

# Buscas e ordenação em vetores

quinta-feira, 16 de novembro de 2023 15:03

## Busca

Recuperação de dados armazenados em um repositório ou "Base de Dados". A busca depende do tipo de dados armazenados (valores duplicados, dados ordenados e não ordenados, dados estruturados, etc).

### Métodos de busca:

- Busca linear (começa pelo primeiro elemento e termina quando é encontrado ou até o fim do vetor)

### Busca linear

Começa no primeiro elemento e termina quando o elemento desejado é encontrado ou quando o fim do vetor é alcançado, é usado com um vetor de inteiros com comprimento conhecido.

- Dados desordenados e ordenados;
- Devolve o índice onde o elemento está ou  $-1$  senão existir;
- Em caso médio testará metade dos elementos de um vetor;
- Testará em média  $1/2n$  elementos.

```
int buscaLinear(int *vetor, int n, int elem){
    int i;
    for(i = 0; i < n; i++){
        if(elem == vetor[i]){
            return i;
        }
    }
    return -1;
}
```

### Busca ordenada

Semelhante a anterior, porém a pequena diferença de que a cada iteração onde o elemento não é encontrado, há uma verificação afim de detectar se o elemento no vetor realmente existe (if elem < vetor[i]), caso a verificação seja verdadeira, a busca é encerrada e é retornado  $-1$ .

```
int buscaOrdenada(int *vetor, int n, int elem){
    int i;
    for(i = 0; i < n; i++){
        if(elem == vetor[i]){
            return i;
        }
        else{
            if(elem < vetor[i]){
                return -1;
            }
        }
    }
    return -1;
}
```

### Busca Binária

Ideia de divisão e conferência, divide o vetor ao meio e vai analisando somente partes, para assim, encontrar o elemento mais rápido.

Enquanto o início for menor que o fim

Meio recebe (início + fim) / 2

Se o elemento for menor que o vetor na posição meio

Início recebe meio - 1 (busca na metade esquerda)

Senão, mas se for maior que meio

Início recebe meio + 1 (busca na metade direita)

Senão

Retorna meio

```
int buscaBinaria(int *vetor, int n, int elem){
    int i, inicio, meio, fim;
    inicio = 0;
    fim = n - 1;

    while(inicio <= fim){
        meio = (inicio + fim)/2;
        if(elem < vetor[meio]){
            fim = meio - 1; //busca na metade esquerda
        }
        else{
            if(elem > vetor[meio]){
                inicio = meio + 1; // busca na metade direita
            }
            else{
                return meio;
            }
        }
    }
    return -1; // elemento não encontrado
}
```

## Ordenação

Existem muitos métodos para ordenação, cabe a nós sabermos responder certas perguntas

- Qual a velocidade (no pior, melhor e caso médio) para ordenar as informações?
- Apresenta comportamento **natural** ou **não-natural**?
- Ele rearranja elementos iguais?

Processo de organizar conjunto de informações semelhantes, em uma ordem crescente ou decrescente, permite que o acesso aos dados seja feito de forma eficiente.

Usa como base uma chave específica (ex: código, id), mas quando uma troca se torna necessária, toda estrutura de dados é transferida

### Natural e Não-Natural

Dizemos que um algoritmo de ordenação tem um comportamento natural se ele trabalha o menos possível quando a lista já está ordenada, e quanto mais desordenada a lista estiver, mais trabalho o algoritmo terá, e q trabalhará o maior tempo ainda quando a lista estiver em ordem inversa.

### Ordenação Interna

O bloco a ser ordenado está na sua totalidade na memória do computador, logo qualquer registro pode ser imediatamente acessado

### Ordenação Externa

Trata-se de um arquivo e são acessos sequencialmente ou em bloco, não reorganiza os dados. O bloco a ser ordenado não cabe na memória principal disponível, e neste caso, trata-se de um arquivo.

## Métodos básicos de Ordenação

### Troca

Espalhe as cartas na mesa, troque sequencialmente as cartas fora de ordem, vá repetindo a operação.

### Seleção

Espalhe as cartas na mesa e selecione a carta de menor valor, retire-a e segure na mão, repita até que todas as cartas estejam na sua mão.

### Inserção

Segure as cartas na mão, ponha uma carta por vez na mesa, sempre na posição correta, repita até não restarem cartas na sua mão

### BubbleSort

É uma ordenação por **trocac**. Compara pares de elementos que estão lado a lado (2 elementos sequenciais), e os troca de lugar se estiverem na ordem errada

Não devolve nada, ordena direto pelo vetor

A cada vez que realocamos um elemento, diminuimos o valor do vetor

(analisa o primeiro e o segundo, se o primeiro for maior que o segundo, troca de lugar, analisa o segundo e o terceiro, se o segundo... Até o fim do vetor)

[Vídeo de Exemplo](#)

- ★ Determinar quantas comparações serão realizadas
  - ★ Tempo de execução aumenta exponencialmente com a quantidade de elementos
- Ordenação oscilante

### SelectionSort

Localiza o menor e coloca na primeira posição, localiza o segundo menor e coloca na segunda posição, assim vai. A cada iteração é calculado o menor valor dos elementos que ainda faltam ordenar. Repete-se o processo até que todos os elementos estejam ordenados.

- Lenta para grande volume de itens;

[Vídeo de Exemplo](#)

### InsertionSort

Pega um elemento de cada vez e põe em seu devido lugar. Usa o modelo de seleção.

- Comportamento natural;

[Vídeo de Exemplo](#)

Deste modo podemos eliminar metade da busca (se tivéssemos 100 posições, seria necessário somente em metades, dividindo essa metade em partes cada vez menores)

# Complexidade e Análise de Algoritmos

quinta-feira, 16 de novembro de 2023 15:03

## Análise de algoritmos

Área cujo o foco são os algoritmos e sua eficiências. Algoritmos diferentes podem resolver o mesmo problema mas não com a mesma eficiência. Essas diferenças de eficiência podem ser irrelevantes (para um pequeno volume de elementos processados) e crescer proporcionalmente (com o número de dados processados).

## Complexidade Computacional

CUSTO = MEMÓRIA + TEMPO

- Esforço computacional
- Desempenho

A complexidade depende do tamanho de entrada, os principais critérios são o pior caso e o caso médio. O pior caso dá o valor máximo que ela pode atingir.

## Abordagem para determinar se um algoritmo é eficiente:

- Empírica: comparar programas
- Matemática: estudo das propriedades dos algoritmos (complexidade intrínseca)

## Análise Empírica

Complexidade a partir da execução de um programa

- Avalia o desempenho em uma determinada configuração de computador/linguagens;
- Comparar linguagens e computadores;
- Considerar custos não aparentes, como por exemplo "custos de alocação de memória".

Envolve como foi implantado

O hardware pode mascarar o resultado ou eventos ocorridos no momento da avaliação

## Dados:

- Dados reais
- Aleatórios
- Perversos

## Análise matemática

Estudo formal, estudo da ideia por trás de um algoritmo, analisa apenas a ideia do algoritmo, pequenos detalhes da implementação são desconsiderados.

- Detalhes de baixo nível, como a linguagem de programação utilizada, o hardware no qual o algoritmo é executado ou o conjunto de instruções da CPU, são ignorados;
- Permite entender como o algoritmo se comporta a medida que o conjunto de entrada cresce.

## Contando instruções de um algoritmo

Contar as instruções simples:

- Atribuição a uma variável;
- Acesso ao valor de um elemento do vetor;
- Comparação entre valores;
- Incremento de um vetor;
- Operações matemáticas;

Todas essas instruções terão o mesmo custos (no caso 1) e os comandos de seleção (if), não tem custo, somente suas comparações.

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/     M = V[i];
/*5*/   }
/*6*/ }
```

- No exemplo acima, o custo da "linha 1" é de 1 instrução;
- Na "linha 1", o valor da primeira posição do Vetor é copiado para a variável "M" (acessar o valor "V[0]" e atribuí-lo a "M");
- O custo da inicialização do laço for (linha 2), é de 2 instruções (o comando for precisa ser inicializado: 1 instrução (i = 0), mesmo que o vetor tenha tamanho zero, ao menos uma comparação será executada (i < n), o que custa mais 1 instrução);
- O custo para executar o comando de laço for (linha 2), é de: 2n instruções (ao final de cada iteração do laço for, precisamos executar uma instrução de:
  - Incremento (i++), a comparação para verificar se vamos continuar a executar o laço for (i < n), o laço for será executado "n" vezes, assim, essas 2 instruções também serão executadas "n" vezes);
- O comando if: 1 instrução - acesso ao valor do vetor e a sua comparação (V[i] >= M);

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/     M = V[i];
/*5*/   }
/*6*/ }
```

Atribuição: 1 instrução

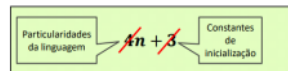
Atribuição e comparação para inicializar o for: 2 instruções

Comparação e incremento: mais 2 instruções que serão executadas "n" vezes.

## Comportamento Assintótico

Definida pelo crescimento da complexidade para entradas suficientemente grandes.

Exemplo: o custo de um algoritmo é dado pela função  $f(n) = 4n + 3$ . Mas será que todos os termos da função  $f$  são necessários para termos noção do custo? Podemos descartar certos termos na função e manter apenas os que nos informam o que acontece com a função quando o tamanho dos dados de entrada ("n"), cresce muito. Descartando todos os termos constantes e mantendo apenas o de maior crescimento, obtemos o comportamento assintótico. Podemos ignorar então p 4 e o 3 na função  $4n + 3$ !



Função de Custo	Comportamento assintótico
$f(n) = 227$	$f(n) = 1$
$f(n) = 15n + 4$	$f(n) = n$
$f(n) = n^2 + 3n + 8$	$f(n) = n^2$
$f(n) = 8n^3 + 700n^2 + 467$	$f(n) = n^3$

## Big O / Grande-O

Representa o custo do algoritmo no pior caso possível, para todas as entradas de tamanho "n. Ou seja, analise o limite superior de entrada. A notação  $O(n^2)$  nos diz que o custo do algoritmo não é assintoticamente, pior do que  $n^2$ .

- Grande-Omega;
- Grande-O;
- Grande-Theta;
- Pequeno-O.

## n log n

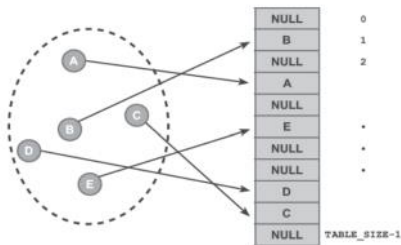
Típica de algoritmos que trabalham com particionamento de dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos. Logaritmos de ordenação mais sofisticados e rápidos tem crescimento do tempo nessa execução, onde isso é o logaritmo na base 2 de n.

# Tabela Hash

terça-feira, 5 de março de 2024 19:39

## Tabela Hash

Espalhar elementos que queremos armazenar na tabela. Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do array que define a tabela



Podemos assim:

- Acessar de forma rápida uma determinada posição do array;
- Evita o gasto excessivo de memória, pois os elementos ficam dispersos, ou seja, várias posições do array podem não possuir nenhum dado;
- Alta eficiência na operação de busca;
- O tempo de busca é independente do número de chaves armazenadas na tabela;
- Implementação simples.

## Desvantagens:

- Alto custo para recuperar os elementos ordenados pela chave;
- O pior caso é aquele onde se tem um alto número de colisões.

## Usos:

- Busca de elementos;
- Verificação de integridade de dados e autenticação de mensagens;
- Armazenamento de senhas com segurança;
- Implementação da tabela de símbolos dos compiladores;
- Criptografia.

## Colisões

Uma colisão é uma ocorrência de duas ou mais chaves na tabela hash com o mesmo valor de posição. Independentemente da função de hashing utilizada existe a possibilidade da função retornar a mesma posição para duas chaves diferentes (colisão).

A colisão não é exatamente ruim, mas é indesejável e diminui o desempenho do sistema

## Implementando...

Para implementar nossa tabela hash, primeiramente é preciso definir o tipo de dado que será armazenado nela. Para isso, é necessário definir o tipo `opaco (struct)` que representa nossa tabela. Além disso, é necessário definir o encapsulamento.

## hashTable.h

Tipo Struct Aluno;

Funções disponíveis ao programador para se trabalhar com essa tabela hash.

## hashTable.c

Chamadas às bibliotecas necessárias;  
Definição do tipo de dado que será a tabela.  
Implementações das funções cujo os protótipos foram definidos no arquivo hashTable.h

Principais funções desses arquivos:

- Cria o Hash;
- Insere Hash (com e sem colisão);
- Busca hash (com e sem colisão);
- Libera o Hash;
- Métodos de Hash;
- String como chave.

## Métodos de Hash:

Tanto na operação de inserção de elementos, quanto na operação de busca de elementos na tabela Hash, é necessário calcular essa posição a partir de uma chave escolhida entre os dados manipulados.

É essa função que distribui as informações de forma equilibrada pela tabela. Ela calcula, a partir do valor do dado, a posição delem dentro da tabela.

## Método da divisão

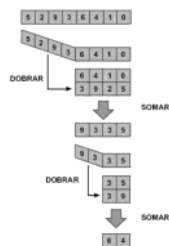
Calcula o resto da divisão do valor inteiro que representa o elemento, pelo tamanho da tabela.

## Método da Multiplicação

Usa uma constante fracionária para multiplicar o valor da chave que representa o elemento.

## Método da dobra

Utiliza o esquema de dobrar e somar os dígitos do valor para calcular sua posição.



## String como chave

Caso tenhamos uma string que haja como chave, devemos calcular um valor numérico a partir dessa string, onde os valores ascii dos caracteres que compõe a string são multiplicados por um determinado valor a depender de sua posição.

## Hashing Universal

Estratégia que busca amenizar o problema de colisões. A proposta é escolher aleatoriamente, em tempo de execução, a função que será usada a partir de um conjunto de funções de hashing previamente definidos.

## Tipos

A função de hashing é dita perfeita se nunca ocorre uma colisão e imperfeita de para duas chaves diferentes a saída gerada pela função é a mesma posição na tabela.

## Tratando colisões

Um dos motivos para colisões ocorrerem é porque existem mais chaves a serem armazenadas do que o tamanho da tabela suporta.

- Encadeamento aberto;
- Encadeamento separado.

## Encadeamento Aberto

A ideia estratégica do endereçamento aberto é percorrer a tabela hash buscando uma posição ainda não ocupada. Apesar de ter diversas vantagens, essa técnica tem a grande desvantagem de **requerer maior esforço de processamento**, que se deve ao fato de que quando ocorre uma colisão, um novo cálculo precisa deve ser efetuado para uma nova posição na tabela.

## Sondagem linear:

O algoritmo tenta espalhar os elementos de **forma sequencial** a partir da posição calculada utilizando a função de hashing.

Porém, essa abordagem gera o problema de agrupamento primário, à medida que a tabela fica cheia, o tempo para incluir ou buscar aumenta.

	CHAVE	POSICÃO	INTERPREÇÃO	
NULL	A	2	Posição 2 vazia. Insere elemento	NULL
NULL	B	6	Posição 6 vazia. Insere elemento	A
NULL	C	2	Posição 2 ocupada. Procura na próxima posição: 3	C
NULL	D	10	Posição 10 vazia. Insere elemento	NULL
NULL	E	10	Posição 10 ocupada. Procura na próxima posição. Como a posição 10 é a última, volta para o início: 0	B
NULL			Posição 0 vazia. Insere elemento	NULL
NULL				NULL
NULL				NULL
NULL				NULL
NULL				D

## Sondagem quadrática

O algoritmo tenta espalhar os elementos utilizando uma **equação do segundo grau**.

A sondagem quadrática resolve o problema de agrupamento primário, porém ela gera outro problema conhecido como agrupamento secundário (ocorre porque todas as chaves que produzem a mesma posição inicial na tabela, também produzem a mesma posição na sondagem quadrática).

## Duplo Hash

O algoritmo tenta espelhar os elementos utilizando **duas funções de hashing**, a primeira é usada para calcular a posição inicial do elemento, e a segunda para calcular os deslocamentos em relação a posição inicial, caso haja uma colisão.

# Heap Sort

domingo, 17 de março de 2024

12:51

Analisar a quantidade de passos ou iterações que um algoritmo leva para executar sua função

O big o é um tipo de notação que leva em conta grande volumes de dados

- Levar em consideração apenas as repetições do código
- Verificar a complexidade das funções/métodos da própria linguagem
- Ignorar as constantes e usar o termo de maior grau

Complexidade de Tempo:

- No pior caso e no caso médio, o Heap Sort possui uma complexidade de tempo de  $O(n * \log n)$ , onde 'n' é o número de elementos a serem ordenados. Isso ocorre porque a inserção e remoção em uma heap binária têm complexidade de tempo  $O(\log n)$ , e o algoritmo requer 'n' dessas operações para ordenar completamente a lista.
- No melhor caso, também é  $O(n * \log n)$ . Isso ocorre porque mesmo que a lista já esteja ordenada, o algoritmo ainda precisa construir a heap, o que leva  $O(n * \log n)$  operações.

Complexidade de Espaço:

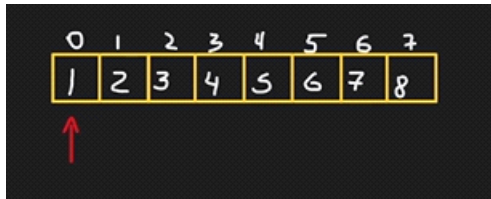
- O Heap Sort é um algoritmo in-place, o que significa que ele não requer espaço adicional além do espaço usado para armazenar os elementos da lista original. Portanto, a complexidade de espaço é  $O(1)$ , ou seja, constante. No entanto, o Heap Sort ainda requer espaço adicional para a operação de construção da heap, mas isso é negligenciável em comparação com o espaço usado para armazenar a lista original.

Características Adicionais:

- O Heap Sort é estável, o que significa que a ordem relativa dos elementos iguais na lista original é preservada na lista ordenada.
- É um algoritmo de classificação in-place, o que significa que não requer espaço de memória adicional para armazenar a lista de entrada.
- O Heap Sort é eficiente em termos de tempo para grandes conjuntos de dados e é particularmente útil quando a memória é limitada, pois sua complexidade de espaço é mínima.

Em termos práticos, quando 'n' aumenta, a complexidade do Heap Sort cresce, mas em um ritmo que é mais lento do que uma função linear ( $O(n)$ ). Isso faz com que o Heap Sort seja eficiente para grandes conjuntos de dados em comparação com algoritmos com complexidade de tempo linear ou quadrática.

## Entendendo $O(\log n)$



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

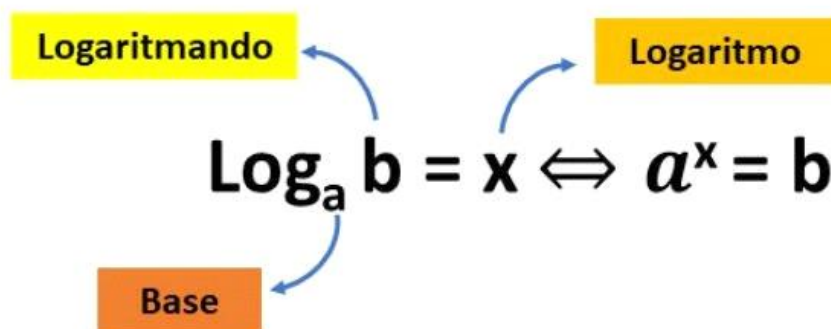
Usando o método de sequenciação para achar o elemento 8, fazemos  $n$  comparações

Já com o método binário, dividimos ao meio três vezes para achar o elemento 8, ou seja, fazemos 3 comparações

Se tivéssemos um array de 16 posições, com o método de sequenciação faríamos 16 comparações. Já no método binário usaremos 4 comparações.

Mas o que isso tem a ver com  $\log n$ ?

Considerando o log, temos por exemplo  $\log$  de 8 na base 2 equivaleria a  $2^x = 8$



Sendo  $b$  o número de elementos,  $a$  2 que é fixo e  $x$  o número de comparações sendo feitas

Que tal irmos aumentando o número de elementos para visualizarmos melhor?

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

....

Conforme o número de elementos aumenta exponencialmente, nosso número de comparações aumenta constantemente, de um em um.

Se tivéssemos por exemplo:

$$2^{20} = 1.048.576$$

Teríamos 1 milhão de posições para apenas vinte comparações

$$2^{30} = 1.073.741.824$$

Aqui já seriam 1 bilhão de posições

Por esse motivo, um algoritmo que tem  $\log n$  como complexidade é altamente eficiente.

Mas no nosso caso, nosso algoritmo não tem  $\log n$  e sim  $n * \log n$

Suponha que temos uma lista de números que precisamos ordenar usando o algoritmo Heap Sort. O tamanho dessa lista é representado por ' $n$ '.

Digamos que ' $n$ ' seja igual a 8. Então, temos 8 números para ordenar.

Agora, vamos calcular ' $n * \log n$ ' para ' $n = 8$ ':

1.  $n = 8$
2.  $\log n = \log_2(8) = 3$
3.  $n * \log n = 8 * 3 = 24$

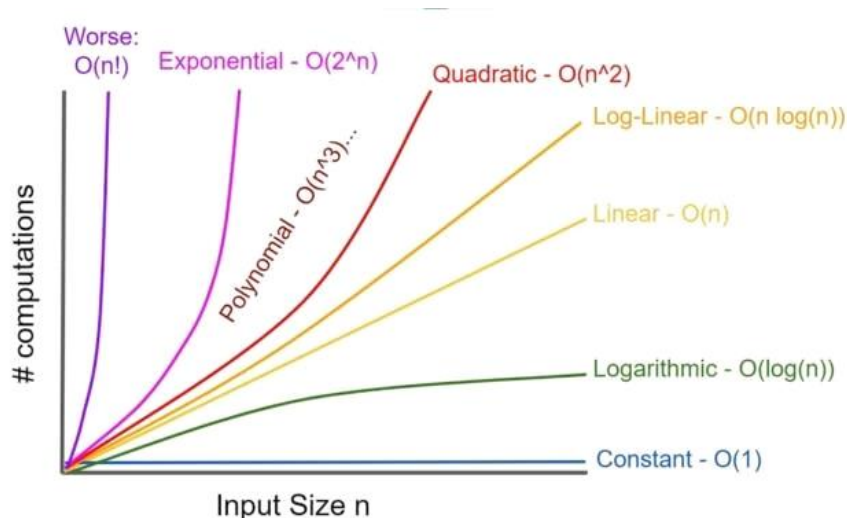
Portanto, para uma lista de 8 elementos, ' $n * \log n$ ' seria igual a 24.

Essa multiplicação nos dá uma ideia da complexidade do algoritmo Heap Sort em termos do número de operações necessárias para ordenar uma lista de tamanho ' $n$ '. À medida que ' $n$ ' aumenta, o número de operações necessárias aumenta, mas em um ritmo que é mais lento do que uma função linear, tornando o Heap Sort eficiente para conjuntos de dados maiores.

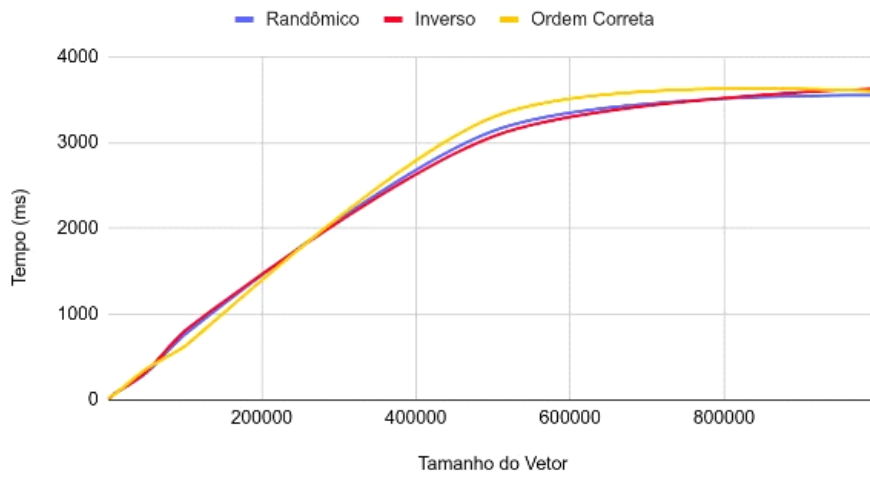
Vamos calcular ' $n * \log n$ ' para ' $n = 16$ ':

4.  $n = 16$
5.  $\log n = \log_2(16) = 4$
6.  $n * \log n = 16 * 4 = 64$

Portanto, para uma lista de 16 elementos, ' $n * \log n$ ' seria igual a 64.



## Heap Sort





# Árvores

sábado, 20 de abril de 2024 09:52

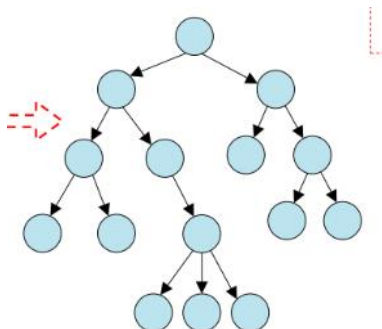
## Árvores

### O que são?

São um tipo especial de Grafo, são definidas como um conjunto não vazio de vértices (ou nós) e arestas que satisfazem certos requisitos para se conectarem.

É um esquema organizacional

As árvores são adequadas para representar estruturas hierárquicas não lineares, como [pastas \(estrutura de diretórios\)](#), busca de dados, campeonatos de modalidade esportiva, etc.



### Pontos importantes:

**Vértices:** é cada uma das entidades representadas na árvore, pode ser chamado de nó

**Aresta** - é uma conexão entre dois vértices

**Grafo conexo** - caminho entre os vértices

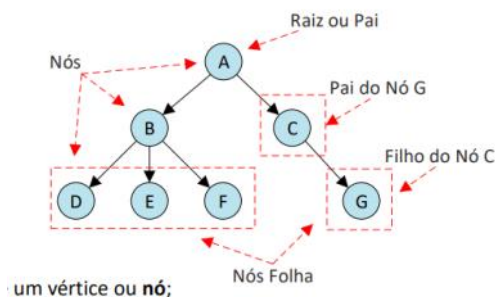
**Pai** - antecessor imediato de um vértice nó

**Filho** - sucessor imediato de um vértice

**Raiz** - vértice pai, que não há nenhum antes dele

**Nos terminais ou folhas** - qualquer vértice que não possui filhos

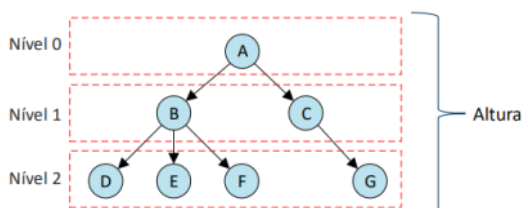
**Nós não terminais ou internos** - qualquer vértice que possua ao menos um filho



**Sub-árvores** - qualquer vértice pode ser a raiz de uma sub árvore

**Altura da árvore** - comprimento mais longo entre a raiz e as folhas

**Níveis** - número de nós no caminho entre o vértice e a raiz



## Árvore binária

Cada vértice pode possuir duas sub-árvores: sub-árvore esquerda e sub-árvore Direita

## Árvore binária

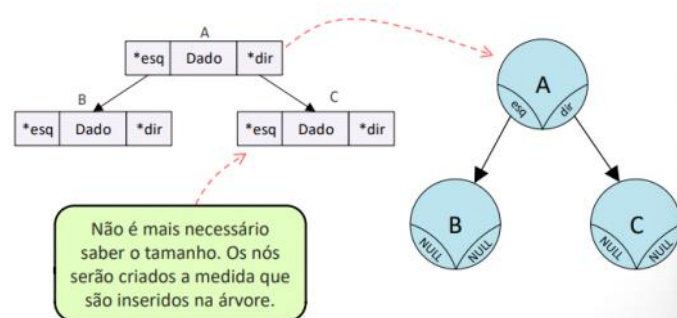
As operações nesse tipo de árvore dependem do tipo de alocação de memória usada:

- Estática (heap)
- Dinâmica (lista encadeada)

A **estática** é a mesma usada pelo **heap sort**, temos um array e acessamos as posições dos filhos por meio de uma conta.

Esse método tem desvantagens principalmente porque temos mais filhos a esquerda, além de gerar muito espaço vazio alocado

Já a **encadeada** usa o sistema de ponteiros e struct para apontar para seus filhos



## Implementação

**arvoreBinaria.h** – serão definidos tudo que ficará visível ao usuário programador desta biblioteca:

- Os protótipos das funções que manipulam o dado encapsulado;
- O tipo de dado que será armazenado na árvore;
- O ponteiro árvore que estará disponível para o main().

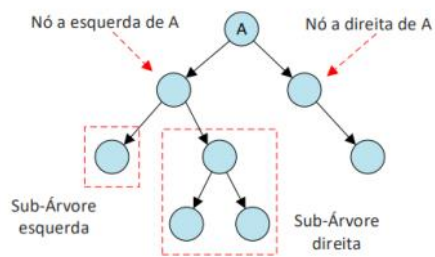
**arvoreBinaria.c** – serão definidos:

- O tipo de dado árvore (tipo opaco);
- A implementação de suas funções que manipulam o tipo opaco, fechando assim o encapsulamento.

## Operações principais

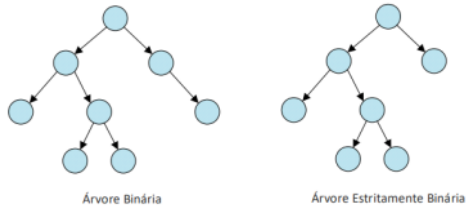
Em uma árvore Binária podemos realizar as seguintes operações

- Criação da árvore;
- Inserção de um elemento;
- Remoção de um elemento;
- Acesso à um elemento;
- Destruição da árvore.



### Árvore estritamente binária

Cada nó (vértice) possui 0 ou 2 sub-árvores;



### Árvore binária completa

É Estritamente Binária e todos os seus nós-Folha estão no mesmo nível.