

Estrutura de Dados II

Professora: Inés Restovic



UNEB

UNIVERSIDADE DO
ESTADO DA BAHIA
SISTEMAS DE INFORMAÇÃO

A decorative network diagram at the top of the slide, featuring a series of interconnected nodes and lines. A central node is highlighted with a blue double quote symbol (").

“

© *Para construir os sistemas de software vitais do futuro, precisamos ser capazes de manipular a informação efetivamente e eficientemente.*

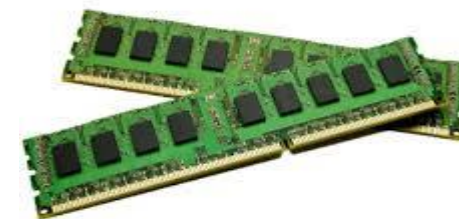
© *Alan L. Tharp*

Organização de Arquivos

Como Armazenar os Dados?

◎ Sistema de memória

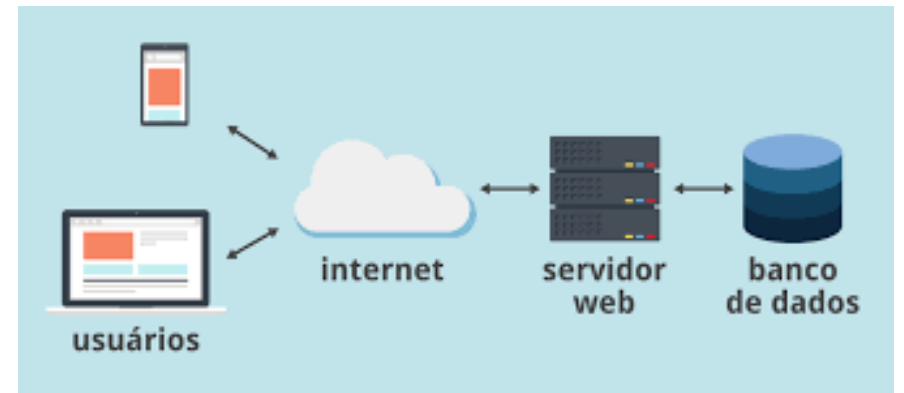
- Provê solução para a execução de processos
 - Disponibiliza espaço para a manipulação de dados
 - Mantidos enquanto processo existe
- Não atende a necessidade de todas aplicações
 - Pode-se requerer além da capacidade da memória virtual
 - Exemplo: Um sistema corporativo
 - Pode-se precisar garantir persistência dos dados quando processo termina
 - Exemplo: Um banco de dados
 - Pode-se precisar que diferentes processos acessem a mesma informação
 - Exemplo: Uma página web



Organização de Arquivos

© Solução

- Emprego de sistema de arquivos
 - Permitir armazenar grandes quantidades de informações
 - Informação permanecem após término do processo
 - Permitir o acesso por distintos processos



Organização de Arquivos

- ◎ O objetivo da organização de arquivos é: o processamento da informação, usualmente registros armazenados em arquivos. Para esse fim é necessário estudar as estruturas de dados apropriadas para a organização, que possam lidar com:
 - Grandes quantidades de registros;
 - Informações que requeiram espaços especiais e requisitos de desempenho;
 - Informações que podem ser processadas de diferentes formas por diferentes aplicações;
 - Informações que permitam novas tarefas serem cumpridas.

Organização de Arquivos-Conceitos Básicos

- ◎ A estrutura necessária não somente deve cumprir os requisitos como também devera fazê-lo eficientemente considerando aspectos de espaço e desempenho.

Definições

- ◎ Um *arquivo* é formado por uma coleção de registros lógicos, cada um deles representando um objeto ou entidade.
- ◎ Mapeada pelo S.O. em dispositivos físicos.
- ◎ Um *registro lógico*, ou simplesmente registro, é formado por uma sequencia de itens, sendo cada item chamado campo ou atributo. Cada item corresponde a uma característica ou propriedade do objeto representado.
- ◎ Cada campo possui um *nome*, um *tipo* e um *comprimento*

Organização de Arquivos - Conceitos Básicos

◎ Registro

Campo 1	Campo 2	...	Campo n
---------	---------	-----	---------

◎ Exemplo: Registro de aluno

Nome	Nº Matricula	RG	CPF	Curso	...
------	--------------	----	-----	-------	-----

Chave Primaria

Organização de Arquivos- Conceitos Básicos

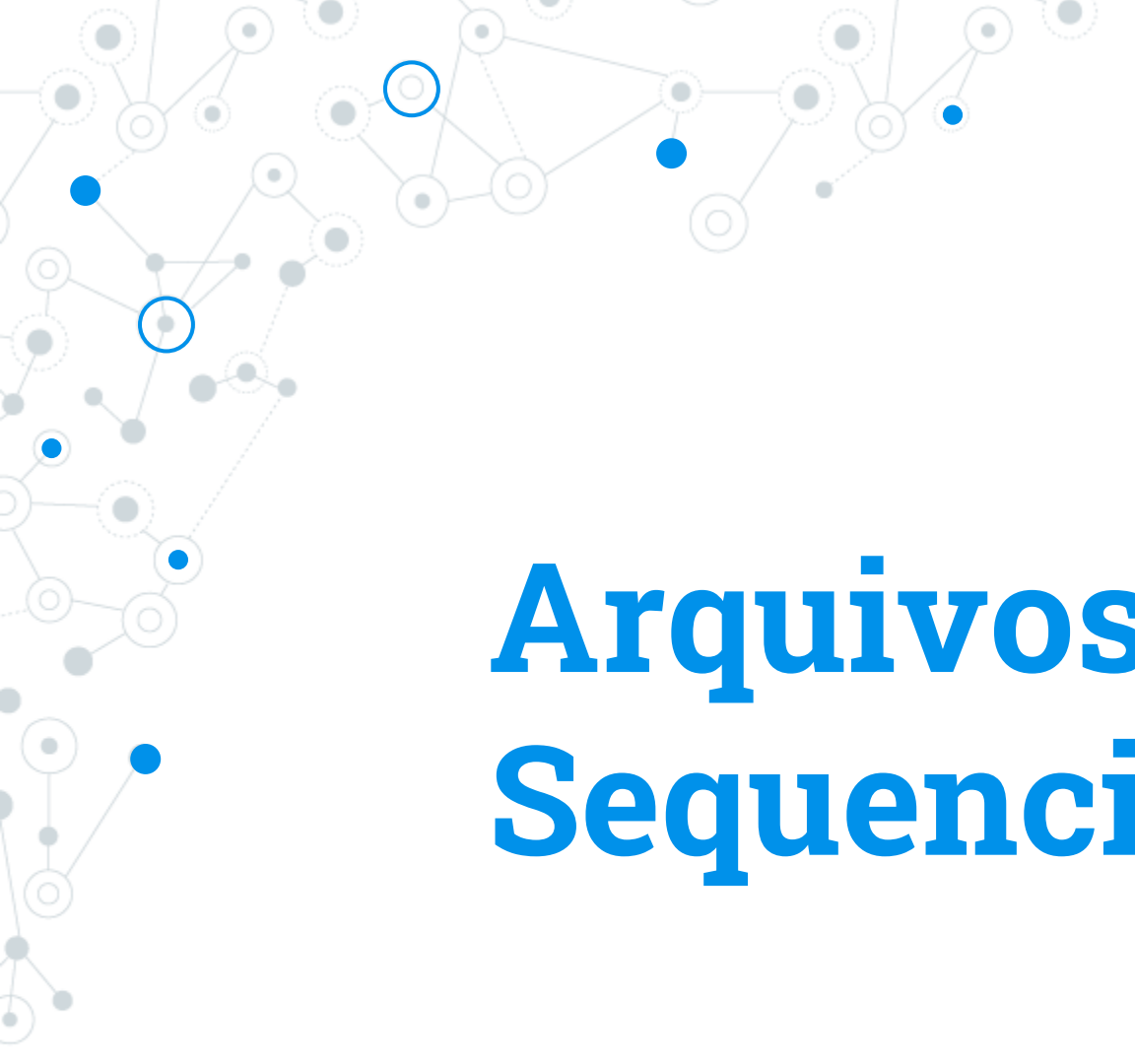
- © Uma *chave* é uma sequencia de um ou mais campos em um arquivo.
- © Uma *chave primária* é uma chave que apresenta um valor diferente para cada registro do arquivo, de tal forma que, dado um valor da chave primária, é identificado um único registro do arquivo. Usualmente a chave primária é formada por um único campo.
- © *Chave de acesso* é a chave utilizada para identificar o(s) registro(s) desejado(s) em uma operação de acesso a um arquivo.

Organização de Arquivos - Conceitos Básicos


Organizações Básicas de Arquivos

Organização	Acesso
Sequencial	Sequencial
Indexado	Sequencial e Direto
Direto	Direto

- ⊙ **Acesso sequencial** – refere-se ao acesso de múltiplos registros ou o arquivo inteiro, normalmente de acordo a uma ordem predefinida.
- ⊙ **Acesso direto** ou *random* – refere-se à localização de um único registro.
- ⊙ Para obter uma organização efetiva, é necessário que o tipo de organização e o tipo pretendido de acesso coincidam.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. Some nodes are highlighted with blue circles, and others with solid blue dots.

Arquivos Sequenciais

A decorative network diagram in the bottom-right corner, featuring a complex web of interconnected nodes and lines. Some nodes are highlighted with blue circles, and others with solid blue dots.

Arquivos Sequenciais

- Em um arquivo sequencial, os registros são dispostos ordenadamente, obedecendo a sequência determinada por uma chave primária, chamada chave de ordenação.

NÚMERO	NOME	IDADE	SALÁRIO
100	Pedro	23	1000
150	Leandro	20	500
200	Rodrigo	19	270
250	Maria	30	5000
300	Celso	27	2500

A decorative background featuring a network diagram. It consists of numerous nodes, represented by circles of varying sizes and shades of gray, connected by thin, light gray lines. Some nodes are highlighted with a solid blue fill, while others have a blue outline. The network is more densely packed on the left and right sides of the slide, with the central area being mostly empty space containing the title.

Complexidade de Algoritmos

Complexidade de Algoritmos

- ◎ Complexidade computacional: é a classificação de problemas segundo sua dificuldade
- ◎ Relação entre o tamanho do problema; o tempo e o espaço necessários para resolvê-lo
- ◎ Fundamental para projetar e analisar algoritmos

Complexidade de Algoritmos

- ◎ **Complexidade Espacial:**
 - quantidade de recursos para resolver o problema (Ex: memória)
- ◎ **Complexidade Temporal:**
 - quantidade de instruções necessárias para resolver o problema
 - Perspectivas: melhor caso, médio e pior caso
- ◎ **Como calcular a Complexidade:**
 - Experimentalmente
 - Notação Assintótica

Complexidade de Algoritmos

- ◎ Comportamento Assintótico:
- ◎ Objetivo: simplificar a análise descartando informações desnecessárias
 - Quando n tem um valor muito grande ($n \gg \infty$)
 - Termos inferiores e constantes multiplicativas contribuem pouco na comparação e podem ser descartados
- Exemplo:
 - $f_1(n) = 2n^2 + 400 \Rightarrow n^2$
 - $f_2(n) = 500n + 2000 \Rightarrow n$

Complexidade de Algoritmos

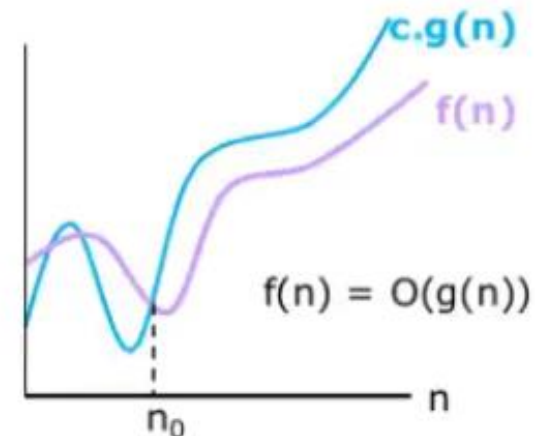
A notação O (Ômicron)
Oferece um limite
superior assintótico

Fonte: <https://pt.slideshare.net/necioveras/introducao-analise-e-complexidade-de-algoritmos>

Notação O

- Então uma função $f(n)$ é $O(g(n))$ se existem duas constantes positivas c e n_0 tais que:

$$f(n) \leq c \cdot g(n), \text{ para todo } n \geq n_0$$



Complexidade de Algoritmos

Algumas complexidades conhecidas

Notação	Complexidade	Característica	Exemplo
$O(1)$	constante	independe do tamanho n da entrada	determinar se um número é par ou ímpar; usar uma tabela de dispersão (hash) de tamanho fixo
$O(\log n)$	logarítmica	o problema é dividido em problemas menores	busca binária
$O(n)$	linear	realiza uma operação para cada elemento de entrada	busca sequencial ; soma de elementos de um vetor
$O(n \log n)$	log-linear	O problema é dividido em problemas menores e depois junta as soluções	heapsort, quicksort, merge sort
$O(n^2)$	quadrática	itens processados aos pares (geralmente loop aninhado)	bubble sort (pior caso); quick sort (pior caso); selection sort ; insertion sort
$O(n^3)$	cúbica		multiplicação de matrizes $n \times n$; todas as triplas de n elementos
$O(n^c)$, $c > 1$	polinomial		caixeiro viajante por programação dinâmica
$O(c^n)$	exponencial	força bruta	todos subconjuntos de n elementos
$O(n!)$	fatorial	força bruta: testa todas as permutações possíveis	caixeiro viajante por força bruta

Fonte: <https://pt.slideshare.net/necioveras/introduco-analise-e-complexidade-de-algoritmos>

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(c^n) < O(n!)$$



ORDENAÇÃO

Ordenação - Conceitos Básicos

- ◎ Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou decendente.
- ◎ A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
 - Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.

Ordenação - Conceitos Básicos

- ◎ Notação utilizada nos algoritmos: Os algoritmos trabalham sobre os registros de um arquivo.
 - Cada registro possui uma chave utilizada para controlar a ordenação.
 - Podem existir outros componentes em um registro.

Ordenação - Conceitos Básicos

- ◎ Um método de ordenação é estável se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação
- ◎ Alguns dos métodos de ordenação mais eficientes não são estáveis.
- ◎ A estabilidade pode ser forçada quando o método é não-estável.

Ordenação-Conceitos Básicos

☐ Classificação dos métodos de ordenação:

- Ordenação interna: arquivo a ser ordenado cabe todo na memória principal.
- Ordenação externa: Ordenação externa: arquivo a ser ordenado não cabe na memória principal.

☐ Diferenças entre os métodos:

- ☐ Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
- ☐ Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos

Ordenação - Conceitos Básicos

- ◎ A maioria dos métodos de ordenação é baseada em comparações das chaves.
- ◎ Existem métodos de ordenação que utilizam o princípio da distribuição.

Ordenação - Conceitos Básicos

- ◎ Exemplo de ordenação por distribuição:
considere o problema de ordenar um baralho com 52 cartas na ordem:

$A < 2 < 3 < \dots < 10 < J < Q < K$
e

$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$

Ordenação - Conceitos Básicos

□ Algoritmo:

1. Distribuir as cartas abertas em treze montes: ases, dois, três, ... , reis.
2. Colete os montes na ordem especificada
3. Distribua novamente as cartas abertas em quatro montes: paus, ouros, copas e espadas.
4. Colete os montes na ordem especificada.

Ordenação - Conceitos Básicos

- © Métodos como o ilustrado são também conhecidos como ordenação digital, radixsort ou bucketsort.
- © O método não utiliza comparação entre chaves. Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- © Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.

Ordenação Interna

- ◎ Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- ◎ Sendo **n** o número registros no arquivo, as medidas de complexidade relevantes são:
 - Número de comparações entre chaves.
 - Número de movimentações de itens do arquivo.
- ◎ O uso econômico da memória disponível é um requisito primordial na ordenação interna.

Ordenação Interna

- ◎ Métodos que utilizam listas encadeadas não são muito utilizados.
- ◎ Métodos que fazem cópias dos itens a serem ordenados possuem menor importância.

Ordenação Interna

- ◎ Classificação dos métodos de ordenação interna:
 - Métodos simples:
 - ◎ Adequados para pequenos arquivos .
 - ◎ Requerem $O(n^2)$ comparações.
 - ◎ Produzem programas pequenos.
 - Métodos eficientes:
 - ◎ Adequados para arquivos maiores.
 - ◎ Requerem $O(n \log n)$ comparações.
 - ◎ Usam menos comparações.
 - ◎ As comparações são mais complexas nos detalhes.
 - ◎ Métodos simples são mais eficientes para pequenos arquivos.



Algoritmos de Ordenação

Ordenação Interna

☐ Estrutura de um registro:

```
typedef int ChaveTipo;  
typedef struct Item {  
    ChaveTipo Chave;  
    outros componentes */  
} Item;
```

☐ Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.

Ordenação Interna

- © Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

typedef int Indice;

typedef Item Vetor[MaxTam + 1];

Vetor A;

- © O índice do vetor vai de 0 até MaxTam, devido às chaves sentinelas.
- © O vetor a ser ordenado contém chaves nas posições de 1 até n.

Ordenação Por Seleção

- ◎ Um dos algoritmos mais simples de ordenação.
- ◎ Algoritmo:
 - Selecione o menor item do vetor.
 - Troque-o com o item da primeira posição do vetor.
 - Repita essas duas operações com os $n - 1$ itens restantes, depois com os $n - 2$ itens, até que reste apenas um elemento.

Ordenação Por Seleção

◎ Exemplo:

	1	2	3	4	5	6
Chaves iniciais	O	R	D	E	N	A
i=1	A	R	D	E	N	O
i=2	A	D	R	E	N	O
i=3	A	D	E	R	N	O
i=4	A	D	E	N	R	O
i=5	A	D	E	N	O	R

Ordenação Por Seleção

```
void Selecao (Item *A, Indice *n)
{  Indice i, j, Min;
   Item x;
   for (i = 1; i <= *n - 1; i++)
   { Min = i;
     for (j = i + 1; j <= *n; j++)
       if (A[j].Chave < A[Min].Chave) Min = j;
     x = A[Min];
     A[ Min] = A[i];
     A[i] = x;
   }
}
```

Ordenação Por Seleção

☐ Vantagens:

- ☐ Custo linear no tamanho da entrada para o número de movimentos de registros.
- ☐ É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- ☐ É muito interessante para arquivos pequenos.

☐ Desvantagens:

- ☐ O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
- ☐ O algoritmo não é estável.

Ordenação Por Inserção

© Algoritmo:

Em cada passo a partir de $i=2$ faça:

- © Selecione o i -ésimo item da sequência fonte.
- © Coloque-o no lugar apropriado na sequência destino de acordo com o critério de ordenação.

Ordenação Por Inserção

□ Exemplo:

	1	2	3	4	5	6
Chaves iniciais	O	R	D	E	N	A
i=2	O	R	D	E	N	A
i=3	D	O	R	E	N	A
i=4	D	E	O	R	N	A
i=5	D	E	N	O	R	A
i=6	A	D	E	N	O	R

Ordenação Por Inserção

```
void Insercao(Item *A, Indice *n)
{
    Indice i, j;
    Item x;
    for (i = 2; i <= *n; i++)
    {
        x = A[i]; j = i - 1;
        A[0] = x; /* sentinela */
        while (x.Chave < A[j].Chave)
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = x;
    }
}
```

Ordenação Por Inserção

- ◎ Considerações sobre o algoritmo:
- ◎ O processo de ordenação pode ser terminado pelas condições:
 - Um item com chave menor que o item em consideração é encontrado.
 - O final da sequencia destino é atingido à esquerda.
- ◎ Solução:
- ◎ Utilizar um registro **sentinela na posição zero do vetor.**

Ordenação Por Inserção

- ☐ O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- ☐ O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- ☐ É o método a ser utilizado quando o arquivo está “quase” ordenado.
- ☐ É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- ☐ O algoritmo de ordenação por inserção é **estável**.

ShellSort

- ❑ É uma extensão do algoritmo de ordenação por inserção.
- ❑ Problema com o algoritmo de ordenação por inserção:
 - ❑ Troca itens adjacentes para determinar o ponto de inserção.
 - ❑ São efetuadas $n - 1$ comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- ❑ O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.

ShellSort

⊙ Algoritmo:

- Os itens separados de h posições são rearranjados.
- Todo h -ésimo item leva a uma sequencia ordenada.
- Tal sequencia é dita estar h -ordenada.

⊙ Como escolher o valor de h :

Sequencia para h :

$$\left. \begin{array}{l} h(s) = 3h(s - 1) + 1, \\ h(s) = 1, \end{array} \right\} \begin{array}{l} \text{para } s > 1 \\ \text{para } s = 1. \end{array}$$

ShellSort

◎ Exemplo:

	1	2	3	4	5	6
Chaves iniciais		O	R	D	E	N A
$h=4$	N	A	D	E	O	R
$h=2$	D	A	N	E	O	R
$h=1$	A	D	E	N	O	R

ShellSort

```
void Shellsort (Item *A, Indice *n)
{
    int i, j; int h = 1;
    Item x;
    do h = h * 3 + 1; while (h < *n);
    do
    { h = h / 3;
      for (i = h + 1; i <= *n; i++)
      { x = A[i]; j = i;
        while (A[j - h].Chave > x.Chave)
        { A[j] = A[j - h]; j = j - h;
          if (j <= h) break;
        }
        A[j] = x;
      }
    } while (h != 1);
}
```

Quicksort

- ◎ É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- ◎ Provavelmente é o mais utilizado.
- ◎ A ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
- ◎ Os problemas menores são ordenados independentemente.

Quicksort

- Algoritmo para o particionamento:
 1. Escolha arbitrariamente um **pivô x**.
 2. Percorra o vetor a partir da esquerda até que $A[i] \geq x$.
 3. Percorra o vetor a partir da direita até que $A[j] \leq x$.
 4. Troque $A[i]$ com $A[j]$.
 5. Continue este processo até os apontadores i e j se cruzarem.
- Ao final, o vetor $A[\text{Esq}..\text{Dir}]$ está particionado de tal forma que:
 - ▣ Os itens em $A[\text{Esq}], A[\text{Esq} + 1], \dots, A[j]$ são menores ou iguais a x ;
 - ▣ Os itens em $A[i], A[i + 1], \dots, A[\text{Dir}]$ são maiores ou iguais a x .

Quicksort

☐ Função Partição:

```
void Particao(Indice Esq, Indice Dir, Indice *i, Indice *j, Item *A)
{
    Item x, w;
    *i = Esq; *j = Dir;
    x = A[(*i + *j)/2]; /* obtem o pivo x */
    do
    {
        while (x.Chave > A[*i].Chave) (*i)++;
        while (x.Chave < A[*j].Chave) (*j)--;
        if ((*i) ≤ (*j))
        {
            w = A[*i]; A[*i] = A[*j]; A[*j] = w;
            (*i)++; (*j)--;
        }
    } while (*i <= *j);
}
```


Quicksort

☐ Função Quicksort

```
void Ordena(Indice Esq, Indice Dir, Item *A)
{
    int i,j;
    Particao(Esq, Dir, &i, &j, A);
    if (Esq < j) Ordena(Esq, j, A);
    if (i < Dir) Ordena(i, Dir, A);
}
```

```
void QuickSort(Item *A, Indice *n)
{
    Ordena(1, *n, A);
}
```

Quicksort

☐ Vantagens:

- ☐ É extremamente eficiente para ordenar arquivos de dados.
- ☐ Necessita de apenas uma pequena pilha como memória auxiliar.
- ☐ Requer cerca de $n \log n$ comparações em média para ordenar n itens.

☐ Desvantagens:

- ☐ Tem um pior caso $O(n^2)$ comparações.
- ☐ Sua implementação é muito delicada e difícil:
- ☐ Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
- ☐ O método não é **estável**.

Quicksort

◎ Complexidade

- **Melhor caso:** $O(n \log n)$ (pivô sempre divide bem o array).
- **Médio caso:** $O(n \log n)$.
- **Pior caso:** $O(n^2)$ (quando o pivô escolhido é sempre o menor ou maior, como array já ordenado).
- **Espaço extra:** $O(\log n)$ por causa da recursão.

Heapsort

- ◎ Possui o mesmo princípio de funcionamento da ordenação por seleção.
- ◎ Algoritmo:
 - 1. Selecione o menor item do vetor.
 - 2. Troque-o com o item da primeira posição do vetor.
 - 3. Repita estas operações com os $n - 1$ itens restantes, depois com os $n - 2$ itens, e assim sucessivamente.
- ◎ O custo do processo de busca do menor item, pode ser reduzido utilizando uma fila de prioridades.

Heapsort

◎ Filas de Prioridades

- É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.
- Aplicações:
 - ◎ Sistemas Operacionais
 - ◎ Roteadores

Heapsort

◎ Filas de Prioridades – Representação

- ◎ A melhor representação é através de uma estruturas de dados chamada *heap*:
- ◎ Para implementar as operações em um heap de forma eficiente é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais.

Heapsort

◎ **Heaps**

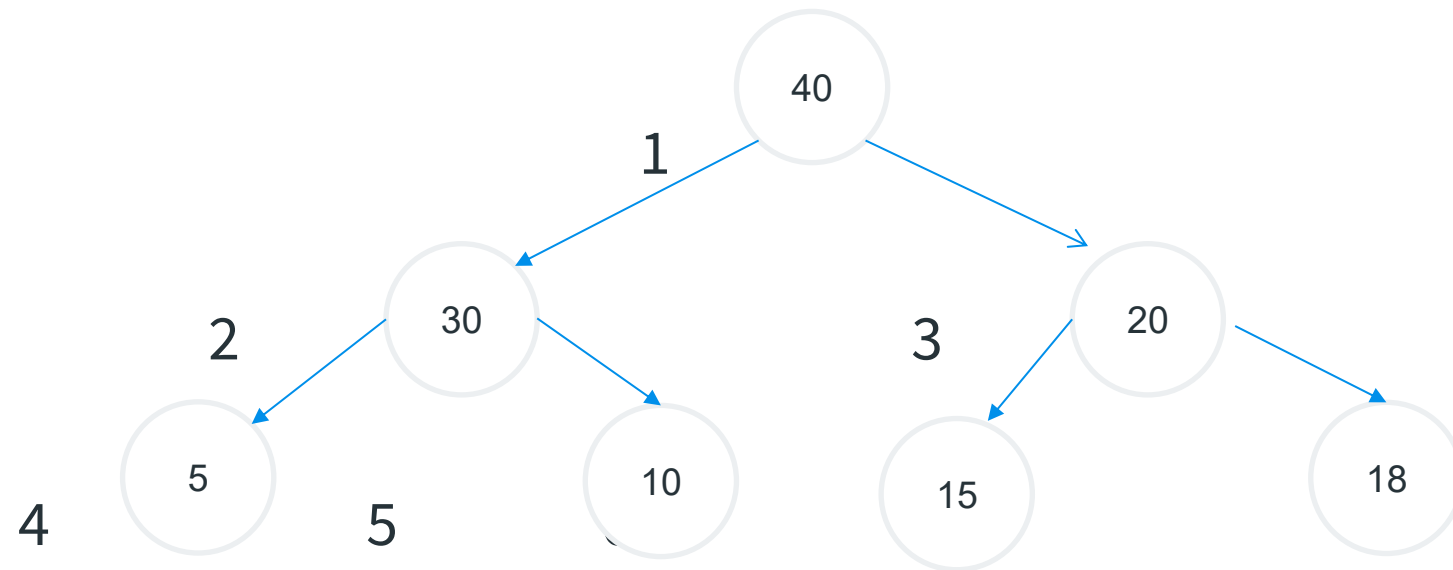
É uma sequência de itens com chaves
 $c[1], c[2], \dots, c[n]$, tal que:

$$c[i] \geq c[2i]$$

$$c[i] \geq c[2i + 1] \quad \text{para todo } i = 1, 2, \dots, n/2$$

◎ A definição pode ser facilmente visualizada em uma árvore binária completa:

Heapsort- representação do heap

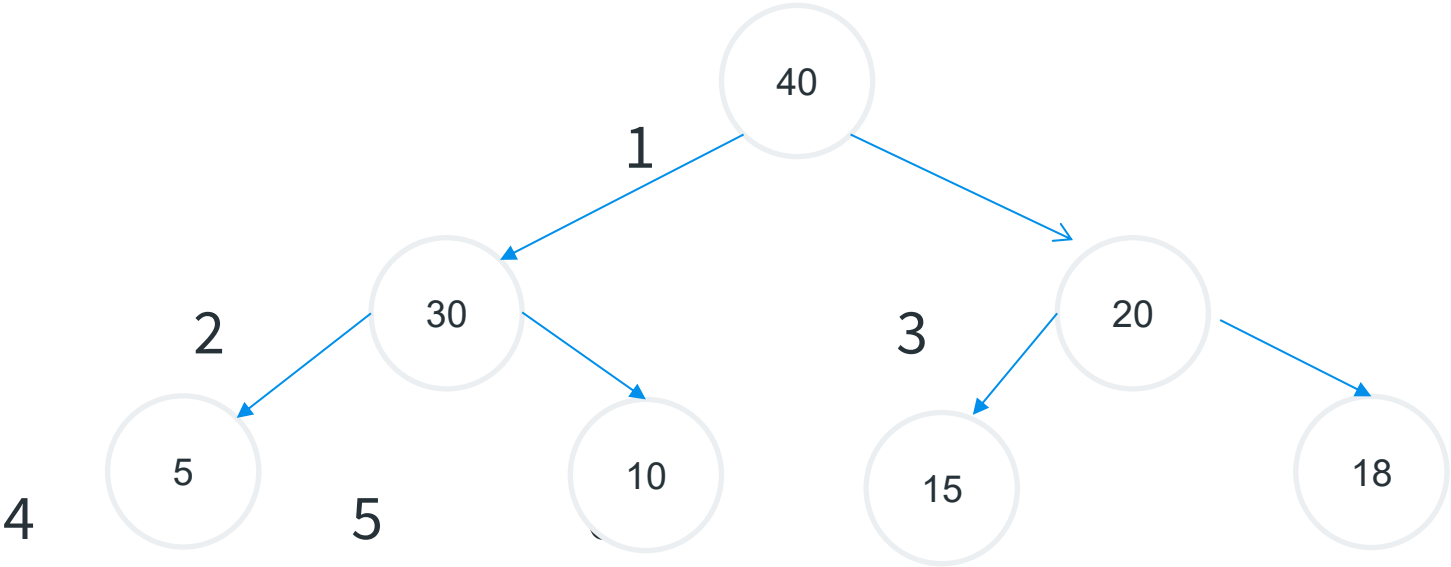


Max-Heap: cada nó é **maior ou igual** aos seus filhos → a raiz é o maior elemento.

Heapsort - Heaps

- árvore binária completa:
- Os nós são numerados de 1 a n
- O primeiro nó é chamado raiz.
- O nó $k/2$ é o pai do nó k , para $1 < k \leq n$.
- Os nós $2k$ e $2k + 1$ são os filhos à esquerda e à direita do nó k , para $1 \leq k \leq n/2$.
- Uma árvore binária completa pode ser representada por um array:

Heapsort



40	30	20	5	10	15	18
----	----	----	---	----	----	----

Heapsort

- © Heaps - algoritmo de Floyd (1964)
- © O algoritmo não necessita de nenhuma memória auxiliar.
- © Dado um vetor $A[1], A[2], \dots, A[n]$.
- © Os itens $A[n/2 + 1], A[n/2 + 2], \dots, A[n]$ formam um *heap*:
- © Neste intervalo não existem dois índices i e j tais que $j = 2i$ ou $j = 2i + 1$.

Heapsort

⦿ Como funciona:

⦿ **Construir o heap**

Transformar o array em um *max-heap* (o maior elemento sobe para a raiz).

⦿ **Trocar a raiz com o último elemento**

O maior elemento (na raiz) vai para o final do array.

⦿ **Reduzir o tamanho do heap**

Ignorar o último elemento (que já está ordenado) e "consertar" o heap (heapify).

⦿ **Repetir**

Até que todos os elementos estejam ordenados.

Heapsort

◎ Exemplo:

Ordenar [4, 10, 3, 5, 1]:

Construir max-heap \rightarrow [10, 5, 3, 4, 1]

Trocar raiz com último \rightarrow [1, 5, 3, 4, 10]

Heapify \rightarrow [5, 4, 3, 1, 10]

Trocar raiz com penúltimo \rightarrow [1, 4, 3, 5, 10]

Heapify \rightarrow [4, 1, 3, 5, 10]

Continua até ficar: [1, 3, 4, 5, 10]

Heapsort

```
void Refaz(Indice Esq, Indice Dir, Item *A)
{  Indice i = Esq;
   int j;
   Item x;
   j = i * 2;
   x = A[i];
   while (j <= Dir)
   {
       if (j < Dir)
       { if (A[j].Chave < A[j+1].Chave) j++;
       }
       if (x.Chave >= A[j].Chave) break;
       A[i] = A[j];
       i = j; j = i * 2;
   }
   A[i] = x;
}
```

Heapsort

© Programa para construir o *heap*:

```
void Constroi(Item *A, Indice n)
{
    Indice Esq;
    Esq = n / 2 + 1;
    while (Esq > 1) {
        Esq--;
        Refaz(Esq, n, A);
    }
}
```

Heapsort

Programa que mostra a implementação do Heapsort:

```
/* -- outras declarações -- */  
/* -- função Refaz      - - */  
/* -- função Constroi   -- */
```

```
void Heapsort(Item *A, Indice *n)  
{ Indice Esq, Dir;  
  Item x;  
  Constroi(A, *n); /* constroi o  
heap */  
  Esq = 1; Dir = *n;
```

```
while (Dir > 1)  
{ /* ordena o vetor */  
  x = A[1];  
  A[1] = A[Dir];  
  A[Dir] = x;  
  Dir--;  
  Refaz(Esq, Dir, A);  
}  
}
```


Heapsort

◎ **Complexidade**

- Construção do heap: $O(n)$
- Cada extração do maior + heapify: $O(\log n)$
- No total: $O(n \log n)$

Heapsort

- ◎ **Vantagens**

- ◎ **Complexidade garantida $O(n \log n)$**

Diferente do Quick Sort (que pode ter pior caso $O(n^2)$), o Heap Sort sempre terá $O(n \log n)$, independentemente da entrada.

- ◎ **In-place (baixo uso de memória extra)**

Ele não precisa de estruturas adicionais grandes, pois o heap é implementado dentro do próprio array (usa só $O(1)$ de espaço extra).

- ◎ **Independente da entrada**

Funciona bem mesmo se o array já estiver quase ordenado ou totalmente desordenado (ao contrário do Quick Sort, que pode se degradar se não usar boas estratégias de pivô).

- ◎ **Boa escolha para sistemas com pouca memória**

Justamente por não precisar de alocações extras.

Comparação de Algoritmos

Algoritmo	Complexidade	Estável?	Espaço extra	Observações
Heap Sort	$O(n \log n)$	Não	$O(1)$	Sempre $O(n \log n)$, mas lento na prática
Quick Sort	$O(n \log n)$ médio, $O(n^2)$ pior caso	Não	$O(\log n)$	Muito rápido na prática
Shell Sort	$O(n^2)$	Não	$O(1)$	Melhor caso $O(n \log n)$
Insertion Sort	$O(n^2)$	Sim	$O(1)$	Bom só para arrays pequenos

A decorative graphic in the top-left corner featuring a network of interconnected nodes and lines. Some nodes are highlighted with blue circles or dots.

Arquivos Diretos

A decorative graphic in the bottom-right corner featuring a network of interconnected nodes and lines. Some nodes are highlighted with blue circles or dots.

Arquivos Diretos

- Um arquivo direto consiste na instalação dos registros em endereços determinados com base no valor de uma chave primária, de modo que se tenha acesso rápido a os registros.

Chave = 150



$E = F(\text{chave})$



$E = 3$

	Número	Nome	Salário
1	200	PAULO	3100
2			
3	150	MARIA	2500
4			
5	250	FABIO	2500

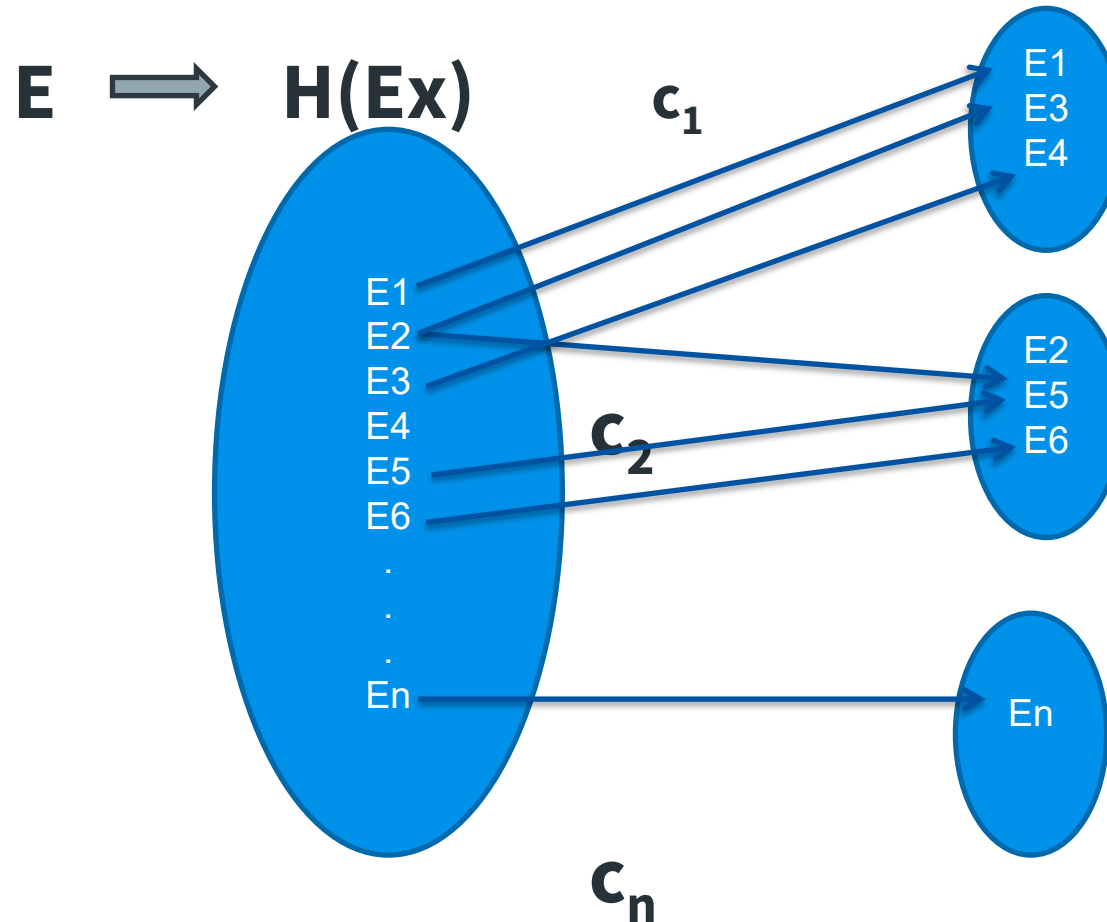
Arquivos de Acesso Direto - Hashing

- © É uma forma extremamente simples e fácil de estruturar o acesso a grandes quantidades de dados
- © Permite armazenar e encontrar rapidamente os dados por chave

Arquivos de Acesso Direto - Hashing



- © É uma forma extremamente simples e fácil de estruturar o acesso a grandes quantidades de dados
- © Permite armazenar e encontrar rapidamente os dados por chave

Hashing



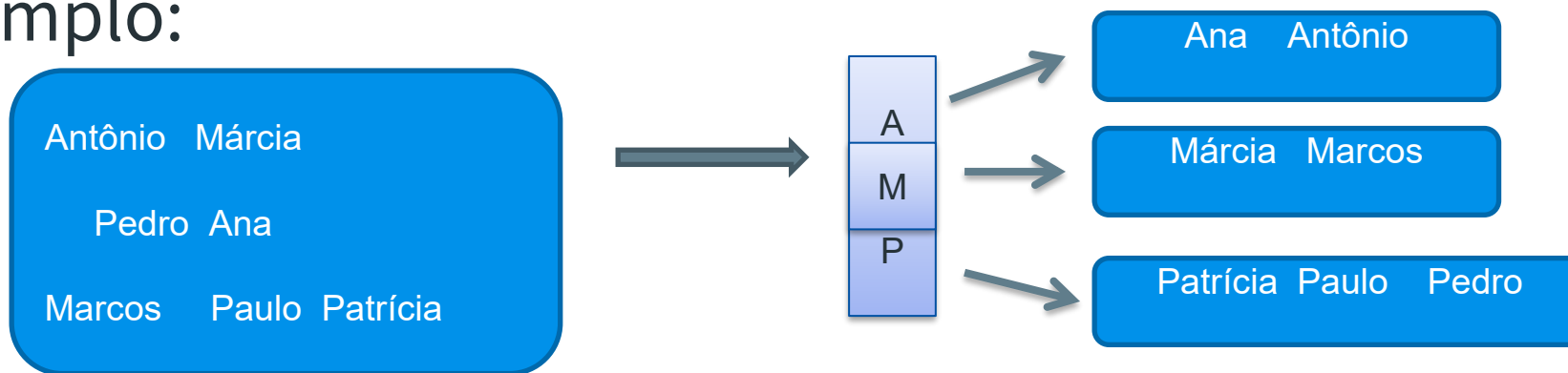
Como é possível associar a cada elemento do conjunto E um valor de hashing $H(e_x)$, calculado em tempo constante, pode-se realizar as operações de inserção, remoção e busca em cada classe gerada também em um tempo constante.

Hashing - Conceito Geral

- 
- Decorative network diagram in the top right corner showing interconnected nodes and lines.
- ◎ Para um universo de dados classificáveis por chave é possível:
 - Criar um critério simples para dividir este universo em subconjuntos com base em alguma qualidade do domínio de chaves.
 - Saber em qual subconjunto procurar e colocar uma chave.
 - Executar operações de acesso sobre estes subconjuntos bem menores com algum método simples.
- 
- Decorative network diagram in the bottom left corner showing interconnected nodes and lines.

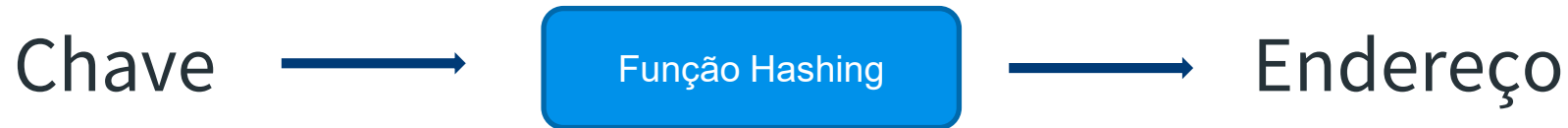
Hashing - Conceito Geral

© Exemplo:



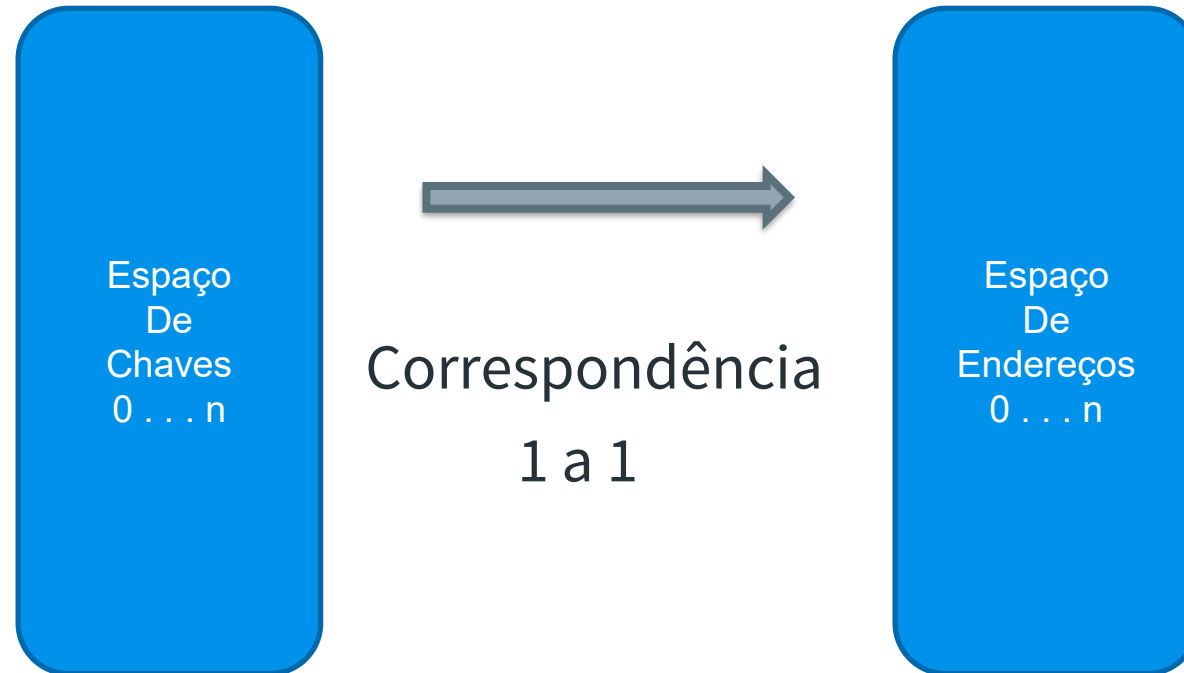
- © É necessário criar uma regra de cálculo que indique, dada uma chave, em qual subconjunto os dados devem ser colocados ou procurados usando essa chave. Isto é chamado de **Função de Hashing**

Função de Hashing



- © Uma função **hashing** deve satisfazer as seguintes condições:
 - Ser simples de calcular;
 - Idealmente, assegurar que elementos distintos possuam índices distintos.;
 - Gerar uma distribuição equilibrada para os elementos dentro de uma tabela de *hashing*.

Função de Hashing



Correspondência entre chaves e endereços

Função de Hashing

◎ Exemplos Funções de *Hashing*

- **Chave mod N**

$$f(\text{chave}) = \text{chave mod } N$$

Onde N é o tamanho da tabela

- **Chave mod P**

$$f(\text{chave}) = \text{chave mod } P$$

Onde P é o menor número primo $\geq N$, P é o novo tamanho de tabela. Resultam algumas colisões deste método

Função de Hashing

Truncation (extração de dígito)

Para uma chave numérica alguns dígitos são usados para gerar o endereço

Exemplo: Chave 1065892378

↓ ↓ ↓
0 8 3

Folding

Exemplo: chave 123456789

Folding by shifting

123
+ 456
789
1 368 →

Folding by boundary

321 ← reversão
+ 456
987 ← reversão
1 764 →

descartado

MÉTODOS DE RESOLUÇÃO DE COLISÕES

- ◎ Em aplicações práticas é difícil evitar o problema de colisões de chaves (sinônimos). Existem diversos mecanismos para resolver colisões.
- ◎ Os métodos podem ser agrupados de acordo com o mecanismo utilizado para localizar o próximo endereço de busca e a capacidade de relocação de um registro que já foi armazenado.

◎ MÉTODOS DE RESOLUÇÃO DE COLISÕES

- Resolução de colisões com enlaces (*links*)
- Resolução de colisões sem enlaces
 - Posicionamento estático dos registros
 - Posicionamento dinâmico dos registros

Hashing

- ◎ Função de Hashing
 - ◎ Endereço de Base = chave **MOD** N
 - ◎ Onde N é o menor número primo maior que o tamanho real da tabela ou arquivo
 - ◎ Exemplo
 - ◎ Para criar um arquivo de 8 registros:
 - N = 11
- Os operações aritméticas com números primos, minimiza o número de colisões

Resolução de Colisões com enlaces

Coalesced Hashing – método das cadeias convergentes

- É um método de resolução de colisões que utiliza ponteiros para conectar os elementos de uma cadeia de sinônimos.

Algoritmo

I. Calcule o endereço do registro a ser inserido no arquivo aplicando a função de *hashing* à chave primária do registro, para obter o endereço de base do registro ou o endereço de armazenamento provável.

II. **Se** o endereço de base está vazio **Então** insira o registro nessa posição.

Senão:

Se o registro está duplicado **Então** termine com uma mensagem de erro.

Senão:

- Análise os registros na lista de busca, verificando se existem duplicatas e procure o fim da lista que está indicado com um ponteiro nulo.
- Encontre a primeira posição vazia do arquivo, a partir do final do arquivo. Se nenhuma foi encontrada, termine com tabela cheia.
- Insira o registro na posição vazia encontrada e atualize o valor do ponteiro da lista para o novo fim de lista.

Resolução de Colisões sem Enlace

Hashing Linear – Hashing Duplo

Algoritmo

- I. Calcule o endereço do registro a ser inserido no arquivo aplicando a função de *hashing* à chave primaria do registro, para obter o endereço de base do registro ou o endereço de armazenamento provável
- II. **Se** o endereço de base está vazio **Então** insira o registro nessa posição.

Senão:

Se o registro está duplicado **Então** termine com uma mensagem de erro.

Senão:

- a. Calcule o deslocamento com $\text{deslocamento} = \text{chave} / \text{tamanho da tabela}$
- b. Encontre a nova posição endereço de base + deslocamento
- c. **Se** a posição esta vazia **Então** Insira o registro na posição vazia encontrada

Senão

Se nova posição = endereço de base **Então** Tabela cheia; FIM

Senão volte para a.

Método de Brent

Hashing linear – Tabela Dinâmica

I. Aplique a função *hash* a chave do registro a ser inserido para obter o endereço de base para armazenar o registro.

II. **SE** o endereço base está vazio **ENTÃO**: inserir registro

SENÃO:

A. Calcule o seguinte endereço potencial para novo registro.

Inicialize $S = 2$

B. **ENQUANTO** (o endereço potencial não está vazio)

1. Se for o endereço principal a tabela está cheia. Fim
2. Se o registro armazenado é o mesmo. Registro duplicado. Fim.
3. Calcule o próximo endereço potencial para armazenar o registro.

$S = S + 1$

/* tentativa de mover um registro previamente inserido */

C. Inicialize $i = 1$ e $j = 1$

D. **ENQUANTO** ($i + j < S$)

1. Determine se o registro armazenado na posição i da cadeia de prova primária pode ser movimentado j offsets através de sua cadeia de prova secundária.

2. **SE** ele pode ser movimentado **ENTÃO**: Movimente o registro e insira o novo registro em a posição livre i na cadeia primária.

SENÃO:

Varie i e/ou j para minimizar a soma $(i+j)$, se $i=j$ minimize em i

/* movimentação impossível */

E. Insira o novo registro na posição S da cadeia de prova primária. Fim.

Comparação entre os Métodos

◎ Complexidade:

	Complexidade
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shellsort	$O(n \log n)$
Quicksort	$O(n \log n)$
Heapsort	$O(n \log n)$

- ◎ A pesar de não se conhecer analiticamente o comportamento do Shellsort, ele é considerado um método eficiente.

Comparação entre os Métodos

⊙ Tempo de execução:

- Observação: O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
- Registros na ordem aleatória:

Número Registros em ordem aleatória	500	5.000	10.000	30.000
Inserção	11,3	87	161	-
Seleção	16,2	124	228	-
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
heapsort	1,5	1,6	1,6	1,6



Arquivos Indexados

Arquivos Indexados

- © Os arquivos sequenciais indexados são um híbrido das organizações de arquivos sequencial e direta, servindo para ambos os propósitos.
- © É útil quando é necessário procurar por um registro específico no arquivo, E outras por muitos registros ao mesmo tempo no mesmo arquivo
- © O problema básico a resolver é que, sendo o arquivo sequencial lento para uma busca direta, torna-se necessário acelerar o processo de busca.

Arquivos Sequencial Indexados

- © Um arquivo sequencial, acrescido em um índice (estrutura de acesso) constitui um *arquivo sequencial indexado*.

Número	Endereço
100	1
150	2
200	3
250	4
300	5

Índice

	Número	Nome	Salário
1	100	PEDRO	3000
2	150	JOÃO	1500
3	200	MARIA	2500
4	250	CARLA	3000
5	300	MAX	2000

Área de dados no Disco

Arquivos Indexados

- © Um *arquivo indexado*, onde os registros são acessados sempre através de um ou mais índices, não havendo qualquer compromisso com a ordem física de instalação dos registros.

Número	Endereço
100	4
150	3
200	1
250	5
300	2

	Número	Nome	Salário
1	200	PAULO	3100
2	300	JOSÉ	4500
3	150	MARIA	2500
4	100	MARISA	5000
5	250	FABIO	2500

Arquivos Indexados

◎ Fundamentos da Indexação

Um índice para um arquivo consiste de uma lista de valores dos campos chave que aparecem no arquivo, cada qual associado à posição do registro correspondente.

Em alguns casos um arquivo precisa ser acessado através de mais de um campo chave. Em tais casos, a solução é um sistema de múltiplos índices.

◎ **Vantagem:** a velocidade de acesso e a independência de acesso por vários tipos de ordenação

◎ **Desvantagem:** à medida que registros vão sendo inseridos e removidos todos os índices devem ser atualizados, o que tende a aumentar o tempo de atualização do arquivo.

◎ **Organização de Índices**

- ◎ Para que a indexação seja eficiente, é desejável que o índice ou parte dele seja movido para a memória principal para poder ser utilizado; ele deverá ter portanto dimensões reduzidas
- ◎ Esta exigência pode causar problemas se o número de registros for muito grande. Um método para superar esse problema é utilizar um índice para encontrar a posição aproximada do registro, em lugar da exata



ÁRVORES B

Árvores B

- ◎ Autores: Rudolf Bayer e Ed McCreight (1971)
- ◎ Árvores n-árias: mais de um registro por nodo.
- ◎ Em uma árvore B de ordem m :
 - **página raiz: 1 e $2m$ registros.**
 - **demais páginas: no mínimo m registros e $m + 1$ descendentes e no máximo $2m$ registros e $2m + 1$ descendentes.**
 - **páginas folhas: aparecem todas no mesmo nível.**

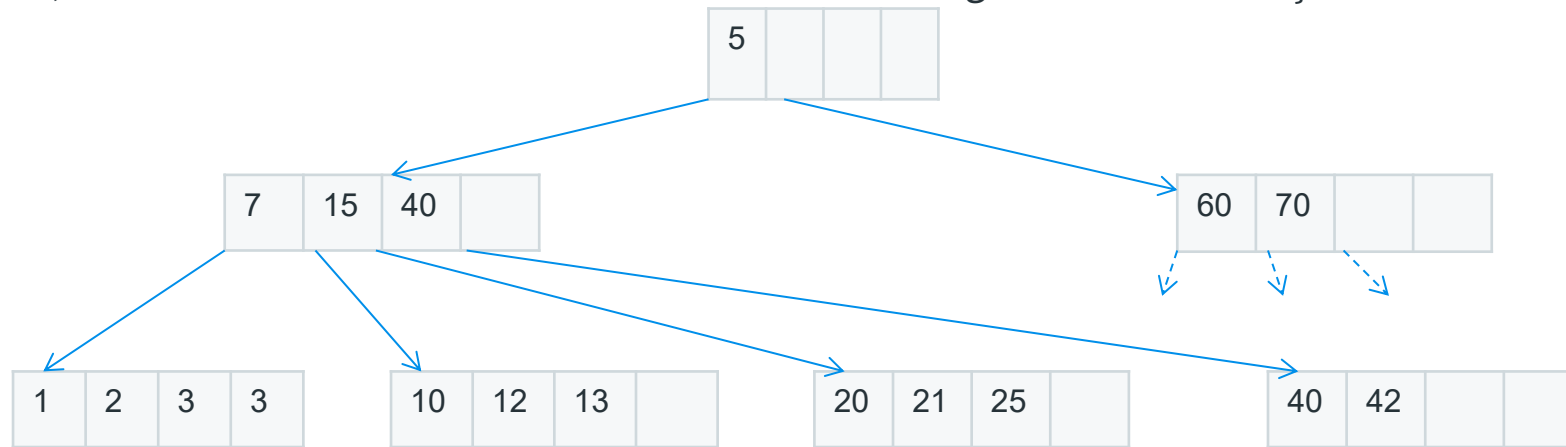
Registros em ordem crescente da esquerda para a direita.

Extensão natural da árvore binária de pesquisa.

Árvores B

Exemplo de árvore B de ordem 2

Neste caso, cada nó tem no mínimo dois e no máximo cinco registros de informação.



Árvore B - Algoritmos

⊙ Operações:

- – Inicializa

void Inicializa (Apontador *Dicionario)

{* Dicionario = NULL ; }

- – Pesquisa
- – Insere
- – Remove

Bibliografia: Projeto de Algoritmos, Nivio Ziviani

Árvore B - Definições

```
typedef long TipoChave;  
typedef struct Registro {  
    TipoChave Chave;  
    /outros componentes/  
} Registro;
```

```
typedef struct Pagina *Apontador;  
typedef struct Pagina {  
    short n;  
    Registro r [MM] ;  
    Apontador p[MM + 1] ;  
} Pagina;
```

```
#define M 2  
#define MM 4 // ordem da  
                árvore
```


Árbores B - Pesquisa

```
void Pesquisa(Registro *x , Apontador Ap)
{
    long i = 1;
    if (Ap == NULL)
    { printf ( "Registro nao esta presente na arvore\n" );
      return;
    }
    while ( i < Ap->n && x->Chave > Ap->r [ i-1].Chave) i++;
    if (x->Chave == Ap->r [ i-1].Chave)
    { *x = Ap->r [ i-1];
      return;
    }
    if (x->Chave < Ap->r [ i-1].Chave)
    Pesquisa(x , Ap->p[ i-1]);
    else Pesquisa(x , Ap->p[ i ] );
}
```

Árvores B- Inserção

1. Localizar a página apropriada aonde o registro deve ser inserido.
2. Se o registro a ser inserido encontra uma página com menos de $2m$ registros, o processo de inserção fica limitado à página.
3. Se o registro a ser inserido encontra uma página cheia, é criada uma nova página, no caso da página pai estar cheia o processo de divisão se propaga.

Árvores B - Primeiro refinamento do algoritmo Insere

```
void Ins(Registro Reg, Apontador Ap, short *Cresceu,  
        Registro *RegRetorno, Apontador *ApRetorno)  
{   long i = 1; long J ; Apontador ApTemp;  
    if (Ap == NULL)  
    { *Cresceu = TRUE; Atribui Reg a RegRetorno;  
      Atribui NULL a ApRetorno; return; }  
    while ( i < Ap-> n && Reg.Chave > Ap-> r [ i-1].Chave) i++;  
    if (Reg.Chave == Ap-> r [ i-1].Chave)  
    { printf ( " Erro : Registro ja esta presente\n" ) ; return ; }  
    if (Reg.Chave < Ap-> r [ i-1].Chave)  
    Ins(Reg, Ap-> p[ i--], Cresceu, RegRetorno, ApRetorno) ;  
    { — Continua na próxima página — }
```

Árvores B – Primeiro refinamento do algoritmo Insere- continuação

```
if (!*Cresceu) return;
if (Numero de registros em Ap < MM)
{ Insere na pagina Ap e *Cresceu = FALSE; return; }
/ Overflow: Pagina tem que ser dividida /
Cria nova pagina ApTemp;
Transfere metade dos registros de Ap para ApTemp;
Atribui registro do meio a RegRetorno;
Atribui ApTemp a ApRetorno;
}

void Insere(Registro Reg, Apontador *Ap)
{ Declarações de variaveis;
  Ins(Reg, * Ap, &Cresceu, &RegRetorno, &ApRetorno);
  if (Cresceu) { Cria nova pagina raiz para RegRetorno e ApRetorno; }
```

Árvores B- Insere na página

```
void InserenaPagina(Apontador Ap, Registro Reg, Apontador ApDir)
```

```
{  short NaoAchouPosicao;  
    Int k;  
    k= Ap -> n;  
    NaoAchouPosicao=(k>0);  
    While (NaoAchouPosicao)  
    { if ( Reg.Chave >= AP -> r[k-1].Chave)  
      { NaoAchouPosicao = FALSE;  
        break; }  
      Ap -> r[k]=Ap -> r[k-1];  
      Ap -> p[k+1] = Ap -> p[k];  
      k--;  
      If (k<1) NaoAchouPosicao =FALSE;
```

```
// continuação
```

```
Ap -> r[k] =Reg;  
Ap -> p[k+1]= ApDir;  
Ap -> n++;  
}
```

Árvores B - Refinamento final do algoritmo Insere

```
void Ins(Registro Reg, Apontador Ap, short *Cresceu,
        Registro *RegRetorno, Apontador *ApRetorno)
{ long i = 1; long J;
  Apontador ApTemp;
  if (Ap == NULL)
  { *Cresceu = TRUE; (*RegRetorno) = Reg; (*ApRetorno) = NULL;
    return;
  }
  while ( i < Ap->n && Reg.Chave > Ap->r [ i-1].Chave) i++;
  if (Reg.Chave == Ap->r [ i-1].Chave)
  { printf ( " Erro : Registro ja esta presente\n" ); *Cresceu = FALSE;
    return; }
  {— Continua na próxima página —}
```

Árvores B - Refinamento final do algoritmo Insere

```
if (Reg.Chave < Ap->r [ i-1].Chave) i--;  
  Ins(Reg, Ap->p[ i ] , Cresceu, RegRetorno, ApRetorno) ;  
  if ( !*Cresceu) return;  
  if (Ap->n < MM) / Pagina tem espaco /  
  { InsereNaPagina(Ap, *RegRetorno, *ApRetorno) ;  
    *Cresceu = FALSE;  
    return;  
  }  
  / Overflow: Pagina tem que ser dividida /  
  ApTemp = (Apontador)malloc(sizeof(Pagina) ) ;  
  ApTemp->n = 0; ApTemp->p[0] = NULL;  
  {— Continua na próxima página —}
```

Árvores B - Refinamento final do algoritmo Insere

```
if ( i < M + 1 )
{ InsereNaPagina(ApTemp, Ap->r [MM-1], Ap->p[MM] );
  Ap->n--;
  InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
}
else InsereNaPagina(ApTemp, *RegRetorno, *ApRetorno);
for ( J = M + 2; J <= MM; J++)
  InsereNaPagina(ApTemp, Ap->r [ J -1], Ap->p[ J ] );
Ap->n = M; ApTemp->p[0] = Ap->p[M+1];
*RegRetorno = Ap->r [M] ; *ApRetorno = ApTemp;
```

}

Árvores B - Refinamento final do algoritmo Insere

```
void Insere(Registro Reg, Apontador *Ap)
{
    short Cresceu;
    Registro RegRetorno;
    Pagina *ApRetorno, *ApTemp;
    Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
    if (Cresceu) / Arvore cresce na altura pela raiz /
    {
        ApTemp = (Pagina) malloc(sizeof(Pagina));
        ApTemp->n = 1;
        ApTemp->r[0] = RegRetorno;
        ApTemp->p[1] = ApRetorno;
        ApTemp->p[0] = *Ap;
        *Ap = ApTemp;
    }
}
```



Compressão de Dados

Compressão de Dados

- ◎ Apesar da alta disponibilidade e custo em constante decréscimo de dispositivos de armazenamento de dados como discos rígidos e memória de acesso aleatório, o volume de dados sempre será um fator importante para aplicações intensivas sobre os dados.
- ◎ Por esta razão, a compressão ou compactação de dados pode ser considerada como uma estratégia para:
 - reduzir o espaço de armazenamento físico;
 - melhorar as taxas na transferência de dados, especialmente em aplicações distribuídas;
 - para obter melhor desempenho em aplicações intensivas sobre os dados.

Compressão de Dados

- ◎ **Compressão de dados é a transformação ou codificação de um** conjunto de símbolos em outro com um tamanho reduzido.
- ◎ a compressão de dados busca detectar redundância nos dados e tenta removê-la através de uma representação reduzida.
- ◎ As estratégias de compactação podem ser classificadas de um modo geral em reversíveis e não-reversíveis:
 - **não-reversíveis** são geralmente utilizadas com arquivos de imagem, som e vídeo. Com esta técnica a redução do tamanho da representação é obtida descartando alguns dados e mantendo somente os mais relevantes. Sendo assim, no processo de descompressão não é possível obter o arquivo original.
 - **reversíveis** são aquelas que ao aplicar o processo contrário à compressão é possível obter os dados originais sem perda de informação. Esta técnica é geralmente usada para arquivos de texto.

Algoritmo LZW (Lempel-Ziv-Welch)

- © O algoritmo LZW é uma extensão ao algoritmo LZ proposto por Lempel e Ziv, em 1977. A diferença em relação ao LZ original é que o dicionário não está vazio no início, contendo todos os caracteres individuais possíveis.
- © Código ASCII

Algoritmo LZW

© Definições adotadas:

- raiz caracter individual
- String uma sequência de um ou mais caracteres
- palavra código valor associado a uma string
- dicionário tabela que relaciona palavras código e strings
- P string que representa um prefixo
- C caracter
- cW palavra código
- pW palavra código que representa um prefixo
- string(w) string correspondente à palavra código w
- SC sequência codificada

Algoritmo LZW - Codificação

1. No início o dicionário contém todas as raízes possíveis e P é vazio;
2. C = próximo caracter da sequência de entrada;
3. **Se** a string P+C existe no dicionário
 Então
 P = P+C;
 Senão
 - i. coloque a palavra código correspondente a P na sequência codificada;
 - ii. adicione a string P+C ao dicionário;
 - iii. P = C;
4. **Se** existirem mais caracteres na sequência de entrada
 Então
 - i. volte ao passo 2;**Senão**
 - i. coloque a palavra código correspondente a P na sequência codificada;
 - ii. **FIM.**

Algoritmo LZW - Exemplo

© Deseja-se codificar a sequência abaixo:

Posição	1	2	3	4	5	6	7	8	9
Caracter	a	b	b	a	b	a	b	a	c

© Seguindo-se o algoritmo de codificação e chamando SC a sequência codificada, temos:

1. SC = \emptyset

2. palavra código = \emptyset

2. Inicialização do dicionário com as raízes:

Palavra Código	String
1	a
2	b
3	c

Algoritmo LZW - Exemplo

sequência

Dicionario

Posição	1	2	3	4	5	6	7	8	9
Caracter	a	b	b	a	b	a	b	a	c



Palavra Código	String
1	a
2	b
3	c

4	ab
5	bb
6	ba
7	aba

P	C	P+C	SC
∅	a	a	∅
a	b	ab	1
b	b	bb	1 2 2
b	a	ba	
a	b	ab	
ab	a	aba	1 2 2 4
a	b	ab	
ab	a	aba	
aba	c	abac	1 2 2 4 7
c			1 2 2 4 7 3

Algoritmo LZW

◎ Sequencia Codificada

Posição	1	2	3	4	5	6
Palavra Código	1	2	2	4	7	3

Algoritmo LZW - Decodificação

1. No início o dicionário contém todas as raízes possíveis;
2. cW = primeira palavra código na sequência codificada (sempre é uma raiz);
3. Coloque a string(cW) na sequência de saída;
4. $pW = cW$;
5. cW = próxima palavra código da sequência codificada;
6. **Se** a string(cW) existe no dicionário

Então

- i. coloque a string(cW) na sequência de saída;
- ii. $P \leq \text{string}(pW)$;
- iii. $C \leq \text{primeiro caracter da string}(cW)$;
- iv. adicione a string $P+C$ ao dicionário;

Senão

- i. $P = \text{string}(pW)$;
- ii. $C = \text{primeiro caracter da string}(pW)$;
- iii. coloque a string $P+C$ na sequência de saída e adicione-a ao dicionário;

7. **Se** Existirem mais palavras código na sequência codificada

Então

- i. volte ao passo 4;

Senão


- i. FIM.



Grafos


Grafos



- ◎ Muitas aplicações em computação necessitam considerar conjunto de conexões entre pares de objetos:
 - Existe um caminho para ir de um objeto a outro seguindo as conexões?
 - Qual é a menor distância entre um objeto e outro objeto?
 - Quantos outros objetos podem ser alcançados a partir de um determinado objeto?
 - ◎ Existe um tipo abstrato chamado grafo que é usado para modelar tais situações.
- 

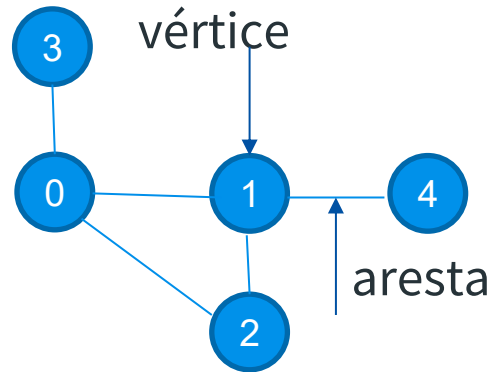
Grafos



- ◎ Alguns exemplos de problemas práticos que podem ser resolvidos através de uma modelagem em grafos:
 - Ajudar máquinas de busca a localizar informação relevante na Web.
 - Descobrir os melhores casamentos entre posições disponíveis em empresas e pessoas que aplicaram para as posições de interesse.
 - Descobrir qual é o roteiro mais curto para visitar as principais cidades de uma região turística.
- 

Grafos- Definições

- ◎ Grafo: conjunto de vértices e arestas.
- ◎ Vértice: objeto simples que pode ter nome e outros atributos.
- ◎ Aresta: conexão entre dois vértice.

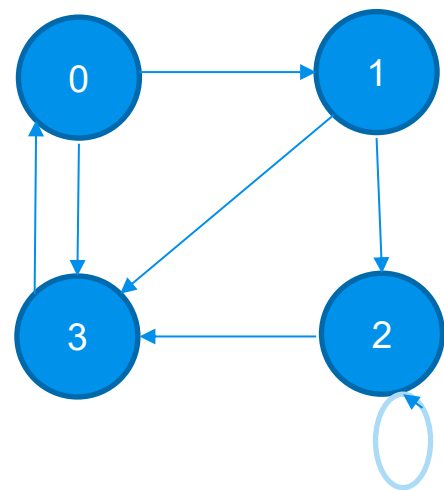


- ◎ Notação: $G = (V, A)$
 - G: grafo, V: conjunto de vértices, A: conjunto de arestas

Grafo Direcionados

- © Um grafo direcionado **G** é um par **(V, A)**, onde **V** é um conjunto finito de vértices e **A** é uma relação binária em **V**.
- © Uma aresta (u, v) sai do vértice u e entra no vértice v . O vértice v é adjacente ao vértice u .
- © Podem existir arestas de um vértice para ele mesmo, chamadas de self-loops.

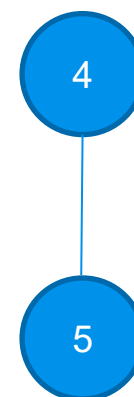
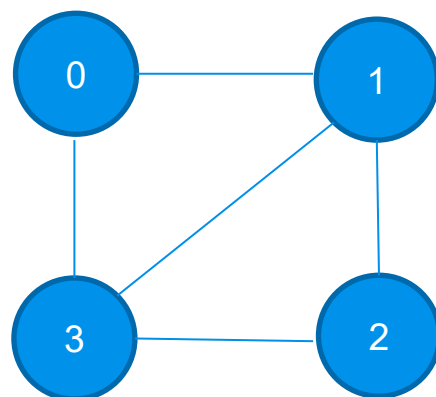
Grafos Direccionados



Grafos não Direcionados

- © Um grafo não direcionado **G** é um par **(V, A)**, onde o conjunto de arestas **A** é constituído de pares de vértices não ordenados.
 - As arestas (u, v) e (v, u) são consideradas como uma única aresta. A relação de adjacência é simétrica.
 - Self-loops não são permitidos.

Grafos Não Direcionados



Grafos – grau de um vértice

- ◎ Em grafos não direcionados:
 - O grau de um vértice é o número de arestas que incidem nele.
- ◎ Um vértice de grau zero é dito isolado ou não conectado.
- ◎ Em grafos direcionados
 - – O grau de um vértice é o número de arestas que saem dele (out-degree) mais o número de arestas que chegam nele (in-degree).

Grafos-Caminhos

- Um caminho de comprimento k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma sequência de vértices $(v_0, v_1, v_2, \dots, v_k)$ tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \dots, k$.
- O comprimento de um caminho $v_0, v_1, v_2, \dots, v_k$ é o número de arestas nele, isto é, o caminho contém os vértices e as arestas (v_0, v_1) , $(v_1, v_2), \dots, (v_{k-1}, v_k)$.
- Se existir um caminho c de x a y então y é alcançável a partir de x via c .
- Um caminho é simples se todos os vértices do caminho são distintos.

Grafos- ciclos

◎ Em um grafo direcionado:

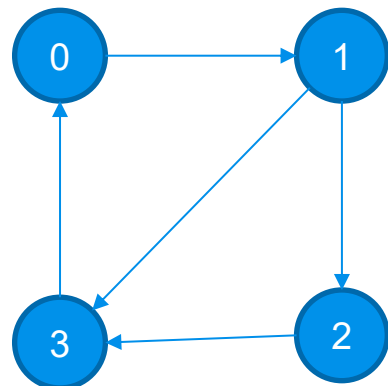
- Um caminho (v_0, v_1, \dots, v_k) forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos uma aresta.
- O ciclo é simples se os vértices v_1, v_2, \dots, v_k são distintos.
- O self-loop é um ciclo de tamanho 1
- Dois caminhos (v_0, v_1, \dots, v_k) e $(v'_0, v'_1, \dots, v'_k)$ formam o mesmo ciclo se existir um inteiro j tal que $v'_i = v_{(i+j) \bmod k}$ para $i = 0, 1, \dots, k-1$.

Grafos ciclos

◎ Em um grafo não direcionado:

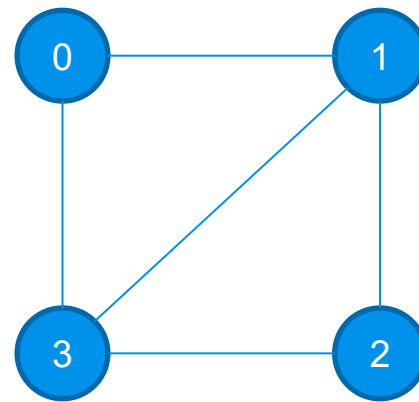
- Um caminho (v_0, v_1, \dots, v_k) forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos três arestas.
- O ciclo é simples se os vértices v_1, v_2, \dots, v_k são distintos.

Exemplos



Caminho: 0,1,2,3

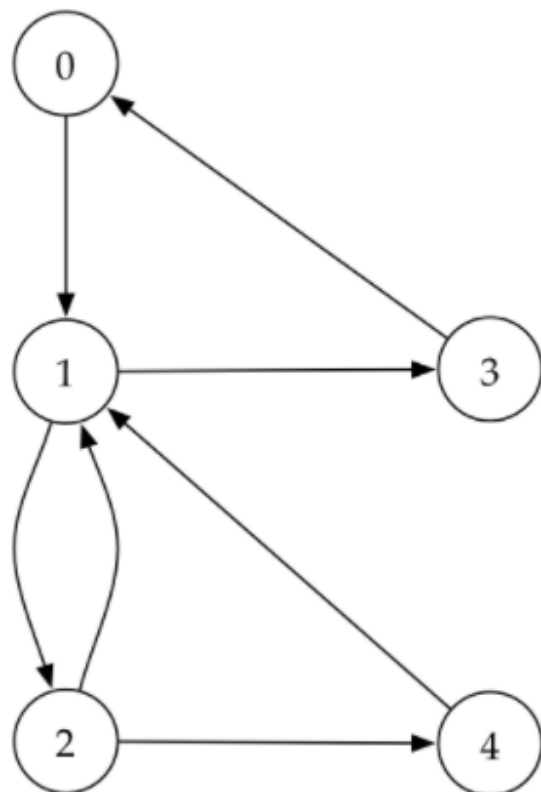
Ciclo : 0,1,2,3,0



caminho: 0,1,2,3,
0,3,2,1

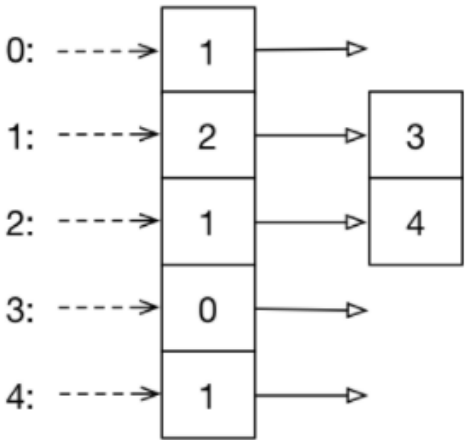
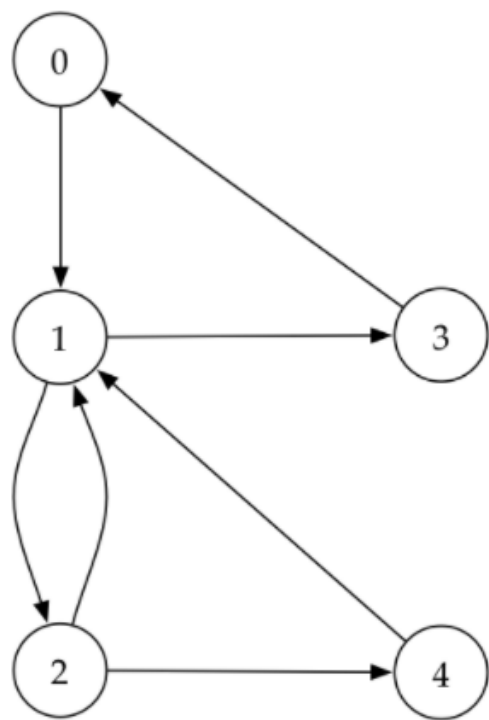
ciclo 0,1,3,0

Representação Computacional de um Grafo Matriz de Adjacências



	0	1	2	3	4
0		1			
1			1	1	
2		1			1
3	1				
4		1			

Lista de Adjacências



Bibliografia

- © *Estruturas de dados usando C-Aaron Ai Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein*
- © *Estrutura de Dados- Nivio Ziviani*
- © *Sistemas de Arquivos*
https://www.inf.pucrs.br/~emoreno/undergraduate/CC/sisop/classes_files/Aula13.pdf