

ALGORÍTMIA

Os mais importantes algoritmos

1. A* Search algorithm → Busca de caminho (grafos)
2. Beam search → Busca
3. Binary search → Busca
4. Data compression → Codificação
5. Dijkstra algorithm → Caminhos
6. Dynamic programming
7. Hashing → Mapas
8. Heaps (heap sort) → Ordenação
9. Karatsuba multiplication
10. Maximum flow

3 Perguntas fundamentais:

- ↳ Funciona?
- ↳ É rápido?
- ↳ Dá para fazer melhor?

A* search algorithm

↳ algoritmo de busca de caminho

Desvantagens → complexidade do espaço pois examina todos os nós gerados na memória.

↳ Isto é superado por algoritmos que podem pré-processar o gráfico para obter melhor desempenho.

A* search x Dijkstra

→ encontra o caminho mais curto para um objetivo específico

→ como todo o caminho do caminho mais curto é gerado, cada nó é um objetivo e não pode haver uma heurística direcionada a um objetivo específico.

Dijkstra

↳ fixa um único nó como o nó "fonte" e encontra caminhos mais curtos da fonte para todos os outros nós no grafo, produzindo um caminho mais curto árvore.

→ uso rotulos que são inteiros positivos ou ∞ reais, que são totalmente ordenados

→ Uso rótulos que são inteiros positivos ou números reais, que são totalmente ordenados.

Beam Search → algoritmo ganancioso.

↳ algoritmo de busca heurística que explora um grafo expandindo um nó mais promissor em um conjunto limitado

→ Beam Search é uma otimização do pesquisa do best-first search que reduz seus requisitos de memória

Best-first

↳ é uma busca de grafos que encontra todos os soluções parciais (estados) de acordo com alguma heurística. Nos no Beam search, operam um número predeterminado de melhores soluções parciais só mantidas como candidatos.

DATA Compression

→ processo de codificação de informações usando menos bits do que a representação original

→ Existe compressão com ou sem perdas

Compressão

Com perdas → reduz os bits removendo informações desnecessárias ou menos importantes

Sem perdas → Reduz os bits identificando e eliminando a redundância estatística. Nenhuma informação é perdida.

Compressão → processo de redução do tamanho de um arquivo de dados.

Dispositivo que realiza a compressão de dados ↗

Codificador

↓ reverso

descodificador

Dynamic programming

- ↳ método de programação de computadores
- ↳ Refere-se à simplificação de um problema complicado, dividindo-o em subproblemas mais simples de maneira recursiva
 - ↳ Alguns problemas de decisão que podem ser separados

Subestrutura ótima

- ↳ Separar-se recursivamente
- ↳ Problema é dividido em subproblemas e em seguida, encontrando recursivamente os soluções ótimas para os subproblemas.

Dijkstra é DP? → Sim (pense pq)

- Métodos de DP
 - ↳ De cima para baixo com memoização
 - ↳ Tentar-se resolver o problema de cima para baixo.

↳ Armazenamos em cache o resultado sempre que se resolve um problema para que não seja resolvido repetidamente. Se for chamado várias vezes. Esse técnico de chamar os resultados de subproblemas já resolvidos é chamado de memoização.

- Debaixo para cima com tabulação
 - ↳ Evita a recursão, resolvemos de baixo para cima, logo resolvemos todos os subproblemas primeiro.

- Sempre começaremos com uma função recursiva de forma bruta, que é a melhor maneira de começar a resolver qualquer problema de DP. Temos solução recursiva? → aplicar os técnicas abordadas em cima.

Karatsuba

↳ algoritmo de multiplicação rápida. Rodar a multiplicação de dois números de m dígitos...

→ Assimptoticamente mais rápido que o algoritmo tradicional

Algoritmo ingênuo	→	tempo de execução
Karatsuba	→	$\Theta(m^2)$ $\Theta(m \log_2 3) \approx \Theta(m^{1.585})$

→ Objetivo: tornar números grandes em números menores para que quaisquer multiplicações que ocorram em números menores.

Hashing

↳ separam dados de tamanho arbitrário para valores de tamanho fixo. Os valores rotacionados por uma função de hash são chamados de Valores de hash

→ Os valores geralmente são usados para indexar uma tabela de tamanho fixo chamada de tabela hash.



São usados em aplicativos de armazenamento

→ recuperar de dados para acessar dados em um tempo pequeno e quase constante para recuperação.

→ Compartilhamento de pinceladas é intensivamente ruim com uma probabilidade muito pequena

→ O comportamento do caso médio pode ser quase ótimo.

Binary Search

↪ algoritmo de busca que encontra a posição de um valor alvo dentro de uma matriz ordenada

→ compara o valor de destino com o elemento do meio da matriz.

↓
Se não forem iguais

a metade em que o alvo não pode estar é eliminada.



busca continua no método restante



→ Repetir o processo até encontrar.

→ $O(\log n)$

→ É mais rápida que a pesquisa linear, exceto para pequenos arrays.

→ Existem estruturas de dados especializadas projetadas para pesquisa rápida, como tabelas de hash, que podem ser pesquisadas com mais eficiência do que a pesquisa binária



→ pode ser usada para resolver um grande conjunto de problemas como encontrar o próximo menor.

Existem diversos métodos de classificação de algoritmos

Precisamos de saber ②



- ①. Tipo de implementação
- ②. Paradigma da abordagem (design)

① ————— || ————— |) ————— || —————

Recursivos / Iterativos

→ As implementações Recursivas têm uma versão iterativa (mais ou menos complexa) equivalente, e vice-versa.

Logicos

→ Paradigma deductivo

O programa declara os objetivos da computação, mas não detalha o algoritmo pelo qual esses objetivos deverão ser alcançados.

→ Os programas são expressos através de lógica simbólica

Sequenciais / Paralelos

→ Usam um só processador de forma sequencial / usam mais de um computador ou um computador com múltiplos processadores

IMPORTANTE → Os algoritmos iterativos são, em geral, implementáveis de modo paralelo.

Se não tiverem uma solução paralela só desigualdos problemas sequenciais imatos

→ Uma solução eficaz posso pelo utilização de computação paralela em que os algoritmos podem especificar múltiplos operações num só passo.

- Desempenho do algoritmo = Pior caso de tempo de execução da sequência mais rápida contida
- N.º de processadores executados Para um problema particular
- Custo total Pior caso de tempo de execução do algoritmo paralelo.

Determinísticos / Não determinísticos

→ O resultado de uma qualquer operação é definido de maneira única só desigualdos Determinísticos.

Não determinismo = paralelismo sem restrições

Exactos / Aproximados

Um algoritmo exato oferece apenas soluções ótimas.

Se o problema tiver uma solução, o algoritmo encontrará a solução óptima.

→ Como existe problemas tão difíceis, tentar uma solução exata é ambicioso demais para o poder computacional atualmente disponível.

Portanto, usamos algoritmos aproximados, que encontram maneiras inteligentes ("mais baratas") de chegar ao mais próximo possível da solução ideal.

② ————— || ————— || ————— || —————

Fórmula Bruta (exhaustive search)

Consiste em enumerar todos os possíveis candidatos de solução e checar cada candidato para ver se ele o satisfaz o enunciado do problema.

→ Simples implementação

→ Se houver solução ele encontrará

→ Custará proporcionalmente ao número de candidatos à solução.

→ Use-se quando o tamanho do problema é limitado e também usado quando a simplicidade de implementação é mais importante que a velocidade.

Divisão - e - Conquista

Divide sucessivamente de um problema em um ou mais problemas menores (habitualmente de modo recursivo).

- quicksort
- mergesort

Redução - e - Conquista

→ Resolve um sub-problema (por Reduzir / diminuir) sucessiva da dimensão da entrada.

→ modo recursivo ou iterativo

→ Por exemplo 2, ou seja, diminuir a dimensão pelo método.
↳ constante

1. métodos de ordenação por inserção
2. pesquisa binária (decimento com fator constante, 2)
3. algoritmos de inserção, remoção e pesquisa em árvores.

mais complexo o primeiro

Divisão - e - conquista

VS Redução - e - conquista

→ mergesort → divide a lista em duas listas separadas para ordenar recursivamente cada uma das sub-listas

→ pesquisa binária → utiliza sequencialmente metade da lista anterior para efetuar a pesquisa.

Técnicas de programação - e - conquista

- Modifica o problema (e pode diminuir a dimensão do enunciado), para obter um problema de mais fácil resolução (e de menor dimensão.)
- modo recursivo e iterativo.

1. heapsort

2. Rod-Balok

Programação dinâmica

- Resuelve problemas compostos por sub-problemas duplicados (normalmente em resultado da Recursividade.)
- O cálculo destes sub-problemas é feito uma só vez, sendo "memorizados" para posterior utilização.

1. algoritmos memorizicos repetitivos
2. algoritmo de Floyd em grafos.

Programação Dinâmica vs Divisão-e-Conquista

- ↳ os sub-problemas são (mais ou menos) independentes enquanto que na programação dinâmica eles são compostos.

Pragmatismo: Dinâmica vs Recursividade

→ diferença está no caching dos valores já calculados (ou memoization) dos chamados recursivos.

→ Uma função recursiva, que não utiliza memoization, vai calcular diversas vezes o mesmo problema, sendo assim ineficaz

Algoritmos Gulosos (greedy)

→ Decompondo o problema num conjunto de etapas e tenta optimizar a solução partindo do pressuposto que a escolha sucessiva das melhores soluções em cada etapa conduz à melhor solução do problema.

→ Algoritmos MST de Prim e Kruskal

→ algoritmo de caminho mais curto de Dijkstra.

Algoritmos Gulosos vs Programação dinâmica

↳ é semelhante a um algoritmo de programação dinâmica mas, em vez de ser necessário o prévio conhecimento das soluções para os sub-problemas, é feita uma escolha de maior lucro em cada altura, baseada na melhor decisão (e não todos) da etapa anterior.

Algoritmo Gúloso

↳ Cálculo do troco de uma compra

↳ Começar com moeda

Em cada altura, e não ultrapassando o montante necessário:

Aumentar a moeda maior às moedas já escondidas.

Programação Linear

↳ a resolução do problema envolve maximizar (ou minimizar) dos entradas de um conjunto de desigualdades lineares.

↳ A programação linear é importante área da optimização por vários razões.

Pesquisa e Enumeração

1. Branch and Bound

→ É organizado através de uma árvore de sub-problemas (candidatos) que se vão constituindo à medida que o algoritmo avança.

Problema inicial → problema raiz.

→ O algoritmo vai explorando os ramos (branches) da árvore, que contém os candidatos e cuja solução resultante é antes verificada em relação aos limites superior e inferior estimados da solução ótima.

→ Os candidatos são descartados se não produzirem uma solução melhor do que encontrada até aí.

2. Backtracking (Recursividade com Retorno)

→ A pesquisa exaustiva verifica todos as soluções possíveis do problema.

→ A recursividade com retorno elimina recursivamente soluções possíveis (em análise) que não respeitem os critérios do problema, otimizando a pesquisa.

→ Os candidatos são descartados se não produzirem uma solução melhor do que a encontrada aí.

Probabilísticos e Heurísticos

→ Para alguns problemas (computacionais) pode ser difícil a formulação de uma solução com boa eficiência (tempo de execução) ou essa solução pode apresentar um grande crescimento do tempo de execução com o número de entradas.

1 - Algoritmos Probabilísticos

↳ faz pelo menos uma escolha aleatória. Esta abordagem é útil na obtenção de soluções aproximadas, em casos em que a determinação de soluções exatas seja impraticável

2 - Algoritmos Heurísticos

↳ objetivo é encontrar uma solução aproximada de forma fácil e rápida (para problemas em que o tempo ou os recursos são limitados).

Buscados nos Dados

→ Nós tem desempenho genérico
→ Ajusta os parâmetros desses algoritmos usando um processo / conjunto de treino de amostras

Exemplo : clustering

ALGORITMOS DE ORDENAMENTO

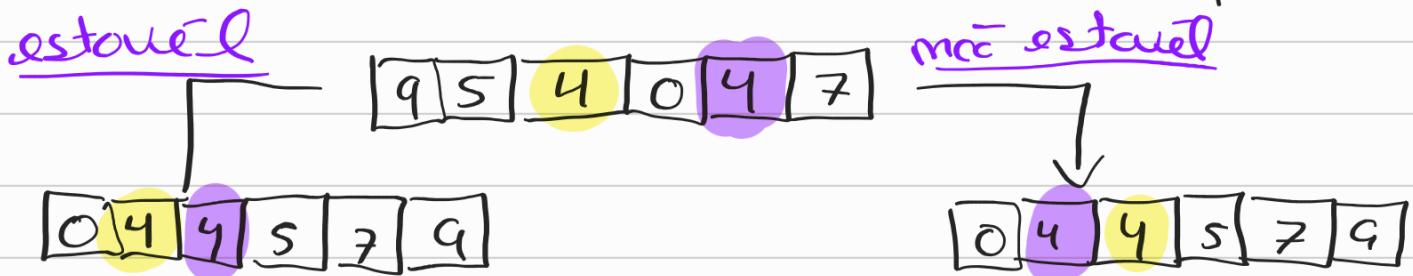
→ é um algoritmo que organiza de uma lista (conjunto sequencial de dados) com uma determinada ordem.

Ordenamento interno → em que tudo se passe na memória principal

Ordenamento externo → quando o número e dimensão dos objetos a ordenar é tal que inviabiliza o uso da memória principal

Métodos não estáveis → a ordem relativa dos elementos com chaves iguais é alterada durante o processo de ordenamento

Métodos estáveis → a ordem relativa de elementos com chaves iguais mantém-se inalterada durante o processo



- Ordenamento por inserção (insertion sort)
- Ordenamento por seleção (selection sort)
- Ordenamento por flutuação (bubble sort)

INSERT SORT

5	3	1	8	7	2
---	---	---	---	---	---

Em cada passagem, o primeiro elemento do ponto não ordenada da lista "procura" a sua posição na parte ordenada (à sua esquerda) da lista, movendo-se sucessivamente os elementos de valor superior uma posição para a direita.

3	5	1	8	7	2
---	---	---	---	---	---

1	3	5	8	7	2
---	---	---	---	---	---

- O número de passagens é fixo ($m-1$)
- O número de inversões em cada passagem é variável.

Tempo de execução depende: → tamanho da entrada

→ do tipo de ordenação dos valores da entrada.

Realizam o desempate

Comparações e trocas

Caso-pior

- Quando este ordenado de forma decrescente
- $O(m^2)$

Comparações: $\approx \frac{1}{2}m^2$
Trocas: $\approx \frac{1}{2}m^2$

(Oso - melhor)

- Quando já está ordenado
- $O(m)$

Comparações : $m - 1$
Índices : 0

(Oso médio)

$$\rightarrow O(m^2)$$

Características : → Fácil implementação

→ Eficiente em conjuntos de dados pequenos

→ Mais eficiente na prática do que os outros métodos
básicos : $O(m^2)$ - tais como ordenação por seleção ou bubble sort

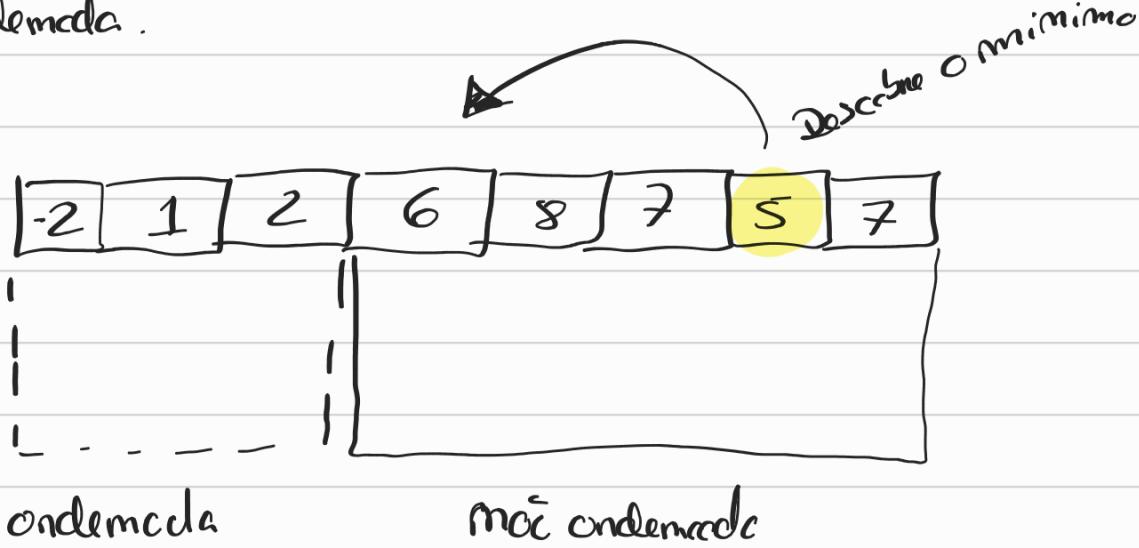
→ Estável.

Selectiom Sort

→ Encontra o valor mínimo da lista

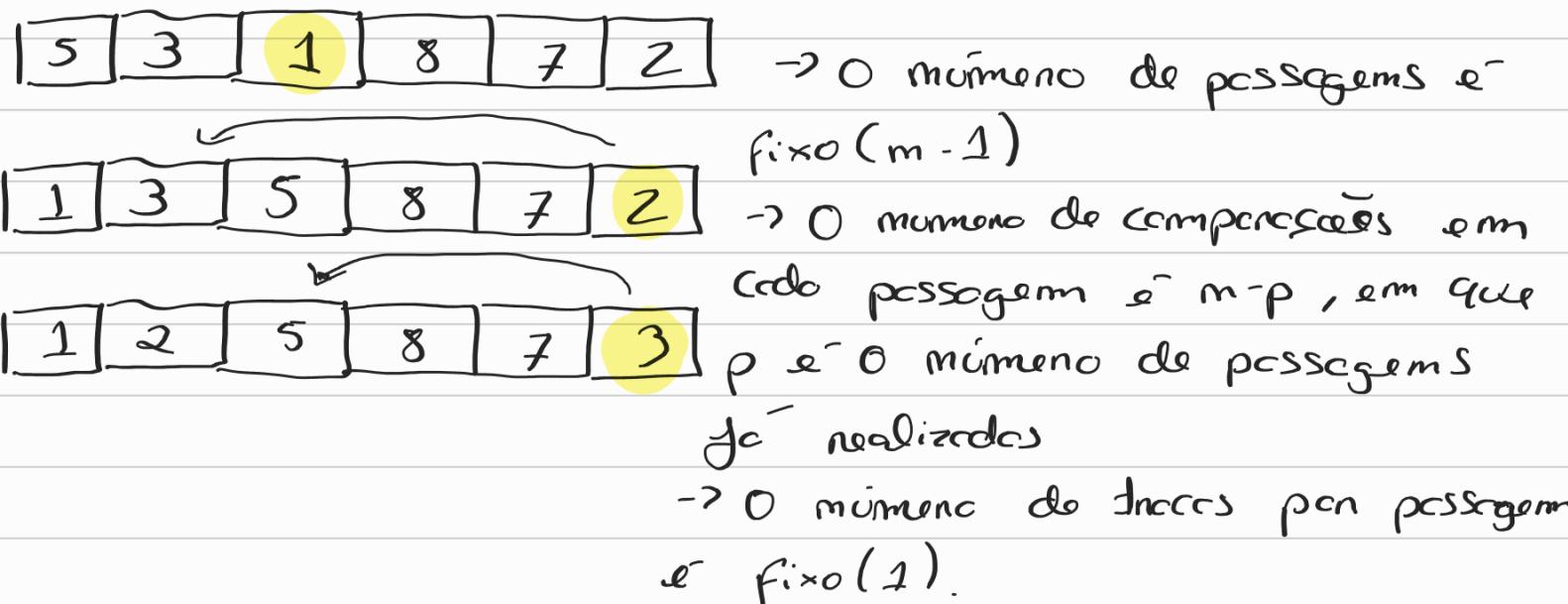
→ Inseriu este elemento com o primeiro elemento da lista

dividindo assim a lista na sua parte ordenada e não ordenada.



→ Determinam o mínimo em cada passagem

implica uma série de comparações entre elementos



Qualquer CSC : $O(m^2)$

Complexidade : $\approx \frac{1}{2} m^2$

trocas : $m - 1$

- Características :
- fácil implementação
 - Complexidade invencível : $O(m^2)$
 - Não é geralmente eficiente, mas pode ser implementado como tal
 - Implementação (necessita de um espaço extra de memória constante) : $O(1)$

Variante Bidimensional \rightarrow cocktail sort



encontra o máximo e o mínimo de cada Passagem.

Reduz o número de passagens por um fator de 2, mas não diminui o número de comparações ou trocas..

Variante estavel : stable sort



Insere o mínimo na primeira posição
(em vez de realizar a troca)

Bubble Sort

→ Utiliza como princípio básico a comparação entre pares dos elementos da lista, trocando-os caso a sua ordem esteja errada.

→ No fim da primeira passagem garante-se que o maior elemento "subiu" ou "flutuou" até ao fim da lista.

6	2	5	1	8	7	-2	7
2	6	5	1	8	7	-2	7
2	5	6	1	8	7	-2	7
2	5	1	6	8	7	-2	7
2	5	1	6	8	7	-2	7
2	5	1	6	7	8	2	7
2	5	1	6	7	-2	8	7

2	5	1	6	7	-2	8
---	---	---	---	---	----	---



ordenado

→ O processo repete-se até não ocorrer mais trocas em que a lista está ordenada.

→ O número de passagens é $\leq n - 1$

→ O número de comparações em cada passagem é $n - p$, em que p é o número de passagens já realizadas.

→ O número de trocas por passagem depende da ordenação relativa prévia.

Qualquer caso ou caso-pior

$O(m^2)$ excepto quando a lista está ordenada ou quase ordenada.

Comparações: $\approx \frac{1}{2}m^2$

trocos: $\approx \frac{1}{2}m^2$

Características:

→ Dos 3 métodos é o único que consegue determinar quando porre se o conjunto estiver ordenado.



→ A rapidez com que os elementos muito desordenados atingem o seu lugar final depende da sua valen e posição

↳ Elementos grandes no inicio rapidamente "subem" até ao seu lugar no final do conjunto ("rabbits")

↳ Elementos pequenos no fim só avançam imanicamente lentos a atingir a sua posição ("turtles")

→ Fácil implementação

→ Complexidade $O(m^2)$, menos em listas (quase) ordenadas (cuja complexidade é melhor)

→ $O(m)$ no melhor caso

→ Estável

→ In place

VARIANTE bidimensional → cocktail ou shaker sort

• Faz alternadamente uma passagem da esquerda para a direita (fazendo subir os elementos maiores) e uma passagem da direita para a esquerda (fazendo descer os elementos menores)

→ Mais difícil que o original, resolu o problema dos turtlles, mas mantém complexidade $O(m^2)$ em geral

Desempenho dos diferentes algoritmos

Nome	Caso Médio	Caso Pior	Memória	Estável	Notas
Bubble sort	—	$O(n^2)$	$O(1)$	Sim	
Cocktail sort	—	$O(n^2)$	$O(1)$	Sim	
Comb sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Não	
Gnome sort	—	$O(n^2)$	$O(1)$	Sim	
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	Não	Pode ser implementado como estável
Insertion sort	$O(n + d)$	$O(n^2)$	$O(1)$	Sim	d é o número de inversões, que é $O(n^2)$
Shell sort		$O(n \log^2 n)$	$O(1)$	Não	
Binary tree sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sim	Quanto é utilizada uma árvore binária (auto)equilibrada
Library sort	$O(n \log n)$	$O(n^2)$	$O(n)$	Sim	
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sim	
In-place merge sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Não	
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Não	
Smoothsort	—	$O(n \log n)$	$O(1)$	Não	
Quicksort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Não	Variantes "ingênuas" utilizam espaço $O(n)$; Pode ter um pior caso $O(n \log n)$ se utilizada a mediana como pivot.
Introsort	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	Não	
Patience sorting	—	$O(n^2)$	$O(n)$	Não	
Strand sort	$O(n \log n)$	$O(n^2)$	$O(n)$	Sim	
Radix sort	non-comparative integer sorting algorithm				Ordena n inteiros na base k com, no máximo, d dígitos em $O(d(n + k))$.

algoritmos de ordenação

04 MERGE SORT 05 QUICK SORT 06 HEAP SORT 07 RADIX SORT

Nome	Caso Médio	Caso Pior	Memória	Estável	Notas
Bubble sort	—	$O(n^2)$	$O(1)$	Sim	
Cocktail sort	—	$O(n^2)$	$O(1)$	Sim	
Comb sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Não	
Gnome sort	—	$O(n^2)$	$O(1)$	Sim	
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	Não	Pode ser implementado como estável
Insertion sort	$O(n + d)$	$O(n^2)$	$O(1)$	Sim	d é o número de inversões, que é $O(n^2)$
shell sort		$O(n \log^2 n)$	$O(1)$	Não	
Binary tree sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sim	Quanto é utilizada uma árvore binária (auto)equilibrada
Smoothsort	$O(n \log n)$	$O(n^2)$	$O(1)$	Não	
Quicksort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Não	Variante "ingênuas" utilizam espaço $O(n)$ Pode ter um pior caso $O(n \log n)$ se utilizada a mediana como pivot.
Introsort	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	Não	
Patience sorting	—	$O(n^2)$	$O(n)$	Não	
Strand sort	$O(n \log n)$	$O(n^2)$	$O(n)$	Sim	
Radix sort	non-comparative integer sorting algorithm				Ordena n inteiros na base k com, no máximo, d dígitos em $O(d(n + k))$.

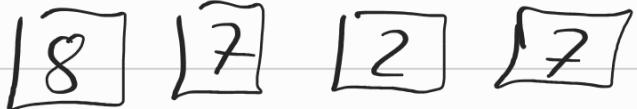
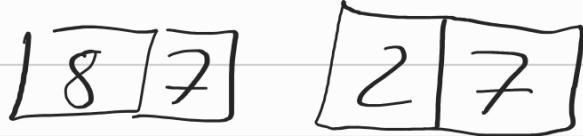
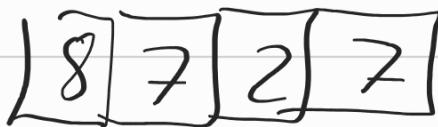
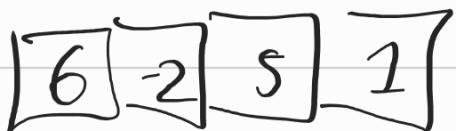
Merge Sort \rightarrow Divisão e Conquista

Divide-se a lista desordenada em dois sublistas de tamanho igual.

Baseia-se em dois princípios

→ Um lista pequena seu menor possos até estar ordenada do que uma lista maior.

→ A união de duas listas ordenadas menor lista resultante ordenada seu menor possos do que ordenar esta lista



→ Divide-se a lista em tamanho igual

→ Divide-se recursivamente, cada uma das sublistas

→ O processo deve cessar quando a sublistas resultante tiver dimensão igual a um.

[6] [2] [5] [1]

[8] [7] [2] [7]

[3] [6]

[1] [5]

[7] [8] [2] [7]

[2] [1] [5] [6]

[2] [7] [7] [8]

[2] [1] [2] [5] [6] [7] [7] [8]

- O merge-se sucessivamente cada dois sublistas numa lista comum
- Acaba quando houver lista comum.

Análise Geral

- Éficoz em geral
- Tempo esperado: $\Theta(m \lg m)$
- Não efetua a ordenação in place
- Estavel

Caso - melhor

Comparações

- (*) Ocorre quando o último elemento da sub-lista da esquerda é menor ou igual que o 1º elemento do sublista da direita.

(Comparações): $\approx m/2$

Caso - pior

Comparações

→ Ocurre quando o 1º elemento da sub-lista da esquerda é maior

Comparações → $\approx m$

Caso - geral

Comparações

→ Os elementos estão distribuídos aleatoriamente

Comparações → $\approx m$

Caso - melhor

Movimentos

→ Ocurre quando o último elemento da sublista da esquerda é menor ou igual que o 1º elemento da sublista da direita.

Comparações → $\approx m/2$

Movimentos → $\approx 2 \times m/2$

Caso pior

Movimentos

→ Ocurre quando o 1º elemento da sublista da esquerda é maior do que o ultimo elemento da sublista da direita

Comparações → $\approx m$

Movimentos → $\approx m/2 + m$

Caso - geral

↳ São distribuídos aleatoriamente

Componções : $\leq m$

Movimentos : $\leq m/2 + m //$

- O tempo de cada iteração vai ser a soma do tempo das duas chamadas recursivas mais o tempo da reunião
- O desempenho final vai depender do número de partções.

Quick Sort

↳ método de ordenação muito rápido e eficiente mas não é muito estável.

Pior caso

→ quando o elemento pivô divide a lista de forma desbalanceada, uma sub lista com tamanho 0 e a outra $m-1$ (m = tamanho da lista original)



isso pode ocorrer quando o elemento pivô é o maior ou menor elemento da lista

Quicksort é mais familiarizado com o Heapsort

mais rápido
pouco otimizado

mais lento
poco otimizado
Pior caso $O(m \log 2m)$

Quicksort

\times

Mergesort

pouco otimizado
mais rápido
 \checkmark
(constantes menores)

mais otimizado
mais lento

Pior caso \rightarrow Quicksort

\downarrow
 $O(m^2)$

Mergesort & Heapsort

$m \times \log(m)$

\checkmark
melhor

escolher qq um
 menores ou ← | PIVOT | → maiores
 igual

- Θ Como o pivot caso necessita de duas condições: os arrays já estão ordenados e mais escolhermos sempre o próximo pivot


 podemos diminuir a prob. da ocorrência do pivot caso.

Escolhendo um pivot melhor.

(• aleatoriamente pais tinha
 Probabilidade 1/m,

(• escolher o pivot como sendo
 a mediana do array

Radix Sort \rightarrow ordenação linear

(• ordena cada dígito do menor significativo ao mais significativo

- Θ $O(mk)$ $m =$ tamanho da matriz

- Θ é estável $k =$ mº de dígitos da maior num

porque se baseia em mº ou letras

Radix X Quicksort
mais lento } mais rápido
menos flexivel]

nqu

<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tbody> <tr><td>1</td><td>3</td><td>2</td></tr> <tr><td>5</td><td>4</td><td>3</td></tr> <tr><td>7</td><td>8</td><td>3</td></tr> <tr><td>0</td><td>6</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>7</td></tr> </tbody> </table>	1	3	2	5	4	3	7	8	3	0	6	3	0	0	7	→	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tbody> <tr><td>0</td><td>0</td><td>7</td></tr> <tr><td>1</td><td>3</td><td>2</td></tr> <tr><td>5</td><td>4</td><td>3</td></tr> <tr><td>0</td><td>4</td><td>9</td></tr> <tr><td>0</td><td>6</td><td>3</td></tr> </tbody> </table>	0	0	7	1	3	2	5	4	3	0	4	9	0	6	3	→	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tbody> <tr><td>0</td><td>0</td><td>7</td></tr> <tr><td>0</td><td>4</td><td>9</td></tr> <tr><td>0</td><td>6</td><td>3</td></tr> <tr><td>1</td><td>3</td><td>2</td></tr> <tr><td>5</td><td>4</td><td>3</td></tr> </tbody> </table>	0	0	7	0	4	9	0	6	3	1	3	2	5	4	3
1	3	2																																															
5	4	3																																															
7	8	3																																															
0	6	3																																															
0	0	7																																															
0	0	7																																															
1	3	2																																															
5	4	3																																															
0	4	9																																															
0	6	3																																															
0	0	7																																															
0	4	9																																															
0	6	3																																															
1	3	2																																															
5	4	3																																															

Pior Caso: Todos os elementos têm o mesmo número de dígitos, exceto um. $O.(m^2)$

Se n° de dígitos no maior elemento = m $\rightarrow O.(m^2)$

Melhor Caso: Todos os elementos têm o mesmo número de dígitos, exceto o menor. $O(a(m+b))$

Se b = O(m) $\rightarrow O.(a * m)$

VANTAGENS

→ Rápido quando os todos são curtos.

(A) LSD - do algoritmo menos significativo para o mais significativo.
 MSB - do algoritmo mais significativo para o menos significativo.

Radix Sort \rightarrow Counting Sort



Não envolve comparações entre elementos

- (A) Para cada elemento x , determinar o número de elementos menores ou iguais a x
- (B) Colocar x na posição correta no array de saída.



exemplo $x = 6$



nº de elementos $\leq x = 4$

se k for constante $O(m)$

Complexidade $O(m + k)$

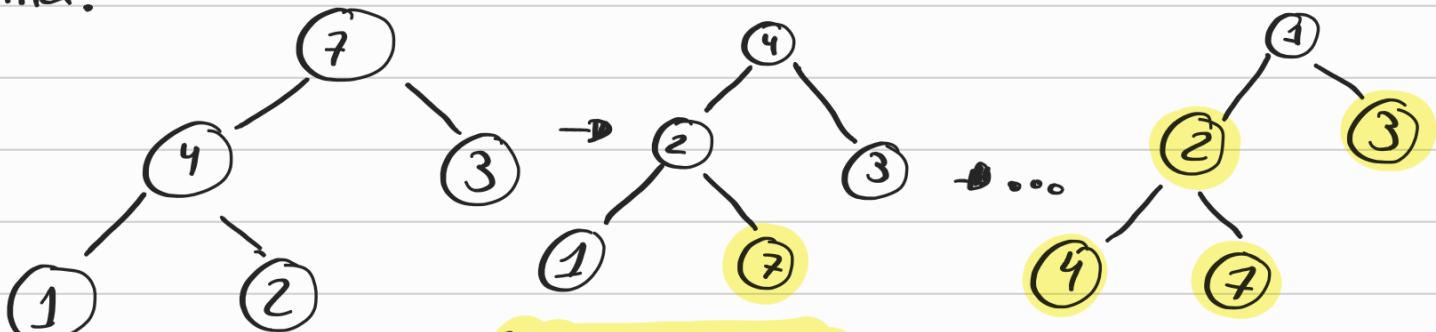
- \rightarrow Método estavel.
- \rightarrow Não é in-place

melhor método de ordenação das colunas/dígitos.

\rightarrow Têm a desvantagem de necessitar de vetores.

Heap Sort

• Utiliza o paradigma da transformação-e-conquista, utilizando uma diferente representação (através do heaps) do mesmo problema.



- Constrói-se uma heap de máximo a partir da lista
- Troca-se o primeiro com o último elemento da heap
- Retira-se o último elemento da heap
- Reconstrói-se a heap
- Repete-se até não haver elementos da heap.

Analise Geral

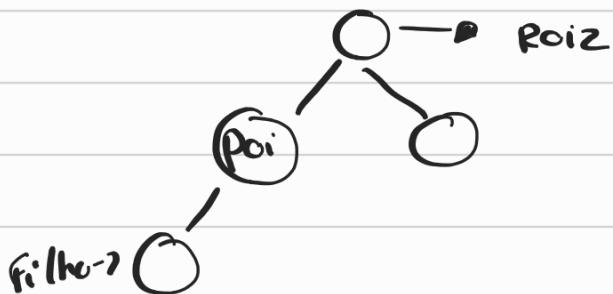
- Eficaz em geral
- Tempo gasto em execução: $\Theta(n \lg n)$
- Mais lento que o Quick Sort
- Não tem um caso - pior
- Ordenação in-place: $O(1)$
- Não é estável

Árvores Binárias

Árvores (free trees)

→ Conjunto não vazio de vértices e ligações que satisfaça a certas condições, existe exatamente um caminho a ligar a dois vértices da árvore.

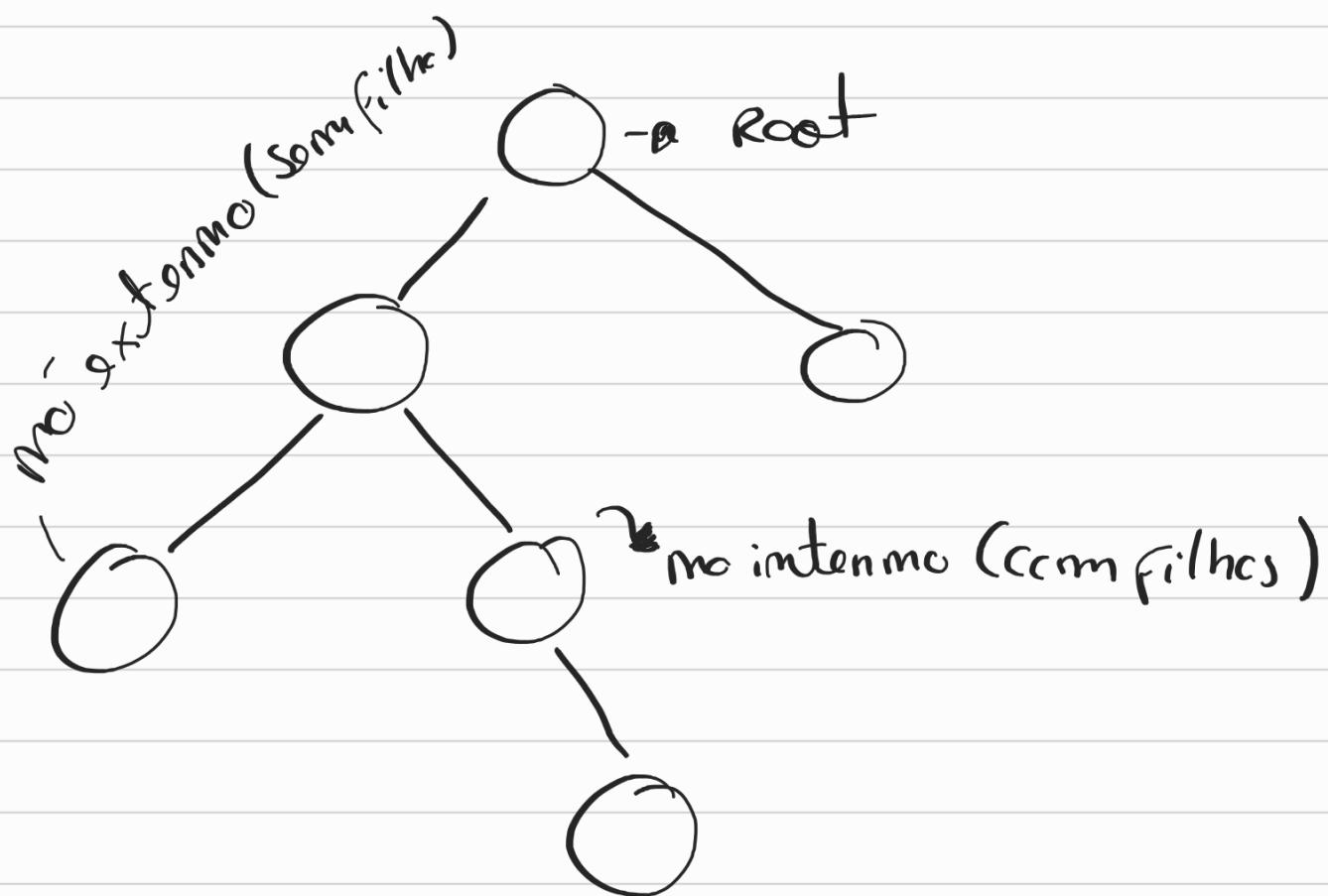
Árvores com raiz (rooted trees)



Árvores ordenadas (ordered trees)

→ é uma árvore com raiz
→ a ordem dos filhos em cada nó tem significado

Anúncios binários



Anúncio binomial cheia (perfect BT)

→ todos os folhas tem o mesmo nível e todos os nós internos têm dois filhos.

Anúncio binomial completo

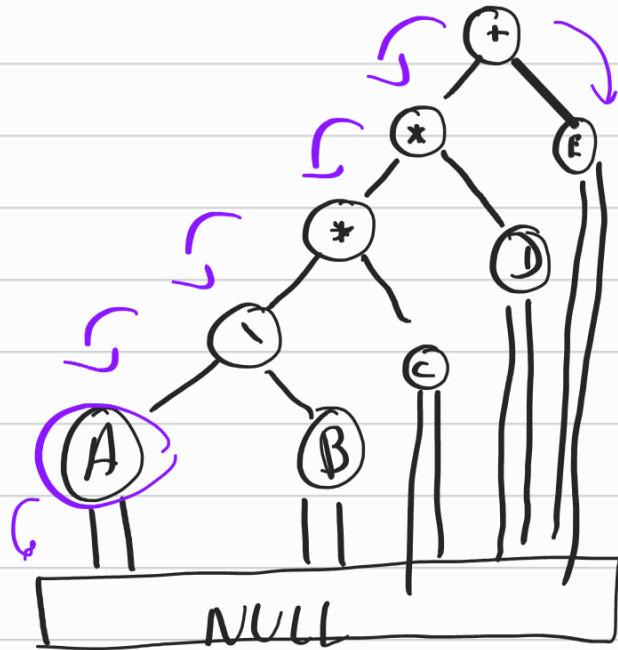
→ todos os níveis, excepto o último, estão completos, e os níveis estão o mais à esquerda possível.

Anúncio binária própria (full BT; prepos. BT) e uma
árvore binária em que cada nó tem 0 ou 2 filhos

Void imorder(tree pointer){
if(ptr) {

```
imorder( ptr-> left-child );  
printf(" %d ", ptr-> data );  
imorder( ptr-> right-child );
```

return;



(A) () (B) ()

Heaps

max-heaps e min-heaps

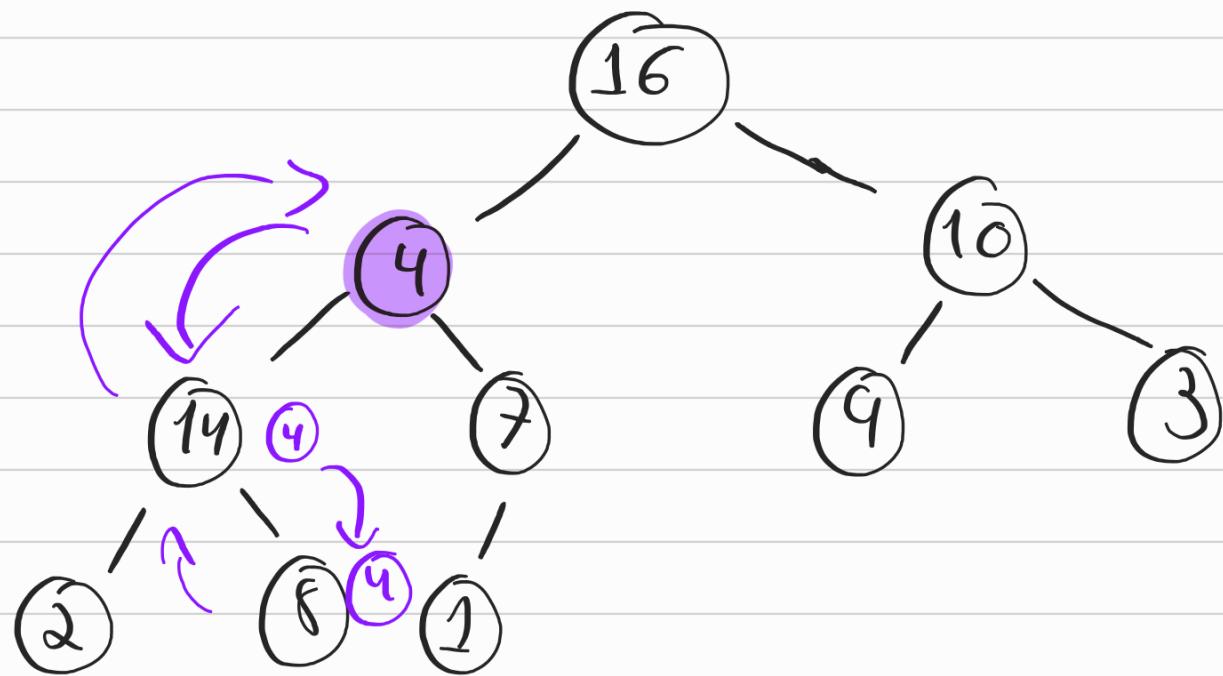
- O valor de um nó é maior ou igual ao valor de todos os seus filhos.
- O valor máximo da estrutura encontra-se na raiz.

mim-heaps

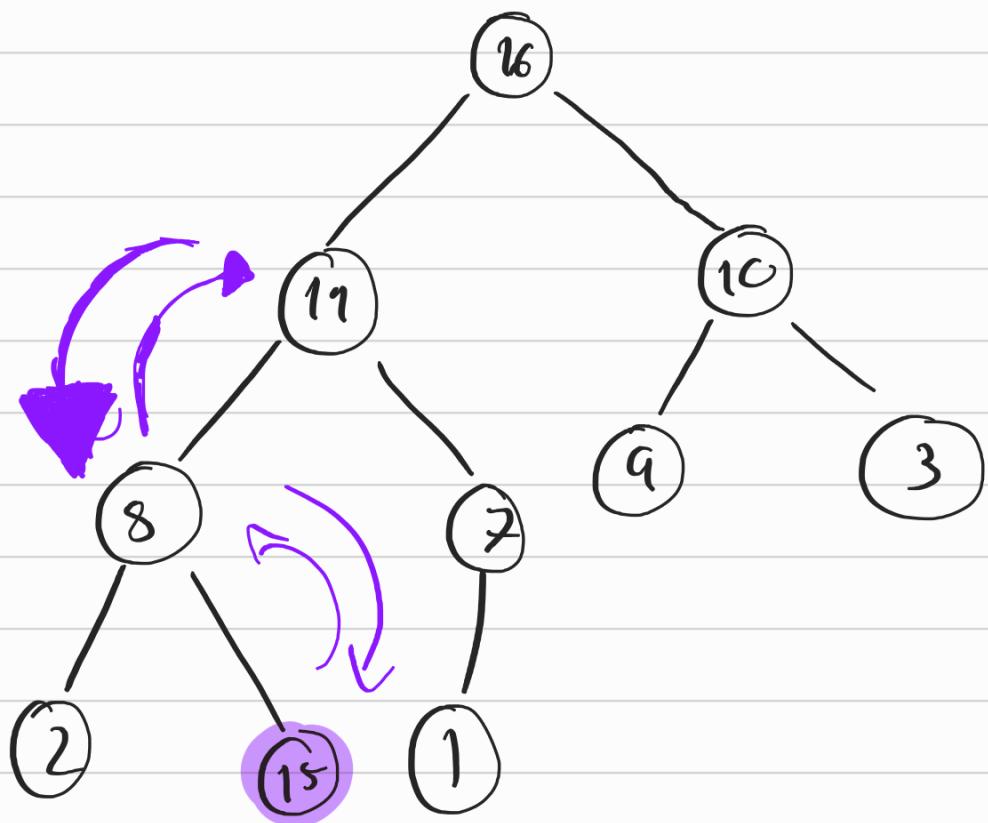
- o valor de um vértice mai pode ser menor do que do seu pai
- valor mínimo da cinchete em contêiner raiz.

Inicialização e manutenção

Top-Down Heapify (Sink)



Bottom - Up - Heapify (swim)



Operações típicas:

- inicialização e manutenção
- inserção e eliminação do mês
- método de ordenação heapsort

Anúncios binários de pesquisa



Observe em:- Seja um S^t m-

- todos os més da sub - árvore esquerda

de x têm de ter valores menores ou iguais a x

- todos os més da sub - árvore direita de

x têm de ter valores maiores ou iguais a x.

Operações típicas

- 1- ➔ Eliminar um mō sem filhos (folha)
- 2- ➔ Eliminar um mō com um filho (mantendo este as suas sub-árvore).
- 3- ➔ Eliminar um mō com dois filhos.

1- Basta eliminar o mō da árvore
2- substitui-se esse mō pelo seu filho mantendo
este os seus sub-árvore.
3- Substitui-se pelo seu antecessor ou sucessor
R (ao permanecer a árvore utilizando o método imediatamente
eliminando este da sua posição anterior).

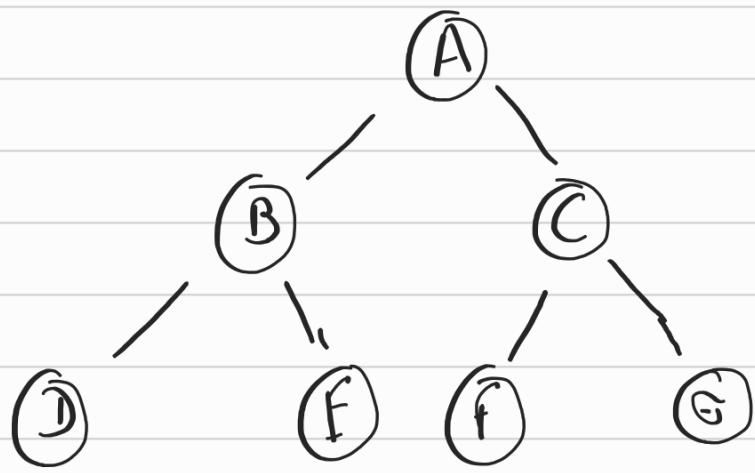
↓ ➔ Nota: representação ppt 52/60

antecessor → mō mais à direita da sua
sub-árvore esquerda

sucessor → mō mais à esquerda da sua
sub-árvore direita.

Estratégias de equilíbrio de árvores binárias

1. Não equilibrar
2. Equilíbrio Rigoroso
3. Um bom equilíbrio
4. Adaptação ao acesso.
➔ melhora a amortização



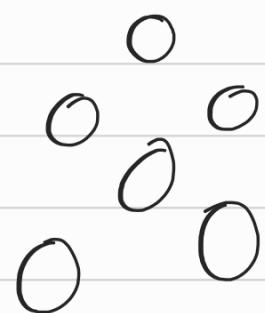
inorder \rightarrow D, B, E, A, F, C, G

Preorder \rightarrow A, B, D, E, C, F, G

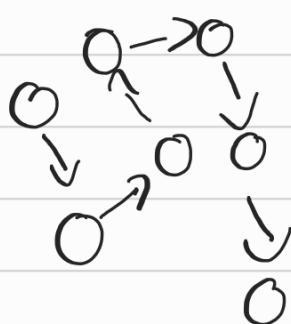
Postorder \rightarrow D, E, B, F, G, C, A

LISTAS

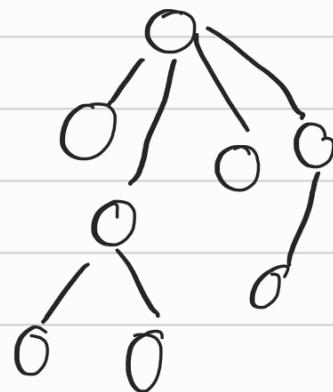
Relações entre os dados



Conjunto de dados



Relação linear
lista



Relação não linear
árvore

Conjunto → Coleção de objetos diferentes
→ Contém elementos não repetidos e sem uma
ordem estabelecida.

Lista → pode conter elementos repetidos
→ os elementos são identificados por índices.

$$\{3, -2, 5, 0, 11, 5\} \neq \{11, 5, 0, -2, 3, 5\}$$

1º elemento (head)

2º elemento (tail)

Propriedades → Pode ter zero ou mais elementos
→ um novo elemento pode ser adicionado em qualquer posição.

Arrays

- VANTAGENS
 - Acesso direto a qualquer elemento
 - Facilidade em percorrer a lista
- Desvantagens
 - Operações de inserção (ou remoção) lentas
 - Uso inefficiente dos recursos de memória

Listas encadeadas

- VANTAGENS
 - Uso eficiente dos recursos de memória
 - Operações de inserção e remoção rápidos

- Desvantagens
 - Acesso sequencial aos elementos

- Pode haver dificuldade em percorrer a lista.

• Sequência de nós

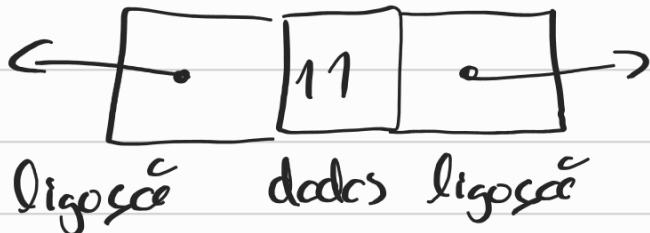
—
↳ Contém: os dados (de qualquer tipo)
uma ligação ao elemento
seguinte

- O último nó aponta para NULL

Sucessor (seguinte do nó)
Antecessor (anterior do nó)

Lista encadeada

- (A) Os dados podem ser de qualquer tipo
- (B) Uma ligação ao elemento seguinte
- (C) Uma ligação ao elemento anterior



- Cada nó tem uma ligação para o nó anterior e para o nó seguinte
- A ligação para o nó anterior do primeiro nó aponta para NULL
- A ligação para o nó seguinte do último nó aponta para NULL

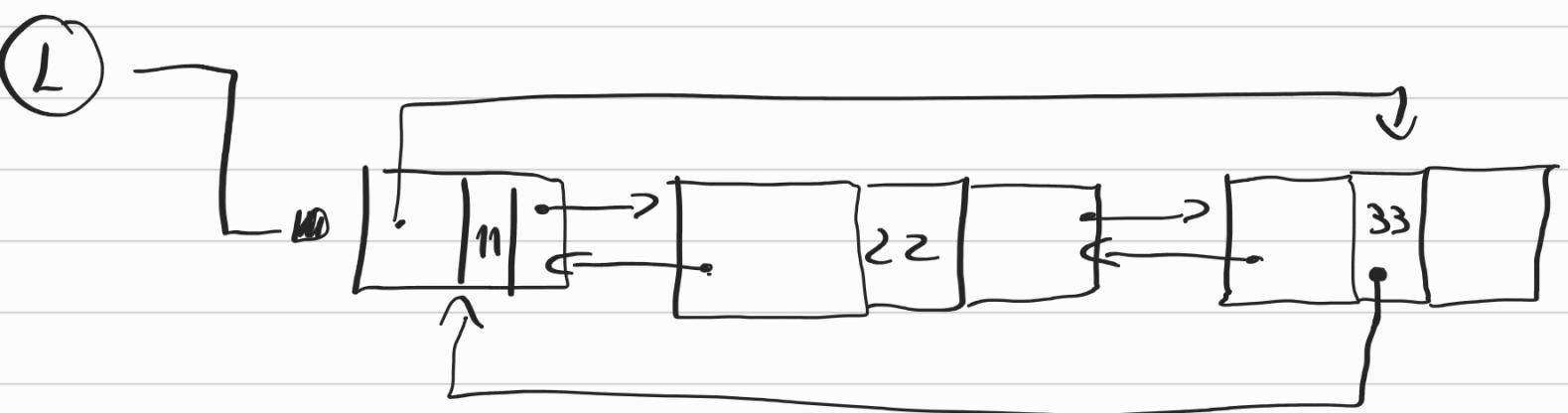
VANTAGENS

- ↳ Reverter a lista duplamente ligada é muito fácil
- ↳ A exclusão de nós é fácil com campanhas com uma lista encadeada. Uma exclusão de lista encadeada simples requer um ponteiro para o nó e o nó anterior a ser excluído, mas em uma doubly linked list, é necessário apenas o ponteiro que deve ser excluído.
- Pode alocar ou realocar memória facilmente durante sua execução.

Listo Circular

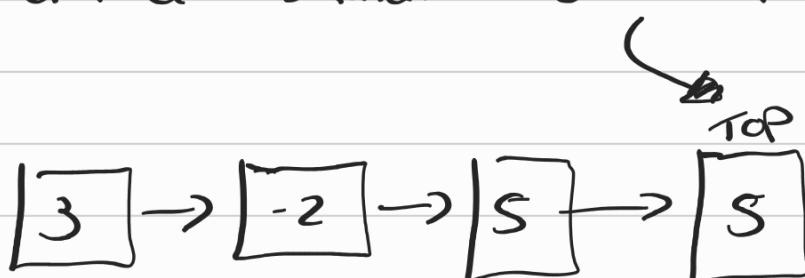
→ O ultimo mó aponta para o ultimo.

Listo Circular biemcodeada



STACKS

→ Operações de entrada e saída de elementos são efetuados por um dos extremos da lista



Definições: → Estas operações são efetuadas somente por uma das extremidades da lista, normalmente designado por top

→ A operação de entrada de dados é designada por push

→ A operação de saída de dados é designada por pop.

Um stock diz-se LIFO (last in first out)
(O ultimo elemento a entrar é o primeiro a sair).

Propriedades:

- pode ter 0 ou mais elementos
- O elemento posicionado numa das extremidades designa-se por TOP
 - Só o elemento TOP deve ser accedido
 - Um novo elemento é adicionado a seguir ao elemento TOP
 - em cada instante, só pode ser eliminado o elemento TOP

① arrays

VANTAGENS: → entrada / saída de dados rápida
→ facilidade de utilização

Desvantagens: → Uso ineficaz dos recursos de memória.

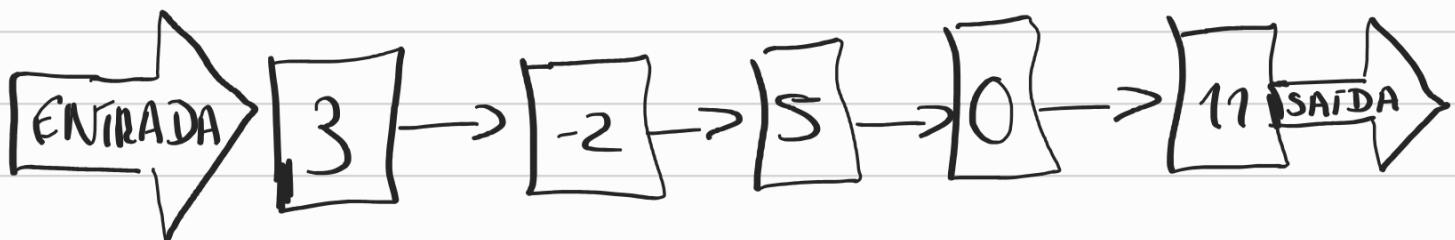
② Listas encadeadas

VANTAGENS: → Entrada / saída de dados rápidas
→ Uso eficaz dos recursos de memória.

Desvantagens: → Menor facilidade de utilização

Filos de espera

→ Operações de entrada e saída de elementos são efetuadas por extremos opostos da lista



Definições:

- Estas operações são efetuadas por extremidades opostas da lista.
- A operação de entrada de dados é designada por enqueue.
- A operação de saída de dados é designada por dequeue.

Uma fila de espera diz-se FIFO (first in first out)
(o primeiro elemento a entrar é o primeiro a sair)

Propriedades:

- pode ter zero ou mais elementos
- o elemento de uma das extremidades designa-se por Front e o da outra por REAR
- só os elementos referidos em cima podem ser accedidos
- Adicionam elementos e outras do REAR
- Em cada instante só pode ser eliminado o elemento front.

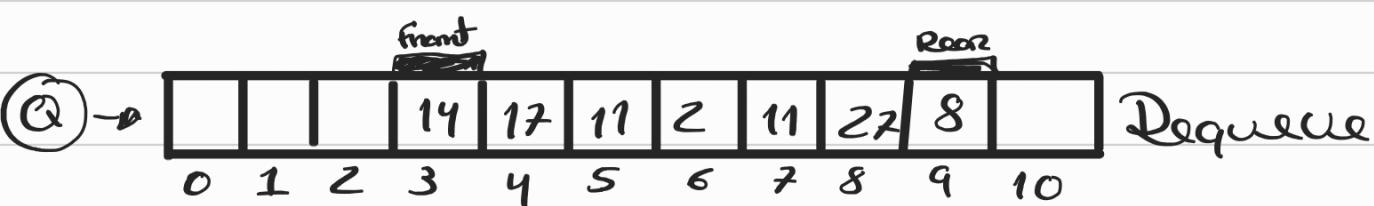
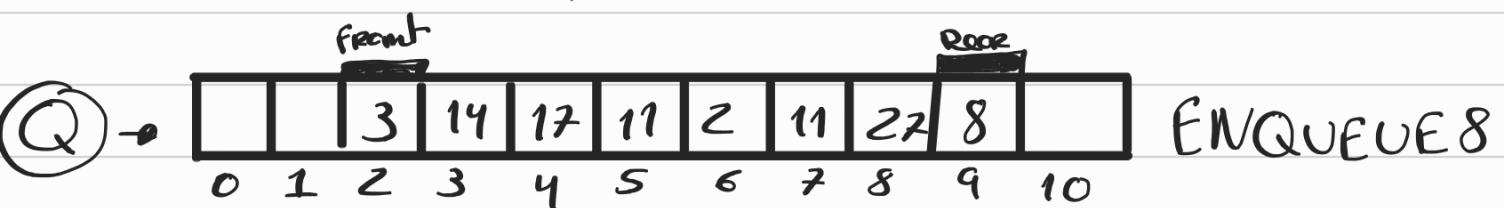
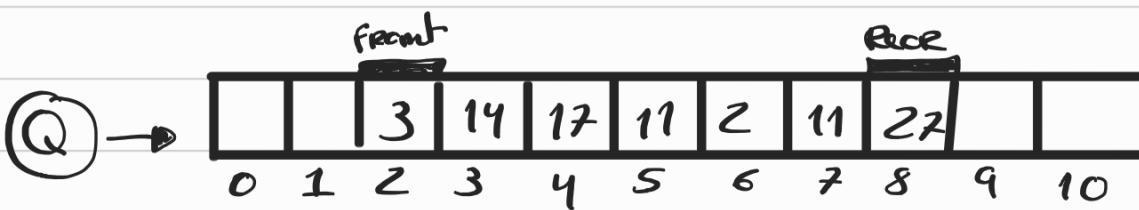
① ARNOYS

VANTAGENS:

- Entrada/Saída de dados rápida
- Facilidade de utilização

Desvantagens:

- Uso ineficaz dos recursos de memória.



- Após um conjunto de entradas e saídas, a fila de "move-se" para a direita.
- Este aspecto pode gerar "falsos" problemas de overflow.

② Listas encadeadas

VANTAGENS → Possível entrada/saída de dados rápida

- Uso eficaz dos recursos de memória.

Desvantagens:

- Possível entrada/saída de dados lenta.

