()
{  leDados(&n1,&n2);
   printf("\n n1 = %f",n1);
   printf("\n n2 = %f",n2);
   r=calcula(n1,n2);
   printf("\n O resultado he = %f",r);
   getchar();
   getchar();
}
```

Chamada de uma função por referência

Chamada da função por valor

Passagem de parâmetro por referência

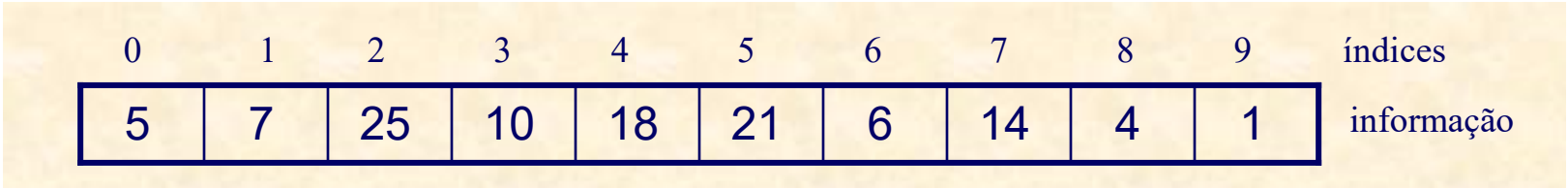
```
void leDados(float *a, float *b)
{  printf("\n Entre com o primeiro número :");
   scanf("%f",a);
   printf("\n Entre com o segundo número :");
   scanf("%f",b);
}
```

Passagem de parâmetro por valor

```
float calcula(float x, float y)
{  float sub, mul, divi, resul;
   sub=x-y;
   mul=x*y;
   divi=x/y;
   resul=sub+mul+divi;
   return resul;
}
```

Vetores – Estruturas Estáticas

- **Definição**
  - São tipos de dados compostos ou estruturados.
  - É um conjunto finito e ordenado de dados.
  - São chamadas de estruturas estáticas porque não podem mudar de tamanho durante a execução do programa, ou seja, preservam o tamanho definido pelo programador no momento do desenvolvimento do software.
  - São formados por índices e informações.
    - Índices: Definem as posições de armazenamento da estrutura
    - Informações: São os dados armazenados e identificados pelos índices.



Vetores – Estruturas Estáticas - Exemplo

```
#include <stdio.h>

void inserir(int []);
void intercala(int[], int[]);

main()
{ int v1[5], v2[5];
  printf("\n Incluir dados no vetor 1");
  inserir(v1);
  printf("\n Incluir dados no vetor 2");
  inserir(v2);
  printf("\n Vetores Intercalados");
  intercala(v1,v2);
  getchar();
  getchar();
}
```

```
void inserir(int vetor[])
{  int i;
   for (i=0; i < 5; i++)
   {  scanf("%d",&vetor[i]);
   }
}

void intercala(int vetor1[], int vetor2[])
{  int i, k = 0;
   int intercalado[10];
   for (i=0; i < 5; i++)
   { intercalado[k]=vetor1[i];
     k++;
     intercalado[k]=vetor2[i];
     k++;
   }
   mostra(intercalado);
}
```

Ponteiros e Vetores

- **Definição :** Inicialmente temos que esclarecer que o nome de um vetor é um ponteiro. Dessa forma podemos trabalhar com vetores de duas formas. veja os códigos a seguir.

```
main ()
{  float vet[10];
   int i;
   for (i=0;i<10;i++)
   {  vet[i]=0.0;
      printf("\n %f",vet[i]);
   }
   getchar();
}
```

```
main ()
{  float vet[10], *p;
   int i;
   p=vet;
   for (i=0;i<10;i++)
   {  *p=0.0;
      p++;
   }
   for (i=0;i<10;i++)
   {  printf("\n %f",vet[i]);
   }
   getchar();
}
```

  - No 1ª código, cada vez que se faz vet[i] o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 10 deslocamentos. No segundo o único cálculo que deve ser feito é o de um incremento de ponteiro, que é muito mais rápido que calcular 10 deslocamentos completos.

Ponteiros e Vetores

- **Manipulação de vetores através de ponteiros**
  - Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como conseqüência podemos também indexar um ponteiro qualquer. Veja o código a seguir.

```
main()
{   int vet [10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *p;
    p=vet;
    printf ("\n O terceiro elemento do vetor e: %d",p[2]);
    printf ("\n O quinto elemento do vetor e: %d",*(p+4));
    printf ("\n O sétimo elemento do vetor e: %d",vet[7]);
    getchar();
}
```

Mostra 3

Mostra 5

Mostra 8

Podemos ver que **p[2]** equivale a ***(p+2)** que equivale a **vet[2]**.

Estruturas – Tipos Abstratos de Dados

- **Definição**
  - Uma estrutura agrupa várias variáveis numa só. Funciona como uma ficha pessoal que tenha nome, telefone e endereço. A ficha seria uma estrutura. A estrutura, então, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

- **Declaração**

```
struct nome_do_tipo_da_estrutura
{  tipo_1   nome_1;
   tipo_2   nome_2;
   ...
   tipo_n   nome_n;
}
variáveis_estrutura;
```

- **Exemplo**

```
struct conta
{  int   num_conta;
   char tipo_conta;
   char nome[80];
   float saldo;
};
struct conta clientenovo, clienteant;
```

  - No exemplo foram declaradas duas variáveis, clientenovo e clienteant que são do tipo da estrutura, isto é, possuem os campos num_conta, tipo_conta, nome e saldo.

Estruturas – Tipos definidos pelos usuários

- **Uma estrutura pode conter outra estrutura.**

```
struct data
{   int   mes;
    int   dia;
    int ano;
};
```

```
struct conta
{ int    num_conta;
   int   tipo_conta;
   char nome[80];
   float saldo;
   struct data ultpag;
}
```

- **Inicializando estruturas**
  - struct conta cliente = {12345, 'R', "Joao", 586.30, 5, 24, 30};
- **Processando uma estrutura**
  - Acessando num_conta: cliente.num_conta
  - Acessando o 3ª caracter do vetor nome: cliente.nome[2]
- **O uso do operador ponto pode ser estendido a vetores**
  - struct conta cliente[100];
  - número da conta do 14º cliente: cliente[13].num_conta
  - mês do último pagamento do 14º cliente: cliente[13].ultpag.mes

Tabelas

```
#include<stdio.h>

typedef struct produto
{  int codigo;
   char nome[50];
   float preco;
} t_produto;

void incluir(t_produto [], int *);
void listagem(t_produto [], int);
void menu();

main()
{  menu();
}
```

Tabelas

```
void menu()
{  int total=0, op;
   t_produto tab[10];
   do
   { printf("\n [1] incluir  [2] listar [3] fim : ");
     scanf("%d",&op);
     switch(op)
     {  case 1: incluir(tab,&total);
        break;
        case 2: listagem(tab, total);
        break;
     }
   }while (op != 3);
}
```

Tabelas

```
void incluir(t_produto t[], int *i)
{  printf("\n Entre com o codigo : ");
   scanf("%d",&t[*i].codigo);
   printf("\n Entre com o nome : ");
   scanf("%s",&t[*i].nome);
   printf("\n Entre com o preco : ");
   scanf("%f",&t[*i].preco);
   *i = *i+1;
}

void listagem(t_produto t[], int total)
{  int i=0;
   while (i < total)
   { printf("\n codigo = %d nome = %s preco = %f",
                                          t[i].codigo,t[i].nome,t[i].preco);
      i++;
   }
}
```

Recursividade

- **Definição**
  - A recursividade pode ser considerada um método eficaz para resolver um problema originalmente complexo, reduzindo-o em pequenas ocorrências do problema principal. Assim, segue a ideia de dividir para conquistar. Resolvendo, isoladamente, cada uma das pequenas partes, podemos obter a solução do problema original como um todo.
- **Características de uma função recursiva**
  - Definição de parâmetros;
  - Condição de parada da recursão, para que a rotina não seja chamada infinitamente;
  - Chamada da função dentro dela própria;
- **Rotinas recursivas e pilhas**
  - O controle de chamadas e de retorno de rotinas é efetuado por uma pilha (criada e mantida dinamicamente pelo sistema). Quando uma rotina é chamada, empilha-se o endereço da rotina e todas as variáveis locais são recriadas. Quando ocorre o retorno da rotina as variáveis locais que foram criadas deixam de existir.
- **Vantagens**
  - Facilidade na resolução de alguns tipos de problemas.
- **Desvantagens**
  - Uso demasiado dos recursos computacionais de um computador.

Recursividade - Exemplo

```
#include <stdio.h>

int fat(int);

main()
{  int i, numero;
   printf("\n Entre com um numero para calcular o fatorial \n ");
   scanf("%d",&numero);
   numero=fat(numero);
   printf("\n O fatorial e : %d ",numero);
}

int fat(int n)
{  int res;
   if (n == 0)
      return 1;
   else
   {  res = fat(n-1);
      res = res * n;
      return res;
   }
}
```

Pesquisa - Introdução

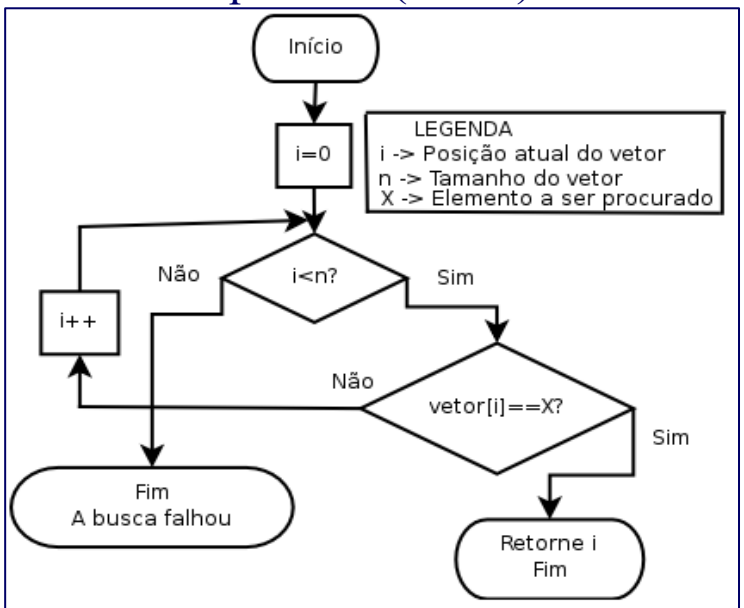
- **Definição**
  - Quando temos um Vetor (Array, Matriz, Tabela), com muitos elementos e precisamos descobrir se um determinado elemento que procuramos se encontra no vetor, uma solução que certamente nos vem à mente é comparar o elemento que procuramos com cada elemento do vetor, até que encontremos ou até que concluamos que o elemento procurado não está no vetor.

  - Esta é a base do raciocínio dos algoritmos de pesquisa ou busca ("Search"), que como sugere o nome, "Pesquisam", em um vetor, a existência ou não existência de um elemento procurado. A diferença entre um e outro algoritmo de busca, fica por conta da rapidez com que "varremos" o vetor para encontrar o elemento ou para concluirmos que ele não existe.

  - Um fator que influencia em muito nessa rapidez é a disposição dos elementos no vetor. Se estão desordenados, seguramente teremos que verificar do primeiro ao último elemento para concluir, com certeza, que o elemento não existe. Já se estão ordenados, ao encontrarmos um elemento maior (ou menor) que o elemento procurado, poderemos concluir pela sua não existência.

Pesquisa Sequencial

- **Definição**
  - Encontrar informações em um vetor desordenada requer uma pesquisa sequencial, começando no primeiro elemento e parando quando o elemento procurado ou o final do vetor é encontrado. Esse método deve ser usado em dados desordenados, mas também pode ser aplicado a dados ordenados. É fácil de ser codificada. Funciona da seguinte forma: a partir do primeiro índice, pesquise sequencialmente até encontrar a chave procurada; então pare.
  - No fluxograma da figura abaixo é possível identificar todos os passos executados em uma busca sequencial (linear).



Pesquisa Sequencial - Implementação

```
int pesquisaSequencial(int vetor[], int tamanho, int elementoProcurado)
{    int i;
     for (i = 0; i < tamanho; i++)
     {      if (vetor[i] == elementoProcurado)
            {      return i;              // retorna a posição caso encontre o elemento procurado
            }
     }
     return -1;                          // retorna -1 caso não encontre o elemento procurado
 }
```

Análise de Complexidade: No melhor caso, o elemento a ser buscado é encontrado logo na primeira tentativa da busca. No pior caso, o elemento a ser buscado encontra-se na última posição e são feitas N comparações, sendo N o número total de elementos. No caso médio, o elemento é encontrado após (N+1)/2 comparações. O algoritmo de busca linear é um algoritmo O(n).

Pesquisa Sequencial - Implementação

- **Algoritmo Recursivo**

```
int pesquisaSequencialRec(int vet[], int total, int elementoProcurado, int i)
{  if ((i < total) && (vet[i] != elementoProcurado))
        return (pesquisaSequencialRec(vet, total, elementoProcurado, i+1));
   else
   {  if (i == total)
      {
            return(-1);
      }
      else
      {
            return(i);
      }
   }
}
```

Pesquisa Binária

- **Definição**

  - A busca binária é um eficiente algoritmo para encontrar um item em uma lista ordenada de itens. Ela funciona dividindo repetidamente pela metade a porção da lista que deve conter o item, até reduzir as localizações possíveis a apenas uma.

  - Um dos modos mais comuns de se usar a busca binária é para encontrar um item em um vetor. Por exemplo, o catálogo estelar Tycho-2 contém informações sobre as 2.539.913 estrelas mais brilhantes na nossa galáxia. Suponha que você queira buscar por uma estrela em particular no catálogo, com base no nome da estrela. Se o programa examinar cada estrela do catálogo em ordem, começando com o primeiro nome, utilizando busca linear, no pior caso, o computador irá examinar todas as 2.539.913 estrelas para encontrar a desejada. Se o catálogo estivesse organizado alfabeticamente pelos nomes das estrelas, a busca binária não teria que examinar mais do que 22 estrelas, mesmo no pior caso.

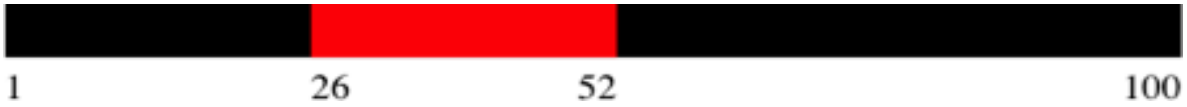
Pesquisa Binária

- **Vamos ver como descrever cuidadosamente a busca binária.**
  - A ideia principal da busca binária é controlar a faixa atual de suposições razoáveis. Vamos dizer que estou pensando em um número entre 1 e 100, assim como o jogo de adivinhação. Se você já tiver tentado o 25 e eu tiver dito que meu número é maior, e já tiver tentado o 81 e eu tiver dito que meu número é menor, então os números na faixa de 26 a 80 são as únicas suposições razoáveis. Observe na régua abaixo que a seção vermelha da reta numérica contém as suposições razoáveis, e a seção preta as suposições que já foram descartadas.

  - A cada vez, você faz uma suposição que divide o conjunto de suposições razoáveis em duas faixas de tamanho aproximadamente igual. Se sua suposição não estiver correta, eu lhe direi se ela é alta ou baixa demais, e você vai poder eliminar cerca de metade das suposições razoáveis.

Pesquisa Binária

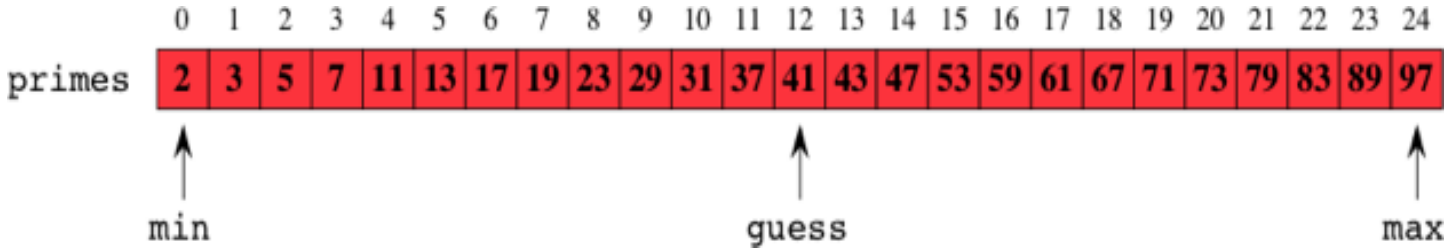
- Por exemplo, se o intervalo corrente de suposições razoáveis é de 26 a 80, você pode adivinhar o ponto do meio, (26+80)/2, ou 53. Então, se eu digo a você que 53 é muito alto, você pode eliminar todos os números de 53 a 80, deixando 26 a 52 como o novo intervalo de valores razoáveis, reduzindo pela metade o intervalo.



- Vamos ver como pensar na busca binária em um vetor organizado. Para a exemplificação vamos utilizar os primeiros 25 números primos, em ordem:
- primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
- Suponha que desejemos saber se o número 67 é primo. Se 67 estiver no vetor, então ele é primo.
- Podemos também querer saber quantos números primos são menores que 67. Se descobrirmos a posição do número 67 no vetor, podemos usar essa informação para descobrir quantos números primos menores que ele existe.

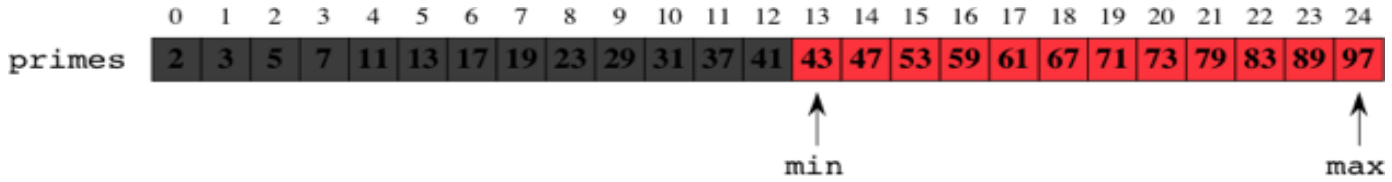
Pesquisa Binária

- A posição de um elemento em um vetor é conhecida como seu índice. Os índices dos vetores começam em 0 e são contados de forma crescente. Se um elemento estiver no índice 0, ele é o primeiro elemento do vetor. Se um elemento estiver no índice 3, então ele tem 3 elementos antes dele no vetor.
- Uma vez que sabemos que o número primo 67 está no índice 18, podemos identificá-lo como um primo. Também podemos identificar rapidamente que há 18 elementos antes do 67 no vetor, o que significa que há 18 números primos menores que 67.
- Usando as variáveis min = 0 e max = 24 para exemplificar a pesquisa do primo 67, a primeira suposição na busca binária seria o índice 12 (que é igual a (0+24)/2). primes[12] é igual a 67? Não, primos[12] é igual a 41.

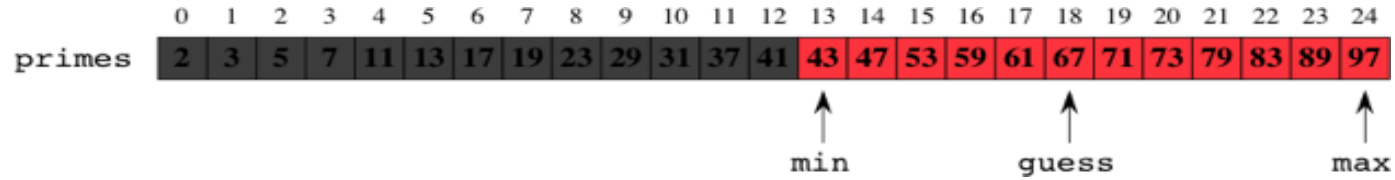


Pesquisa Binária

- O índice que estamos procurando é maior ou menor do que 12? Como os valores no vetor estão em ordem crescente, e 41 < 67, o valor 67 deve estar à direita do índice 12. Em outras palavras, o índice que estamos tentando encontrar deve ser maior do que 12. Atualizamos, o valor de min para 12 + 1, ou 13, e mantemos o max igual a 24.



- Qual é a próxima suposição de índice? A média entre 13 e 24 é 18,5, que podemos arredondar para 18, já que os índices dos vetores devem ser números inteiros. Descobrimos que primes[18] é 67.



- O algoritmo da busca binária para neste ponto, já que encontrou a resposta. Foram necessárias apenas duas tentativas em vez das 19 tentativas de que a busca linear precisaria.

Pesquisa Binária - Implementação

```
int PesquisaBinaria (int vet[], int chave, int Tam)
{
      int min = 0;     // limite inferior (o primeiro índice de vetor em C é zero)
      int max = Tam-1; // limite superior (termina em um número a menos. 0 a 9 são 10)
      int meio;
      while (min <= max)
      {
            meio = (min + max)/2;      // calcula o meio
            if (chave == vet[meio])
            {
                  return meio;          // retorna o índice do elemento procurado
            }
            if (chave < vet[meio])
            {
                  max = meio-1;
             }
             else
             {
                   min = meio+1;
             }
             return -1;            // indica que o elemento não foi encontrada
      }
}
```

Pesquisa Binária - Implementação

- **Algoritmo Recursivo**

```
int busca_binaria(int n, int inicio, int fim)
{    int meio;
     if (inicio <= fim)
     {
        meio = (inicio+fim)/2;
        if (vet[meio] == n)
                return meio;
        else
              if (vet[meio] < n)
                 return busca_binaria(n,meio+1,fim);
              else
                 return busca_binaria(n,inicio,meio-1);
     }
     else
         return -1;
}
```

Pesquisa Binária – Análise de Complexidade

- Sabemos que a busca linear em um vetor de *n* elementos pode precisar fazer até *n* suposições. Você provavelmente já tem uma ideia intuitiva de que a busca binária faz menos suposições do que a busca linear. Você pode até ter percebido que a diferença entre o pior número possível de suposições da busca linear e da busca binária se torna mais evidente conforme aumenta o tamanho do vetor.

- Vamos ver como analisar o número máximo de suposições que a busca binária faz.
  - A ideia principal é que quando a busca binária faz uma suposição incorreta, a porção do vetor que contém as suposições razoáveis se reduz pelo menos pela metade. Se a porção razoável tiver 32 elementos, então uma suposição incorreta a corta para ter no máximo 16. A busca binária divide o tamanho da porção razoável a cada suposição incorreta.
  - Se começarmos com um vetor de comprimento 8, então as suposições incorretas reduzem o tamanho da porção razoável para 4, para 2 e então para 1. Quando a porção razoável contém apenas um elemento, não ocorrem mais suposições. A suposição para a porção com 1 elemento está correta ou incorreta, e então terminamos. Desta forma, com um vetor de comprimento 8, a busca binária precisa de no máximo quatro suposições.

Pesquisa Binária – Análise de Complexidade

- O que você acha que aconteceria com um vetor de 16 elementos?
  - Se você disse que a primeira suposição eliminaria pelo menos 8 elementos, de forma que restassem no máximo 8 elementos, você está começando a entender. Então, com 16 elementos, precisamos de no máximo cinco suposições.
  - Você já deve estar começando a ver o padrão. Sempre que dobramos o tamanho do vetor, precisamos de, no máximo, mais uma suposição.
  - Suponha que precisamos de no máximo ***m*** suposições para um vetor de comprimento ***n***. Então, para um vetor de comprimento ***2n***, a primeira suposição reduz a porção razoável do vetor para o tamanho ***n***, e no máximo ***m*** suposições terminam o processo, nos dando um total de no máximo ***m+1*** suposições.
  - Vamos examinar o caso geral de um vetor de comprimento ***n***. Podemos expressar o número de suposições, no pior caso, como "o número de vezes que podemos reduzir pela metade, começando em ***n***, até obter o valor ***1***, mais um.

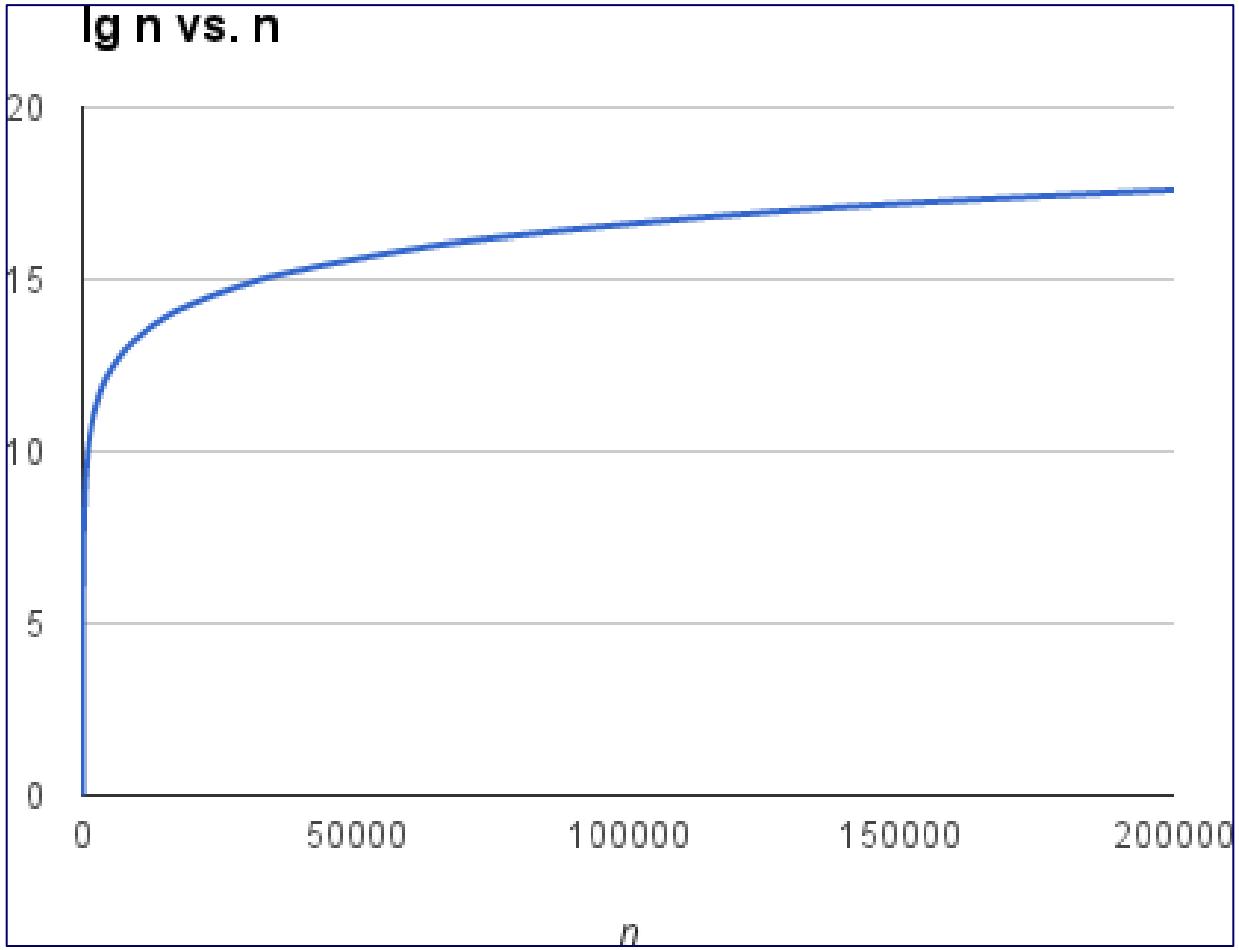
Pesquisa Binária – Análise de Complexidade

- Mas é inconveniente escrever isso por extenso.
  - Felizmente, há uma função matemática que representa o mesmo que o número de vezes que dividimos pela metade, começando em ***n***, até obtermos o valor ***1***: o ***logaritmo de base 2 de n***. Nós o escrevemos como ***lg n***.
  - Aqui está uma tabela que mostra os logaritmos de base 2 de diversos valores de n:

n	lg n
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9

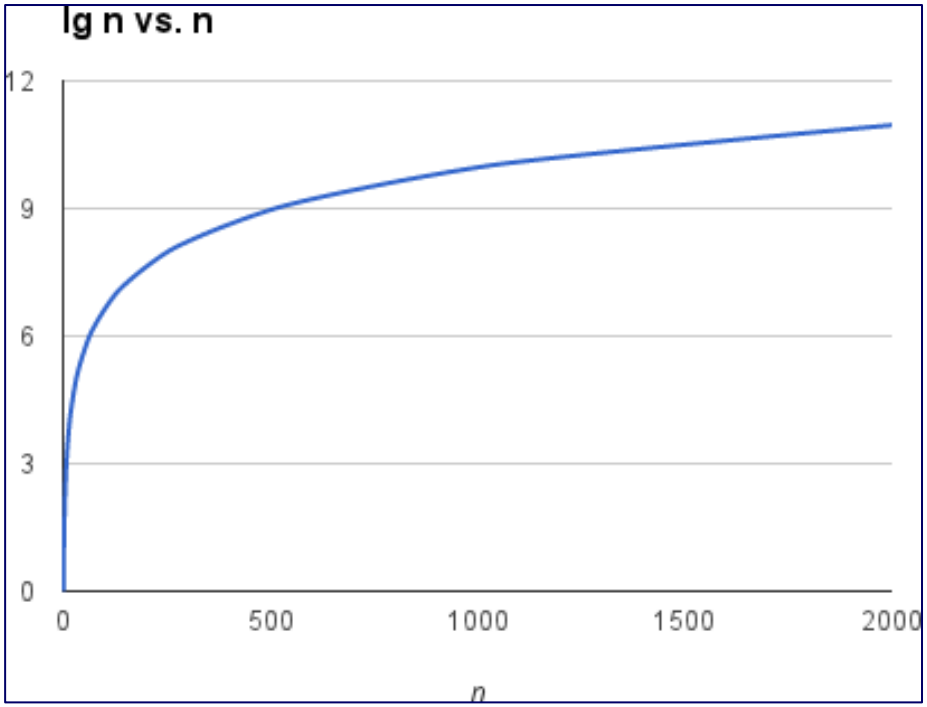
Pesquisa Binária – Análise de Complexidade

- Podemos visualizar essa mesma tabela na forma de um gráfico:



Pesquisa Binária – Análise de Complexidade

- Dando zoom nos valores mais baixos de n:

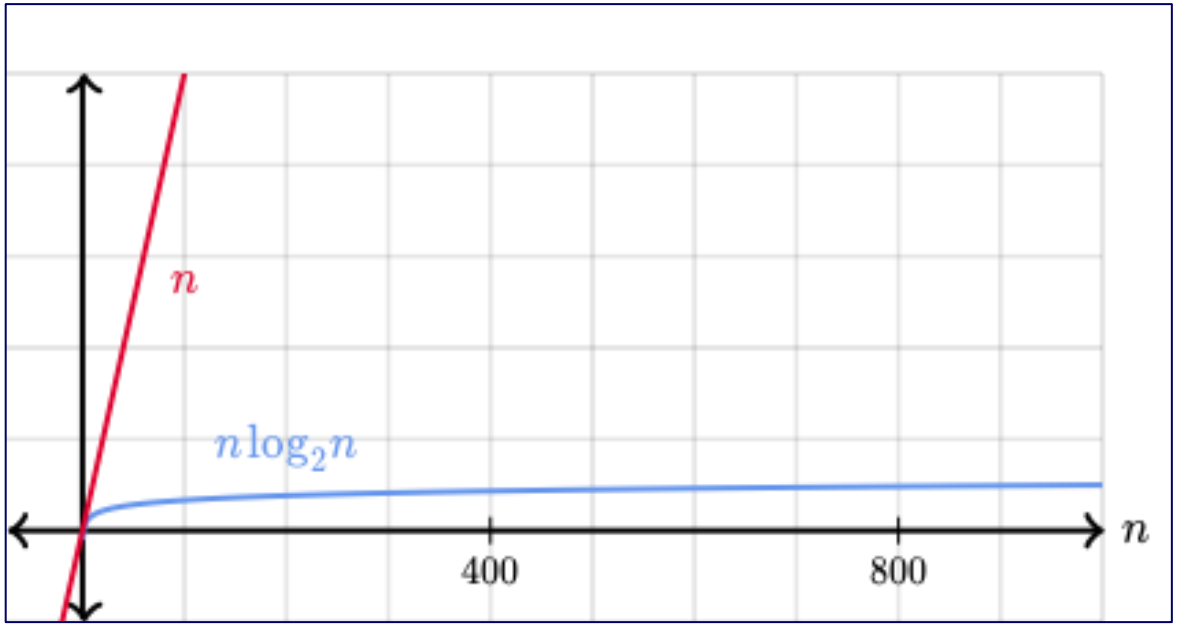


  - A função logarítmica cresce muito lentamente. Os logaritmos são o inverso dos exponenciais, que crescem muito rapidamente, então se $lg\ n = x$, então $n = 2^X$, Por exemplo, sabendo que $lg\ 128 = 7$, constatamos que $2^7 = 128$.

Pesquisa Binária – Análise de Complexidade

- Quando ***n*** não é uma **potência de 2**, podemos simplesmente ir para a próxima maior potência de 2. Para qualquer vetor de comprimento 1000, a próxima maior potência de 2 é 1024, que é igual a 2^{10}. Portanto, para um vetor de 1000 elementos, a busca binária precisaria de, no máximo, 11 (10 + 1) suposições.

  – A Busca Binária é muito melhor que a busca linear! Compare as duas no próximo gráfico.



Pesquisa Hash

- **Definição**
  - Na apresentação da estrutura de tabelas (vetores de tipos não primitivos), a busca por uma chave ocorre sempre através de comparações, utilizando Pesquisa Linear/Sequencial ou se os dados estiverem ordenados, é possível utilizar a Pesquisa Binária.
  - Uma alternativa de busca em tabelas dá-se através do cálculo da posição que uma chave ocupa na tabela através de uma função. Esse tipo de função que mapeia um símbolo para valores inteiros é denominado função ***hash*** (função de espalhamento) e o tipo de tabela manipulada dessa forma é uma **tabela *hash***.
  - A grande vantagem na utilização da **tabela *hash*** está no desempenho, enquanto a Pesquisa Linear tem complexidade temporal ***O(n)*** e a busca binária tem complexidade ***O(log n)***, o tempo de busca na **tabela *hash*** é praticamente independente do número de chaves armazenadas na tabela, ou seja, tem complexidade temporal ***O(1)***. Aplicando a **função *hash*** no momento de armazenar e no momento de buscar a chave, a busca pode se restringir diretamente àquela posição da tabela gerada pela função.

Pesquisa Hash

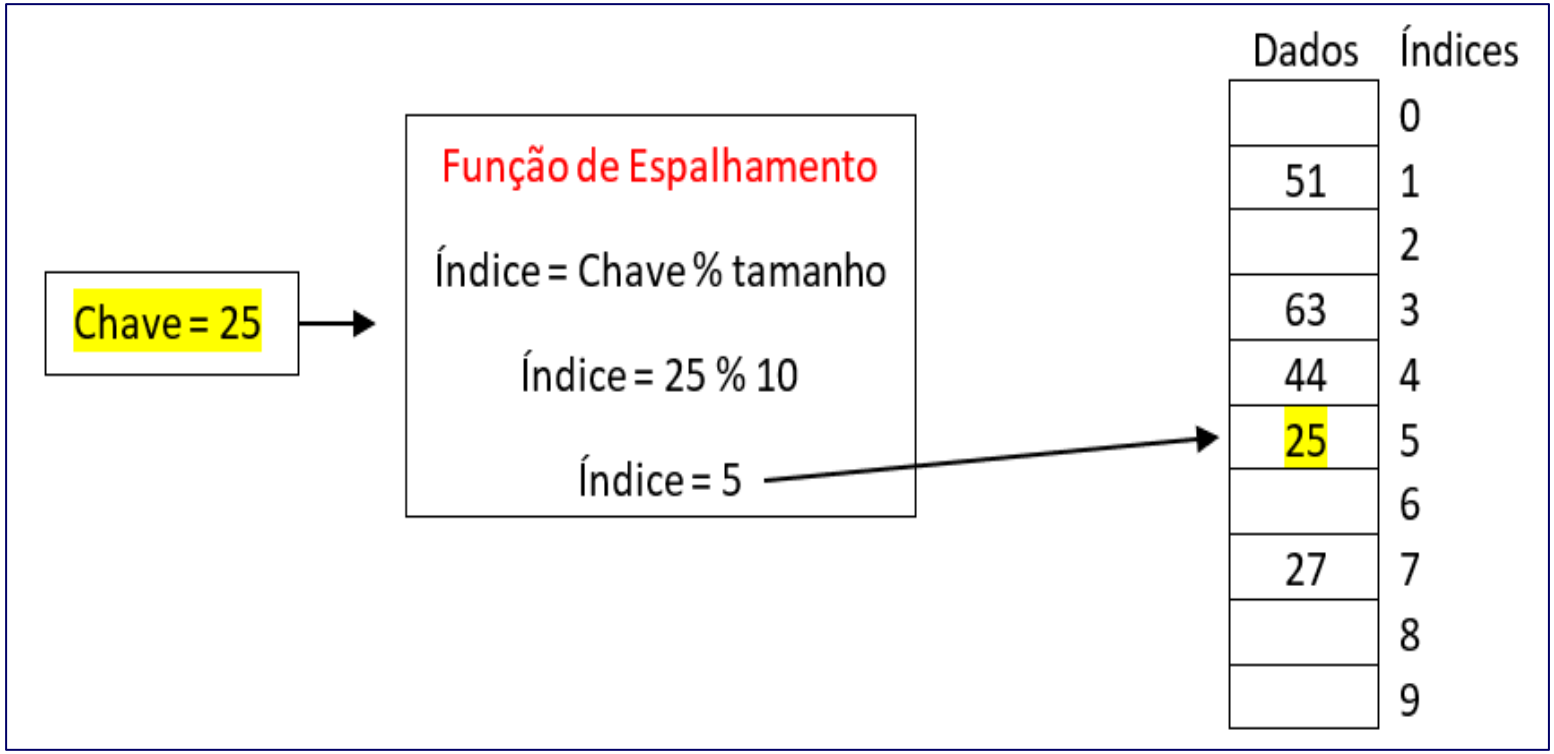
- Idealmente, cada chave processada por uma ***função hash*** geraria uma posição diferente na tabela. No entanto, na prática existem sinônimos (chaves distintas que resultam em um mesmo valor de ***hashing***). Quando duas ou mais chaves sinônimas são mapeadas para a mesma posição da tabela, diz-se que ocorre uma colisão.

- Uma boa ***função hash*** deve apresentar duas propriedades básicas: seu cálculo deve ser rápido e deve gerar poucas colisões. Além disso, é desejável que ela leve a uma ocupação uniforme da tabela para conjuntos de chaves quaisquer.

- Um exemplo de uma ***função hash*** simples seria a Divisão. Nesse tipo de função, a chave é interpretada como um valor numérico que é dividido por um valor ***M***. O resto dessa divisão inteira, um valor entre ***0*** e ***M-1***, é considerado o endereço em uma tabela de ***M*** posições. Para reduzir colisões, é recomendável que ***M*** seja um número primo.

Pesquisa Hash – Função de Espalhamento

- Exemplo de Função de Espalhamento:



Pesquisa Hash – Tratamento de Colisões

- O processamento de **tabelas hash** demanda a existência de algum mecanismo para o tratamento de colisões. As formas mais usuais de tratamento de colisão são por **endereçamento aberto** ou por **encadeamento**.

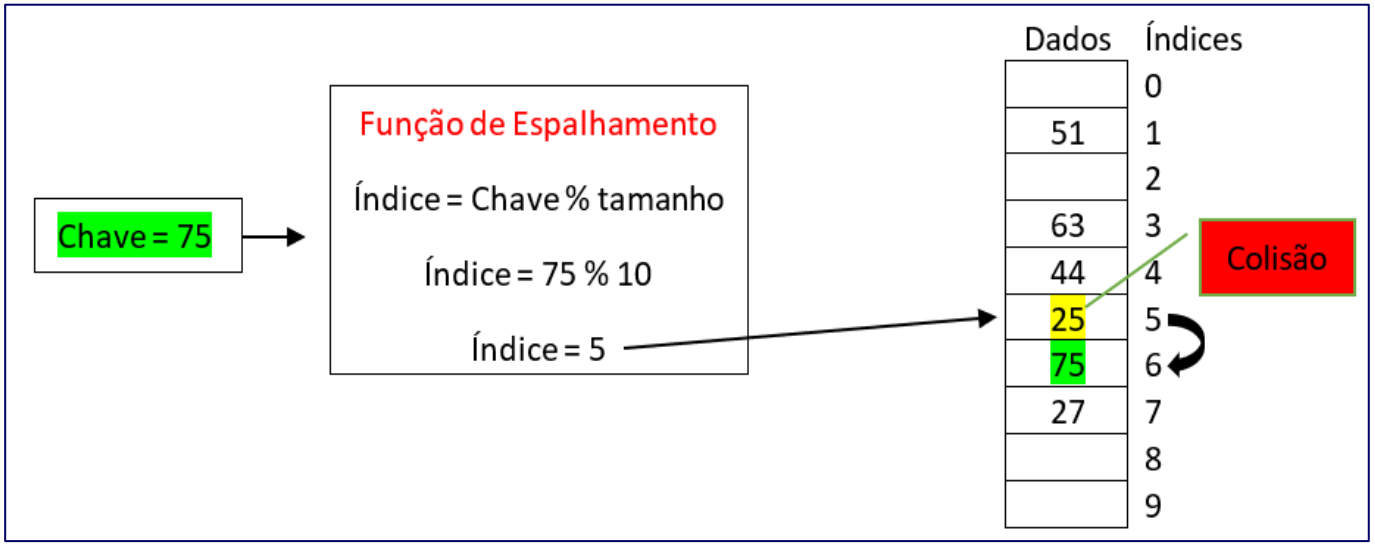
- Na técnica de tratamento de colisão por **endereçamento aberto**, a estratégia é utilizar o próprio espaço da tabela que ainda não foi ocupado para armazenar a chave que gerou a colisão.

- Quando a função hash gera para uma chave, uma posição que já está ocupada, o procedimento de armazenamento verifica se a posição seguinte também está ocupada; se estiver ocupada, verifica a posição seguinte e assim por diante, até encontrar uma posição livre.

- Nesse tipo de tratamento, considera-se a tabela como uma estrutura circular, onde a primeira posição sucede a última posição. A entrada é então armazenada nessa posição. Se a busca termina na posição inicialmente determinada pela função hash, então a capacidade da tabela está esgotada e uma mensagem de erro deve ser gerada.

Pesquisa Hash – Tratamento de Colisões

- Exemplo de tratamento de colisão com **Endereçamento Aberto**.

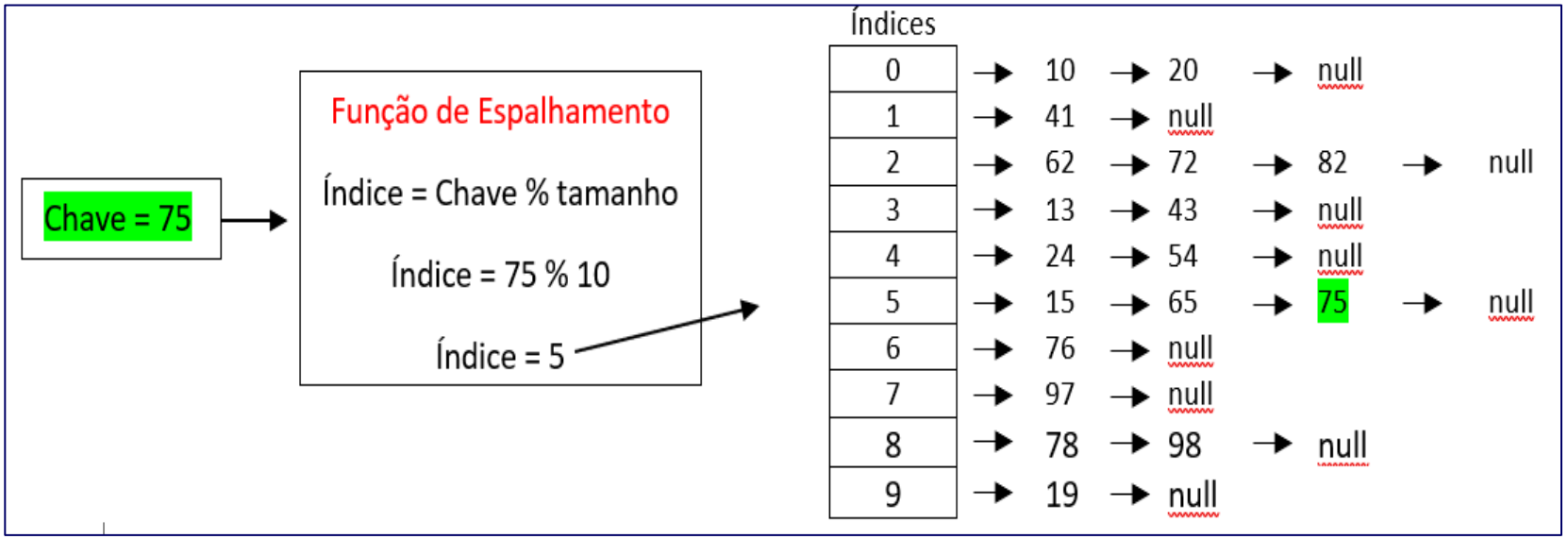


  - No momento da busca, essa varredura da tabela pode ser novamente necessária. Se a chave buscada não está na posição indicada pela função *hashing* e aquela posição está ocupada, a chave pode eventualmente estar em outra posição na tabela. Assim, é necessário verificar se a chave não está na posição seguinte. Se, por sua vez, essa posição estiver ocupada com outra chave, a busca continua na posição seguinte e assim por diante, até que se encontre a chave procurada ou uma posição sem nenhum símbolo armazenado.

Pesquisa Hash – Tratamento de Colisões

- No caso do **tratamento de colisões por encadeamento**, requer que seja utilizada uma outra estrutura para armazenar os elementos que colidirem, tais como, lista encadeada, árvore binária, árvore AVL, entre outras.

- Exemplo de tratamento de colisões por encadeamento utilizando Lista Encadeada.



Pesquisa Hash – Implementação

```
#include<stdio.h>
int funcaoEspalhamento(int);
int insere(int, int []);
int consulta(int, int []);
void listagem(int []);

main()
{ int n, op, tabela[10],i, indice;
  // inicializa a tabela com -1
  for (i=0; i < 10; i++)
  {
      tabela[i]=-1;
  }
  Do  // Menu
  {
      printf("\n *** Menu ***");
      printf("\n [1] Inserir");
      printf("\n [2] Consultar");
      printf("\n [3] Lista");
      printf("\n [4] Fim\n");
      printf("\n Qual sua opcao ? ");
      scanf("%d",&op);
```

```
switch (op)
{   case 1: printf("\n Informe um número p/ inserir:");
            scanf("%d",&n);
            indice = insere(n, tabela);
            if (indice != -1)
            {   printf("\n Elemento inserido %d \n",indice);
            }
            else
            {   printf("\n Tabela sem espaço \n");
            }
       break;
       case 2: printf("\n Informe o numero p/ consultar:");
               scanf("%d",&n);
               indice = consulta(n, tabela);
               if (indice != -1)
               {    printf("\n Elemento encontrado %d",indice);
               }
               else
               {     printf("\n Elemento não encontrado \n");
               }
        break;
        case 3: printf("\n Listagem da Tabela Hash : \n");
                listagem(tabela);
        break;
    }
 } while (op != 4);   }
```

Pesquisa Hash – Implementação

```
int funcaoEspalhamento(int n)
{
        int indice;
        indice = n % 10;
        return indice;
}
```

```
void listagem(int tab[])
{   int i;
    printf("\n Indice   Dado");
    for (i=0; i < 10; i++)
    {
        printf("\n  %d     %d",i,tab[i]);
    }
}
```

```
int consulta(int n, int tab[])
{
  int indice,i=0;
  indice = funcaoEspalhamento(n); //calcula o índice da chave
  if (tab[indice] == n)   // achou o número
  {
     return indice;  // retorna o índice que encontrou o número
  }
  else   // Deu colisão
  {
     i=(indice+1)%10;   // vai para o próximo índice
     while ((i != indice) && (tab[i] != n))  // procura na tabela
     {
       i=(i+1)%10;  // caminha na tabela
     }
     if (i == indice)
     {
       return -1; // indica que não encontrou
     }
     else
     {
       return i;  // retorna o índice que encontrou o número
     }
  }
}
```

Pesquisa Hash – Implementação

```
int insere(int n, int tab[])
{   int indice,i=0;
    indice = funcaoEspalhamento(n); //calcula o índice para inserção
    if ((tab[indice] == -1) || (tab[indice] == n))  // insere sem colisão
    {
        tab[indice] = n;
        return indice;  // retorna o índice de inserção
    }
    else   // Deu colisão
    {
        i=(indice+1)%10;   // vai para o próximo índice
        while ((i != indice) && (tab[i] != -1))  // procura espaço vazio na tabela
        {
            i=(i+1)%10;  // caminha na tabela
        }
        if (i == indice)
        {
             return -1; // indica que não tem espaço na tabela
        }
        else
        {
             tab[i] = n;  // insere em um local vago na tabela
             return i;  // retorna o índice de inserção apesar da colisão
        }
    }
}
```

Árvores

- **Definição:**
  - Relação de hierarquia ou de composição entre os dados (***nós***).
  - Conjunto finito T de um ou mais nós, tais que:

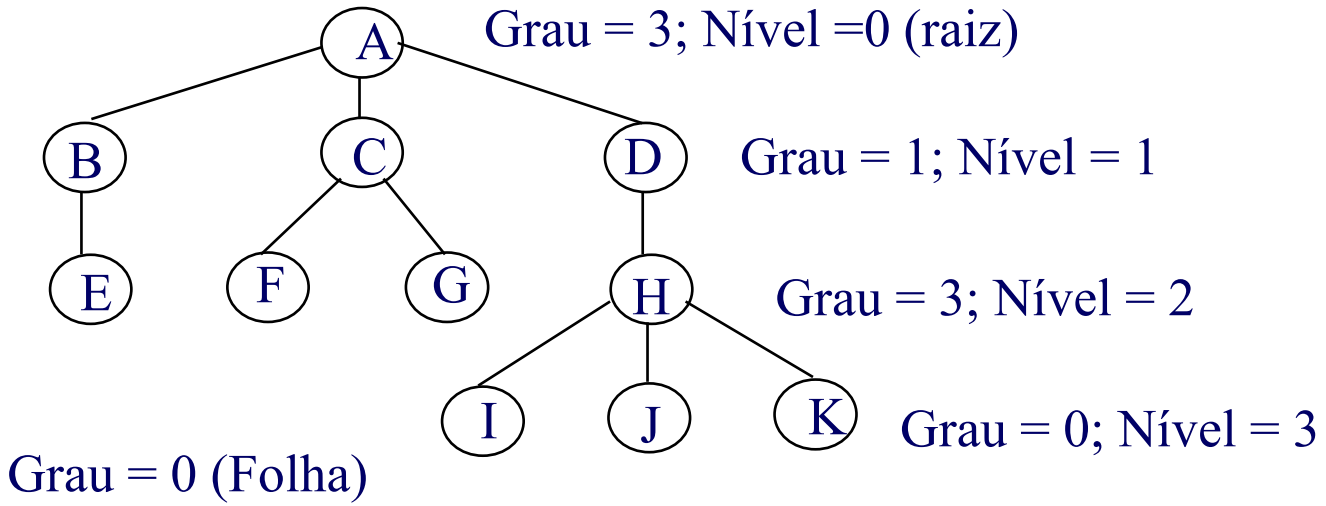
    (a) existe um nó denominado ***raiz*** da árvore;

    (b) os demais nós formam m >= 1 conjuntos disjuntos S1,...,Sm, onde cada um desses conjuntos é uma árvore.
  - As árvores Si recebem a denominação de ***Sub-árvores.***

- **Terminologia:**
  - Cada nó da árvore é a raiz de uma Sub-árvore.
  - O número de Sub-árvores de um nó é o grau daquele nó.
  - Um nó de grau igual a zero é denominado folha ou nó terminal.
  - A raiz da árvore tem nível 0.
  - Os demais nós: nível = número de "linhas" que o liga à raiz.
  - Altura: nível mais alto da árvore.

Árvores

- **Representação Estrutural:**



Árvore com altura igual a 3.

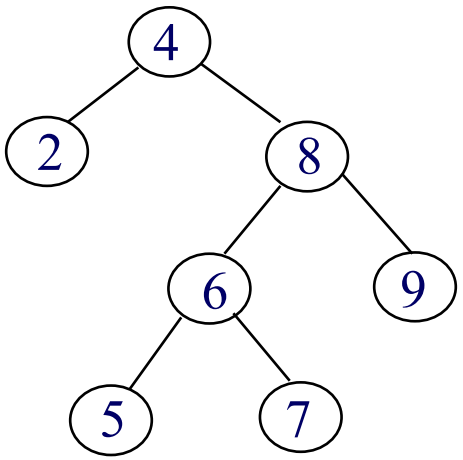
Árvores Binárias

- **Definição:**
  - Uma árvore binária é uma estrutura de dados útil quando precisam ser tomadas decisões bidirecionais em cada ponto de um processo.
  - O Grau de cada nó é menor ou igual a 2 (Sub-árvores da esquerda e da direita).
  - Se grau = 1, deve ser especificado se a sua Sub-árvore é a da esquerda ou a da direita.
  - **Árvore Estritamente Binária**: é a árvore onde todo o nó que não é folha possuí Sub-árvores a esquerda e a direita.
  - Uma **árvore binária completa** é uma árvore estritamente binária sendo que todas as folhas devem estar no mesmo nível.

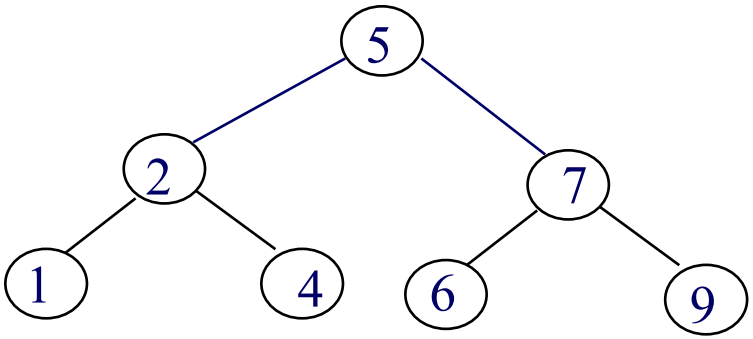
Árvores Binárias

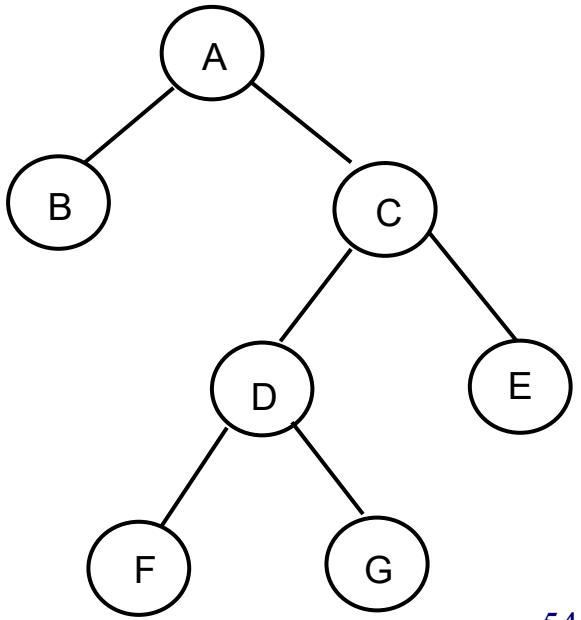
- **Representação Estrutural:**

Árvore Estritamente Binária



Árvore Binária Completa





- **Como construir uma árvore?**
- **Como percorrer uma árvore ?**
- **Aplicações de árvores binárias**
- **Exemplos:**

Árvores Binárias - Percurso

- **A natureza recursiva de uma árvore binária:**
  - Existem três métodos recursivos para que possamos percorrer uma árvore passando por todos os seus elementos:

  - **Em Pré-ordem**

1º. visitamos a raiz
2º. Sub-árvore esq. em pré-ordem (Centro, Esquerda, Direita)
3º. Sub-árvore dir. em pré-ordem

  - **Em Ordem**

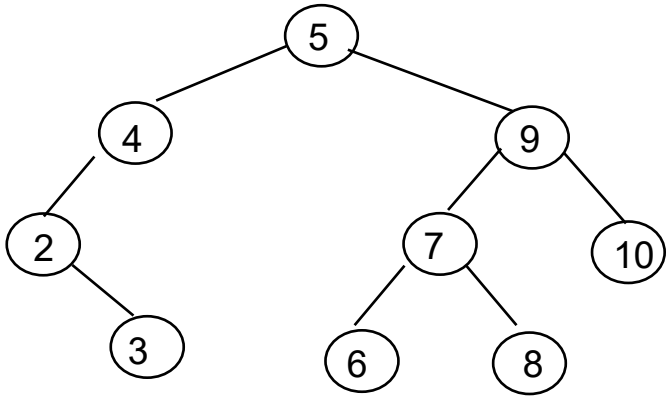
1º. Sub-árvore esq. em ordem
2º. visitamos a raiz (Esquerda, Centro, Direita)
3º. Sub-árvore dir. em ordem.

  - **Em Pós- Ordem**

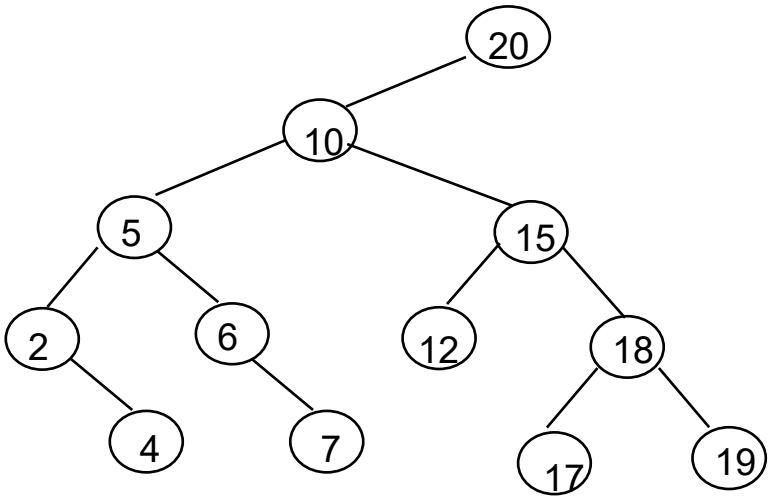
1º. Sub-árvore esq. em pós-ordem
2º. Sub-árvore dir. em pós-ordem (Esquerda, Direita, Centro)
3º. Visitamos a raiz

Árvores Binárias - Percurso

- **Exemplos:**



- Pré-ordem: 5, 4, 2, 3, 9, 7, 6, 8, 10
- Em ordem: 2, 3, 4, 5, 6, 7, 8, 9, 10
- Pós-ordem: 3, 2, 4, 6, 8, 7, 10, 9, 5

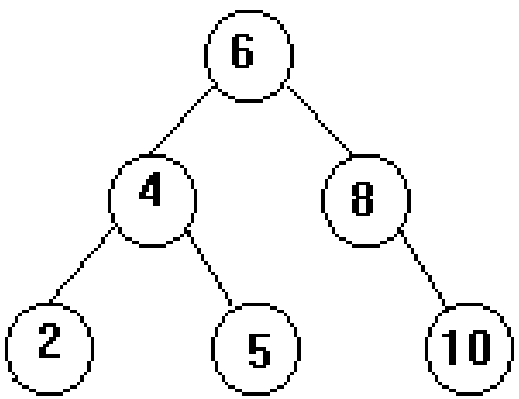


- Pré-ordem: 20, 10, 5, 2, 4, 6, 7, 15, 12, 18, 17, 19
- Em ordem: 2, 4, 5, 6, 7, 10, 12, 15, 17, 18, 19, 20
- Pós-ordem: 4, 2, 7, 6, 5, 12, 17, 19, 18, 15, 10, 20

Árvores Binárias de Pesquisa

- **Regra Geral de Inserção:**
  - Os valores menores devem ficar a esquerda da raiz e os maiores a direita.
  - Os valores repetidos não devem ser inseridos.
  - As inserções sempre são feitas nas folhas, dessa forma, deve se percorrer a árvore até encontrar a folha que será o pai do novo elemento a ser inserido.
  - O percurso é baseado no valor da informação que está sendo inserida. Se o novo elemento for menor que o nó comparado, deve andar para a esquerda, caso contrário deve andar para a direita.

- **Exemplo**



- **Exercício:**
  - Crie uma árvore com os seguintes nós:

    14, 15, 4, 9, 7, 18, 2, 5, 16, 4, 20, 17, 9, 5.

Árvores Binárias de Pesquisa

```
#include <stdio.h>

typedef struct nodo
{  int         info;
   struct nodo *pai;
   struct nodo *f_esq;
   struct nodo *f_dir;
   char        deletado;
} T_nodo;

typedef struct arvore
{  struct nodo *raiz;
   int         qtd;
} T_arvore;

T_arvore *parvore;
```

```
T_arvore *inicializa_arvore();
T_nodo *cria_nodo(int, T_nodo *);
void insere(int, T_arvore *);
T_nodo *consulta(int, T_arvore *);
void ordem(T_nodo *);
void pre_ordem(T_nodo *);
void pos_ordem(T_nodo *);
void retira(int, T_arvore *);
```

```
T_arvore *inicializa_arvore()
{  T_arvore *nova_arvore;
   nova_arvore=(T_arvore *) malloc(sizeof(struct arvore));
   if (nova_arvore == NULL)
   {  printf("\n Memoria insuficiente para alocar estrutura");
      exit(1);
   }
   nova_arvore->raiz=NULL;
   nova_arvore->qtd=0;
   return(nova_arvore);
}
```

Árvores Binárias de Pesquisa

```
main()
{  char opcao;  int informacao=0;
   parvore = inicializa_arvore();
   do
   {  printf("\n (I)ncluir (C)onsultar (R)emover (O)rdem  Pr(E)-Ordem  Po(s)-Ordem  (F)im: ");
      scanf("%c",&opcao);
      if ((opcao == 'I')||(opcao== 'i'))
      {  printf("\n Entre com a informacao : ");
         scanf("%d",&informacao);
         insere(informacao, parvore);
      }
      else if ((opcao == 'C')||(opcao=='c'))
      {  printf("\n Entre com a informacao : ");
         scanf("%d",&informacao);
         consulta(informacao,parvore);
      }
      else if ((opcao == 'O')||(opcao == 'o'))
      {  ordem(parvore->raiz);
      }
      else if ((opcao == 'E')||(opcao == 'e'))
      {  pre_ordem(parvore->raiz);
      }
      else if ((opcao == 'S')|| (opcao == 's'))
      {  pos_ordem(parvore->raiz);
      }
      else if ((opcao == 'R')|| (opcao == 'r'))
      {  printf("\n Entre com a informacao : ");
         scanf("%d",&informacao);
         retira(informacao, parvore);
       }
       getchar();
   } while ((opcao != 'F') && (opcao != 'f'));
}
```

Árvores Binárias de Pesquisa


```
T_nodo *cria_nodo(int n, T_nodo *p)
{  T_nodo *novo;
   novo=(T_nodo *) malloc(sizeof(struct nodo));
   if (novo == NULL)
   {  printf("\n Memoria insuficiente para alocar estrutura");
      exit(1);
   }
   novo->info=n;
   novo->pai=p;
   novo->f_esq=NULL;
   novo->f_dir=NULL;
   novo->deletado='f';
   return(novo);
}
```

```
void pre_ordem(T_nodo *sub_raiz)
{  if (sub_raiz != NULL)
   {  if (sub_raiz->deletado == 'V')
         printf("\n * %d",sub_raiz->info);
      else
         printf("\n %d",sub_raiz->info);
      pre_ordem(sub_raiz->f_esq);
      pre_ordem(sub_raiz->f_dir);
   }
}
```

Árvores Binárias de Pesquisa

```
void insere(int n, T_arvore *parvore)
{  T_nodo *p, *aux;
   if (parvore->raiz == NULL)
   {   parvore->raiz = cria_nodo(n,NULL);
   }
   else
   {  p=parvore->raiz;
      aux=parvore->raiz;
      while (n != p->info && aux != NULL)
      {  p=aux;
                  if (n < p->info)
                     aux = p->f_esq;
                  else
                     aux = p->f_dir;
      }
      if (n == p->info)
      {  printf("\n Numero Repetido");
         return;
      }
```

```
      else if (n < p->info)
      {  p->f_esq=cria_nodo(n, p);
      }
      else
      {  p->f_dir=cria_nodo(n, p);
      }
   }
   parvore->qtd++;
}
```

```
void ordem(T_nodo *sub_raiz)
{  if (sub_raiz != NULL)
   {  ordem(sub_raiz->f_esq);
      if (sub_raiz->deletado == 'V')
         printf("\n *%d",sub_raiz->info);
      else
         printf("\n %d",sub_raiz->info);
      ordem(sub_raiz->f_dir);
   }
}
```

ARTUR KRONBAUER

Árvores Binárias de Pesquisa

```
void retira(int n, T_arvore *parvore)
{ T_nodo *aux=parvore->raiz, *remover=parvore->raiz;
  if (parvore->raiz == NULL)
  {  printf("\n Arvore sem elementos");
  }
  else
  {  while ((aux != NULL) && (n != remover->info))
     {    remover = aux;
          if (n < remover->info)
              aux = remover->f_esq;
          else
              aux = remover->f_dir;
     }
     if (n == remover->info)
     {  remover->deletado='V';  // remove logicamente
        parvore->qtd--;
```

Continuação na próxima transparência

Árvores Binárias de Pesquisa

```
        while ((remover != NULL) && (remover->deletado == 'V'))
        {    if ((remover->f_esq == NULL) && (remover->f_dir == NULL))
             {   if (remover != parvore->raiz)
                 {  aux=remover->pai;         // remove fisicamente um nodo
                    if (aux->f_esq == remover)
                         aux->f_esq=NULL;
                    else
                         aux->f_dir=NULL;
                 }
                 else
                 {   parvore->raiz=NULL;       // remove fisicamente a raiz
                 }
                 aux = remover;
                 free(aux);
             }
             remover=remover->pai
        }
}
else
{       printf("\n Elemento nao encontrado");
}
```

Árvores Binárias de Pesquisa

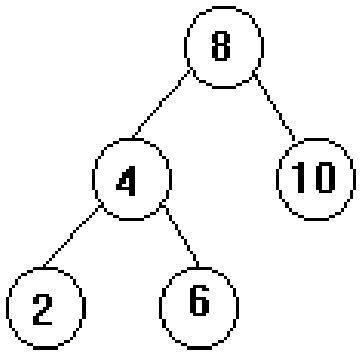
```
T_nodo *consulta(int n, T_arvore *parvore)
{  T_nodo *p, *aux;
   p=parvore->raiz;
   aux=parvore->raiz;
   while ((n != p->info) && (aux != NULL))
   {  p=aux;
      if (n < p->info)
            aux = p->f_esq;
      else
            aux = p->f_dir;
   }
   if (n == p->info)
   { if (p->deletado != 'V')
        printf("\n Informacao Existente");
      else
        printf("\n Informacao removida logicamente");
      return(p);
   }
   else
    {   printf("\n Informacao Inexistente");
        return(NULL);
    }
}
```

```
void pos_ordem(T_nodo *sub_raiz)
{  if (sub_raiz != NULL)
   {  pos_ordem(sub_raiz->f_esq);
      pos_ordem(sub_raiz->f_dir);
      if (sub_raiz->deletado == 'V')
          printf("\n *%d",sub_raiz->info);
      else
          printf("\n %d",sub_raiz->info);
   }
}
```

Árvores AVL

- **Definição:**
  - Uma árvore AVL é uma árvore binária de busca construída de tal modo que a altura de sua Sub-árvore direita difere da altura da Sub-árvore esquerda de no máximo 1.

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



  - Nós 9 ou 11 podem ser inseridos sem balanceamento . Sub-árvore com raiz 10 passa a ter uma Sub-árvore e Sub-árvore com raiz 8 vai ficar melhor balanceada !
  - Inserção dos nós 1, 3, 5 ou 7 requerem que a árvore seja rebalanceada!

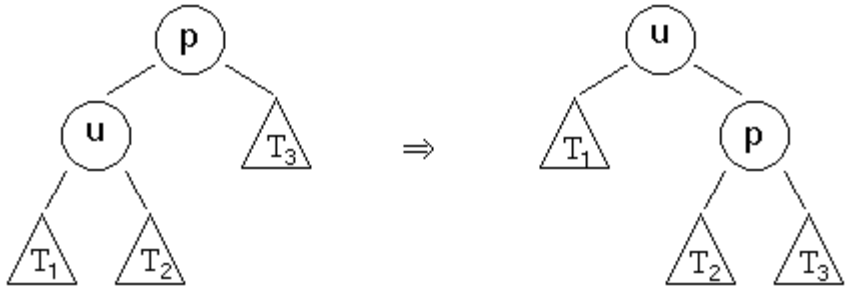
- **Fator de Balanceamento de um nó:**
  - É a altura da Sub-árvore direita do nó menos a altura da Sub-árvore esquerda do nó .

FB= altura direita - altura esquerda

Se todos os FB forem [-1, 0, 1] a árvore está balanceada.

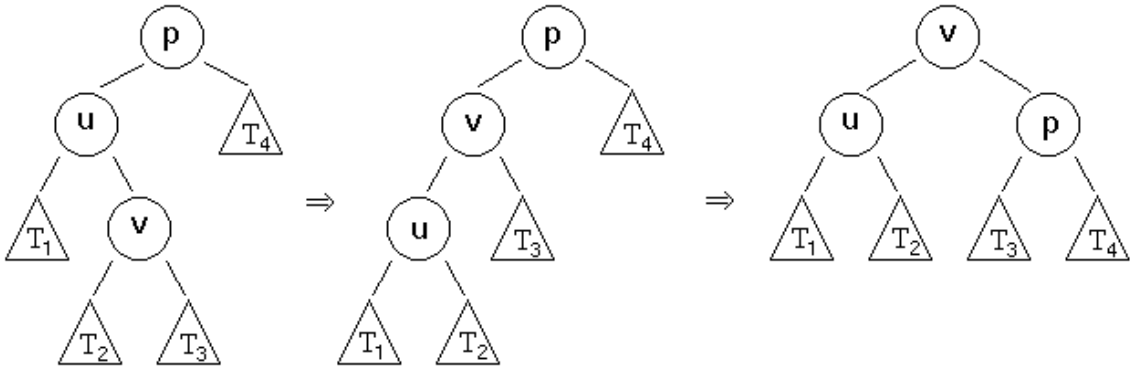
Árvores AVL

- **Rebalanceamento:**
  - Nos casos abaixo considere **P** como sendo o nó raiz de uma Sub-árvore desbalanceada e **U** como sendo o nó filho dessa raiz.
  - **Caso 1:** Altura Esquerda de P > Altura Direita de P
    - **Caso 1.1 :** Altura Esquerda de U > Altura Direita de U



Rotação a direita

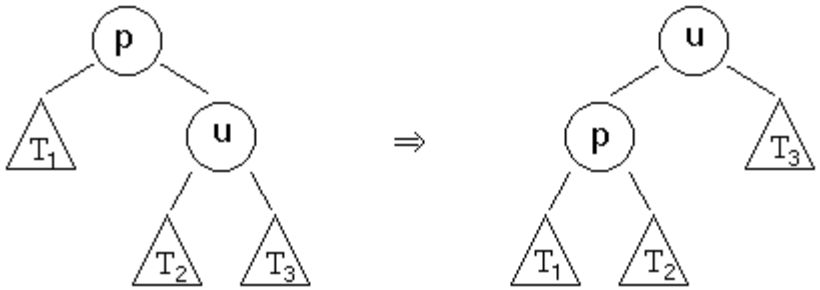
- **Caso 1.2 :** Altura Esquerda de U < Altura Direita de U



Rotação para a esquerda e em seguida para a direita

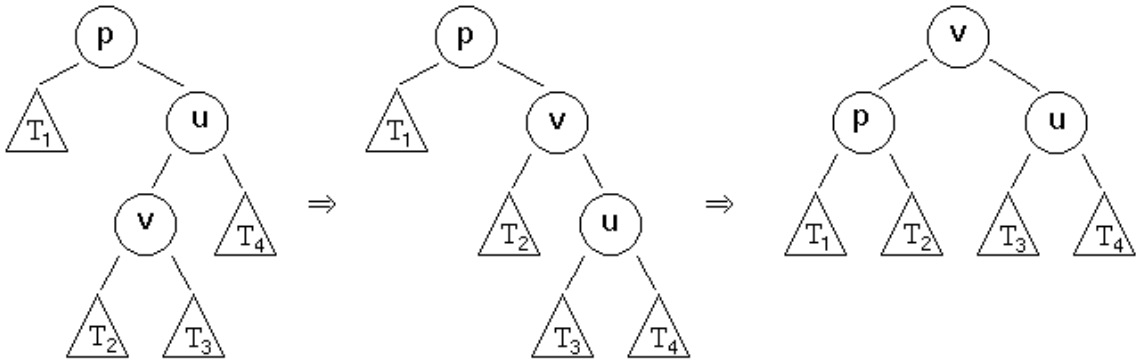
Árvores AVL

- **Caso 2:** Altura Direita de P > Altura Esquerda de P
  - **Caso 1.2:** Altura Direita de U > Altura Esquerda de U



Rotação a esquerda

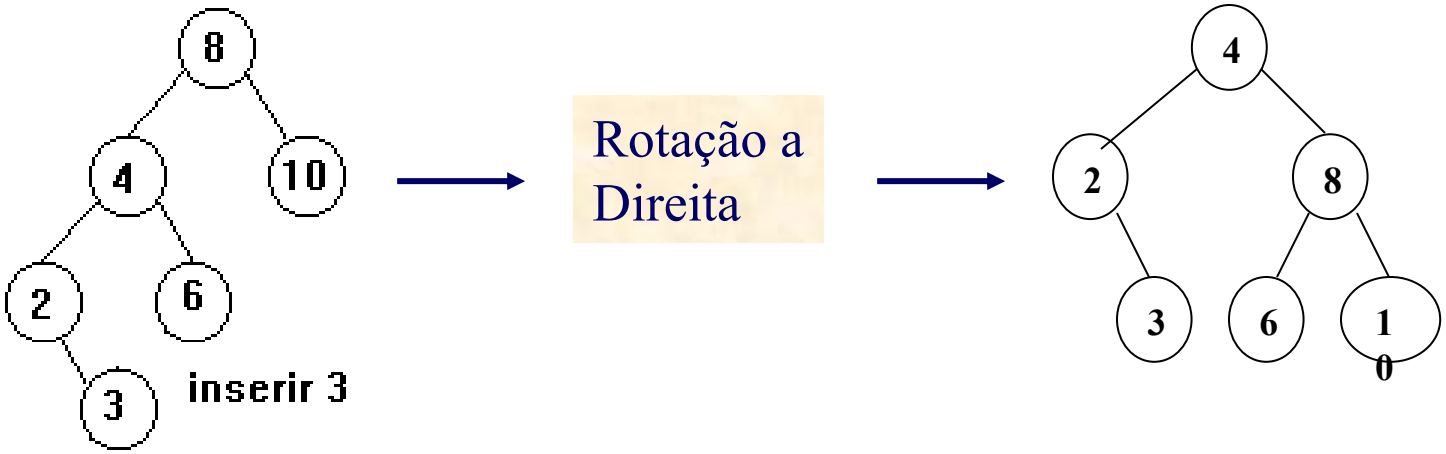
  - **Caso 2.2 :** Altura Esquerda de U > Altura Direita de U



Rotação para a direita e em seguida para a esquerda

Árvores AVL

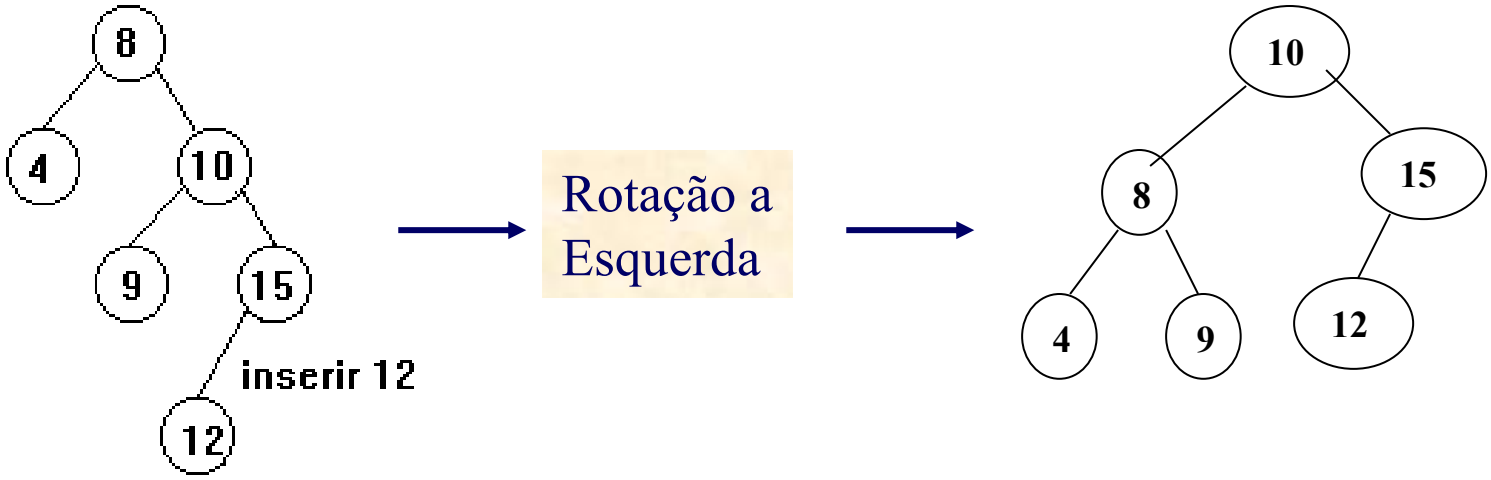
- **Exemplos de Rotações (Rotação simples a direita):**



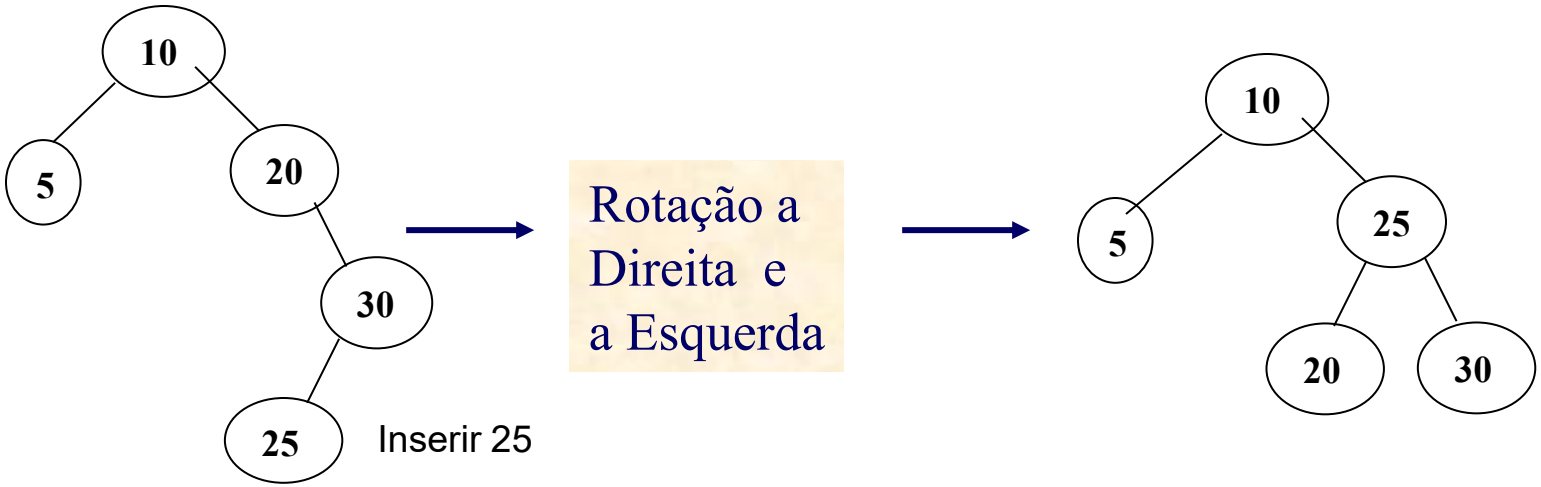
- **Exemplos de Rotações (Rotação dupla a direita):**

Árvores AVL

– **Exemplos de Rotações (rotação simples a esquerda) :**



– **Exemplos de Rotações (Rotação dupla a esquerda):**



Árvores AVL

- **Regras para as rotações:**
  - **Rotação Simples a Direita:** rotaciona o U sobre o P para a direita, sendo que o filho da direita de U passa a ser o filho da esquerda de P.
  - **Rotação Simples a Esquerda:** rotaciona o U sobre o P para a esquerda, sendo que o filho da esquerda de U passa a ser o filho da direita de P.
  - **Rotação Dupla a Direita:** rotaciona o V sobre U para a esquerda, sendo que o filho da esquerda de V passa a ser o filho da direita de U. Após, rotaciona V sobre P para a direita, sendo que o filho da direita de V passa a ser o filho da esquerda de P.
  - **Rotação Dupla a Esquerda:** rotaciona o V sobre U para a direita, sendo que o filho da direita de V passa a ser o filho da esquerda de U. Após, rotaciona V sobre P para a esquerda, sendo que o filho da esquerda de V passa a ser o filho da direita de P.

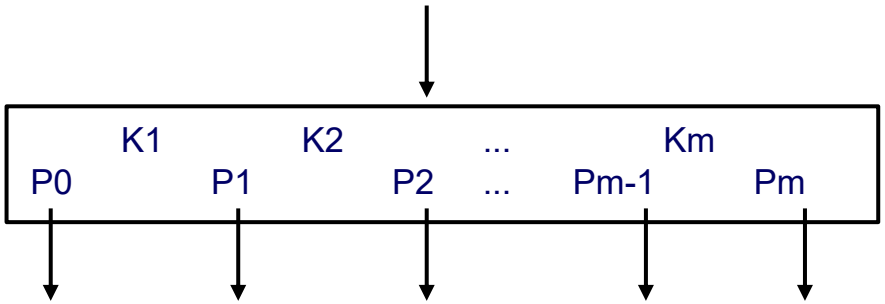
Árvores - B

- **Definição:**
  - É a Construção e manutenção de árvores de busca de grandes dimensões.
  - Ponteiros referem-se a áreas de memória secundária, em vez de representarem endereços da memória principal.
  - Busca: acesso a disco (com os inerentes atrasos de acesso).
  - Sub-árvores representadas em unidades, do ponto de vista de acesso
    ⟶ ***páginas***
  - Reduz o número de acessos ao disco.
  - Necessita de esquema de crescimento controlado.
  - Todo nó, exceto a raiz, deve possuir entre ***n*** e ***2n*** chaves, para uma dada constante ***n***.

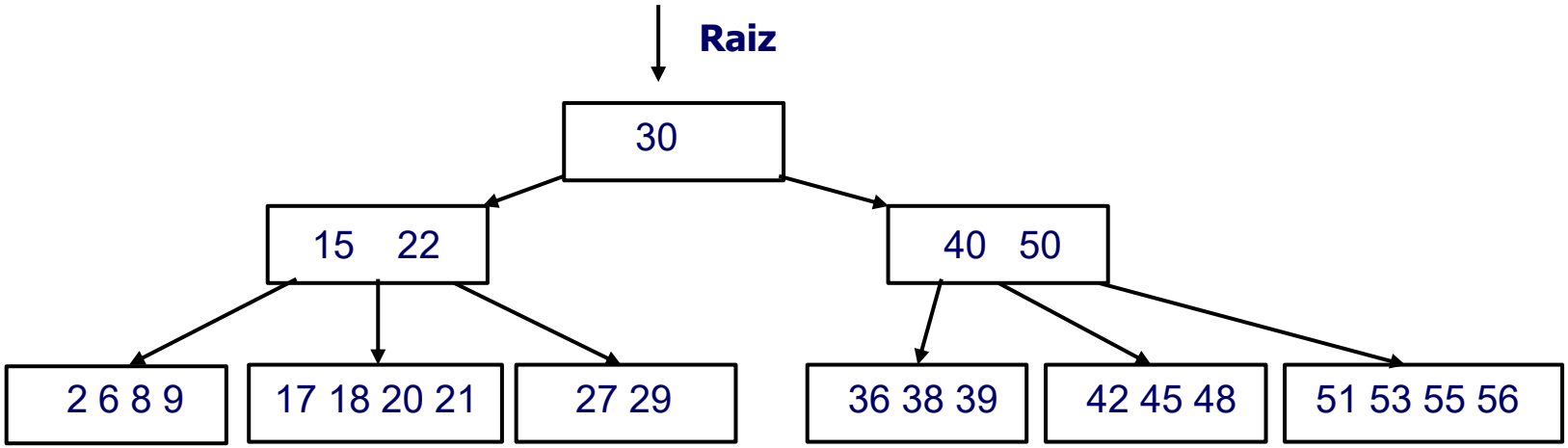
- **Características:**
  - Cada página (nó) contém no máximo, **2n** elementos (chaves);
  - cada página, exceto a que contém a raiz, contém no mínimo **n** chaves;
  - os nós chamados folhas não têm descendentes e os demais(nós de derivação) possuem **m + 1** descendentes, onde **m** é a quantidade de chaves;
  - todas as folhas têm o mesmo nível.

Árvores - B

- **Representação Estrutural:**



- **Exemplo: Árvore B (ordem 2):**



Árvores B

- **Regras para redistribuir chaves:**

1) Redistribuir a chave mais a esquerda de uma página que extrapolou os limites da ordem. Suba o elemento mais a esquerda para a página pai e desça o elemento de ligação para a página irmã esquerda.

   - Caso não seja possível executar a redistribuição porque a página irmão está cheia ou não exista, passe para a segunda regra.

2) Redistribuir a chave mais a direita de uma página que extrapolou os limites da ordem. Suba o elemento mais a direita para a página pai e o elemento de ligação para a página irmã direita.

   - Caso não seja possível executar a redistribuição porque a página irmão está cheia ou não exista, passe para a terceira regra.

3) Quebre em duas a página que extrapolou os limites da ordem e suba o elemento do meio para a página pai.