Estrutura de Dados e Algoritmos

PROFº Augusto Galego

SUMÁRIO

[**1. Módulo 1: Estrutura de Dados**](#_4vgzxyidzywn) **8**

[1.1. Big O Notation](#_exddz93oc24i) 8

[Conceito](#_75znvsatum2o) 8

[Complexidade Temporal](#_czvkdounf1le) 8

[Complexidade Espacial (Space Complexity)](#_cxvsjmmd3sv2) 9

[Exemplo:](#_73m4bhgany9) 9

[Boas práticas para análise](#_p74pftqlhdba) 9

[1.2. Arrays](#_78x4ggoilse9) 10

[Conceito](#_to2fhnplpqu9) 10

[Estrutura de Memória](#_3hq7kj4e0wz3) 10

[Imutabilidade vs. Mutabilidade](#_3lqaj3o2rtx1) 10

[Arrays Mutáveis](#_so0rm0mmynro) 10

[Arrays Imutáveis](#_3d41hqyd48oh) 10

[Operações e Complexidade](#_653as53rv644) 10

[Quando usar Arrays?](#_o6eh952rbt4h) 11

[1.3. Linked List](#_rlk7dv1k9tvc) 11

[Estrutura de um Nó](#_hs7a5psz8azu) 12

[Em pseudocódigo:](#_3jsqz64oy529) 12

[Em C:](#_cek3ilxlx0tc) 12

[Tipos de Linked List](#_nli4lnngup04) 12

[1. Simplesmente Encadeada (Singly Linked List)](#_60asdg2w8qd) 12

[2. Duplamente Encadeada (Doubly Linked List)](#_vnt9p3qvyknh) 12

[3. Circular](#_882qt2i60sw4) 12

[Complexidade](#_oilrhco615sr) 12

[Comparação com Arrays](#_p62f7eubmz1c) 14

[Exemplo de implementação em C (Lista Simples)](#_gz4035i1j3x7) 14

[Aplicações Comuns](#_4hfdxth2n6uj) 15

[Desvantagens](#_bsdrikwnbmb9) 15

[1.4. Queue | Filas](#_puseombdmj8m) 15

[Definição](#_x6i16r4615j9) 15

[Características Principais](#_g5af1yf9pxzo) 15

[Operações Fundamentais](#_alu15x2lfsyp) 15

[Tipos de Filas](#_mv72lbwhgs6h) 16

[Fila Linear (Linear Queue)](#_7diiyonuk2lk) 16

[Fila Circular (Circular Queue)](#_vfqoozb4tjtq) 16

[Fila Duplamente Terminada (Deque)](#_ouam3afvqavd) 16

[Fila de Prioridade](#_q3k400ralxw5) 16

[Complexidade](#_eu9401np5ddk) 16

[1.5. Hashmap](#_9cc9l6p6dxgi) 17

[Definição](#_a58d841i4vxn) 17

[Como funciona?](#_2z2ffdt0i0e7) 17

[Estrutura Interna](#_5rphsuf65jth) 18

[Operações básicas](#_h6gpz0pebxdp) 18

[Colisões](#_8yg5h9mogkot) 18

[Métodos para tratar colisões:](#_3yjnbigtbacs) 19

[1. Encadeamento (Chaining)](#_pjtcz0upnk6u) 19

[2. Endereçamento Aberto (Open Addressing)](#_zftrxnky9hso) 19

[Funções de Hash](#_xd79yowj55in) 19

[Tamanho e Redimensionamento](#_g5l0qrfyipz3) 19

[Exemplo em Python (dicionário)](#_4e8k89cn3ksd) 20

[Exemplo visual](#_nawfyq5fb00q) 20

[Comparações](#_29f9uphm2jik) 20

[Aplicações](#_l76953ncfark) 21

[1.6. Stack | Pilha](#_xt64h0byqy3a) 21

[Definição](#_kfxd8m5iwepb) 21

[Características Principais](#_q09yhpsatha9) 21

[Operações Fundamentais](#_h6qctw3n2pah) 22

[Implementações](#_kosszpif68wy) 22

[Usando Array/Vetor](#_e7xgeerl4scu) 22

[Usando Lista Encadeada](#_him9fepwa513) 22

[Complexidade](#_pql48rx3bdxk) 23

[Uso de Pilhas na Prática](#_3qikjg8zvtab) 23

[Pilha x Fila](#_2w742wqycbty) 23

[Pilhas na Memória](#_60g2q4el3bxp) 24

[1.7. Binary Trees | Árvores Binárias](#_gt5b0dv9gibs) 24

[Definição](#_c68gewpiyo70) 24

[Representação de um nó](#_jshkpi9n50l4) 25

[Tipos de árvores binárias](#_9yijsto3sn3u) 25

[Complexidade (tempo e espaço)](#_jsnaalm02yok) 26

[Travessias (percursos)](#_l6l96hpzt1wf) 26

[1. Pré-Ordem (Pre-order)](#_buibw586ulk3) 26

[2. Em-Ordem (In-order)](#_oas1kgo612zi) 26

[3. Pós-Ordem (Post-order)](#_c0yt0mgex33c) 27

[4. Nível (Level-order)](#_6yk8bnxki38d) 27

[Inserção em uma BST](#_dno4ggrnjmm6) 27

[Busca em uma BST](#_nnww1q1eolca) 27

[Aplicações práticas](#_z65h0srhc7dq) 28

[1.8. Grafos](#_1uxxo4eqqwe9) 28

[Definição](#_adpf5bk17j0d) 28

[Tipos de grafos](#_6rqotb7g14ay) 28

[Representação de grafos](#_bpoxggtc6ynh) 29

[1. Lista de Adjacência (mais eficiente para grafos esparsos)](#_fhfcnt9msg4m) 29

[2. Matriz de Adjacência (eficiente para grafos densos)](#_4dg9ac8afv9c) 29

[Complexidade de operações](#_8y7pzptvmxmz) 29

[Algoritmos clássicos](#_sl8e3s3uq83b) 30

[1. Busca em Largura (BFS)](#_w1i2lyfcnb3s) 30

[2. Busca em Profundidade (DFS)](#_n48nzo7aeyj5) 30

[Algoritmos de caminhos mínimos](#_5d8q3lwx8qxv) 31

[Aplicações práticas](#_ye3x8t183lns) 31

[Desafios e armadilhas comuns](#_rl6pqtw2fixr) 31

[1.9. Trie](#_ir78w15nfagy) 31

[O que é uma Trie?](#_4agdd5qbe9xv) 31

[Características principais](#_ar3bymhkup9) 32

[Complexidade](#_dji9likl02uj) 32

[Variações de Trie](#_500hn7sqwd6l) 32

[Comparação com HashMap](#_xzjjysfvt75t) 32

[Conclusão](#_4isuc7kdt73l) 32

[1.10. B-Tree](#_crrg7mnol7fd) 33

[O que é uma B-Tree?](#_erfbdqgz83ie) 33

[Principais aplicações:](#_rnxumimybo4e) 33

[Características](#_y42xp7oyhec0) 33

[Regras da B-Tree de ordem m](#_exsy117ha69r) 34

[Complexidade](#_cll4n1q29go4) 34

[Comparação: B-Tree vs BST vs AVL vs Trie](#_h060ix5njm71) 35

[Conclusão](#_dy7we23bitms) 35

[**2. Módulo 2: Arrays**](#_kk6w1ic6x27p) **35**

[2.1. Two Pointer](#_wn1842yie3fm) 35

[O que é?](#_kf10ovzdzcqw) 35

[Quando usar?](#_tjhksj8mob5u) 35

[Tipos de Two Pointers](#_f5selwy2phdt) 36

[Vantagens](#_n15zphctepwl) 36

[Exemplo 1: Soma de dois números (array ordenado)](#_fpbalypm0hsy) 36

[Implementação em C:](#_domugy730ovy) 36

[Implementação em JavaScript:](#_8otkycaadh4i) 37

[Exemplo 2: Verificar se uma string é palíndromo](#_8gddgza35f9l) 37

[Exemplo 3: Remover duplicatas de array ordenado](#_wh0pnbvnoqwr) 37

[Cuidados e armadilhas](#_wwwqe3m8tpr7) 37

[Complexidades típicas](#_ylvv0u8jmrdi) 38

[Conclusão](#_rdq7ui6pytjf) 38

[2.2. Binary Search](#_qo0vsi9iq3oh) 38

[Objetivo](#_jom7pu1q2h43) 38

[Estratégia](#_oxre7te29ou) 38

[Condições](#_oq0tof2uh6jb) 38

[Complexidade](#_76nv6xmjt10x) 38

[Exemplo em C#:](#_6xtw9r5h3whv) 39

[2.3. Sliding Window](#_vhi8ufugdml) 39

[Objetivo:](#_w8buy89irg87) 39

[Tipos comuns:](#_16j1y0x9jr1q) 39

[Complexidade:](#_w4e14sjujvre) 39

[Exemplo em C# (maior substring sem caracteres repetidos):](#_9iy9e81t0gqw) 40

[2.4. Exponential Search](#_l6n5gd2hmklu) 40

[Objetivo:](#_soypj9ti9isg) 40

[Etapas:](#_ja30douea631) 40

[Complexidade:](#_e4lyw76nnu99) 40

[Exemplo básico (c#):](#_4fcwo9w1q1t7) 41

[2.5. Problema com HashMap](#_oi589ktpekvd) 41

[HashMap (dicionário) é útil para:](#_mrwfwer3kp8) 41

[Exemplo 1: Two Sum (Two Pointers + HashMap)](#_g1qkv8n5g09t) 41

[Exemplo 2: Frequência Máxima de Caracteres](#_jus9k6jzrqvp) 42

[Resumo Tabela](#_hx4hia7nwrbc) 42

[**3. Módulo 3: Linked List**](#_an5cwqeid9f) **42**

[3.1. Implementação Linked Lists](#_c5z1g0cdxcp2) 43

[Estrutura de um nó (em pseudocódigo/C#)](#_scr3zjx2p3l2) 43

[Estrutura da lista](#_rl87u6xmwvaf) 43

[3.2. Inverter uma linked list](#_u0wt87cv1e2q) 44

[Objetivo](#_a3avlgcb2o51) 44

[Exemplo](#_3cgd36jlqzag) 44

[Abordagem Iterativa (eficiente em tempo e espaço)](#_7ywpnq26og9p) 44

[Complexidade](#_4r54667tchdo) 44

[3.3. Encontrar o meio de uma linked list](#_aej6y7jqnm4n) 44

[Objetivo](#_s5zzmdn5kdhv) 44

[Abordagem com dois ponteiros (slow e fast)](#_cflpq6is3e1b) 44

[Como funciona](#_kklnuo9obrov) 45

[Complexidade](#_de1wmsikzemf) 45

[3.4. Encontrar ciclos na linked list](#_iq6pgjk6jpae) 45

[Problema](#_9qfuiiw8olvx) 45

[Abordagem de Floyd (Tartaruga e Lebre)](#_bechpykxxecm) 45

[Como funciona](#_9abkr6pgns5p) 45

[Complexidade](#_6bpt2y9610fg) 46

[Dicas Gerais para Entrevistas e Exercícios](#_h60us7b4p0ja) 46

[Desafios de Prática](#_snxerd3ye1st) 46

[**4. Módulo 4: Sorting**](#_666i0h3ec0hw) **46**

[4.1. BubbleSort](#_5t4ly74tiy9m) 46

[4.2. QuickSort](#_r45m21gyzoxn) 46

[4.3. MergeSort](#_hnufvrsahnsf) 46

[**5. Módulo 5: Binários**](#_3mmipkkqt87x) **46**

[5.1. Left e right shift](#_9sp3gzvh88pa) 46

[Left Shift (<<)](#_qxbye5bsiezm) 46

[Conceito:](#_8hi5ofgi3dbw) 46

[Exemplo:](#_vaym4hr6rxzg) 46

[Right Shift (>>)](#_3y3xqrtb5ckc) 46

[Conceito:](#_8xfs4icv163i) 47

[Exemplo:](#_ir7mli28vbm7) 47

[5.2. AND, OR, NOT, XOR](#_kywope3lfci5) 47

[AND (&)](#_hsmx2ek624ur) 47

[OR (|)](#_j8bgwwqtxwa4) 47

[XOR (^)](#_1wbxgik40obc) 47

[NOT (~)](#_uiaj971mdh7p) 47

[**6. Módulo 6: Binary Trees**](#_dpvob05a08s) **48**

[Definição](#_cw89cj6khc9g) 48

[Propriedades Fundamentais](#_zek5op69rma6) 48

[Classificações de Árvores Binárias](#_u4n77llasjs0) 48

[Percursos (Traversals)](#_u6v3n6l1ba63) 49

[Em Profundidade (Depth-First Traversal)](#_fzw5raienya1) 49

[Em Largura (Breadth-First Traversal)](#_2ncyg3n3qkk) 49

[Árvores Binárias de Busca (Binary Search Trees – BST)](#_qmvyh2y5htns) 49

[Operações Fundamentais](#_ltya66lo9x6) 50

[Complexidade](#_ojfkm3g9td2s) 50

[Árvores Binárias Balanceadas](#_2u6o5sjs3d7e) 50

[Aplicações Práticas](#_2m1vvetejeua) 50

[Considerações Finais](#_cdgba1rsuux3) 51

[Implementação (C#)](#_pw2zycbemcub) 51

[**7. Módulo 7: Grafos**](#_v27lp9slbivv) **55**

[7.1. Clonar um Grafo](#_op1rxsmdw6xy) 55

[Algoritmo de Clonagem](#_j8xt9ceucnx3) 55

[7.2. Algoritmo de Dijkstra](#_vkx9e9a9k52m) 56

[Funcionamento](#_rm7jdjprg7i0) 56

[Complexidade](#_duljwf56n509) 56

[7.3. Implementação do algoritmo de Dijkstra](#_hfev3q7rtnl5) 56

[**8. Módulo 8: Stack**](#_tceh9foob7jb) **57**

[8.1. Primeira implementação de stack](#_qaq2h3bk9iu7) 57

[Implementação em C# usando Lista](#_a1mspjfmoj7q) 57

[8.2. Implementando stack com linked list](#_oly5bypcmnl3) 57

[8.3. Implementando min stack](#_19u433wcfj6q) 59

[Conceito](#_qysmffzapwo5) 59

[Implementação em C#](#_8wj65eimvjfj) 59

[Exemplo de Uso](#_ogch4ahqrn8c) 61

[Características](#_78lr8vyp6os6) 61

[**9. Módulo 9: Heap**](#_aebuh5lwca3u) **61**

[9.1. Implementação de Heap](#_sev1j48kgoit) 61

[Conceito](#_jhsq2t4r16k9) 61

[Representação](#_f7vwy8yxloq9) 62

[Operações Principais](#_myxoqludvb9z) 62

[Exemplo de Max-Heap em C# usando Array](#_ruwx4g4jxskm) 62

[Exemplo de Uso](#_glvcidcfvdjc) 64

[Aplicações Comuns de Heap](#_p836jctw97pa) 64

[**Conclusão**](#_y0clpc8eenlp) **64**

[Livros Avançados:](#_76xtj9juhg8d) 64

[Cursos e Aulas Online:](#_76xtj9juhg8d) 64

# Módulo 1: Estrutura de Dados

## Big O Notation

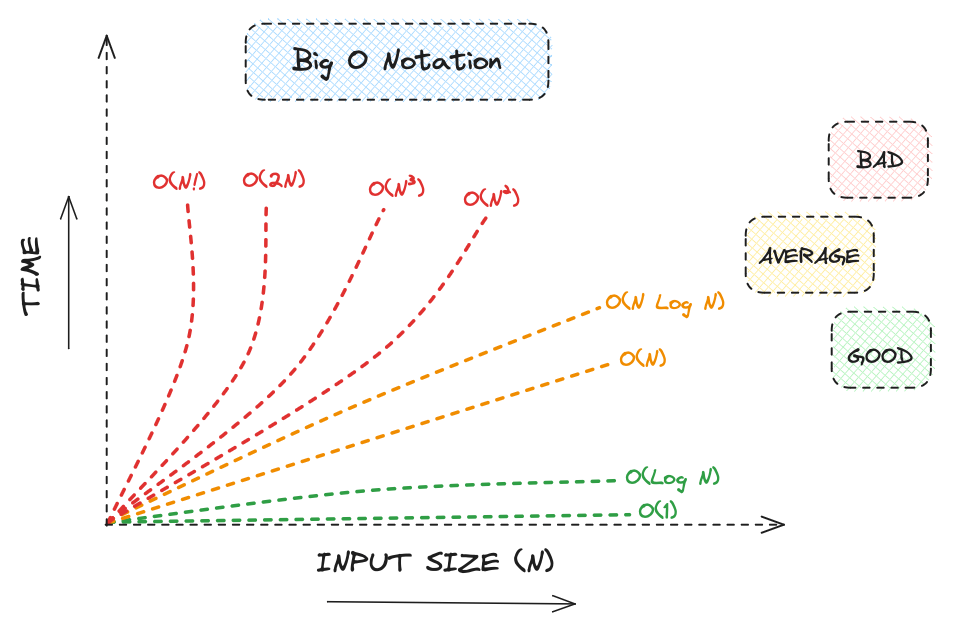
### Conceito

A ***Big O Notation*** é uma notação matemática que descreve o **comportamento assintótico** de um algoritmo. Ela representa como o tempo de execução (*tempo*) ou o uso de memória (*espaço*) de um algoritmo cresce conforme o tamanho da entrada aumenta.

**Foco:** Avaliar **eficiência/escalabilidade** de algoritmos, especialmente para grandes volumes de dados.

### Complexidade Temporal

A complexidade temporal mede **quantas operações** um algoritmo executa à medida que a entrada cresce.



| **Notação** | **Nome** | **Exemplo de uso** | **Descrição breve** |
| --- | --- | --- | --- |
| **O(1)** | Constante | return arr[0] | Sempre leva o mesmo tempo, não importa o tamanho da entrada. |
| **O(log n)** | Logarítmica | binary search(arr, x) | Divide a entrada pela metade a cada passo. |
| **O(n)** | Linear | for (i = 0; i < n; i++) | O tempo cresce proporcionalmente à entrada. |
| **O(n log n)** | Quase-linear | merge sort(arr) | Divisão e conquista eficientes. |
| **O(n²)** | Quadrática | bubble sort(arr) | Duplo laço – cresce muito rápido com n. |
| **O(2ⁿ)** | Exponencial | solve(subset problem) | Inviável para entradas grandes. |
| **O(n!)** | Fatorial | permute(n elementos) | Custo extremo — cresce mais rápido que qualquer outra. |

### Complexidade Espacial (Space Complexity)

A complexidade espacial analisa **quanta memória adicional** o algoritmo consome.

### **Exemplo:**

def soma\_array(arr):

soma = 0 # ocupa espaço constante → O(1)

for i in arr: # laço consome O(1) espaço extra

soma += i

return soma

→ Complexidade espacial: **O(1)**→ Complexidade temporal: **O(n)** (o laço percorre todos os elementos)

### Boas práticas para análise

1. **Ignore constantes**:  
    O(2n) → **O(n)** O(5 + n) → **O(n)**
2. **Considere apenas o termo dominante**:  
    O(n² + n + 1) → **O(n²)**
3. **Loops aninhados multiplicam complexidade**:  
    Duplo for → O(n²)  
    Triplo for → O(n³)
4. **Loops sequenciais somam**:  
    Dois for em sequência → O(n) + O(n) = **O(n)**

## Arrays

### **Conceito**

Um ***array*** é uma **estrutura de dados linear** que armazena múltiplos elementos do mesmo tipo em posições contíguas na memória.

### Estrutura de Memória

Os elementos de um array são armazenados **lado a lado** na RAM.

Isso permite **acesso direto** via índice:

endereco\_base + ( índice × tamanho\_do\_tipo)

É por isso que **o acesso por índice em arrays é O(1)**.

### Imutabilidade vs. Mutabilidade

#### Arrays Mutáveis

* Suportam **modificações em tempo real**.
* Exemplo: int arr[5] = {1, 2, 3, 4, 5}; em C.
* Linguagens como C, C++, Java e Python (listas) permitem modificar valores de um array existente.

#### Arrays Imutáveis

* Após a criação, **não podem ser modificados** (sem recriar outro array).
* Exemplo: Tuplas em Python ((1, 2, 3)).
* Em linguagens funcionais (como Haskell), arrays são geralmente imutáveis.

### Operações e Complexidade

| **Operação** | **Complexidade** | **Observações** |
| --- | --- | --- |
| Acesso (por índice) | O(1) | Direto via ponteiro |
| Atualização (por índice) | O(1) | Similar ao acesso |
| Inserção (no fim sem redimensionar) | O(1) | Apenas incrementa o índice |
| Inserção (no fim com realocação) | O(n) | Cria novo array e copia todos os dados |
| Inserção (no início/meio) | O(n) | Deslocamento de elementos posteriores |
| Remoção (início ou meio) | O(n) | Necessita deslocar elementos à esquerda |
| Busca linear | O(n) | Sem ordenação, deve percorrer todos os elementos |
| Busca binária | O(log n) | Exige array **ordenado** |

## 

### Quando usar Arrays?

* Quando você sabe o **tamanho fixo da coleção**.
* Precisa de **acesso rápido** por índice.
* Poucas inserções/remoções no meio da estrutura.
* Precisa de **baixa sobrecarga de memória**.

## Linked List

Uma ***Linked List*** é uma estrutura de dados linear composta por **nós (nodes)**, onde cada nó armazena:

* Um **valor/dado**
* Um **ponteiro** (ou referência) para o **próximo nó** da lista

Ao contrário dos arrays, os elementos **não ocupam posições contíguas na memória**.

A navegação ocorre através de ponteiros, formando uma **cadeia de nós**.

### Estrutura de um Nó

#### Em pseudocódigo:

Node {

valor

próximo → Node

}

#### Em C:

typedef struct Node {

int valor;

struct Node\* proximo;

} Node;

### Tipos de Linked List

#### 1. Simplesmente Encadeada (Singly Linked List)

Cada nó aponta para o próximo:

[1|\*] → [2|\*] → [3|NULL]

#### 2. Duplamente Encadeada (Doubly Linked List)

Cada nó aponta para o anterior e o próximo:

NULL ← [1|\*|\*] ↔ [2|\*|\*] ↔ [3|\*|NULL]

#### 3. Circular

O último nó aponta de volta para o primeiro:

[1|\*] → [2|\*] → [3|\*] ↻

Circular ou não, simples ou dupla, essas variações servem para **otimizar operações específicas**.

### Complexidade

| **Operação** | **Tempo (Singly)** | **Tempo (Doubly)** |
| --- | --- | --- |
| Acesso (por índice) | O(n) | O(n) |
| Inserção (início) | O(1) | O(1) |
| Inserção (fim)\* | O(n)\* ou O(1)\*\* | O(n)\* ou O(1)\*\* |
| Remoção (início) | O(1) | O(1) |
| Remoção (fim) | O(n) | O(1) |
| Busca por valor | O(n) | O(n) |

\*O(n) se for necessário **percorrer até o fim**

\*\*O(1) se a lista **armazenar ponteiro para o fim (tail)**

### Comparação com Arrays

| **Característica** | **Array** | **Lista Encadeada** |
| --- | --- | --- |
| Acesso por índice | O(1) | O(n) |
| Inserção (meio/início) | O(n) | O(1)\* |
| Remoção (meio/início) | O(n) | O(1)\* |
| Uso de memória | Contíguo | Não contíguo |
| Cache da CPU | Mais eficiente | Menos eficiente |
| Redimensionamento | Complexo | Fácil (dinâmico) |

\*Inserção/remoção O(1) **somente se o nó de referência for conhecido**.

### Exemplo de implementação em C (Lista Simples)

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int valor;

struct Node\* proximo;

} Node;

void inserir\_inicio(Node\*\* cabeca, int valor) {

Node\* novo = malloc(sizeof(Node));

novo->valor = valor;

novo->proximo = \*cabeca;

\*cabeca = novo;

}

void imprimir(Node\* atual) {

while (atual != NULL) {

printf("%d -> ", atual->valor);

atual = atual->proximo;

}

printf("NULL\n");

}

### Aplicações Comuns

1. Implementação de **pilhas (stack)** e **filas (queue)**
2. **Sistemas operacionais**: gerenciamento de processos, buffers
3. **Playlists**, **navegação com histórico (back/forward)**
4. Estruturas como **hash tables com encadeamento**

### Desvantagens

1. Acesso aleatório é **ineficiente** (precisa percorrer a lista).
2. Maior uso de memória (cada nó armazena ponteiros).
3. **Custo de gerenciamento de ponteiros** (especialmente ao remover nós).

## Queue | Filas

### Definição

Uma **fila (queue)** é uma estrutura de dados linear que segue o princípio **FIFO**:

**First In, First Out** O primeiro elemento a entrar é o primeiro a sair — como uma fila de banco, tráfego ou impressão.

### Características Principais

1. **Ordem** importa: elementos são processados na ordem de chegada.
2. Inserções ocorrem no **fim da fila** (traseira).
3. Remoções ocorrem no **início da fila** (frente).

### Operações Fundamentais

| **Operação** | **Descrição** | **Complexidade** |
| --- | --- | --- |
| enqueue | Adiciona elemento ao final | O(1) |
| dequeue | Remove e retorna o elemento da frente | O(1) |
| peek | Retorna o elemento da frente sem remover | O(1) |
| isEmpty | Verifica se a fila está vazia | O(1) |

### Tipos de Filas

#### Fila Linear (Linear Queue)

* Simples: adiciona no final e remove do início.
* Problemas: o espaço inicial pode ficar “perdido” após remoções.

#### Fila Circular (Circular Queue)

* Usa um array fixo, onde o final da fila se conecta ao início.
* Evita desperdício de espaço.

#### Fila Duplamente Terminada (Deque)

* Permite inserções e remoções nas **duas extremidades**.
* Usada em algoritmos como o de janela deslizante.

#### Fila de Prioridade

* Cada elemento tem uma **prioridade**.
* O elemento com maior prioridade é processado primeiro (não necessariamente o primeiro a entrar).

### Complexidade

| **Implementação** | **enqueue** | **dequeue** | **peek** | **espaço** |
| --- | --- | --- | --- | --- |
| Array | O(1)\* | O(n) | O(1) | O(n) |
| Lista Encadeada | O(1) | O(1) | O(1) | O(n) |
| Fila Circular | O(1) | O(1) | O(1) | O(n) |

\* O(1) se não precisar redimensionar (em arrays dinâmicos)

## Hashmap

### Definição

Um **hashmap** é uma estrutura de dados que **associa pares chave → valor**, oferecendo acesso extremamente rápido aos dados por meio de uma **função de hash**.

Exemplo:  
 map["nome"] = "Rafael"  
 "nome" é a **chave**, "Rafael" é o **valor**

### Como funciona?

1. **Chave (key)** é passada para uma **função de hash**.
2. A função retorna um **índice inteiro**.
3. O valor é armazenado nesse índice dentro de um array interno.

"Rafael" → hash("Rafael") = 2 → armazenado em índice 2

### 

### 

### Estrutura Interna

Internamente, um hashmap geralmente é:

* Um **array de buckets** (listas ou ponteiros)
* Cada bucket pode conter:  
  + Nenhum valor
  + Um único item
  + Uma **lista encadeada** (em caso de colisão)
  + Uma **árvore balanceada** (em implementações modernas, como no Java 8+)

### Operações básicas

| **Operação** | **Descrição** | **Complexidade média** | **Complexidade pior caso** |
| --- | --- | --- | --- |
| t(k,v) | Insere ou atualiza valor da chave k | O(1) | O(n) |
| t(k) | Retorna o valor associado a k | O(1) | O(n) |
| e(k) | Remove o par chave/valor | O(1) | O(n) |
| ins(k) | Verifica se a chave existe | O(1) | O(n) |

*No pior caso (muitas colisões), o desempenho pode cair para O(n)* → Implementações modernas minimizam isso com árvores ou hashing eficiente.

### Colisões

Duas chaves diferentes podem gerar o **mesmo índice**. Isso é chamado de **colisão**.

### Métodos para tratar colisões:

#### 1. Encadeamento (Chaining)

Cada posição do array contém uma **lista de pares chave/valor**.

array[5] → [ ("nome", "Rafael"), ("idade", 22) ]

#### 2. Endereçamento Aberto (Open Addressing)

Se a posição estiver ocupada, procura-se **outra posição livre**.

* **Linear probing**: próxima posição (i+1, i+2, …)
* **Quadratic probing**: salta em potências (i+1², i+2², …)
* **Double hashing**: usa uma segunda função de hash

### Funções de Hash

Uma **boa função de hash**:

* Deve ser **determinística**
* Deve espalhar as chaves **uniformemente**
* Deve ser **rápida** de computar
* Evita muitas colisões

Exemplos:

* hash(k) = k % N (simples)
* std::hash (C++), hash() (Python), Objects.hash() (Java)

### Tamanho e Redimensionamento

* Ao atingir um certo **fator de carga** (load factor), o hashmap é **redimensionado** (ex: dobra de tamanho).
* Fator de carga = número de elementos / tamanho da tabela
* Ex: Se a tabela tem tamanho 16 e 12 elementos → load factor = 0.75

**Redimensionamento é custoso (O(n))**, mas ocorre raramente.

### 

### 

### Exemplo em Python (dicionário)

agenda = {}

agenda["joao"] = "9999-1111"

agenda["ana"] = "8888-2222"

print(agenda["joao"]) # → 9999-1111

print("ana" in agenda) # → True

→ Em Python, dict é uma **hashmap** otimizada

### Exemplo visual

Tamanho da tabela: 10

Chave | Hash | Índice

------------|------------|--------

"joao" | 834122 | 2

"ana" | 583918 | 8

"rafael" | 382918 | 2 (colisão!)

Bucket 2 conterá uma **lista encadeada** com "joao" e "rafael".

### Comparações

| **Estrutura** | **Acesso por chave** | **Acesso por índice** | **Ordenação** |
| --- | --- | --- | --- |
| Hashmap | O(1) | Não | Não |
| Array | Não | O(1) | Simples |
| Lista ligada | O(n) | O(n) | Linear |
| Árvores BST | O(log n) | Não | Sim |

### Aplicações

* Implementar dicionários de palavras
* Contar frequência de caracteres ou palavras
* Caches (ex: LRU cache)
* Lookup rápido (como alunos por RA, clientes por CPF)
* Implementar conjuntos (set)

## Stack | Pilha

### Definição

Uma **pilha** é uma estrutura de dados linear que segue o princípio **LIFO**:

**Last In, First Out** O último elemento inserido é o primeiro a ser removido.

Imagine uma pilha de pratos: você empilha de cima, e sempre retira o último colocado primeiro.

### Características Principais

* Acesso **apenas ao topo da pilha**.
* Não permite acesso aleatório a outros elementos.
* Toda inserção e remoção é feita no topo.

### Operações Fundamentais

| **Operação** | **Descrição** | **Complexidade** |
| --- | --- | --- |
| push(x) | Insere o elemento x no topo | O(1) |
| pop() | Remove e retorna o topo | O(1) |
| peek() | Retorna o topo sem remover | O(1) |
| isEmpty() | Verifica se a pilha está vazia | O(1) |
| size() | Retorna o número de elementos na pilha | O(1) |

### Implementações

#### Usando Array/Vetor

stack = []

stack.append(10) # push

stack.append(20)

print(stack.pop()) # 20 (pop)

print(stack[-1]) # 10 (peek)

* Simples e eficiente.
* Mas pode ter redimensionamento (custo amortizado O(1)).

#### Usando Lista Encadeada

typedef struct Node {

int valor;

struct Node\* proximo;

} Node;

Node\* topo = NULL;

void push(int val) {

Node\* novo = malloc(sizeof(Node));

novo->valor = val;

novo->proximo = topo;

topo = novo;

}

int pop() {

if (topo == NULL) return -1;

int val = topo->valor;

Node\* temp = topo;

topo = topo->proximo;

free(temp);

return val;

}

* Ideal para quando não se sabe o tamanho final da pilha.
* Boa performance, sem redimensionamento.

### Complexidade

| **Operação** | **Array** | **Lista Encadeada** |
| --- | --- | --- |
| push | O(1)\* | O(1) |
| pop | O(1) | O(1) |
| peek | O(1) | O(1) |
| espaço | O(n) | O(n) |

\*O(1) amortizado. Em arrays, o push pode disparar realocação (resize).

### Uso de Pilhas na Prática

Pilhas são usadas quando há **retrocesso**, **ordem reversa**, ou **gerenciamento de contexto**. Exemplos:

1. **Recursão** (a pilha de chamadas do sistema)
2. **Backtracking** (sudoku, labirintos)
3. **Navegador (voltar página)**
4. **Desfazer (CTRL+Z)**
5. **Conversão de expressões (infixa → pós-fixa)**
6. **Avaliação de expressões pós-fixas**
7. **Balanceamento de parênteses**

### Pilha x Fila

| **Conceito** | **Stack (Pilha)** | **Queue (Fila)** |
| --- | --- | --- |
| Ordem | LIFO | FIFO |
| Inserção | Topo | Final |
| Remoção | Topo | Início |
| Uso comum | Backtracking, Recursão | Agendamento, Buffers |

### Pilhas na Memória

A pilha (stack) é também usada na **execução de programas**, especialmente em chamadas de funções recursivas:

int fatorial(int n) {

if (n == 0) return 1;

return n \* fatorial(n - 1);

}

Cada chamada ocupa **um quadro (stack frame)** na pilha, contendo:

1. Parâmetros da função
2. Variáveis locais
3. Endereço de retorno

## Binary Trees | Árvores Binárias

### Definição

Uma **árvore binária** é uma estrutura de dados hierárquica em que cada **nó** possui no máximo **dois filhos**, denominados:

* **Esquerdo (left)**
* **Direito (right**

A árvore possui um **nó raiz (root)**, e cada filho também pode ser a raiz de uma subárvore.

### Representação de um nó

typedef struct No {

int valor;

struct No\* esquerdo;

struct No\* direito;

} No;

### Tipos de árvores binárias

* **Árvore binária cheia**: cada nó tem 0 ou 2 filhos
* **Árvore binária completa**: todos os níveis estão completos, exceto talvez o último (preenchido da esquerda para a direita)
* **Árvore balanceada**: a diferença de altura entre subárvores esquerda e direita é no máximo 1
* **Árvore de busca binária (BST)**: os nós obedecem à propriedade de ordenação:  
  + Para cada nó n: todos os valores na subárvore esquerda são **menores** que n.valor
  + Todos os valores na subárvore direita são **maiores**

### Complexidade (tempo e espaço)

| **Operação** | **Árvore balanceada (AVL/Red-Black)** | **Árvore desbalanceada (BST comum)** |
| --- | --- | --- |
| Inserção | O(log n) | O(n) |
| Busca | O(log n) | O(n) |
| Remoção | O(log n) | O(n) |
| Travessia (inorder) | O(n) | O(n) |
| Espaço | O(n) | O(n) |

O desempenho ideal depende do **balanceamento** da árvore.

### Travessias (percursos)

As travessias visitam todos os nós da árvore em uma determinada ordem.

#### 1. Pré-Ordem (Pre-order)

Visita o nó atual → Esquerda → Direita

A

├── B

│ ├── D

│ └── E

└── C

├── F

└── G

Pré-ordem: A B D E C F G

#### 2. Em-Ordem (In-order)

Esquerda → Nó atual → Direita  
 Útil em **BST**, pois resulta em **ordem crescente**.

In-ordem: D B E A F C G

#### 3. Pós-Ordem (Post-order)

Esquerda → Direita → Nó atual

Pós-ordem: D E B F G C A

#### 4. Nível (Level-order)

Travessia em **largura** (Breadth-First Search – BFS)

Nível: A B C D E F G

### Inserção em uma BST

No\* inserir(No\* raiz, int valor) {

if (raiz == NULL) {

No\* novo = malloc(sizeof(No));

novo->valor = valor;

novo->esquerdo = novo->direito = NULL;

return novo;

}

if (valor < raiz->valor)

raiz->esquerdo = inserir(raiz->esquerdo, valor);

else if (valor > raiz->valor)

raiz->direito = inserir(raiz->direito, valor);

return raiz;

}

### Busca em uma BST

No\* buscar(No\* raiz, int valor) {

if (raiz == NULL || raiz->valor == valor)

return raiz;

if (valor < raiz->valor)

return buscar(raiz->esquerdo, valor);

return buscar(raiz->direito, valor);

}

### Aplicações práticas

* **Árvores de busca** (BST) para dados ordenados
* **Expressões matemáticas** (árvores de expressão)
* **Árvores de decisão**
* **Compiladores** (sintaxe e semântica)
* **Sistemas de arquivos hierárquicos**
* **Heaps** (para algoritmos de ordenação e filas de prioridade)

## Grafos

### 

### Definição

Um **grafo** é uma estrutura que representa **relações** entre pares de elementos.  
 Ele é composto por:

* **Vértices (ou nós)**: entidades ou pontos do grafo
* **Arestas (ou arcos)**: conexões entre os vértices

### Tipos de grafos

| **Tipo** | **Descrição** |
| --- | --- |
| **Não direcionado** | Arestas não têm direção: (u, v) = (v, u) |
| **Direcionado (dígrafo)** | Arestas têm direção: (u → v) ≠ (v → u) |
| **Ponderado** | Arestas têm pesos (custos, distâncias, etc.) |
| **Não ponderado** | Arestas sem peso |
| **Cíclico** | Possui ciclos (caminhos que retornam ao mesmo nó) |
| **Acíclico (DAG)** | Sem ciclos |
| **Conexo** | É possível ir de qualquer nó a qualquer outro (em grafos não direcionados) |
| **Fortemente conexo** | Em grafos direcionados, há caminhos de ida e volta entre quaisquer dois vértices |

### Representação de grafos

#### 1. Lista de Adjacência (mais eficiente para grafos esparsos)

// Em C (exemplo simples):

typedef struct No {

int destino;

struct No\* proximo;

} No;

No\* grafo[MAX\_VERTICES];

#### 2. Matriz de Adjacência (eficiente para grafos densos)

// Para grafo com 5 vértices

int matriz[5][5] = {

{0, 1, 1, 0, 0},

{1, 0, 1, 1, 0},

...

};

| **Representação** | **Espaço** | **Vantagem** |
| --- | --- | --- |
| Lista de adjacência | O(V + E) | Eficiente para grafos grandes e esparsos |
| Matriz de adjacência | O(V²) | Boa para grafos densos ou com acesso direto a vizinhos |

### 

### 

### 

### Complexidade de operações

| **Operação** | **Lista de Adjacência** | **Matriz de Adjacência** |
| --- | --- | --- |
| Verificar vizinho direto | O(grau do nó) | O(1) |
| Inserir aresta | O(1) | O(1) |
| Remover aresta | O(grau do nó) | O(1) |
| AIterar sobre vizinhos | O(grau do nó) | O(V) |

## 

## 

## 

## 

### Algoritmos clássicos

#### 1. Busca em Largura (BFS)

* Explora por níveis (ideal para encontrar caminhos mínimos não ponderados)

from collections import deque

def bfs(grafo, inicio):

visitado = set()

fila = deque([inicio])

while fila:

atual = fila.popleft()

if atual not in visitado:

visitado.add(atual)

fila.extend(grafo[atual])

* **Complexidade**: O(V + E)

#### 2. Busca em Profundidade (DFS)

* Explora ao máximo antes de voltar

def dfs(grafo, v, visitado=None):

if visitado is None:

visitado = set()

visitado.add(v)

for viz in grafo[v]:

if viz not in visitado:

dfs(grafo, viz, visitado)

* **Complexidade**: O(V + E)

### Algoritmos de caminhos mínimos

| **Algoritmo** | **Ponderado?** | **Negativo?** | **Complexidade** |
| --- | --- | --- | --- |
| Dijkstra | Sim | Não | O((V + E) log V) |
| Bellman-Ford | Sim | Sim | O(VE) |
| Floyd-Warshall | Sim | Sim | O(V³) |
| A\* | Sim | Não | Heurística |

### Aplicações práticas

* **Redes de computadores** (roteamento, conexões)
* **Mapas e navegação GPS**
* **Engenharia elétrica** (circuitos)
* **Planejamento de tarefas** (DAGs)
* **Relacionamentos sociais** (redes sociais)
* **Compiladores** (ordem de dependências)
* **IA e jogos** (busca de caminhos)

### Desafios e armadilhas comuns

1. Detecção de ciclos (em DFS)
2. Componentes fortemente conectados
3. Grafos direcionados x não direcionados
4. Caminhos negativos (exigem algoritmos apropriados)

## Trie

### O que é uma Trie?

**Trie** (pronuncia-se “trai”) é uma estrutura de dados em forma de **árvore** especializada para armazenar **conjuntos de strings**, com foco em **busca por prefixo**.  
 Ela é muito utilizada em:

* **Sistemas de autocomplete**
* **Corretores ortográficos**
* **Busca em dicionários**
* **IP routing (em redes)**
* **Compressão de texto**

### Características principais

* Cada **nó** representa **um caractere** de uma palavra.
* Um **caminho da raiz até um nó folha** representa uma **palavra completa**.
* Cada **nó pode ter até N filhos**, onde **N = número de caracteres possíveis** (geralmente 26 para letras minúsculas a-z).
* Não há repetição de palavras.
* Armazena **compartilhamento de prefixos** eficientemente.

### Complexidade

| **Operação** | **Tempo** | **Observações** |
| --- | --- | --- |
| Inserção | O(m) | m = comprimento da string |
| Busca | O(m) | Muito rápida |
| Remoção | O(m) | Requer verificar se é folha |
| Espaço | O(ALPHABET\_SIZE × m × n) | Pode ser alto, especialmente sem compressão |

## 

## 

## 

## 

### Variações de Trie

* **Compressed Trie (Radix Tree)**: otimiza espaço ao unir caminhos únicos
* **Suffix Trie**: armazena todos os sufixos de uma string
* **Ternary Search Trie**: mistura busca binária com árvore de prefixos

### Comparação com HashMap

| **Aspecto** | **Trie** | **HashMap** |
| --- | --- | --- |
| Busca por prefixo | Sim | Não (ineficiente) |
| Inserção ordenada | Sim | Não |
| Espaço | Maior | Menor |
| Velocidade de busca | Muito alta | Alta (por palavra exata) |

### 

### 

### 

### 

### 

### Conclusão

**Trie** é extremamente útil quando precisamos de:

* Busca rápida por palavras completas e prefixos
* Autocompletar sugestões com base em input parcial
* Armazenar muitos textos com prefixos semelhantes

Apesar de consumir mais memória que outras estruturas, seu desempenho é excelente em aplicações linguísticas e busca inteligente.

## B-Tree

### O que é uma B-Tree?

A **B-Tree (Árvore-B)** é uma **árvore de busca balanceada de múltiplos caminhos**, generalizando a estrutura de árvore binária de busca (BST). Diferente das árvores binárias, cada nó pode ter **vários filhos e várias chaves**, o que a torna ideal para **acesso em disco ou SSD**, onde minimizar a quantidade de leituras é crucial.

### Principais aplicações:

* Sistemas de banco de dados (ex: MySQL, PostgreSQL)
* Sistemas de arquivos (ex: NTFS, HFS+)
* Indexação de dados em memória secundária

### Características

* Cada nó pode conter **m-1 chaves** e até **m filhos**, onde **m** é a ordem da árvore.
* Os dados são mantidos **ordenados**.
* A árvore é sempre **balanceada**: todas as folhas estão no mesmo nível.
* Cresce e diminui de forma **eficiente e controlada**.
* Permite **busca, inserção e remoção** em tempo logarítmico.

### Regras da B-Tree de ordem m

1. Cada nó interno pode ter, no máximo, m - 1 chaves.
2. Cada nó (exceto a raiz) deve ter, no mínimo, ceil(m/2) - 1 chaves.
3. Todos os nós folha devem estar no mesmo nível.
4. A raiz pode ter entre 1 e m - 1 chaves.
5. As chaves dentro de cada nó estão **ordenadas**.
6. A subárvore entre duas chaves k[i] e k[i+1] contém **apenas valores entre** essas chaves.

### Complexidade

| **Operação** | **Tempo** | **Observações** |
| --- | --- | --- |
| Busca | O(log n) | Alta performance mesmo com grandes dados |
| Inserção | O(log n) | Ocorre split quando necessário |
| Remoção | O(log n) | Pode envolver fusão de nós |
| Espaço | O(n) | Armazena eficientemente dados grandes |

## 

## 

## 

## 

## 

## 

### Comparação: B-Tree vs BST vs AVL vs Trie

| **Estrutura** | **Melhor uso** | **Balanceamento** | **Tempo de busca** |
| --- | --- | --- | --- |
| BST | Dados pequenos e dinâmicos | Não garantido | O(n) no pior |
| AVL | Dados dinâmicos e balanceamento rigoroso | Rígido | O(log n) |
| Trie | Strings e prefixos | Parcial | O(m) por string |
| B-Tree | Dados grandes em disco ou banco de dados | Simples | O(log n) |

## 

## 

## 

### Conclusão

A **B-Tree** é uma estrutura altamente eficiente para dados em **largas quantidades**, onde o acesso a disco é lento e o custo de leitura precisa ser minimizado. Seu uso é padrão em bancos de dados relacionais, sistemas de arquivos e indexadores.

# Módulo 2: *Arrays*

## Two Pointer

### O que é?

A técnica **Two Pointers** consiste em usar **dois índices (ou ponteiros)** para percorrer estruturas lineares como arrays, listas ou strings. O objetivo é resolver problemas de forma mais eficiente, frequentemente reduzindo a complexidade de tempo de algoritmos de O(n²) para O(n).

### Quando usar?

Essa técnica é útil em problemas que envolvem:

* Comparar elementos de extremidades opostas
* Encontrar pares com soma igual a um valor (Two Sum)
* Identificar subarrays ou substrings com restrições
* Detectar palíndromos
* Processar elementos repetidos (remoção de duplicatas)
* Encontrar janelas máximas ou mínimas em intervalos (sliding window)

### Tipos de Two Pointers

| **Tipo** | **Posição dos ponteiros** | **Aplicações comuns** |
| --- | --- | --- |
| Início e fim | left = 0, right = n - 1 | Busca de pares, palíndromos, compressão |
| Dois crescentes (slow, fast) | Ambos do início para frente | Remoção de duplicatas, detectores de ciclos |
| Sliding window | Ponteiros ajustam uma janela | Substrings, soma de subarrays, janelas fixas |

### Vantagens

* Redução significativa na complexidade de tempo
* Simples de implementar
* Consome espaço constante (O(1)), ideal para otimizações
* Evita estruturas adicionais como listas ou mapas em muitos casos

### Exemplo 1: Soma de dois números (array ordenado)

**Problema:** Dado um array ordenado e um valor-alvo target, determinar se existe um par de elementos cuja soma é igual a target.

**Estratégia:**

* Um ponteiro começa no início (left), outro no fim (right)
* Compare arr[left] + arr[right]  
  + Se for igual ao target, par encontrado
  + Se for menor, incremente left
  + Se for maior, decremente right

### Implementação em C:

bool existeParComSoma(int arr[], int n, int target) {

int left = 0, right = n - 1;

while (left < right) {

int soma = arr[left] + arr[right];

if (soma == target) return true;

else if (soma < target) left++;

else right--;

}

return false;

}

### Implementação em JavaScript:

function temParComSoma(arr, target) {

let left = 0, right = arr.length - 1;

while (left < right) {

const soma = arr[left] + arr[right];

if (soma === target) return true;

else if (soma < target) left++;

else right--;

}

return false;

}

### Exemplo 2: Verificar se uma string é palíndromo

**Estratégia:**

* Ponteiros em left = 0 e right = len - 1
* Compare caracteres s[left] e s[right] enquanto left < right
* Se todos forem iguais, é palíndromo

### Exemplo 3: Remover duplicatas de array ordenado

**Estratégia:**

* Use dois ponteiros (slow e fast)
* fast percorre todos os elementos
* slow aponta para a próxima posição de gravação de valor único

### Cuidados e armadilhas

* Não esquecer de mover o ponteiro adequado dentro do laço
* Evite ultrapassar os limites do array
* Confirme se o array está ordenado (quando necessário)
* Trate casos especiais: array vazio, string com espaços múltiplos, etc.

### Complexidades típicas

| **Operação** | **Tempo** | **Espaço** |
| --- | --- | --- |
| Comparação de pares | O(n) | O(1) |
| Verificação de palíndromo | O(n) | O(1) |
| Sliding window | O(n) | O(1) |
| Busca com soma alvo | O(n) | O(1) |

### 

### 

### 

### 

### 

### Conclusão

A técnica **Two Pointers** é uma estratégia poderosa para resolver diversos problemas que envolvem análise de elementos em estruturas sequenciais. Seu uso adequado permite otimizar algoritmos, melhorar desempenho e reduzir complexidade de forma elegante e eficiente.

## Binary Search

### Objetivo

Encontrar a posição de um valor target em um array ordenado.

### Estratégia

* Utiliza os ponteiros left e right para limitar o intervalo de busca.
* Calcula o índice do meio: mid = left + (right - left) / 2.
* Compara nums[mid] com o target e reduz o intervalo pela metade.

### Condições

* A busca **só funciona em arrays ordenados**.

### Complexidade

* **Tempo:** O(log n)
* **Espaço:** O(1)

### Exemplo em C#:

int BinarySearch(int[] nums, int target) {

int left = 0, right = nums.Length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target)

return mid;

else if (nums[mid] < target)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

## Sliding Window

### Objetivo:

Processar substrings, subarrays ou sequências de forma eficiente, com base em uma **janela móvel de elementos**.

### **Tipos comuns:**

* **Janela fixa:** usada para somas médias, contagem etc.
* **Janela variável:** expande ou reduz dinamicamente conforme uma condição (ex: frequência de caracteres).

### **Complexidade:**

* **Tempo:** O(n)
* **Espaço:** O(k) onde k é o número de elementos distintos na janela (ex: 26 letras).

### **Exemplo em C# (maior substring sem caracteres repetidos):**

int LengthOfLongestSubstring(string s) {

var set = new HashSet<char>();

int left = 0, maxLen = 0;

for (int right = 0; right < s.Length; right++) {

while (set.Contains(s[right])) {

set.Remove(s[left]);

left++;

}

set.Add(s[right]);

maxLen = Math.Max(maxLen, right - left + 1);

}

return maxLen;

}

## Exponential Search

### Objetivo:

Buscar em **arrays muito grandes**, onde o tamanho pode ser desconhecido ou o custo da leitura é alto (como em streams, arquivos, APIs etc.).

### **Etapas:**

1. Expandir a janela de busca exponencialmente (2^i) até encontrar um valor maior que o target ou ultrapassar o array.
2. Aplicar **Binary Search** no intervalo identificado.

### **Complexidade:**

* **Tempo:** O(log i) onde i é o índice onde target se encontra.
* **Espaço:** O(1)

### **Exemplo básico (**c#**):**

int ExponentialSearch(int[] arr, int target) {

if (arr[0] == target)

return 0;

int i = 1;

while (i < arr.Length && arr[i] <= target)

i \*= 2;

return BinarySearch(arr, target, i / 2, Math.Min(i, arr.Length - 1));

}

int BinarySearch(int[] arr, int target, int left, int right) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) return mid;

else if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

## Problema com HashMap

### HashMap (dicionário) é útil para:

* Frequência de elementos.
* Verificação de duplicatas.
* Mapear relações chave-valor (ex: índice, posição).
* Resolver problemas em tempo linear com espaço adicional.

### Exemplo 1: Two Sum (Two Pointers + HashMap)

**Objetivo:** encontrar dois índices cujo valor soma seja igual ao target.

int[] TwoSum(int[] nums, int target) {

var map = new Dictionary<int, int>();

for (int i = 0; i < nums.Length; i++) {

int complement = target - nums[i];

if (map.ContainsKey(complement))

return new int[] { map[complement], i };

map[nums[i]] = i;

}

return new int[0];

}

### Exemplo 2: Frequência Máxima de Caracteres

int MaxCharFrequency(string s) {

var map = new Dictionary<char, int>();

foreach (char c in s) {

if (!map.ContainsKey(c))

map[c] = 0;

map[c]++;

}

return map.Values.Max();

}

### Resumo Tabela

| **Técnica** | **Tempo** | **Quando usar** |
| --- | --- | --- |
| **Binary Search** | O(log n) | Arrays ordenados |
| **Sliding Window** | O(n) | Substrings/subarrays com restrições |
| **Exponential Search** | O(log i) | Arrays muito grandes ou streams |
| **HashMap** | O(n) | Frequência, buscas rápidas, mapas reversos |

# Módulo 3: *Linked List*

## Implementação Linked Lists

### Estrutura de um nó (em pseudocódigo/C#)

class Node {

public int data;

public Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

### Estrutura da lista

class LinkedList {

public Node head;

public void AddFirst(int value) {

Node newNode = new Node(value);

newNode.next = head;

head = newNode;

}

public void AddLast(int value) {

Node newNode = new Node(value);

if (head == null) {

head = newNode;

return;

}

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

public void Print() {

Node current = head;

while (current != null) {

Console.Write(current.data + " -> ");

current = current.next;

}

Console.WriteLine("null");

}

}

## Inverter uma linked list

### Objetivo

Transformar a lista de forma que o último elemento se torne o primeiro

### Exemplo

Entrada: 1 -> 2 -> 3 -> 4 -> null  
Saída: 4 -> 3 -> 2 -> 1 -> null

### Abordagem Iterativa (eficiente em tempo e espaço)

public Node Reverse(Node head) {

Node prev = null;

Node current = head;

while (current != null) {

Node nextTemp = current.next;

current.next = prev;

prev = current;

current = nextTemp;

}

return prev; // novo head

}

### Complexidade

* Tempo: O(n)
* Espaço: O(1)

## Encontrar o meio de uma linked list

### Objetivo

Encontrar o nó central de uma lista encadeada (ou o segundo, em listas com número par de elementos)

### Abordagem com dois ponteiros (slow e fast)

public Node FindMiddle(Node head) {

Node slow = head;

Node fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow; // middle node

}

### Como funciona

* O ponteiro slow anda 1 passo de cada vez
* O ponteiro fast anda 2 passos de cada vez
* Quando fast atinge o fim, slow está no meio

### Complexidade

* Tempo: O(n)
* Espaço: O(1)

## Encontrar ciclos na linked list

#### Problema

Verificar se existe um ciclo (loop) na lista, ou seja, um nó aponta para algum nó anterior

#### Abordagem de Floyd (Tartaruga e Lebre)

public bool HasCycle(Node head) {

Node slow = head;

Node fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

return true; // ciclo detectado

}

}

return false;

}

#### Como funciona

* Se houver um ciclo, os ponteiros slow e fast eventualmente se encontrarão
* Caso contrário, o fast chegará ao fim da lista (null)

#### Complexidade

* Tempo: O(n)
* Espaço: O(1)

### Dicas Gerais para Entrevistas e Exercícios

* Sempre valide se a lista está vazia (head == null) antes de operações
* Em perguntas sobre reversão ou ciclos, priorize abordagens com espaço constante
* Para problemas complexos, desenhar os nós e ponteiros pode ajudar na visualização

### Desafios de Prática

1. Reverter parte de uma lista entre duas posições dadas
2. Verificar se uma lista é palíndroma
3. Remover o n-ésimo elemento do final
4. Mesclar duas listas ordenadas
5. Detectar o nó onde o ciclo começa (usar a extensão de Floyd)

# Módulo 4: *Sorting*

## BubbleSort

[Repositório do Github]

## QuickSort

[Repositório do Github]

## MergeSort

[Repositório do Github]

# Módulo 5: Binários

## Left e right shift

### Left Shift (<<)

#### Conceito:

* Desloca os bits para a **esquerda**.
* Multiplica o número por 2^n, onde n é o número de deslocamentos.

#### Exemplo:

int valor = 5; // 0000 0101

int resultado = valor << 1; // 0000 1010 = 10

### Right Shift (>>)

#### Conceito:

* Desloca os bits para a **direita**.
* Divide o número por 2^n.

#### Exemplo:

int valor = 10; // 0000 1010

int resultado = valor >> 1; // 0000 0101 = 5

## AND, OR, NOT, XOR

#### AND (&)

Compara bit a bit. Retorna 1 somente se os dois bits forem 1.

int a = 5; // 0101

int b = 3; // 0011

int c = a & b; // 0001 = 1

#### OR (|)

Compara bit a bit. Retorna 1 se pelo menos um bit for 1.

int c = a | b; // 0111 = 7

#### XOR (^)

Retorna 1 se os bits forem diferentes.

int c = a ^ b; // 0110 = 6

#### NOT (~)

* Inverte todos os bits (inclusive o bit de sinal).
* Em C#, ~5 resulta em -6 porque inverte os bits e retorna o complemento de dois.

int a = 5; // 0000 0101

int c = ~a; // 1111 1010 = -6

# Módulo 6: *Binary Trees*

## Definição

Uma árvore binária é uma estrutura de dados hierárquica composta por nós, na qual cada nó possui, no máximo, dois filhos:

* Filho à esquerda (left child)
* Filho à direita (right child)

Cada nó pode conter:

* Chave ou valor (data)
* Ponteiro para o filho esquerdo
* Ponteiro para o filho direito

A árvore binária é um caso particular da árvore geral, restrita a dois filhos por nó.

## Propriedades Fundamentais

1. **Altura da árvore (height)**: número de arestas no caminho mais longo da raiz até uma folha.
2. **Profundidade de um nó (depth)**: número de arestas do nó até a raiz.
3. **Nível (level)**: conjunto de nós que se encontram à mesma profundidade.
4. **Número máximo de nós por nível**: em um nível i, pode haver até 2^i nós.
5. **Número máximo de nós em uma árvore binária de altura** h: 2^(h+1) - 1.
6. **Densidade**: uma árvore é considerada "cheia" quando todos os nós, exceto as folhas, possuem exatamente dois filhos.

## Classificações de Árvores Binárias

* **Árvore Binária Completa (Complete Binary Tree)**: todos os níveis, exceto possivelmente o último, estão completamente preenchidos, e os nós do último nível estão o mais à esquerda possível.
* **Árvore Binária Cheia (Full Binary Tree)**: cada nó interno possui exatamente dois filhos.
* **Árvore Binária Perfeita (Perfect Binary Tree)**: árvore cheia em que todas as folhas estão no mesmo nível.
* **Árvore Binária Degenerada (ou patológica)**: cada nó possui apenas um filho, comportando-se como uma lista encadeada.

## 

## 

## Percursos (Traversals)

O percurso de uma árvore é o processo de visitar todos os nós em uma ordem específica. Os principais são:

### Em Profundidade (Depth-First Traversal)

1. **Pré-ordem (Preorder)**: raiz → esquerda → direita  
   * Utilizado para copiar ou serializar árvores.
2. **Em-ordem (Inorder)**: esquerda → raiz → direita  
   * Em árvores binárias de busca (BST), produz elementos em ordem crescente.
3. **Pós-ordem (Postorder)**: esquerda → direita → raiz  
   * Útil para liberar memória ou avaliar expressões.

### Em Largura (Breadth-First Traversal)

* Nível a nível (Level Order): percorre os nós da raiz até as folhas, nível por nível, normalmente implementado com fila (queue).

## Árvores Binárias de Busca (Binary Search Trees – BST)

Uma BST é uma árvore binária em que:

* O filho esquerdo contém apenas valores menores que a chave do nó.
* O filho direito contém apenas valores maiores que a chave do nó.

Propriedades:

* Busca, inserção e remoção possuem, em média, complexidade O(log n), desde que a árvore esteja balanceada.
* No pior caso (árvore degenerada), a complexidade se degrada para O(n).

## 

## Operações Fundamentais

1. Inserção: segue a propriedade da BST, adicionando o nó como folha.
2. Busca: percorre comparando chaves até encontrar ou chegar a um nó nulo.
3. Remoção: três casos devem ser tratados:  
   * Nó é folha (simples remoção).
   * Nó possui um filho (substitui-se pelo filho).
   * Nó possui dois filhos (substitui-se pelo sucessor ou predecessor em ordem).

## Complexidade

* Busca, Inserção, Remoção:  
  + - Melhor caso (árvore balanceada): O(log n)
    - Pior caso (árvore degenerada): O(n)
* Percursos: todos possuem complexidade O(n), pois cada nó é visitado exatamente uma vez.

## Árvores Binárias Balanceadas

Para evitar degradação de desempenho, foram desenvolvidas variações balanceadas:

* AVL Tree: mantém fator de balanceamento entre -1 e 1 para cada nó.
* Red-Black Tree: garante balanceamento aproximado por meio de regras de coloração.
* Splay Tree: ajusta dinamicamente a árvore a partir do acesso mais frequente.
* Treap e B-Trees (generalizadas): aplicam princípios probabilísticos ou voltados a armazenamento em disco.

## Aplicações Práticas

* Representação de expressões matemáticas (árvores de expressão).
* Estruturas de indexação (ex.: compressores, compiladores).
* Implementação de dicionários, mapas e conjuntos (via BST balanceadas).
* Suporte a algoritmos de busca em grafos e IA (ex.: árvores de decisão).
* Sistemas de arquivos e bancos de dados (árvores B/B+ derivadas).

## Considerações Finais

O estudo de árvores binárias é essencial na formação em Estruturas de Dados e Algoritmos, pois elas oferecem uma base sólida para compreender não apenas os fundamentos da hierarquia e organização de dados, mas também para o desenvolvimento de estruturas mais sofisticadas, como heaps, árvores balanceadas e *tries*.

O domínio das operações, propriedades e aplicações permite analisar a eficiência algorítmica, otimizar recursos e construir soluções escaláveis em sistemas complexos.

## Implementação (C#)

using System;

using System.Collections.Generic;

namespace BinaryTreeDemo

{

// Classe que representa um nó da árvore

public class Node

{

public int Value;

public Node Left;

public Node Right;

public Node(int value)

{

Value = value;

Left = null;

Right = null;

}

}

// Classe da Árvore Binária de Busca (BST)

public class BinarySearchTree

{

public Node Root;

public BinarySearchTree()

{

Root = null;

}

// Inserção recursiva

public void Insert(int value)

{

Root = InsertRec(Root, value);

}

private Node InsertRec(Node root, int value)

{

if (root == null)

return new Node(value);

if (value < root.Value)

root.Left = InsertRec(root.Left, value);

else if (value > root.Value)

root.Right = InsertRec(root.Right, value);

return root;

}

// Busca recursiva

public bool Search(int value)

{

return SearchRec(Root, value);

}

private bool SearchRec(Node root, int value)

{

if (root == null) return false;

if (root.Value == value) return true;

if (value < root.Value)

return SearchRec(root.Left, value);

else

return SearchRec(root.Right, value);

}

// Remoção

public void Remove(int value)

{

Root = RemoveRec(Root, value);

}

private Node RemoveRec(Node root, int value)

{

if (root == null) return null;

if (value < root.Value)

root.Left = RemoveRec(root.Left, value);

else if (value > root.Value)

root.Right = RemoveRec(root.Right, value);

else

{

// Caso 1: nó folha

if (root.Left == null && root.Right == null)

return null;

// Caso 2: nó com apenas um filho

if (root.Left == null)

return root.Right;

else if (root.Right == null)

return root.Left;

// Caso 3: nó com dois filhos

root.Value = MinValue(root.Right);

root.Right = RemoveRec(root.Right, root.Value);

}

return root;

}

private int MinValue(Node root)

{

int min = root.Value;

while (root.Left != null)

{

min = root.Left.Value;

root = root.Left;

}

return min;

}

// Percursos em ordem

public void InOrder()

{

InOrderRec(Root);

Console.WriteLine();

}

private void InOrderRec(Node root)

{

if (root != null)

{

InOrderRec(root.Left);

Console.Write(root.Value + " ");

InOrderRec(root.Right);

}

}

public void PreOrder()

{

PreOrderRec(Root);

Console.WriteLine();

}

private void PreOrderRec(Node root)

{

if (root != null)

{

Console.Write(root.Value + " ");

PreOrderRec(root.Left);

PreOrderRec(root.Right);

}

}

public void PostOrder()

{

PostOrderRec(Root);

Console.WriteLine();

}

private void PostOrderRec(Node root)

{

if (root != null)

{

PostOrderRec(root.Left);

PostOrderRec(root.Right);

Console.Write(root.Value + " ");

}

}

// Percurso em largura (Level Order)

public void LevelOrder()

{

if (Root == null) return;

Queue<Node> queue = new Queue<Node>();

queue.Enqueue(Root);

while (queue.Count > 0)

{

Node current = queue.Dequeue();

Console.Write(current.Value + " ");

if (current.Left != null)

queue.Enqueue(current.Left);

if (current.Right != null)

queue.Enqueue(current.Right);

}

Console.WriteLine();

}

}

class Program

{

static void Main()

{

BinarySearchTree bst = new BinarySearchTree();

// Inserindo nós

bst.Insert(50);

bst.Insert(30);

bst.Insert(70);

bst.Insert(20);

bst.Insert(40);

bst.Insert(60);

bst.Insert(80);

Console.WriteLine("In-Order (crescente):");

bst.InOrder();

Console.WriteLine("Pré-Ordem:");

bst.PreOrder();

Console.WriteLine("Pós-Ordem:");

bst.PostOrder();

Console.WriteLine("Level Order:");

bst.LevelOrder();

Console.WriteLine("Busca pelo valor 40: " + bst.Search(40));

Console.WriteLine("Busca pelo valor 100: " + bst.Search(100));

Console.WriteLine("Removendo 20 (folha):");

bst.Remove(20);

bst.InOrder();

Console.WriteLine("Removendo 30 (um filho):");

bst.Remove(30);

bst.InOrder();

Console.WriteLine("Removendo 50 (dois filhos):");

bst.Remove(50);

bst.InOrder();

}

}

}

## 

# Módulo 7: Grafos

## Clonar um Grafo

Clonar um grafo consiste em criar uma cópia exata de um grafo dado, incluindo todos os vértices e arestas, de modo que modificações na cópia não afetem o grafo original. Isso é particularmente útil em algoritmos que requerem manipulação temporária de grafos sem alterar o estado original.

### Algoritmo de Clonagem

Uma abordagem comum é utilizar uma busca em profundidade (DFS) ou em largura (BFS) para percorrer o grafo original e construir a cópia. Durante o processo, é essencial manter um mapeamento entre os vértices originais e seus correspondentes na cópia para garantir a integridade das arestas.

Exemplo em Python

class Node:

def \_\_init\_\_(self, val=0, neighbors=None):

self.val = val

self.neighbors = neighbors if neighbors is not None else []

def cloneGraph(node: 'Node') -> 'Node':

if not node:

return None

mapping = {}

def dfs(node):

if node in mapping:

return mapping[node]

copy = Node(node.val)

mapping[node] = copy

for neighbor in node.neighbors:

copy.neighbors.append(dfs(neighbor))

return copy

return dfs(node)

## Algoritmo de Dijkstra

O algoritmo de Dijkstra é um método clássico para encontrar o caminho mais curto de um vértice origem para todos os outros vértices em um grafo ponderado com arestas de peso não negativo.

### Funcionamento

Inicializa-se a distância de todos os vértices como infinita, exceto a origem, que recebe distância 0.

Utiliza-se uma fila de prioridade para explorar os vértices com a menor distância conhecida.

Para cada vizinho de um vértice explorado, calcula-se a distância potencial e atualiza-se se for menor que a distância conhecida.

### Complexidade

A complexidade do algoritmo é

𝑂((𝑉+𝐸) log 𝑉), onde V é o número de vértices e 𝐸 é o número de arestas, quando implementado com uma fila de prioridade binária.

## Implementação do algoritmo de Dijkstra

A seguir, uma implementação do algoritmo de Dijkstra em Python utilizando um heap binário para a fila de prioridade:

import heapq

def dijkstra(graph, start):

min\_heap = [(0, start)]

distances = {start: 0}

while min\_heap:

current\_dist, current\_vertex = heapq.heappop(min\_heap)

if current\_dist > distances.get(current\_vertex, float('inf')):

continue

for neighbor, weight in graph[current\_vertex]:

distance = current\_dist + weight

if distance < distances.get(neighbor, float('inf')):

distances[neighbor] = distance

heapq.heappush(min\_heap, (distance, neighbor))

return distances

## 

# Módulo 8: *Stack*

## Primeira implementação de stack

### Implementação em C# usando Lista

using System;

using System.Collections.Generic;

public class Stack<T>

{

private List<T> elements = new List<T>();

public void Push(T item)

{

elements.Add(item);

}

public T Pop()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

T item = elements[^1]; // Último elemento

elements.RemoveAt(elements.Count - 1);

return item;

}

public T Peek()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

return elements[^1];

}

public bool IsEmpty()

{

return elements.Count == 0;

}

public int Size()

{

return elements.Count;

}

}

## Implementando stack com linked list

using System;

public class Node<T>

{

public T Data { get; set; }

public Node<T> Next { get; set; }

public Node(T data)

{

Data = data;

Next = null;

}

}

public class StackLinkedList<T>

{

private Node<T> top; // topo da pilha

private int count; // tamanho da pilha

public StackLinkedList()

{

top = null;

count = 0;

}

// Adiciona um elemento ao topo da pilha

public void Push(T data)

{

Node<T> newNode = new Node<T>(data);

newNode.Next = top;

top = newNode;

count++;

}

// Remove e retorna o elemento do topo da pilha

public T Pop()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

T data = top.Data;

top = top.Next;

count--;

return data;

}

// Retorna o elemento do topo sem remover

public T Peek()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

return top.Data;

}

// Verifica se a pilha está vazia

public bool IsEmpty()

{

return top == null;

}

// Retorna o tamanho da pilha

public int Size()

{

return count;

}

}

## Implementando min stack

### Conceito

Uma Min Stack é uma pilha que, além das operações básicas (Push, Pop, Peek), permite obter o menor elemento da pilha em tempo constante O(1).

Para isso, cada nó da pilha armazena:

1. O valor do elemento (Data)
2. O menor valor até aquele nó (MinSoFar)

### Implementação em C#

using System;

public class Node<T> where T : IComparable<T>

{

public T Data { get; set; }

public T MinSoFar { get; set; } // menor valor até este nó

public Node<T> Next { get; set; }

public Node(T data, T minSoFar)

{

Data = data;

MinSoFar = minSoFar;

Next = null;

}

}

public class MinStack<T> where T : IComparable<T>

{

private Node<T> top;

private int count;

public MinStack()

{

top = null;

count = 0;

}

// Adiciona um elemento no topo

public void Push(T data)

{

T min = IsEmpty() ? data : (data.CompareTo(top.MinSoFar) < 0 ? data : top.MinSoFar);

Node<T> newNode = new Node<T>(data, min);

newNode.Next = top;

top = newNode;

count++;

}

// Remove e retorna o topo

public T Pop()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

T data = top.Data;

top = top.Next;

count--;

return data;

}

// Retorna o topo sem remover

public T Peek()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

return top.Data;

}

// Retorna o menor elemento da pilha

public T GetMin()

{

if (IsEmpty())

throw new InvalidOperationException("Stack is empty.");

return top.MinSoFar;

}

public bool IsEmpty()

{

return top == null;

}

public int Size()

{

return count;

}

}

### Exemplo de Uso

class Program

{

static void Main()

{

MinStack<int> stack = new MinStack<int>();

stack.Push(5);

stack.Push(2);

stack.Push(8);

stack.Push(1);

Console.WriteLine("Topo: " + stack.Peek()); // 1

Console.WriteLine("Min: " + stack.GetMin()); // 1

stack.Pop();

Console.WriteLine("Min após pop: " + stack.GetMin()); // 2

stack.Pop();

Console.WriteLine("Min após outro pop: " + stack.GetMin()); // 2

}

}

### Características

* Push, Pop, Peek, GetMin: O(1)
* Crescimento dinâmico sem limite de tamanho
* Cada nó armazena o mínimo até aquele ponto, evitando uso de pilhas auxiliares

# Módulo 9: *Heap*

## Implementação de Heap

### Conceito

Um Heap é uma **árvore binária completa** que satisfaz a propriedade de heap:

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

**Min-Heap:** cada nó é menor ou igual a seus filhos → o menor elemento está na raiz.

Observação: “Árvore binária completa” significa que todos os níveis, exceto possivelmente o último, estão completamente preenchidos, e os nós do último nível estão o mais à esquerda possível.

### Representação

Array: um heap completo pode ser armazenado em array sem ponteiros:

Para um nó em índice i:

**Esquerda:** 2\*i + 1

**Direita:** 2\*i + 2

**Pai:** (i-1)/2

### Operações Principais

### Exemplo de Max-Heap em C# usando Array

using System;

using System.Collections.Generic;

public class MaxHeap

{

private List<int> heap = new List<int>();

public int Size() => heap.Count;

public bool IsEmpty() => heap.Count == 0;

public void Insert(int val)

{

heap.Add(val);

HeapifyUp(heap.Count - 1);

}

public int Peek()

{

if (IsEmpty())

throw new InvalidOperationException("Heap is empty.");

return heap[0];

}

public int ExtractMax()

{

if (IsEmpty())

throw new InvalidOperationException("Heap is empty.");

int max = heap[0];

heap[0] = heap[heap.Count - 1];

heap.RemoveAt(heap.Count - 1);

HeapifyDown(0);

return max;

}

private void HeapifyUp(int index)

{

while (index > 0)

{

int parent = (index - 1) / 2;

if (heap[index] <= heap[parent])

break;

int temp = heap[index];

heap[index] = heap[parent];

heap[parent] = temp;

index = parent;

}

}

private void HeapifyDown(int index)

{

int lastIndex = heap.Count - 1;

while (true)

{

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int largest = index;

if (left <= lastIndex && heap[left] > heap[largest])

largest = left;

if (right <= lastIndex && heap[right] > heap[largest])

largest = right;

if (largest == index)

break;

int temp = heap[index];

heap[index] = heap[largest];

heap[largest] = temp;

index = largest;

}

}

}

### Exemplo de Uso

class Program

{

static void Main()

{

MaxHeap heap = new MaxHeap();

heap.Insert(10);

heap.Insert(20);

heap.Insert(5);

Console.WriteLine("Maior elemento: " + heap.Peek()); // 20

Console.WriteLine("Removendo: " + heap.ExtractMax()); // 20

Console.WriteLine("Novo topo: " + heap.Peek()); // 10

}

}

### Aplicações Comuns de Heap

* Filas de prioridade
* Algoritmo de Dijkstra (min-heap para selecionar o menor caminho)
* Heap Sort (ordenação baseada em heap)
* Problemas de K maiores/menores elementos

# Conclusão

## Livros Avançados:

# 

- "Introduction to Algorithms" de Cormen, Leiserson, Rivest e Stein (CLRS): Um recurso abrangente que cobre uma ampla gama de algoritmos em profundidade.

- "Algorithm Design" de Jon Kleinberg e Éva Tardos: Foca em técnicas de design de algoritmos e estratégias de resolução de problemas.

- "Algorithms" de Robert Sedgewick e Kevin Wayne: Oferece uma exploração detalhada de algoritmos com exemplos de código práticos.

# 

## Cursos e Aulas Online:

# 

- Coursera:

- Algorithms Specialization da Universidade de Stanford.

- Advanced Algorithms and Complexity da Universidade da Califórnia em San Diego.

- edX:

- Algorithms and Data Structures da Microsoft.

- Advanced Data Structures do MIT.

- MIT OpenCourseWare:

- Introduction to Algorithms (6.006).

- Advanced Data Structures (6.851).

System Design:

- Coursera:

- Software Design and Architecture Specialization - University of Alberta

- Outros:

- Grokking the Modern System Design Interview (educative)

- Grokking the System Design Interview (designgurus)

- ByteByteGo System Design Course (bytebytego)

# 