Complexidade de Algoritmos Associados a Estruturas de Dados (Noções e Exemplos)

Pedro Ximenes

Universidade Católica de Pernambuco (UNICAP)

Análise de Algoritmos – 2026.1

Roteiro da Aula

Revisão – Aula 5

Tópicos Abordados

1. Análise completa do **Selection Sort** ($\Theta(n^2)$) e **Bubble Sort** ($O(n^2)$).
2. Comparação entre **Melhor**, **Pior** e **Caso Médio**.
3. Complexidade com **dois parâmetros**: $O(n \cdot m)$.
4. Padrão $O(\sqrt{n})$: verificação de primalidade.
5. **Técnica dos dois ponteiros**: otimização de $O(n^2)$ para $O(n)$.

Revisão – Padrões de Complexidade

Padrão	Exemplo	Complexidade
Sem loop	Sequência de instruções	$O(1)$
1 loop simples	`para i de 0 ate n`	$O(n)$
Loop com divisão	`enquanto n > 1:  n <- n/2`	$O(\log n)$
Loop até $\sqrt{n}$	`enquanto i*i <= n`	$O(\sqrt{n})$
2 loops aninhados	`para i + para j (dep.  de i)`	$O(n^2)$
Série harmônica	$\sum_{i=1}^{n} n/i$	$O(n \log n)$
Dois ponteiros	Two Sum, Palíndromo	$O(n)$

Pergunta Central desta Aula

A complexidade de um algoritmo depende **apenas** do algoritmo? Ou a **estrutura de dados** usada também importa?

A Escolha da Estrutura de Dados Importa

Ideia Central

A **eficiência** de um algoritmo depende diretamente da **estrutura de dados** utilizada. A mesma operação pode ter complexidades completamente diferentes dependendo da ED escolhida.

Exemplo Motivador

Considere a operação **buscar um elemento**:

- Em um **array não ordenado**: $O(n)$ – busca linear.
- Em um **array ordenado**: $O(\log n)$ – busca binária.
- Em uma **tabela hash**: $O(1)$ no caso médio.
- Em uma **árvore binária balanceada**: $O(\log n)$.

Conclusão

Escolher a ED certa pode transformar um algoritmo lento em um algoritmo rápido!

Operações Fundamentais

Operações que analisaremos em cada ED

1. **Acesso/Busca**: Recuperar um elemento (por índice ou por valor).
2. **Inserção**: Adicionar um novo elemento.
3. **Remoção**: Remover um elemento existente.

Trade-off

Nenhuma estrutura de dados é "perfeita" para **todas** as operações. Cada ED faz um **compromisso** (trade-off):

- Array: acesso rápido por índice, mas inserção/remoção lentas.
- Lista encadeada: inserção/remoção rápidas nas pontas, mas acesso lento.
- Tabela hash: operações rápidas no caso médio, mas sem ordenação.

Estruturas que Estudaremos

Roteiro

1. **Array / ArrayList** – Listas baseadas em arrays
2. **LinkedList** – Listas duplamente encadeadas
3. **Pilha (Stack)** – LIFO
4. **Fila (Queue)** – FIFO
5. **Árvore Binária de Pesquisa (BST)**
6. **Heap** – Fila de prioridade
7. **Tabela Hash**

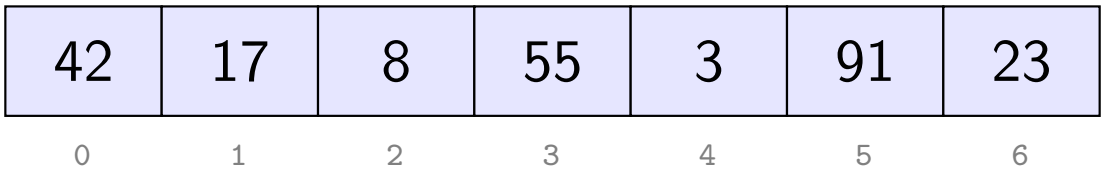
Objetivo

Para cada ED: entender a estrutura, analisar a complexidade de cada operação fundamental, e comparar com as alternativas.

O Array – A Estrutura Mais Elementar

Definição

Um **array** é uma coleção de elementos armazenados em posições **contíguas** de memória, acessíveis por um **índice** inteiro.



Propriedade Fundamental

O acesso por índice é feito em **tempo constante** $O(1)$, pois o endereço de memória é calculado diretamente:

$$\text{endereço}[i] = \text{base} + i \times \text{tamanho\_elemento}$$

Operações em Array Puro – Complexidades

Operação	**Complexidade**	**Razão**
Acesso por índice `V[i]`	$O(1)$	Cálculo direto
Busca por valor	$O(n)$	Busca linear
Inserção no fim	$O(1)$	Se há espaço
Inserção em posição i	$O(n)$	Shift à direita
Remoção da posição i	$O(n)$	Shift à esquerda

Limitações do Array Puro

- **Tamanho fixo**: definido na criação.
- Inserção/remoção em posição arbitrária: precisa deslocar ("shift") todos os elementos subsequentes.
- Programador deve controlar manualmente o número de elementos.

ArrayList – Lista Dinâmica Baseada em Array

O que é?

Uma **ArrayList** encapsula um array e oferece uma API com operações de lista, escondendo detalhes como remanejamento de elementos e crescimento dinâmico.

Atributos internos:

- `lista[]`: o array que armazena os elementos.
- `tamanho`: quantidade de elementos presentes.
- `CAPACIDADE_DEFAULT`: tamanho inicial (ex: 20).

Quando o array enche...

1. Cria-se um novo array com o **dobro** da capacidade.
2. Copiam-se todos os elementos para o novo array – **resize**: $O(n)$.
3. Adiciona-se o novo elemento.

ArrayList – Inserção no Fim (add)

Pseudocódigo

```
1  algoritmo add(lista, tamanho, elemento)
2      se tamanho = capacidade(lista) entao
3          novaLista ← novo vetor[2 * capacidade(lista)]
4          para i de 0 ate tamanho-1 faca
5              novaLista[i] ← lista[i]
6          fim para
7          lista ← novaLista    // resize: O(n)
8      fim se
9      lista[tamanho] ← elemento
10     tamanho ← tamanho + 1
11 fim algoritmo
```

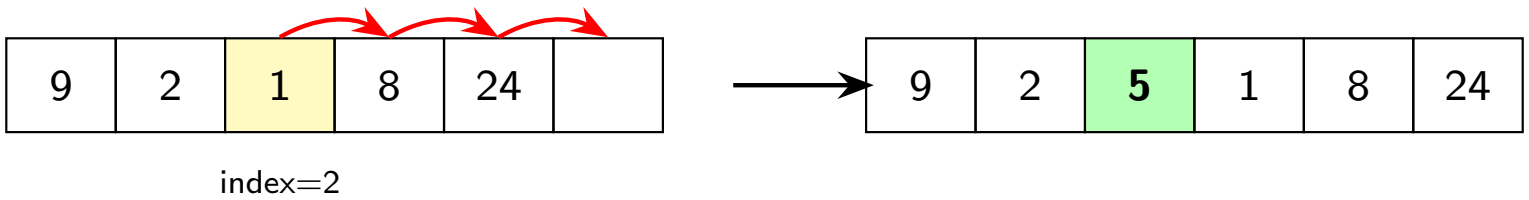
Complexidade

- **Sem resize**: $O(1)$ – apenas uma atribuição.
- **Com resize**: $O(n)$ – copiar todos os elementos.
- **Amortizado**: $O(1)$ – o resize é raro; para n inserções no fim, o custo total é $O(n)$.

ArrayList – Inserção em Posição Arbitrária

Pseudocódigo – add(index, elemento)

```
1 algoritmo addEmPosicao(lista, tamanho, index, elem)
2     // Shift para a direita
3     para i de tamanho ate index+1 (decrementando) faca
4         lista[i] ← lista[i-1]
5     fim para
6     lista[index] ← elem
7     tamanho ← tamanho + 1
8 fim algoritmo
```



Pior caso (index = 0): $O(n)$

ArrayList – Remoção

Pseudocódigo – remove(index)

```
algoritmo remove(lista, tamanho, index)
    elemento ← lista[index]
    // Shift para a esquerda
    para i de index ate tamanho-2 faca
        lista[i] ← lista[i+1]
    fim para
    tamanho ← tamanho - 1
    retorne elemento
fim algoritmo
```

Análise – Pior Caso

- Remover do **início** (index $= 0$): desloca $n - 1$ elementos $\Rightarrow O(n)$.
- Remover do **fim**: sem shift $\Rightarrow O(1)$.

ArrayList – Busca

Busca por Índice – get(index)

Acesso direto ao array: `lista[index]`

$$\boxed{O(1)}$$

Busca por Valor – indexOf(elem) / contains(elem)

Busca linear: percorrer o array comparando elemento a elemento.

$$\boxed{O(n)}$$

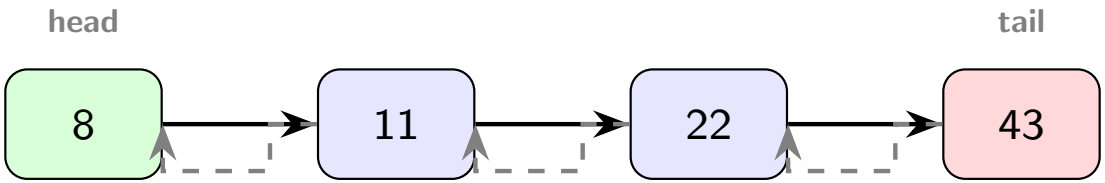
Resumo das Complexidades – ArrayList

Operação	**Complexidade**	**Observação**
`get(i)`	$O(1)$	Acesso direto
`add(elem)` no fim	$O(1)$ amortizado	Resize $O(n)$ eventual
`add(i, elem)`	$O(n)$ pior caso	Shift à direita
`remove(i)`	$O(n)$ pior caso	Shift à esquerda
`indexOf(elem)`	$O(n)$	Busca linear

LinkedList – Estrutura

Ideia

Ao invés de armazenar elementos em posições contíguas de memória, cada elemento é um **nó** que contém o dado e **referências** para o nó anterior e o próximo.



- Cada **nó** contém: valor, referência `next` e referência `prev`.
- A lista mantém referências para `head` (início) e `tail` (fim).
- Os elementos **não** são contíguos na memória.
- Não há como acessar o i-ésimo elemento diretamente (sem percorrer a lista).

LinkedList – Inserção

addFirst(elem) e addLast(elem) – $O(1)$

- **addFirst**: Cria novo nó, liga-o antes de `head`, atualiza `head`.
- **addLast**: Cria novo nó, liga-o após `tail`, atualiza `tail`.
- Apenas manipulação de **referências** – sem deslocamento!

add(index, elem) – $O(n)$

Para inserir em uma posição arbitrária, é preciso **percorrer** a lista até o índice desejado (não há acesso direto por índice).

- Percorrer até a posição: $O(n)$.
- Manipular referências: $O(1)$.
- Total: $O(n)$.

Vantagem sobre ArrayList

Não é necessário fazer **shift** de elementos nem **resize**!

LinkedList – Remoção

removeFirst() e removeLast() – $O(1)$

- **removeFirst**: Atualiza `head` para `head.next`.
- **removeLast**: Atualiza `tail` para `tail.prev`.
- Apenas manipulação de referências.

remove(index) e remove(elem) – $O(n)$

Precisa percorrer a lista para encontrar o nó:

- Percorrer: $O(n)$.
- Remover (desligar referências): $O(1)$.

LinkedList – Busca

get(index), indexOf(elem), contains(elem) – $O(n)$

Todos requerem **iteração** pela lista:

- `get(index)`: percorrer i nós a partir do `head`.
- `indexOf(elem)`: percorrer comparando valores.

getFirst() e getLast() – $O(1)$

Basta retornar `head.valor` ou `tail.valor`.

LinkedList – Resumo das Complexidades

Operação	**Complexidade**	**Observação**
`addFirst / addLast`	$O(1)$	Manipulação de referências
`add(i, elem)`	$O(n)$	Percorrer até posição
`removeFirst / removeLast`	$O(1)$	Manipulação de referências
`remove(i)`	$O(n)$	Percorrer até posição
`get(i)`	$O(n)$	Sem acesso direto
`getFirst / getLast`	$O(1)$	Referências head/tail

Observação

A LinkedList é eficiente nas **extremidades** (head/tail), mas requer percorrer a lista para qualquer acesso no **meio**.

ArrayList vs LinkedList – Comparação

Operação	**ArrayList**	**LinkedList**
Acesso por índice	$O(1)$ ✓	$O(n)$
Inserção no início	$O(n)$	$O(1)$ ✓
Inserção no fim	$O(1)$ amort. ✓	$O(1)$ ✓
Remoção do início	$O(n)$	$O(1)$ ✓
Remoção do fim	$O(1)$ ✓	$O(1)$ ✓
Busca por valor	$O(n)$	$O(n)$
Uso de memória	Contígua	Nós + referências

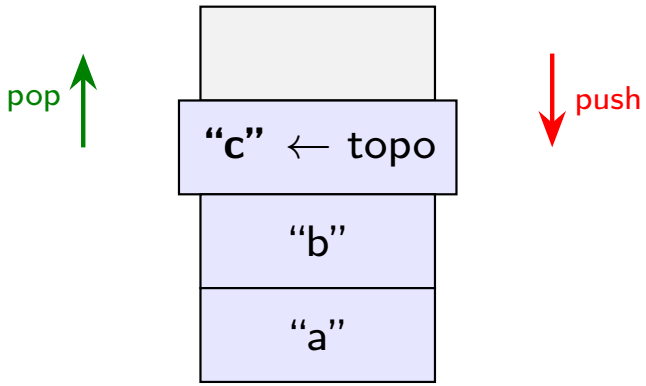
Quando usar cada uma?

- **ArrayList**: quando o acesso por índice é frequente.
- **LinkedList**: quando inserções/remoções no início são frequentes.

Pilha – Last In, First Out

Definição

Uma **pilha** é uma estrutura onde toda adição (`push`) e toda remoção (pop) são feitas no **topo**. O último elemento a entrar é o primeiro a sair.



Aplicações

- Ctrl+Z (desfazer)
- Botão “voltar” do navegador
- Pilha de chamadas recursivas
- Avaliação de expressões

Pilha – Pseudocódigo

push(elem)

```
algoritmo push(pilha, topo, elem)
    se topo+1 = capacidade entao
        // pilha cheia
    fim se
    topo ← topo + 1
    pilha[topo] ← elem
fim algoritmo
```

pop()

```
algoritmo pop(pilha, topo)
    se topo = -1 entao
        // pilha vazia
    fim se
    elem ← pilha[topo]
    topo ← topo - 1
    retorne elem
fim algoritmo
```

Análise

Todas as operações manipulam **apenas o topo** (incremento/decremento de um inteiro e atribuição). Nenhum loop é necessário.

Pilha – Complexidades

Resumo

Operação	Complexidade	Razão
`push(elem)`	$O(1)$	Atribuição + incremento de topo
`pop()`	$O(1)$	Decremento de topo
`peek()`	$O(1)$	Acesso direto a `pilha[topo]`
`isEmpty()`	$O(1)$	Comparação de topo com -1

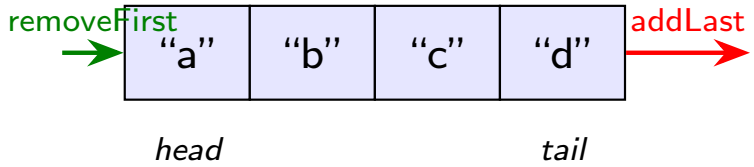
Por que tudo é $O(1)$?

A política LIFO restringe o acesso ao **topo**. Não há necessidade de percorrer a estrutura, apenas manipular um único índice.

Fila – First In, First Out

Definição

Uma **fila** é uma estrutura onde toda adição (`addLast`) é feita no **fim** e toda remoção (`removeFirst`) é feita no **início**. O primeiro a entrar é o primeiro a sair.



Fila – Implementação e Complexidade

Problema: Remoção com Shift

Se sempre removemos do início e fazemos `shiftLeft`, a remoção é $O(n)$. Podemos melhorar!

Solução: Fila Circular

Ao invés de deslocar elementos, usamos aritmética modular:

- `addLast`: $tail \leftarrow (tail + 1)$ % $capacidade$
- `removeFirst`: $head \leftarrow (head + 1)$ % $capacidade$

O array é tratado de forma **circular**. Sem shift!

Complexidades – Fila Circular

Operação	**Complexidade**
`addLast(elem)`	$O(1)$
`removeFirst()`	$O(1)$
`isEmpty()`	$O(1)$
`peek()`	$O(1)$

Fila – Exemplo de Funcionamento Circular

Fila com capacidade 4. Mostramos `head` (H) e `tail` (T):

1. addLast("a"), addLast("b"), addLast("c"):

$$[\underset{H}{a},\ b,\ \underset{T}{c},\ \_]$$

2. removeFirst() $\rightarrow$ "a":

$$[\_,\ \underset{H}{b},\ \underset{T}{c},\ \_]$$

3. addLast("d"), addLast("e"):

$$[\underset{T}{e},\ \underset{H}{b},\ c,\ d]$$

O tail "deu a volta" no array graças a $(tail + 1)\ \%\ 4 = 0$.

Ponto Chave

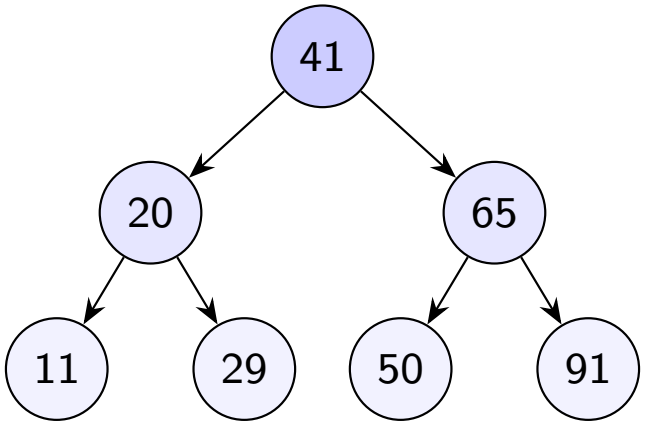
A aritmética modular permite que tanto `addLast` quanto `removeFirst` sejam $O(1)$, eliminando a necessidade de shift.

BST – Definição

Árvore Binária de Pesquisa (BST)

Uma **BST** é uma árvore binária onde, para **todo** nó:

- A subárvore à **esquerda** contém apenas valores **menores**.
- A subárvore à **direita** contém apenas valores **maiores**.



BST – Conceitos Importantes

Terminologia

- **Raiz**: nó superior (41 no exemplo anterior).
- **Folhas**: nós sem filhos (11, 29, 50, 91).
- **Altura** h: maior caminho entre a raiz e uma folha.

Ponto Central

As operações na BST (busca, inserção, remoção) dependem da **altura** h.

- Quanto **menor** h, mais eficiente.
- Uma BST com n nós pode ter altura entre $\log_2 n$ (balanceada) e $n - 1$ (degenerada).

BST – Busca

Ideia

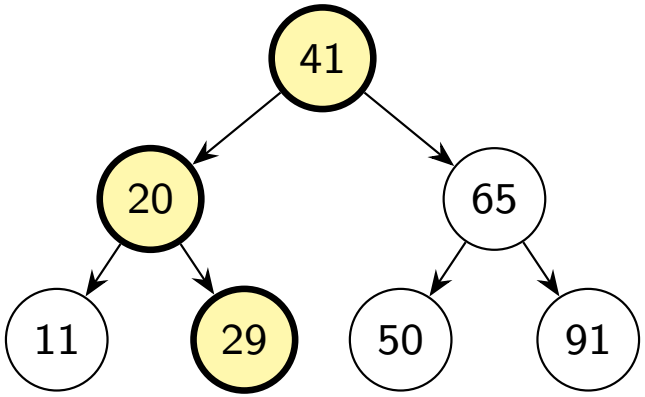
Comparar o valor buscado com o nó atual. Se menor, ir à esquerda. Se maior, ir à direita. Repetir até encontrar ou chegar a nulo.

Pseudocódigo

```
1  algoritmo buscaBST(raiz, valor)
2      no ← raiz
3      enquanto no ≠ nulo faca
4          se valor = no.valor entao
5              retorne no
6          senao se valor < no.valor entao
7              no ← no.esquerda
8          senao
9              no ← no.direita
10         fim se
11     fim enquanto
12     retorne nulo  // nao encontrou
13 fim algoritmo
```

BST – Exemplo de Busca

Buscar o valor 29 na BST abaixo:



1. Compara 29 com 41: $29 < 41 \Rightarrow$ ir à **esquerda**.
2. Compara 29 com 20: $29 > 20 \Rightarrow$ ir à **direita**.
3. Compara 29 com 29: $29 = 29 \Rightarrow$ **Encontrou!**

Percorremos 3 nós $\Rightarrow$ custo proporcional à **altura** h.

Busca em BST: $O(h)$

BST – Inserção

A inserção segue o mesmo caminho da busca: percorre a árvore e insere o novo nó como **folha** na posição correta.

```
1  algoritmo inserirBST(raiz, valor)
2      se raiz = nulo entao
3          raiz ← novo No(valor)
4          retorne
5      fim se
6      no ← raiz
7      enquanto no ≠ nulo faca
8          se valor < no.valor entao
9              se no.esquerda = nulo entao
10                 no.esquerda ← novo No(valor)
11                 retorne
12             fim se
13             no ← no.esquerda
14         senao
15             se no.direita = nulo entao
16                 no.direita ← novo No(valor)
17                 retorne
18             fim se
19             no ← no.direita
20         fim se
21     fim enquanto
22 fim algoritmo
```

BST – Complexidade de Inserção

Análise

A inserção percorre um caminho da raiz até uma folha, fazendo $O(1)$ trabalho por nó visitado.

Inserção em BST: $O(h)$

Exemplo

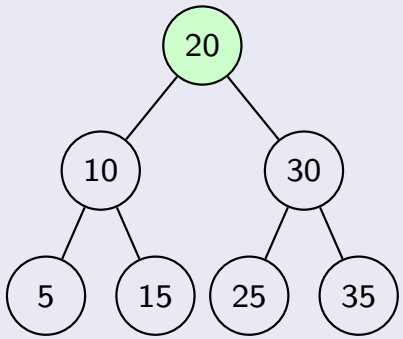
Na BST com raiz 41, inserir o valor 45:

1. $45 > 41 \Rightarrow$ ir à direita (65).
2. $45 < 65 \Rightarrow$ ir à esquerda (50).
3. $45 < 50 \Rightarrow$ ir à esquerda (nulo) $\Rightarrow$ inserir aqui!

Custo: 3 comparações $= O(h)$.

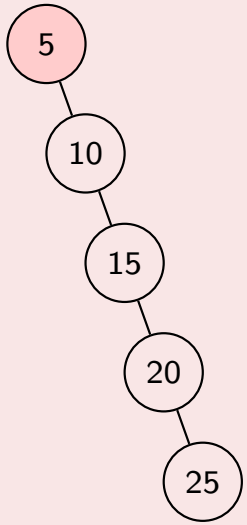
BST – Balanceada vs Degenerada

Árvore Balanceada



$n = 7,\ h = 2 = O(\log n)$

Árvore Degenerada



$n = 5,\ h = 4 = O(n)$

Quando ocorre o pior caso?

Inserir elementos em ordem crescente ou decrescente produz uma "lista" com $h = n - 1$.

BST – Resumo de Complexidades

Operação	**Melhor Caso** (balanceada, $h = O(\log n)$)	**Pior Caso** (degenerada, $h = O(n)$)
Busca	$O(\log n)$	$O(n)$
Inserção	$O(\log n)$	$O(n)$
Remoção	$O(\log n)$	$O(n)$
Mínimo / Máximo	$O(\log n)$	$O(n)$

Conclusão

A eficiência de uma BST depende da **altura** h. Árvores balanceadas (como AVL e Red-Black) garantem $h = O(\log n)$, assegurando operações eficientes.

Heap – O que é?

Motivação: Fila de Prioridade

Queremos uma estrutura que permita:

- Inserir elementos com prioridade.
- Extrair o elemento de **maior** (ou menor) prioridade rapidamente.

Alternativas com estruturas lineares

Estratégia	**Inserção**	**Extração do máx.**
Lista ordenada	$O(n)$	$O(1)$
Lista não ordenada	$O(1)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)$

Heap é o meio-termo ideal!

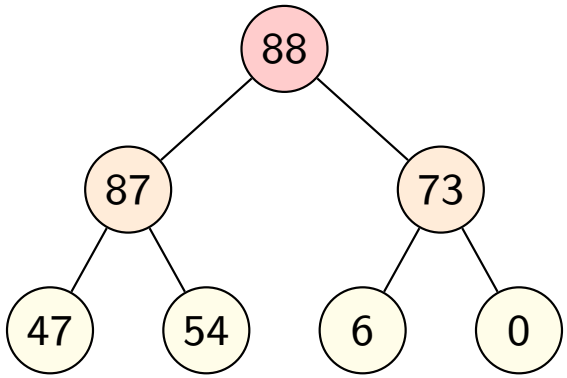
Ambas as operações são $O(\log n)$, e a consulta ao máximo é $O(1)$.

Heap – Propriedades

Definição: Heap Máximo

Uma árvore binária que satisfaz duas propriedades:

1. **Propriedade do Heap**: O valor de cada nó é **maior ou igual** ao de seus filhos. O maior está sempre na raiz.
2. **Completude**: A árvore é **completa** ou **quase-completa da esquerda para a direita**. Isso garante $h = O(\log n)$.



Heap – Representação em Array

Como representar o Heap em um array?

Percurso em largura: $heap = [88, 87, 73, 47, 54, 6, 0]$

Fórmulas de Navegação

Para um nó no índice i:

- Filho à **esquerda**: $2i + 1$
- Filho à **direita**: $2(i + 1)$
- **Pai**: $\lfloor (i - 1)/2 \rfloor$

Por que funciona?

A propriedade de completude garante que não há "buracos" no array. Os nós são dispostos nível a nível, da esquerda para a direita.

Heap – Inserção

Algoritmo

1. Adicionar o novo elemento na **próxima posição livre** (fim do array).
2. Comparar com o pai: se for **maior**, trocar.
3. Repetir até satisfazer a propriedade ou chegar à raiz.

Exemplo: Inserir 100 no heap $[88, 87, 73, 47, 54]$

1. Colocar 100 na posição 5: $[88, 87, 73, 47, 54, \mathbf{100}]$
2. 100 ¿ pai(5) $= 73$? Sim $\Rightarrow$ troca: $[88, 87, \mathbf{100}, 47, 54, 73]$
3. 100 ¿ pai(2) $= 88$? Sim $\Rightarrow$ troca: $[\mathbf{100}, 87, 88, 47, 54, 73]$
4. 100 é raiz $\Rightarrow$ parar.

Inserção no Heap: $O(\log n)$

Por quê $O(\log n)$?

No pior caso, o elemento sobe da folha até a raiz. O caminho tem tamanho $h = O(\log n)$ (árvore completa).

Heap – Remoção (Extract Max) e Heapify

Algoritmo de Remoção

1. Trocar a raiz (máximo) com a **última folha**.
2. Decrementar o tamanho do heap.
3. Aplicar **heapify** na raiz para restaurar a propriedade.

Heapify

Compara o nó com seus filhos esquerdo e direito. O **maior dos três** assume a posição do nó. Repetir descendo na árvore.

Exemplo: Remover max de $[100, 87, 88, 47, 54, 73]$

1. Trocar 100 com 73: $[\mathbf{73}, 87, 88, 47, 54, 100]$. Remover 100 (tail–).
2. Heapify(0): $\max(73, 87, 88) = 88 \Rightarrow$ troca: $[88, 87, \mathbf{73}, 47, 54]$.
3. 73 é folha $\Rightarrow$ parar.

Remoção no Heap: $O(\log n)$

Heap – Resumo de Complexidades

Operação	**Complexidade**
Inserção (add)	$O(\log n)$
Remoção do máximo (extractMax)	$O(\log n)$
Consulta do máximo (peek)	$O(1)$
Build Heap (a partir de array)	$O(n)$

Build Heap é $O(n)$?

Sim! Embora cada heapify individual seja $O(\log n)$, a soma total é $O(n)$ porque a maioria dos nós está próxima das folhas.

Tabela Hash – Motivação

O Sonho: $O(1)$ para tudo

E se pudéssemos mapear cada elemento diretamente a uma posição no array, sem buscar?

Tabela de Acesso Direto

Se as chaves são inteiros pequenos (ex: 0 a 1999), podemos usar a chave como índice:

$$\texttt{tabela[chave]} = \texttt{valor} \quad \Rightarrow \quad O(1)$$

Problema

Chaves grandes (ex: CPF com 11 dígitos $\rightarrow 10^{11}$ posições!) ou não numéricas tornam o acesso direto **inviável**.

Tabela Hash – Função Hash

Solução: Função Hash

Uma função que mapeia chaves para índices válidos no array:

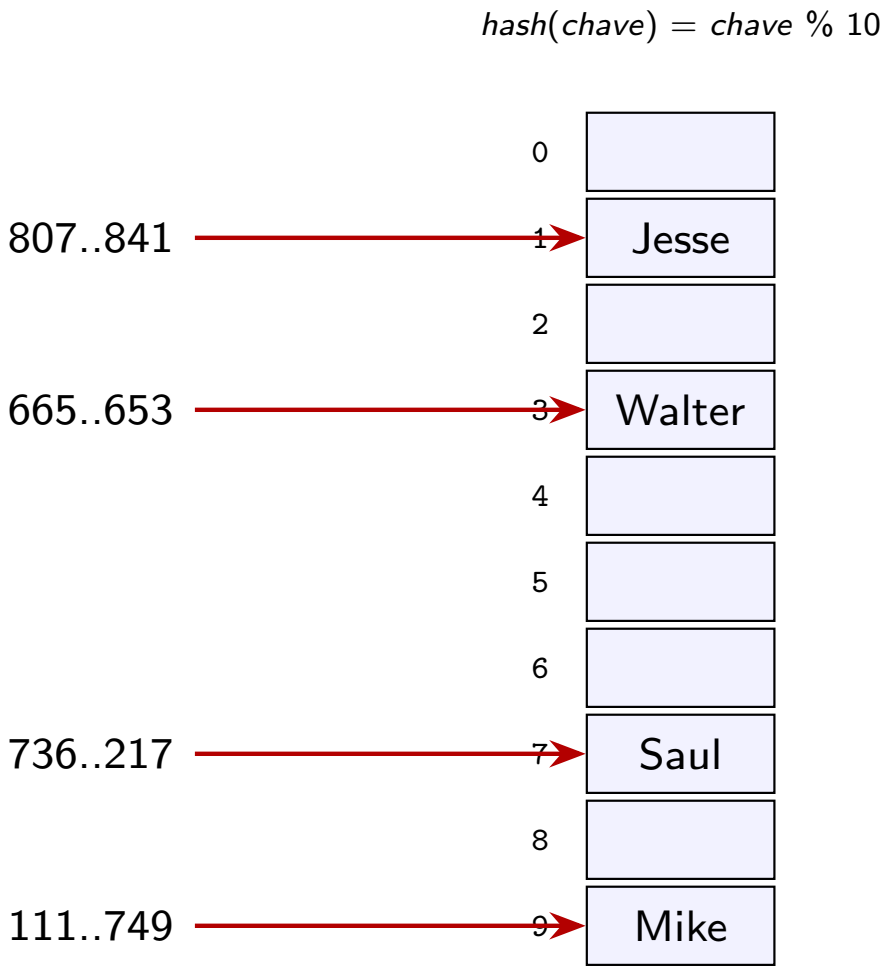
$$hash(chave) \rightarrow \text{índice} \in [0, m-1]$$

Exemplo: $hash(chave) = chave \ \% \ m$, onde m é o tamanho da tabela.

Propriedades de uma boa função hash

1. **Determinística**: mesma chave $\rightarrow$ mesmo hash, sempre.
2. **Eficiente**: executada em $O(1)$.
3. **Uniforme**: distribui as chaves igualmente pela tabela.

Tabela Hash – Funcionamento



- 807365841 % 10 = 1 ⇒ posição 1.
- 665422653 % 10 = 3 ⇒ posição 3.
- A função hash deve ser **determinística**, $O(1)$ e **uniforme**.

Tabela Hash – Colisões

O que é uma colisão?

Duas chaves **diferentes** mapeadas para o **mesmo** índice:

$$hash(chave_1) = hash(chave_2), \quad \text{mas } chave_1 \neq chave_2$$

Resolução por Encadeamento

Cada posição da tabela armazena uma **lista** de elementos com o mesmo hash.

- **Inserção**: calcular hash, adicionar à lista $\Rightarrow O(1)$.
- **Busca**: calcular hash, percorrer a lista $\Rightarrow O(\alpha)$, onde $\alpha = n/m$ é o **fator de carga**.

Tabela Hash – Endereçamento Aberto

Resolução por Endereçamento Aberto

Se a posição está ocupada, procura a **próxima livre** (sondagem linear, quadrática, etc.). Todos os elementos ficam no próprio array.

Sondagem Linear

$$hash(chave, i) = (hash(chave) + i) \% m$$

Com $i = 0, 1, 2, \ldots$ até encontrar posição livre.

Trade-off entre as estratégias

- **Encadeamento**: usa memória extra (listas), mas o fator de carga pode ser > 1.
- **Endereçamento aberto**: sem memória extra, mas o fator de carga deve ser < 1.

Tabela Hash – Complexidades

Operação	Caso Médio	Pior Caso
Inserção (put)	$O(1)$	$O(n)$
Busca (get)	$O(1)$	$O(n)$
Remoção (remove)	$O(1)$	$O(n)$

Quando o pior caso ocorre?

Todas as chaves colidem (mesmo hash) $\Rightarrow$ todos os elementos em uma única lista $\Rightarrow$ busca linear $O(n)$.

Na prática

Com uma boa função hash e fator de carga controlado ($\alpha \leq 0.75$), o **caso médio** $O(1)$ predomina. O resize/rehash é $O(n)$, mas amortizado.

Tabela Hash – Fator de Carga e Resize

Fator de Carga

$$\alpha = \frac{n}{m}$$

onde n = número de elementos e m = tamanho da tabela.

- Se α é muito alto $\Rightarrow$ muitas colisões $\Rightarrow$ degradação do desempenho.
- A biblioteca padrão de Java usa $\alpha = 0.75$ como limite para o resize.

Resize e Rehash

Quando α atinge o limite:

1. Criar nova tabela **maior** (tipicamente $2m$).
2. Recalcular o hash de **cada** elemento com a nova função (rehash).
3. Reinserir todos na nova tabela $\Rightarrow O(n)$.

Esse custo é **amortizado**: ocorre raramente.

Grande Tabela Comparativa

Estrutura	Acesso	Busca	Inserção	Remoção
Array / ArrayList	$O(1)$	$O(n)$	$O(n)^*$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(1)^{**}$	$O(1)^{**}$
Pilha (Stack)	$O(1)^\dagger$	—	$O(1)$	$O(1)$
Fila (Queue)	$O(1)^\dagger$	—	$O(1)$	$O(1)$
BST (balanceada)	—	$O(\log n)$	$O(\log n)$	$O(\log n)$
BST (degenerada)	—	$O(n)$	$O(n)$	$O(n)$
Heap	$O(1)^\ddagger$	—	$O(\log n)$	$O(\log n)$
Tabela Hash	—	$O(1)$ médio	$O(1)$ médio	$O(1)$ médio

* $O(1)$ amortizado se no fim. ** Nas pontas (head/tail). † Apenas topo/head.
‡ Apenas o máximo/mínimo.

Quando Usar Cada Estrutura?

Guia Prático

- **Acesso frequente por índice?** → ArrayList.
- **Inserções/remoções frequentes nas pontas?** → LinkedList ou Deque.
- **Política LIFO (último a entrar, primeiro a sair)?** → Pilha.
- **Política FIFO (primeiro a entrar, primeiro a sair)?** → Fila.
- **Busca eficiente com dados ordenados?** → BST balanceada.
- **Prioridade (obter máximo/mínimo rapidamente)?** → Heap.
- **Busca rápida por chave (sem necessidade de ordem)?** → Tabela Hash.

Exercício Resolvido 1 – Inserções em ArrayList

Enunciado

Uma ArrayList vazia (capacidade inicial 4) recebe as seguintes operações: `add(a)`, `add(b)`, `add(c)`, `add(d)`, `add(e)`.
Quantas operações primitivas (atribuições) são realizadas no total? Qual a complexidade?

Solução:

- `add(a)`: 1 atribuição → total: 1
- `add(b)`: 1 atribuição → total: 2
- `add(c)`: 1 atribuição → total: 3
- `add(d)`: 1 atribuição → total: 4
- `add(e)`: capacidade esgotada! Resize:
  - Copiar 4 elementos para novo array: 4 atribuições.
  - Inserir "e": 1 atribuição.
  - Total do resize: 5.

Total: $1 + 1 + 1 + 1 + 5 = 9$ atribuições para 5 inserções.
Custo amortizado: $9/5 = 1.8$ atribuições por inserção $\Rightarrow O(1)$ amortizado.

Exercício Resolvido 2 – ArrayList vs LinkedList

Enunciado

Considere n inserções no **início** de uma lista. Compare o custo total usando: (a) ArrayList e (b) LinkedList.

Solução (a) – ArrayList:
Cada inserção no início requer shift de todos os elementos atuais:

- 1ª inserção: 0 shifts. 2ª: 1 shift. 3ª: 2 shifts. ... n-ésima: $n-1$ shifts.

$$\text{Total} = 0 + 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Exercício Resolvido 2 – Continuação

Solução (b) – LinkedList:
Cada inserção no início é $O(1)$ (só manipula referências head):

$$\text{Total} = n \times O(1) = O(n)$$

$$\boxed{\text{ArrayList: } \Theta(n^2) \quad \text{vs} \quad \text{LinkedList: } O(n)}$$

Conclusão

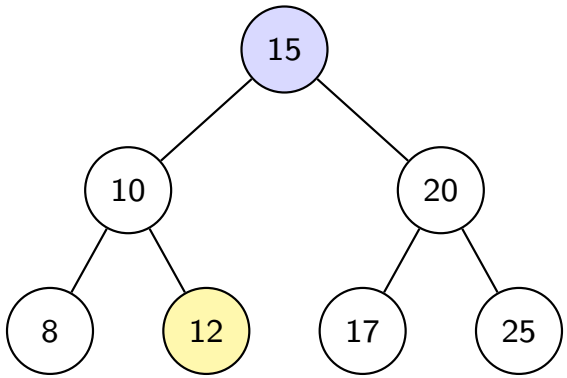
Para inserções frequentes no **início**, LinkedList é muito superior.

Exercício Resolvido 3 – Busca em BST

Enunciado

Os elementos $[15, 10, 20, 8, 12, 17, 25]$ são inseridos nesta ordem em uma BST inicialmente vazia. Desenhe a árvore e determine a complexidade de buscar o elemento 12.

Passo 1 – Construir a árvore:



Passo 2 – Buscar 12:

1. $12 < 15 \Rightarrow$ esquerda.
2. $12 > 10 \Rightarrow$ direita.
3. $12 = 12 \Rightarrow$ encontrou! (3 comparações)

$h = 2$, Custo da busca: $O(h) = O(2)$. Árvore balanceada $\Rightarrow O(\log 7) \approx O(2.8)$.

Exercício Resolvido 4 – Tabela Hash com Colisões

Enunciado

Uma tabela hash de tamanho $m = 7$ usa $hash(k) = k \ \% \ 7$. Insira as chaves $[14, 21, 28, 35, 10]$ e analise o custo de buscar 35. Resolução de colisões: **encadeamento**.

Passo 1 – Calcular hashes:

- $hash(14) = 14 \ \% \ 7 = 0$
- $hash(21) = 21 \ \% \ 7 = 0$ (colisão com 14!)
- $hash(28) = 28 \ \% \ 7 = 0$ (colisão com 14 e 21!)
- $hash(35) = 35 \ \% \ 7 = 0$ (colisão com 14, 21 e 28!)
- $hash(10) = 10 \ \% \ 7 = 3$ (sem colisão)

Exercício Resolvido 4 – Continuação

Passo 2 – Estado da tabela:

- Posição 0: $[14 \to 21 \to 28 \to 35]$ (lista com 4 elementos)
- Posição 3: [10]
- Demais: vazias

Passo 3 – Buscar 35:
$hash(35) = 0$. Percorrer a lista: $14 \to 21 \to 28 \to 35$. **4 comparações**.

Análise

Pior caso com todas as chaves no mesmo bucket $\Rightarrow O(n)$. Usar um m primo (ex: 11) distribuiria melhor.

Exercício Resolvido 5 – Análise de Algoritmo com ED

Enunciado

Analise a complexidade do algoritmo abaixo, considerando que usa: (a) ArrayList e (b) LinkedList.

Pseudocódigo

```
1 algoritmo processaLista(lista, n)
2     para i de 0 ate n-1 faca
3         elemento ← lista.get(i)  // acesso por indice
4         // processa elemento em O(1)
5     fim para
6 fim algoritmo
```

Solução (a) – ArrayList:

`get(i)` é $O(1) \Rightarrow$ Total: $n \times O(1) = O(n)$

Exercício Resolvido 5 – Continuação

Solução (b) – LinkedList:
`get(i)` é $O(i)$ (percorrer i nós) $\Rightarrow$ Total:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

ArrayList: $O(n)$ vs LinkedList: $\Theta(n^2)$

Lição

Usar `get(i)` em loop em uma LinkedList é um **antipadrão** clássico. Para iterar sobre uma LinkedList, deve-se usar um **iterador** (que percorre nó a nó), e não acesso por índice.

Exercício Resolvido 6 – Qual ED Escolher?

Enunciado

Você precisa implementar um sistema com as seguintes operações frequentes: (1) Inserir elementos. (2) Verificar se um elemento existe. (3) Remover elementos por valor. A ordem dos elementos **não importa**. Qual ED escolher?

ED	**Inserção**	**Busca**	**Remoção por valor**
ArrayList	$O(1)$ amort.	$O(n)$	$O(n)$
LinkedList	$O(1)$	$O(n)$	$O(n)$
BST balanceada	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tabela Hash	$O(1)$ médio	$O(1)$ médio	$O(1)$ médio

Melhor escolha: **Tabela Hash**

A ordem não importa, e a Tabela Hash oferece $O(1)$ médio para todas as operações necessárias.

Exercício Proposto 1

ArrayList – Custo de n remoções do início

Dada uma ArrayList com n elementos, qual é o custo total de remover **todos** os elementos do início (posição 0), um por um?

Dica: A cada remoção, há um shift de todos os elementos restantes.

Exercício Proposto 2

BST – Pior caso de inserção

Os elementos $[1, 2, 3, 4, 5, 6, 7]$ são inseridos **nesta ordem** em uma BST vazia.

1. Desenhe a árvore resultante.
2. Qual é a altura da árvore?
3. Qual é a complexidade de buscar o elemento 7?

Dica: Inserção em ordem crescente gera uma árvore degenerada (lista).

Exercício Proposto 3

Heap – Sequência de inserções

Insira os elementos $[10, 20, 15, 30, 25]$ nesta ordem em um Heap Máximo inicialmente vazio.

1. Mostre o estado do heap (array) após cada inserção.
2. Quantas trocas são realizadas no total?

Dica: Após cada inserção, suba o elemento enquanto for maior que o pai.

Exercício Proposto 4

Tabela Hash – Função hash com primo

Uma tabela hash de tamanho $m = 11$ usa $hash(k) = k \ \% \ 11$ com encadeamento. Insira as chaves $[22, 33, 44, 55, 66, 77, 88]$.

1. Em quais posições cada chave é armazenada?
2. Há colisões? Quantas?
3. Qual o fator de carga α?

Dica: Calcule $k \ \% \ 11$ para cada chave.

Exercício Proposto 5

Escolha da ED

Para cada cenário abaixo, indique a **melhor** estrutura de dados e justifique:

1. Um editor de texto precisa implementar `Ctrl+Z` (desfazer) e `Ctrl+Y` (refazer).
2. Um sistema de impressão processa documentos na ordem de chegada.
3. Um jogo precisa manter um ranking dos 10 melhores jogadores, atualizando rapidamente quando alguém pontua.
4. Um compilador precisa verificar se uma variável já foi declarada. A verificação deve ser o mais rápida possível.
5. Um banco de dados precisa manter registros ordenados para consultas por intervalo (ex: "todos os alunos com nota entre 7 e 9").

Dica: Pense nas operações dominantes de cada cenário.

Gabarito – Exercício Proposto 1

ArrayList – Custo de n remoções do início

Dada uma ArrayList com n elementos, remover todos do início, um por um.

Solução:

A cada remoção do início, todos os k elementos restantes são deslocados (shift):

- 1ª remoção: $n-1$ shifts. 2ª: $n-2$ shifts. ... n-ésima: 0 shifts.

$$\text{Total} = (n-1) + (n-2) + \ldots + 1 + 0 = \frac{n(n-1)}{2}$$

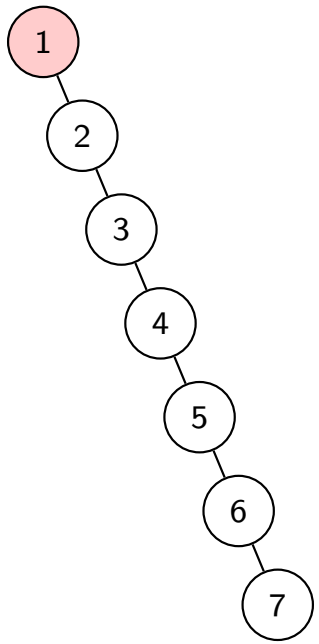
$$\boxed{\Theta(n^2)}$$

Gabarito – Exercício Proposto 2

BST – Pior caso de inserção

Inserir $[1, 2, 3, 4, 5, 6, 7]$ nesta ordem.

(a) Árvore resultante:



(b) Altura: $h = 6 = n - 1$.

(c) Buscar 7: percorrer 7 nós $\Rightarrow$ $\boxed{O(n)}$.

Gabarito – Exercício Proposto 3

Heap Máximo – Inserir $[10, 20, 15, 30, 25]$

Mostrar o estado após cada inserção.

1. Inserir 10: $[10]$ – 0 trocas.
2. Inserir 20: $[10, 20] \rightarrow 20$ ¿ pai(10)? Sim, troca: $[20, 10]$ – 1 troca.
3. Inserir 15: $[20, 10, 15]$ – 15 ¿ pai(20)? Não – 0 trocas.
4. Inserir 30: $[20, 10, 15, 30] \rightarrow 30$ ¿ pai(10)? Sim: $[20, 30, 15, 10] \rightarrow 30$ ¿ pai(20)? Sim: $[30, 20, 15, 10]$ – 2 trocas.
5. Inserir 25: $[30, 20, 15, 10, 25] \rightarrow 25$ ¿ pai(20)? Sim: $[30, 25, 15, 10, 20]$ – 1 troca.

Heap final: $[30, 25, 15, 10, 20]$. Total de trocas: $\boxed{4}$.

Gabarito – Exercício Proposto 4

Tabela Hash $m = 11$, $hash(k) = k \ \% \ 11$

Inserir $[22, 33, 44, 55, 66, 77, 88]$.

(a) Posições:

- $hash(22) = 22 \ \% \ 11 = 0$ $hash(33) = 33 \ \% \ 11 = 0$ $hash(44) = 44 \ \% \ 11 = 0$
- $hash(55) = 55 \ \% \ 11 = 0$ $hash(66) = 66 \ \% \ 11 = 0$ $hash(77) = 77 \ \% \ 11 = 0$
- $hash(88) = 88 \ \% \ 11 = 0$

(b) Todas as 7 chaves colidiram na posição 0. **6 colisões**.

(c) Fator de carga: $\alpha = n/m = 7/11 \approx 0.64$.

Observação

Todos os valores são múltiplos de 11, gerando o pior cenário possível. Escolher m primo que **não divide** os dados é crucial!

Gabarito – Exercício Proposto 5

Escolha da ED – Respostas

1. **Ctrl+Z / Ctrl+Y**: Duas **pilhas** – uma para desfazer (ações), outra para refazer.
2. **Impressão (ordem de chegada)**: **Fila (Queue)** – FIFO garante a ordem.
3. **Ranking top-10**: **Heap mínimo** de tamanho 10 – inserção e remoção do mínimo em $O(\log 10) = O(1)$.
4. **Verificar se variável foi declarada**: **Tabela Hash** – busca em $O(1)$ médio.
5. **Consultas por intervalo**: **BST balanceada** – percurso em-ordem permite extrair elementos no intervalo $[a, b]$ eficientemente em $O(\log n + k)$, onde k é o número de resultados.

Resumo da Aula 6

O que aprendemos

1. A **escolha da ED** afeta diretamente a complexidade dos algoritmos.
2. **ArrayList**: $O(1)$ para acesso, $O(n)$ para inserção/remoção arbitrária.
3. **LinkedList**: $O(1)$ nas pontas, $O(n)$ para acesso por índice.
4. **Pilha e Fila**: todas as operações em $O(1)$ (política restrita de acesso).
5. **BST**: $O(\log n)$ se balanceada, $O(n)$ se degenerada.
6. **Heap**: $O(\log n)$ para inserção e extração; $O(1)$ para consultar máx.
7. **Tabela Hash**: $O(1)$ no caso médio; $O(n)$ no pior caso.

Tabela Resumo Final

ED	Acesso	Busca	Inserção	Remoção
ArrayList	$O(1)$	$O(n)$	$O(1)$–$O(n)$	$O(1)$–$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(1)$ pontas	$O(1)$ pontas
Pilha	$O(1)$ topo	—	$O(1)$	$O(1)$
Fila	$O(1)$ head	—	$O(1)$	$O(1)$
BST (bal.)	—	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap	$O(1)$ max	—	$O(\log n)$	$O(\log n)$
Hash	—	$O(1)$ méd.	$O(1)$ méd.	$O(1)$ méd.

Mensagem Principal

Não existe ED "perfeita". Cada problema exige a análise cuidadosa das operações dominantes para escolher a estrutura mais adequada.

Próxima Aula

Aula 7 – Análise de Algoritmos Recursivos

- Relações de recorrência.
- Método da substituição.
- Método da árvore de recursão.
- Teorema Mestre.
- Exemplos: Fatorial, Fibonacci, MergeSort, QuickSort.

Dúvidas?

Referências


T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Algoritmos: Teoria e Prática*. 3ª ed, Elsevier, 2012. Capítulos 10, 11, 12 e 6.


J. A. B. Monteiro. *Notas de Aula – Estruturas de Dados*. Disponível em: joaoarthurbm.github.io/eda.


R. Sedgewick, K. Wayne. *Algorithms*. 4th ed, Addison-Wesley, 2011.


J. Kleinberg, É. Tardos. *Algorithm Design*. Pearson, 2005.