



# FUNDAMENTOS DE COMPUTAÇÃO E ALGORITMOS

---

Edson Ifarraguirre Moreno – Aula 03

# Professores

## MÁRIO SOUTO

Professor Convidado

Especialista em desenvolvimento web, começou a programar com 11 anos e não parou mais! Atualmente é Engenheiro de Software no Nubank, GitHub Star, Microsoft MVP, Alura Star e criador e diretor do Canal DevSoutinho no YouTube, onde lança vídeos semanalmente com foco no mundo da tecnologia e programação.

## EDSON IFARRAGUIRRE MORENO

Professor PUCRS

Edson Ifarraguirre Moreno é doutor em Ciência da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS). Atualmente é professor adjunto pela mesma PUCRS, estando vinculado a Escola Politécnica da pucrs, sendo responsável por disciplinas da área de hardware para os cursos de ciência da computação, engenharia da computação. Adicionalmente trabalha com a orientação de alunos no desenvolvimento de projetos do curso de Engenharia de Software. Desde 2016 coordena o laboratório iSeed Labs, uma parceria entre a academia e a iniciativa privada que tem por objetivo fomentar a inovação e o empreendedorismo. Os principais temas de pesquisa incluem: Sistemas multiprocessados em chip (em inglês, Multiprocessor System on chip, MPSoC), projeto em nível de sistema e redes em chip (em inglês, Network on chip, NoC).

# *Ementa da disciplina*

Entendimento de algoritmos e estruturas de dados (listas, filas, pilhas e árvores) na solução de problemas. Análise de algorítmica quanto a aplicação e complexidade. Análise da aplicabilidade e manejo de estruturas de dados lineares e hierárquicas.

# Pós graduação em Desenvolvimento Full Stack

---

Por Edson Moreno

# Fundamentos de Computação e Algoritmos

---

Desenvolvimento Full Stack

# Organização da aula

- Fundamentos de computação e Algoritmos
- Complexidade algorítmica
- Estrutura de dados padrão da linguagem
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Organização da aula

- Fundamentos de computação e Algoritmos
- Complexidade algorítmica
- Estrutura de dados padrão da linguagem
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Pensamento computacional

Problema → algoritmo → solução



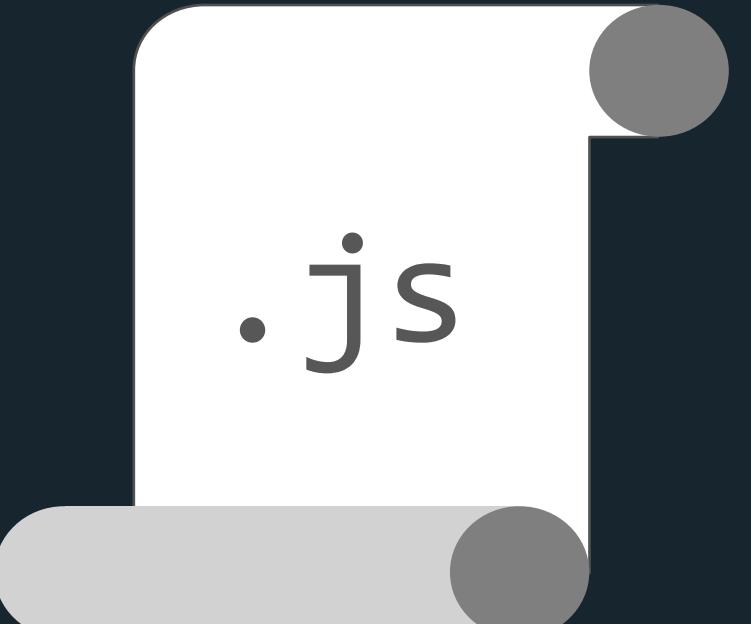
# Conceitos básicos da linguagem

- Configuração do ambiente
  - O que será usado hoje
    - Ambiente de edição dos arquivos javaScript
      - VSCode
        - <https://code.visualstudio.com/>
      - Node
        - <https://nodejs.org/en/download/>
    - Validando o ambiente
      - Criar um projeto hello world
        - `console.log("Hello world");`

The image shows two windows from the Visual Studio Code interface. The top window is the code editor with a dark theme, displaying a single file named 'index.js' containing the code `console.log("Hello world");`. The bottom window is a terminal window titled 'cmd.exe' showing the output of running the script: `Hello world`.

# Conceitos básicos da linguagem

- Noções básicas
  - O que é JS
    - Referência
      - [https://developer.mozilla.org/pt-BR/docs/Learn/Getting started with the web/JavaScript basics](https://developer.mozilla.org/pt-BR/docs/Learn/Getting_started_with_the_web/JavaScript_basics)
    - Linguagem baseada em objetos
    - Dinâmica
    - Fracamente tipada
    - Usualmente interpretada por navegadores
  - Organização
    - Pode-se
      - Criar scripts que manipulam o HTML e CSS (frontend)
      - Criar scripts para serem executados no server side (backend)
    - A extensão do arquivo de script é usualmente .js



.js

# Conceitos básicos da linguagem

- Como se declara uma variável/constante
  - let: Define uma variável mutável dentro de um escopo
  - const: define uma variável IMUTÁVEL dentro de um contexto
  - var: define uma variável mutável de escopo global

```
1
2  {
3      let f_name = 'Alex';
4      const ZIP = 500067;
5      var age = 25;
6  }
7
8  console.log(f_name); // Uncaught ReferenceError: f_name is not defined
9  console.log(ZIP);   // Uncaught ReferenceError: ZIP is not defined
10 console.log(age);  // 25
```

# Conceitos básicos da linguagem

- Comandos de repetição

```
for ([inicialização]; [condição]; [expressão final])  
    declaração
```

```
for (var i = 0; i < 9; i++) {  
    console.log(i);  
    // more statements  
}
```

```
while (condição) {  
    rotina  
}
```

```
var n = 0;  
var x = 0;  
  
while (n < 3) {  
    n++;  
    x += n;  
}
```

```
do  
    statement  
while (condition);
```

```
var result = '';  
var i = 0;  
do {  
    i += 1;  
    result += i + ' ';  
}  
while (i > 0 && i < 5);
```

# Conceitos básicos da linguagem

- Comandos de seleção

```
if (condition) {  
    statement1  
} else {  
    statement2  
}
```

```
if (condition1)  
    statement1  
else if (condition2)  
    statement2  
else if (condition3)  
    statement3  
...  
else  
    statementN
```

```
if (condition1)  
    statement1  
else  
    if (condition2)  
        statement2  
    else  
        if (condition3)  
            ...
```

```
var foo = 0;  
switch (foo) {  
    case -1:  
        console.log('negative 1');  
        break;  
    case 0: // foo is 0 so criteria met here so this block will run  
        console.log(0);  
        // NOTE: the forgotten break would have been here  
    case 1: // no break statement in 'case 0:' so this case will run as well  
        console.log(1);  
        break; // it encounters this break so will not continue into 'case 2:'  
    case 2:  
        console.log(2);  
        break;  
    default:  
        console.log('default');  
}
```

```
condition ? exprIfTrue : exprIfFalse  
var age = 26;  
var beverage = (age >= 21) ? "Beer" : "Juice";  
console.log(beverage); // "Beer"
```

# Conceitos básicos da linguagem

- Criação de funções

```
function name([param[, param, [..., param]]]) {  
    [statements]  
}
```

```
hoisted(); // logs "foo"  
  
function hoisted() {  
    console.log("foo");  
}
```

```
function calc_sales(units_a, units_b, units_c) {  
    return units_a * 79 + units_b * 129 + units_c * 699;  
}
```

# Conceitos básicos da linguagem

- Operadores padrão
  - Operadores aritméticos (+, -, \*, /, %)
    - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#operadores\\_aritm%C3%A9ticos](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#operadores_aritm%C3%A9ticos)
  - Operadores de incremento e decremento (++, --)
    - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#incremento\\_e\\_decremento](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#incremento_e_decremento)
  - Operadores relacionais (==, !=, <, <=, >, >=)
    - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#operadores\\_de\\_igualdade](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#operadores_de_igualdade)
    - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#operadores\\_relacionais](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#operadores_relacionais)
  - Operadores lógicos (&&, ||)
    - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#assignment\\_operators](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators#assignment_operators)

# Conceitos básicos da linguagem

- Código integrando os conceitos apresentados
  - Declaração de variável
  - Uso de comando de seleção
  - Uso de comando de repetição
  - Uso de função

```
1  function integrador(){
2      for(let i=0; i<10; i++){
3          if(i%2==0)
4              console.log(i+": é par")
5          else
6              console.log(i+": é ímpar")
7      }
8  }
9
10 integrador()
```

# Organização da aula

- Fundamentos de computação e Algoritmos
- Complexidade algorítmica
- Estrutura de dados padrão da linguagem
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Análise algorítmica

- Como saber se um algoritmo é eficiente?
- Características de desempenho a serem avaliadas
  - O espaço ocupado é uma característica de desempenho
  - O tempo gasto na execução é outra característica de desempenho
- Complexidade de um algoritmo é a medida do consumo de recursos
  - Tempo de processamento;
  - Memória ocupada;
  - Largura de banda de comunicação;
  - Hardware necessário;
  - etc.

# Avaliação de desempenho

- Contagem de tempo
  - Não é uma boa prática avaliar baseado em tempo de processamento
    - Depende de fatores tais como hardware, software, tamanho e tipo da entrada de dados

1	function avaliaTempo(intervalo){	0;30
2	for(i=0;i<intervalo; i++);	100;32
3	}	200;7
4		300;9
5	var entrySize=1000	400;16
6		500;13
7	for(let size=0; size<entrySize; size+=100){	600;15
8	var initialMoment = performance.now()	700;16
9	avaliaTempo(size)	800;18
10	var finalMoment = performance.now()	900;16
11		
12		
13	console.log(size+";"+Math.trunc((finalMoment-initialMoment)*1000))	
14	//console.log(size+";"+finalMoment-finalMoment)	
15	}	

# Avaliação de desempenho

- Contagem de Operações
  - Regra de contagem
    - Consiste em contar quantas operações primitivas são executadas
    - Operação primitiva
      - Instrução de baixo nível com um tempo de execução constante
    - Considera-se tempos similar mesmos para operações primitivas diferentes
  - Operações primitivas
    - Atribuição de valores a variáveis
    - Chamadas de métodos
    - Operações aritméticas (por exemplo, adição de dois números)
    - Comparação de dois números
    - Acesso a um arranjo
    - Retorno de um método

# Avaliação de desempenho

- Contagem de operações

```
1 /**
2  * Get the smallest number on an array of numbers
3  * @param {Array} array array of numbers
4  * @example
5  *   getMin([3, 2, 9]) => 2
6 */
7 function getMin(array) {
8  let min; ← 1 Operation
9  for (let index = 0; index < array.length; index++) { ← 1 Loop size "array.length" or "n"
10    const element = array[index]; ← 1 Operation
11    if (min === undefined || element < min) { ← 1 Operation
12      min = element; ← 1 Operation (worst case)
13    }
14  }
15  return min; ← 1 Operation
16 }
```

1 Loop size "array.length" or "n"

1 Operation

1 Operation

1 Operation

3 Operations \* n

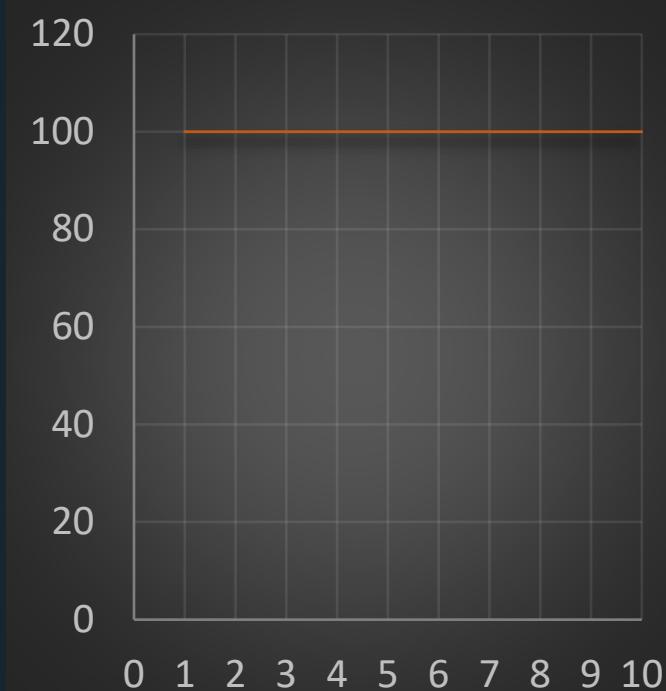
1 Operation

$$3(n) + 3$$

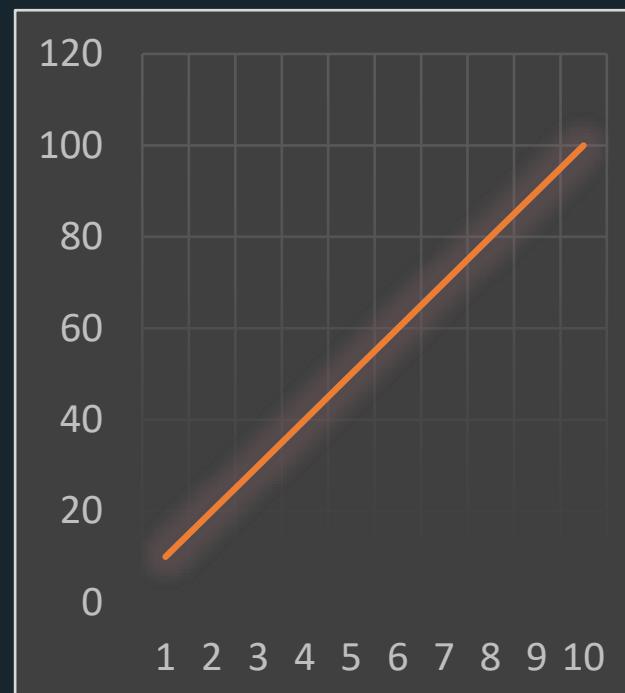
# Exemplo de custo computacional

- Exemplos de complexidade

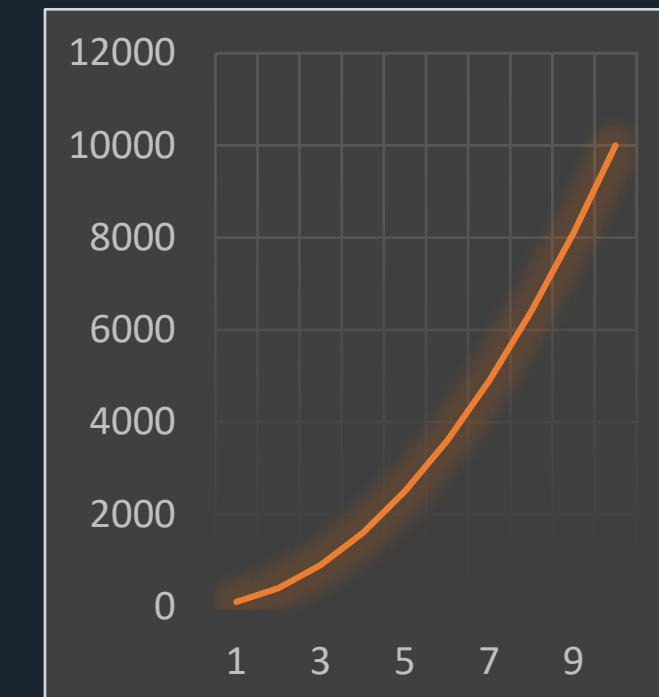
```
1 function Constante(valor){  
2     return ++valor  
3 }  
4  
5 console.log(Constante(10))
```



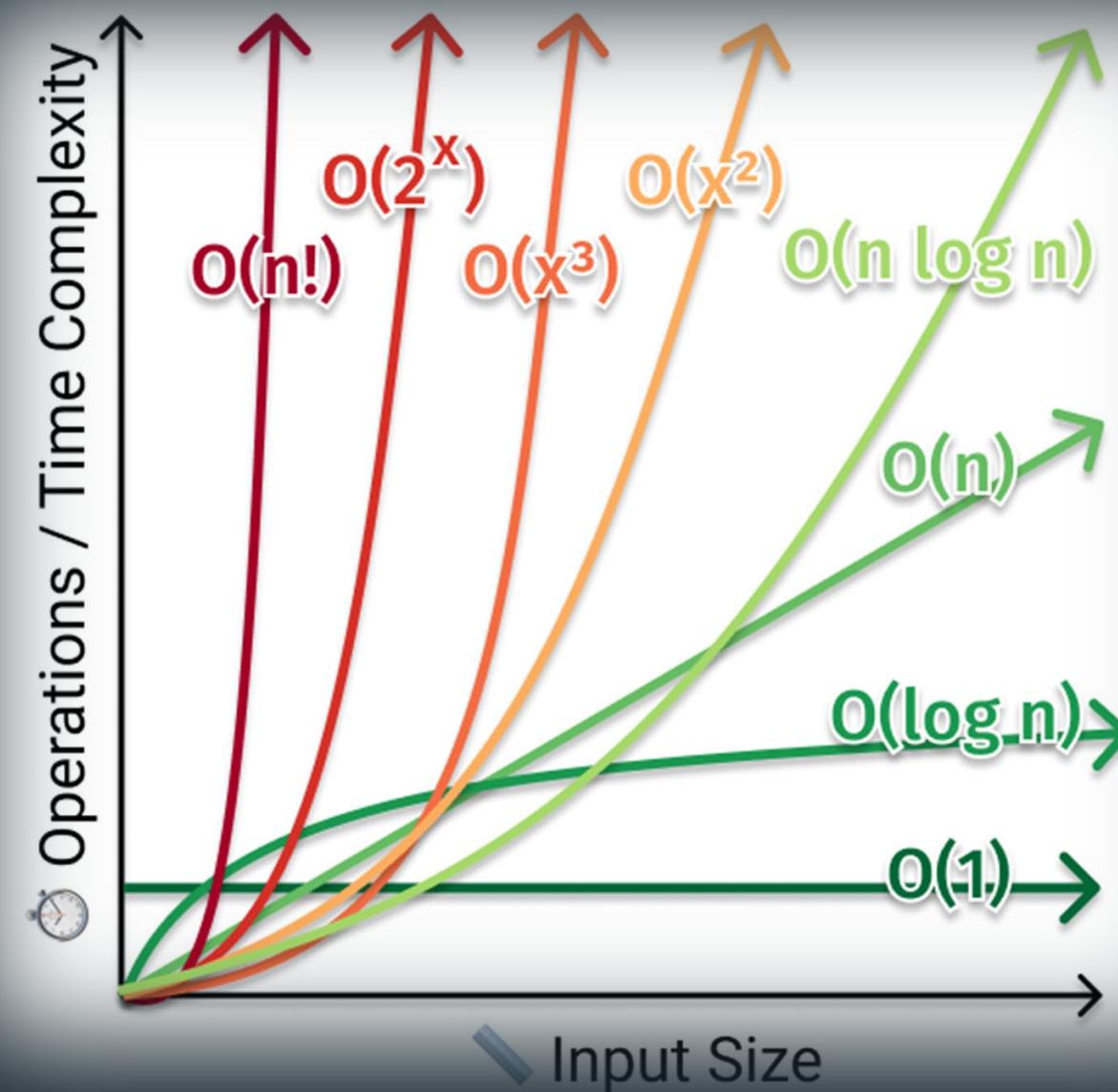
```
1 function somatorio(valor){  
2     var variavel=0;  
3     for(var i=0; i<valor; i++)  
4         variavel+=i  
5     return variavel  
6 }  
7  
8 console.log(somatorio(10))
```



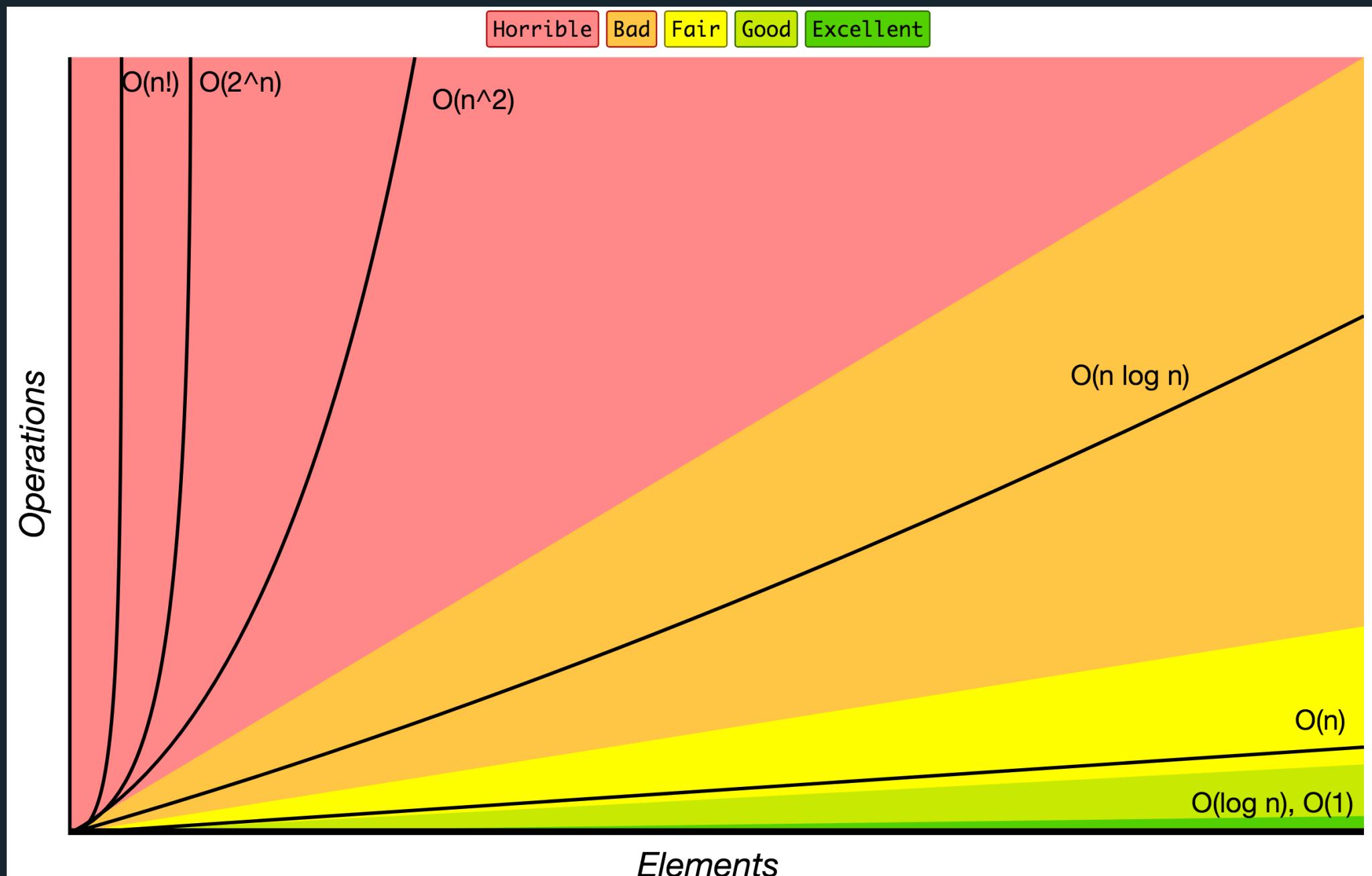
```
1 function busca(valor){  
2     var variavel=0;  
3     for(var x=0; x<valor; x++)  
4         for(var y=0; y<valor; y++)  
5             variavel+=x+y  
6     return variavel  
7 }  
8  
9 console.log(busca(10))
```



# Comparação de casos



# Comparação de casos



- <https://www.bigocheatsheet.com/>

# Organização da aula

- Fundamentos de computação e Algoritmos
- Complexidade algorítmica
- Estrutura de dados padrão da linguagem
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Vetor (Array)

- Definição
  - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_ Objects/Array](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array)
  - Estrutura de dados simples
    - Armazena dados de forma contígua na memória
  - JS tem suporte nativo para esta estrutura de dados
    - Há métodos definidos para manipulação
- Motivação para o uso
  - Evita a criação de múltiplas variáveis para o mesmo fim
  - Facilita “percorrer” os dados armazenados

# Vector (Array)

- Formas de declaração de um array

```
1 let daysOfWeek = [] // {1}
2 console.log(daysOfWeek)
3
4
5 daysOfWeek = [ 'Sunday', 'Monday',
6 | 'Tuesday', 'Wednesday',
7 | 'Thursday', 'Friday',
8 | 'Saturday' ]; // {2}
9 console.log(daysOfWeek)
```

```
let daysOfWeek = new Array(); // {1}
console.log(daysOfWeek)

daysOfWeek = new Array(7); // {2}
console.log(daysOfWeek)

daysOfWeek = new Array
('Sunday', 'Monday',
 'Tuesday', 'Wednesday',
 'Thursday', 'Friday',
 'Saturday'); // {3}
console.log(daysOfWeek)
```

# Vetor (Array)

- Exemplo de operação com emprego de array

```
1  const fibonacci = []; // {1}
2
3  fibonacci[1] = 1; // {2}
4  fibonacci[2] = 1; // {3}
5
6  for (let i = 3; i < 20; i++)
7    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2]; // // {4}
8
9  for (let i = 1; i < fibonacci.length; i++) // {5}
10   console.log(fibonacci[i]); // {6}
11
```

# Vetor (Array)

- Métodos padrão para manipulação de array
  - length: retorna o número de elementos na lista
  - push: Adiciona elemento na última posição do array
  - pop: remove o último elemento da lista
  - shift: remove o primeiro elemento da lista
  - unshift: adiciona elemento na primeira posição da lista
  - splice: adiciona valor em posição específica, permitindo remover outros
  - slice: retorna uma cópia do array
  - at(pos) ou [pos]: retorna elemento de posição especificada por pos
  - ETC...

# Array Bidimensional / Matriz

- Exemplo de declaração

```
1 var averageTemp = []
2 averageTemp[0] = [72, 75, 79, 79, 81, 81]
3 averageTemp[1] = [81, 79, 75, 75, 73, 73]
4 console.log(averageTemp)
```

The diagram illustrates the declaration of a 2D array and its corresponding matrix representation. On the left, a code snippet shows the declaration of a variable `averageTemp` as an empty array, followed by two assignments where the array is populated with temperature data for two different days. A large yellow arrow points from this code to a 2D matrix on the right. The matrix has two rows, each representing a day. The first row, labeled [0], contains temperatures [72, 75, 79, 79, 81, 81]. The second row, labeled [1], contains temperatures [81, 79, 75, 75, 73, 73]. The columns are indexed from [0] to [5].

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	72	75	79	79	81	81
[1]	81	79	75	75	73	73

```
1 var averageTemp = [];
2 console.log(averageTemp)
3
4 averageTemp[0] = [];
5 console.log(averageTemp)
6
7 averageTemp[0][0] = 72;
8 averageTemp[0][1] = 75;
9 averageTemp[0][2] = 79;
10 averageTemp[0][3] = 79;
11 averageTemp[0][4] = 81;
12 averageTemp[0][5] = 81;
13 console.log(averageTemp)
14
15 averageTemp[1] = [];
16 console.log(averageTemp)
17
18 averageTemp[1][0] = 81;
19 averageTemp[1][1] = 79;
20 averageTemp[1][2] = 75;
21 averageTemp[1][3] = 75;
22 averageTemp[1][4] = 73;
23 averageTemp[1][5] = 73;
24 console.log(averageTemp)
```

# Tipos derivados de Array

- Características
  - Explora estrutura de dados de Array
  - Utiliza métodos específico do Array que garantem comportamento adequado
- Pilha
  - Último dado a dar entrada é o primeiro a sair (LIFO)
    - push
    - pop
- Fila
  - Primeiro dado a dar entrada é o primeiro a sair (FIFO)
    - push
    - shift

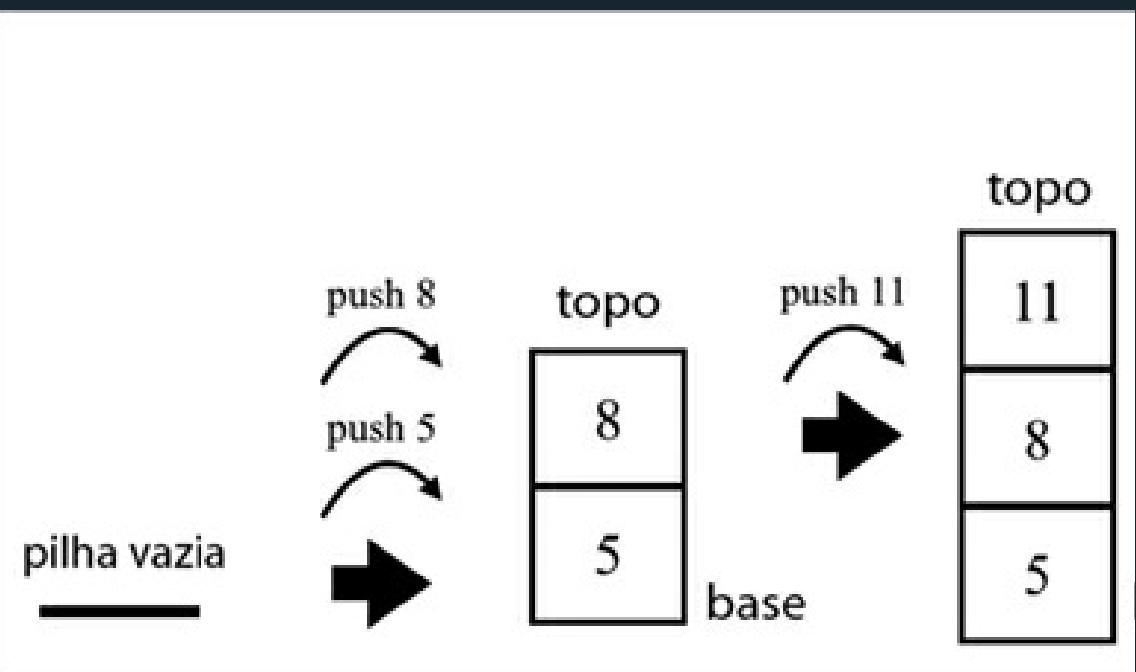


# Tipos derivados de Array

- Pilha

```
1 class Stack {  
2     constructor() {  
3         this.items = []; // {1}  
4     }  
5     push(element) {  
6         this.items.push(element);  
7     }  
8     pop() {  
9         return this.items.pop();  
10    }  
11    peek() {  
12        return this.items[this.items.length - 1];  
13    }  
14    isEmpty() {  
15        return this.items.length === 0;  
16    }  
17    clear() {  
18        this.items = [];  
19    }  
20    size(){  
21        return this.items.length  
22    }  
23 }
```

```
1 var myStack = new Stack()  
2  
3 myStack.push(5);  
4 console.log(myStack.peek())  
5  
6 myStack.push(8);  
7 console.log(myStack.size())  
8  
9 myStack.push(11);  
10 console.log(myStack.isEmpty())  
11  
12 myStack.pop()  
13 console.log(myStack.size())  
14  
15 myStack.clear()  
16 console.log(myStack.isEmpty())
```



# Tipos derivados de Array

- Fila

```
1  class Queue {  
2      constructor() {  
3          this.items = [];  
4      }  
5      enqueue(element) {  
6          this.items.push(element);  
7      }  
8      dequeue() {  
9          return this.items.shift();  
10     }  
11     peek() {  
12         if(this.items.length==0)  
13             return undefined  
14         else  
15             return this.items[0];  
16     }  
17     isEmpty() {  
18         return this.items.length === 0;  
19     }  
20     clear() {  
21         this.items = [];  
22     }  
23     size(){  
24         return this.items.length  
25     }  
26 }
```

```
1  var myQueue = new Queue()  
2  
3  myQueue.enqueue(5);  
4  console.log(myQueue.peek())  
5  
6  myQueue.enqueue(8);  
7  console.log(myQueue.size())  
8  
9  myQueue.enqueue(11);  
10 console.log(myQueue.isEmpty())  
11  
12 myQueue.dequeue()  
13 console.log(myQueue.size())  
14  
15 myQueue.clear()  
16 console.log(myQueue.isEmpty())
```

# Conjuntos (set)

- Definição
  - [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_ Objects/Set](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Set)
  - Estrutura de dados simples como Array
    - Também armazena dados de forma contígua na memória
  - Não permite que os dados armazenados estejam duplicados

```
1  var mySet = new Set()
2  mySet.add(0)
3  console.log(mySet)
4  mySet.add(1)
5  console.log(mySet)
6  mySet.add(1)
7  console.log(mySet)
8  mySet.add(2)
9  console.log(mySet)
10 mySet.add(2)
11 console.log(mySet)
```

# Conjuntos (set)

- Disponibiliza operações básicas de conjuntos

```
var meuSet = new Set();

meuSet.add(1); // meuSet [1]
meuSet.add(5); // meuSet [1, 5]
meuSet.add(5); // 5 já foi adicionando, portanto, meuSet [1, 5]
meuSet.add("texto");
var o = {a: 1, b: 2};
meuSet.add(o);

meuSet.add({a: 1, b: 2}); // o está referenciando outro objeto

meuSet.has(1); // true
meuSet.has(3); // false, 3 não foi adicionado ao set (Conjunto)
meuSet.has(5); // true
meuSet.has(Math.sqrt(25)); // true
meuSet.has("Texto".toLowerCase()); // true
meuSet.has(o); // true

meuSet.size; // 5

meuSet.delete(5); // remove 5 do set
meuSet.has(5); // false, 5 já foi removido

meuSet.size; // 4, nós simplesmente removemos um valor

console.log(meuSet) // Set { 1, 'texto', { a: 1, b: 2 }, { a: 1, b: 2 } }
```

# Conjuntos (set)

- Forma de interação

```
1  var meuSet = new Set();
2
3  meuSet.add(1);
4  meuSet.add(5);
5  meuSet.add(5);
6  meuSet.add("texto");
7  var o = {a: 1, b: 2};
8  meuSet.add(o);
9
10 for (let item of meuSet) console.log(item);
```

# Conjunto (set)

- Remoção de valores duplicados em um array

```
// Use para remover elementos duplicados de um Array

const numeros = [2,3,4,4,2,3,3,4,4,5,5,5,6,6,7,5,32,3,4,5]

console.log([...new Set(numeros)]) // [2, 3, 4, 5, 6, 7, 32]
```

# Dicionários (Maps)

- Definição

- [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_ Objects/Map](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Map)
- Estrutura de dados simples como Array
  - Também armazena dados de forma contígua na memória
- Diferente de array, armazena a informação a partir de um par
  - Chave + valor
  - Chaves
    - Não se repetem
  - Valores
    - Podem ser duplicados
    - Aceita qualquer tipo de dado
  - Requer uso dos métodos
    - Interage com a estrutura

```
const contacts = new Map()
contacts.set('Jessie', {phone: "213-555-1234", address: "123 N 1st Ave"})
contacts.has('Jessie') // true
contacts.get('Hilary') // undefined
contacts.set('Hilary', {phone: "617-555-4321", address: "321 S 2nd St"})
contacts.get('Jessie') // {phone: "213-555-1234", address: "123 N 1st Ave"}
contacts.delete('Raymond') // false
contacts.delete('Jessie') // true
console.log(contacts.size) // 1
```

# Dicionários (Maps)

- Forma de iteração

```
const myMap = new Map()
myMap.set(0, 'zero')
myMap.set(1, 'one')

for (const [key, value] of myMap) {
  console.log(key + ' = ' + value)
}
// 0 = zero
// 1 = um

for (const key of myMap.keys()) {
  console.log(key)
}
// 0
// 1
```

```
for (const value of myMap.values()) {
  console.log(value)
}
// zero
// one

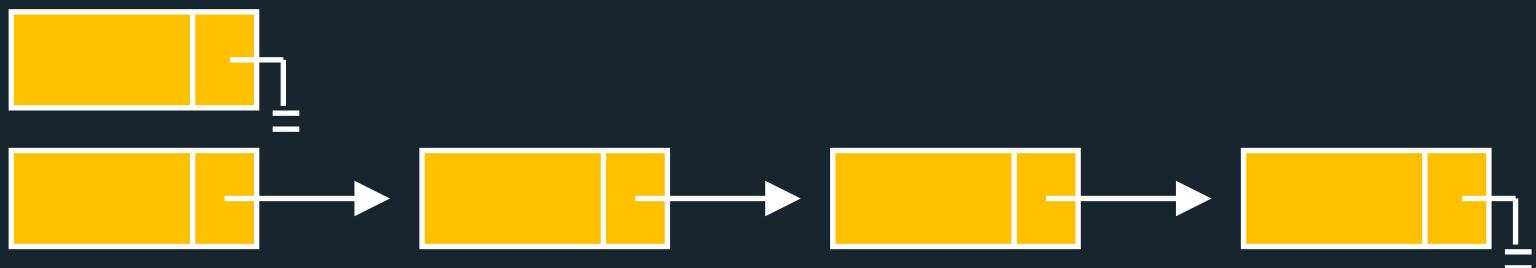
for (const [key, value] of myMap.entries())
  console.log(key + ' = ' + value)
}
// 0 = zero
// 1 = one
```

# Organização da aula

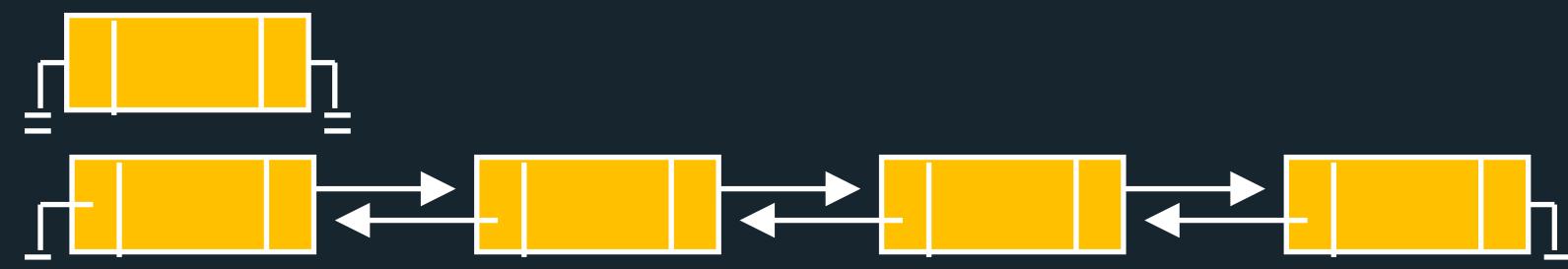
- Fundamentos de computação e Algoritmos
- Complexidade algorítmica
- Estrutura de dados padrão da linguagem
- **Listas encadeadas como estrutura de dados**
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Lista encadeada

- Estrutura encadeada
  - Consiste de um determinado número de nodos, cada um com uma referência para o próximo
  - Duas representações usuais
    - Encadeamento simples

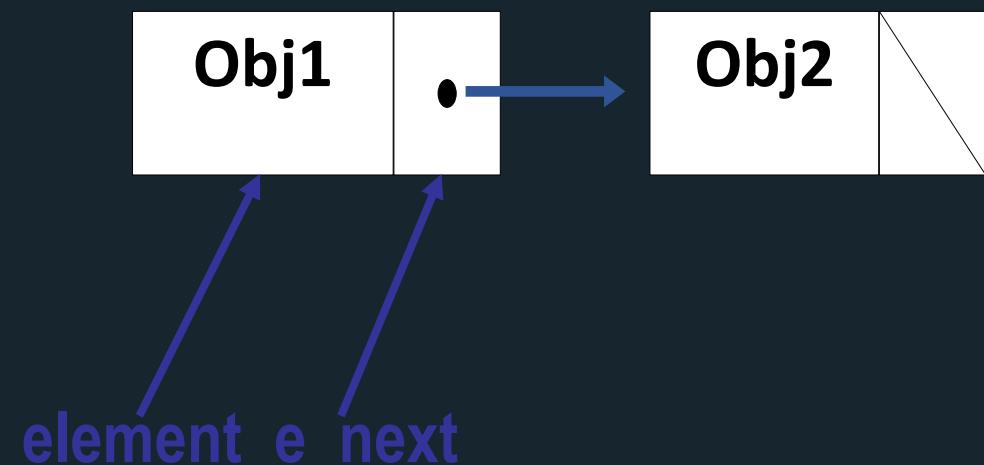


- Encadeamento duplo



# Lista encadeada

- Nodo
  - Nodos são conectados (encadeados) por links
  - Normalmente estes nodos tem dois atributos
    - element: representando o elemento armazenado no nodo
    - next: representando o próximo nodo no encadeamento da lista (contém referência para um objeto da mesma classe)



# Lista encadeada

- Implementação de um nodo
  - Classe Node (auto-referenciada) + Estrutura lista encadeada

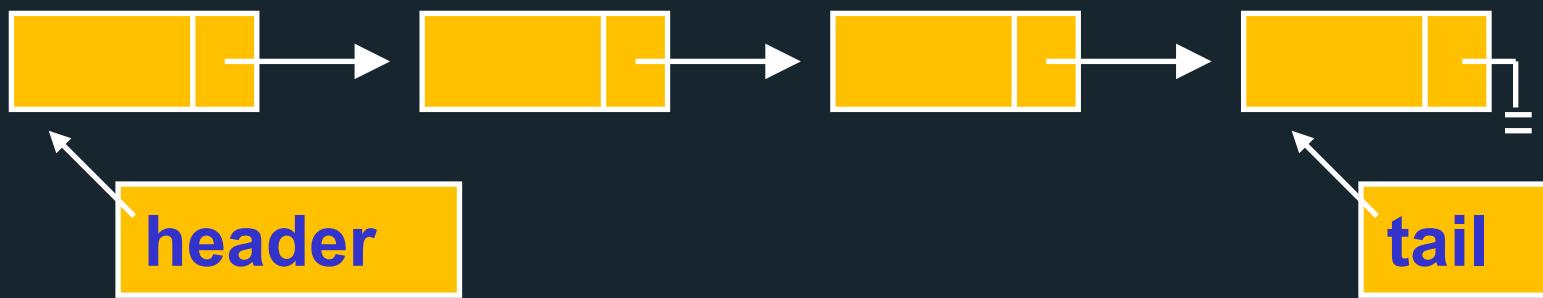
```
1 class Node {  
2     constructor(element) {  
3         this.element = element;  
4         this.next = null;  
5     }  
6 }
```

```
8 class LinkedList{  
9     constructor(){  
10        this.count=0  
11        this.header=null  
12        this.tail=null  
13    }  
14    add(element){  
15        if(this.count==0)  
16            this.header=this.tail;element  
17        else  
18            this.tail.next=element  
19            this.tail=element  
20            this.count++  
21    }  
22    print(){  
23        var aux = this.header  
24        while(aux!=null){  
25            console.log(aux.element)  
26            aux=aux.next  
27        }  
28    }  
29 }
```

```
31 var myLL = new LinkedList()  
32 |  
33 myLL.add(new Node(1))  
34 myLL.print()  
35  
36 myLL.add(new Node(2))  
37 myLL.print()  
38  
39 myLL.add(new Node(4))  
40 myLL.print()  
41  
42 myLL.add(new Node(3))  
43 myLL.print()
```

# Lista encadeada

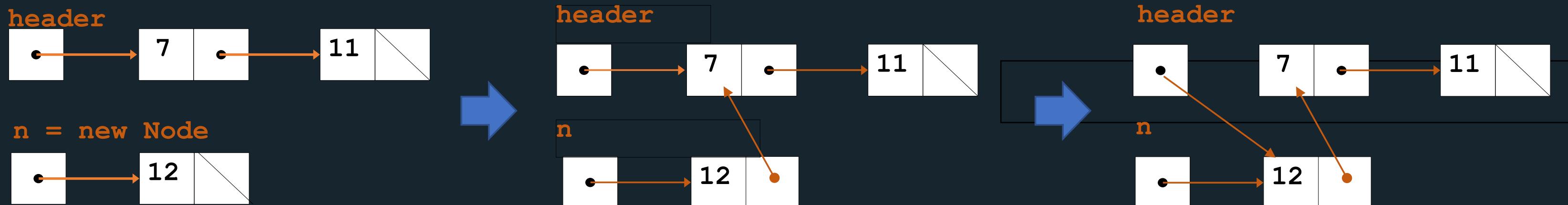
- Listas usando estruturas encadeadas:
  - Acesso direto ao primeiro elemento (*header*) é obrigatório
  - Acesso direto ao último elemento (*tail*) é desejável



- O uso somente da referência *header* é ineficiente para inserção de elementos no final da lista

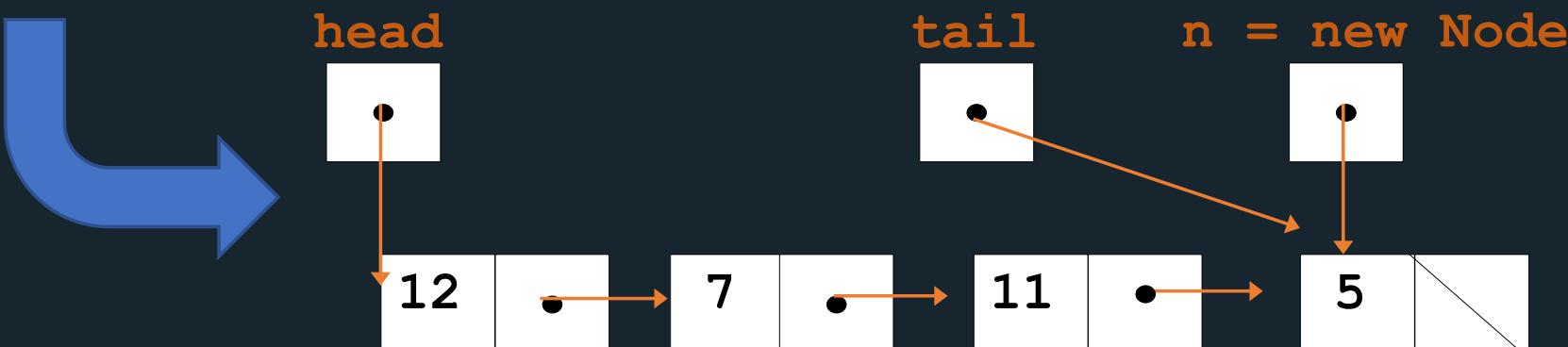
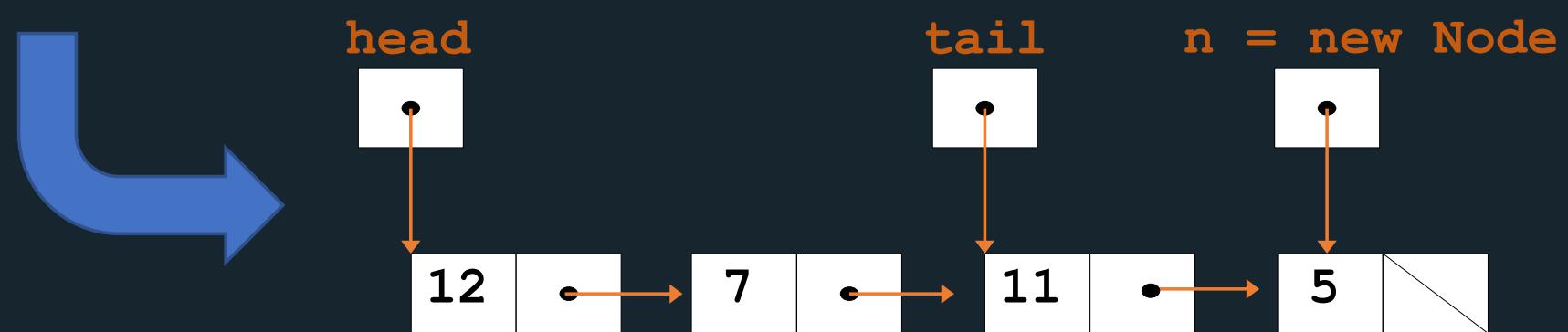
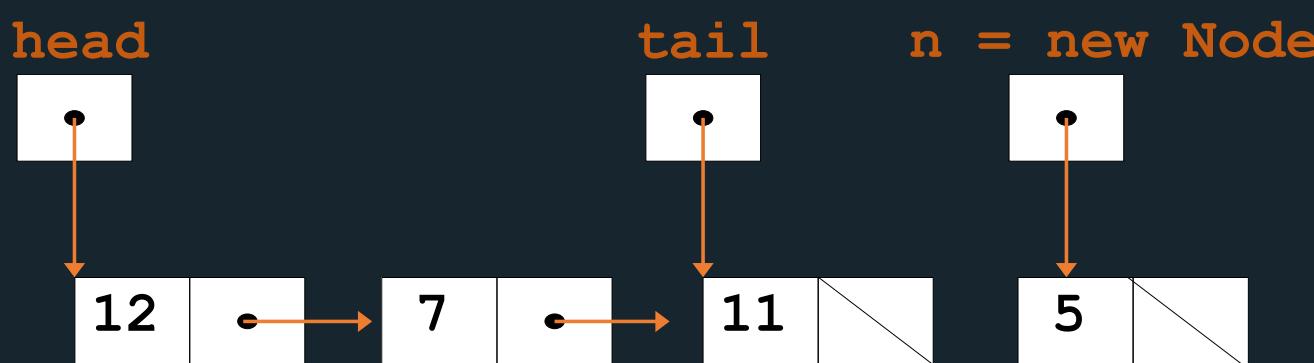
# Lista encadeada

- Passos para inserção de elementos em listas encadeadas:
  - Alocação de um novo nodo
  - Inserção das informações no nodo alocado
  - Inserção do nodo na lista com consequente encadeamento nos nodos já existentes
- Exemplo de inserção



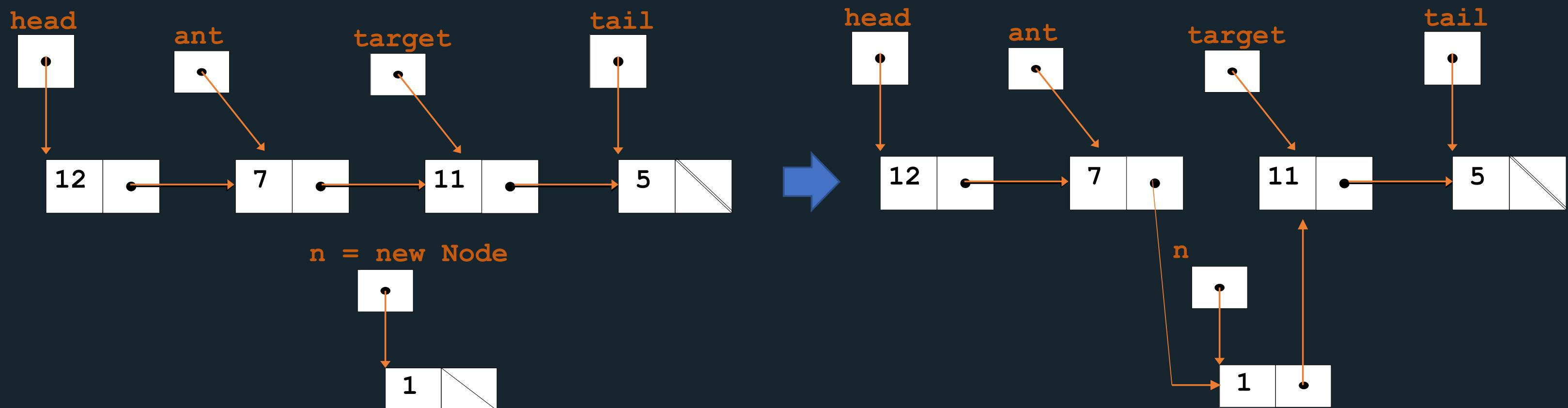
# Lista encadeada

- Inserção no final da lista



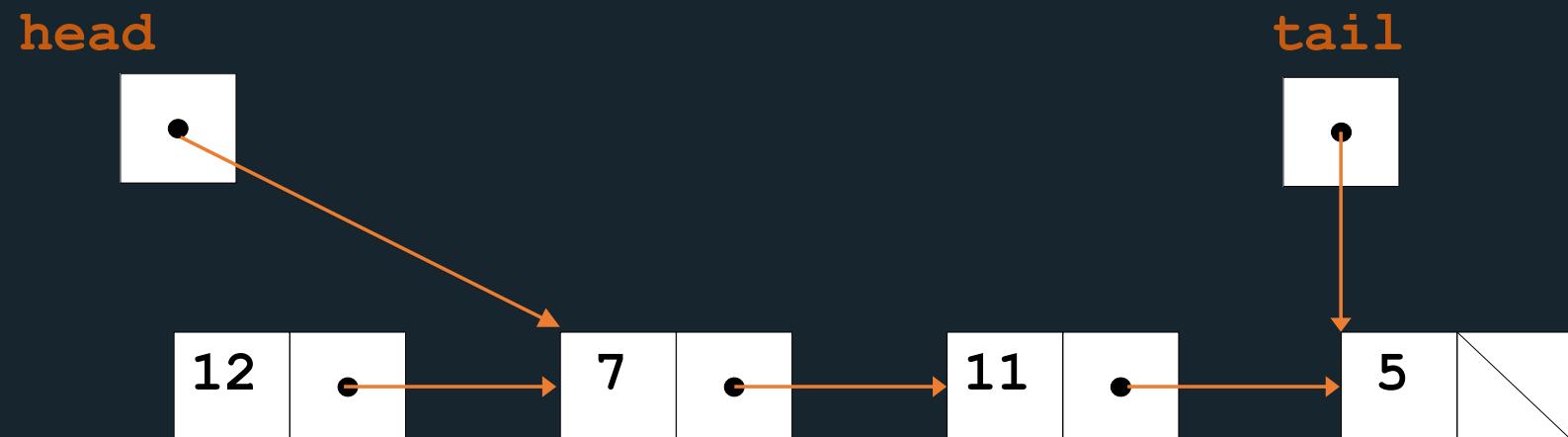
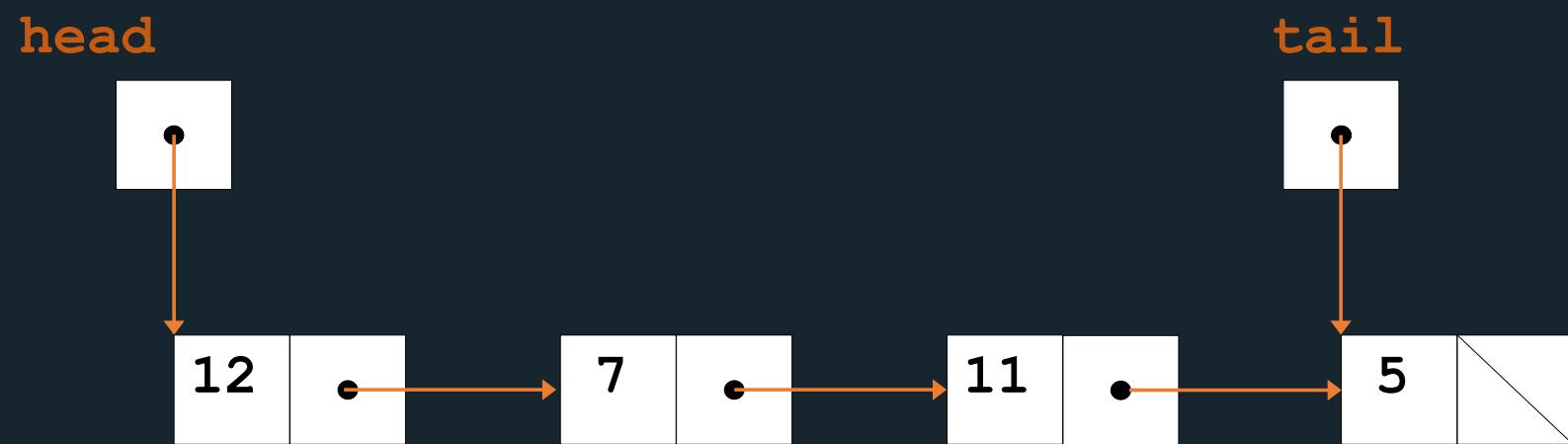
# Lista encadeada

- Inserção no meio da lista



# Lista encadeada

- Remoção do primeiro elemento da lista



# Lista encadeada

- Os seguintes métodos devem ser suportados:
  - add(e): insere um elemento no final da lista
  - add(index, e): insere um elemento em determinada posição (index) da lista
  - get(index)/set(index, e): get/set o elemento na posição index
  - remove(e): remove o elemento da lista
  - isEmpty(): retorna true se a lista está vazia
  - size(): retorna o número de elementos armazenados na lista
  - contains(e): retorna true se a lista contém o elemento
  - indexOf(e): retorna a posição onde o elemento está na lista
  - clear(): remove todos os elementos da lista

# Lista encadeada

- Exemplo de código

- <https://github.com/loiane/javascript-datastructures-algorithms/blob/main/src/js/data-structures/linked-list.js>

The screenshot shows Loiane Groner's GitHub profile. Her bio includes a photo of her, her name, and her GitHub handle (@loiane). She is a Software Engineer at Citi and a published author. Her GitHub stats show 13.2k followers and 129 following. Her pinned repositories are `javascript-datastructures-algorithms` and `curso-java-basico`.

**Overview** tab selected.

**Technologies** section:

- Java, Spring, JavaScript, Nodejs, HTML5, CSS3, Sass, Bootstrap, TypeScript, Angular, Sencha
- Ionic, NestJS, SQL Server, MongoDB, MySQL, Docker, Microsoft Azure, Google Cloud, Firebase
- Oracle Cloud, Git, GitHub, BitBucket, JIRA, JFrog, IntelliJ IDEA, Eclipse, VSCode

**Pinned** section:

- `javascript-datastructures-algorithms` (Public) - collection of JavaScript and TypeScript data structures and algorithms for education purposes. Source code bundle of JavaScript algorithms and data.
- `curso-java-basico` (Public) - [PT-BR] Código fonte apresentado no curso de Java gratuito do blog Loiane.com [FENI] Source code of my free Java training.

# Organização da aula

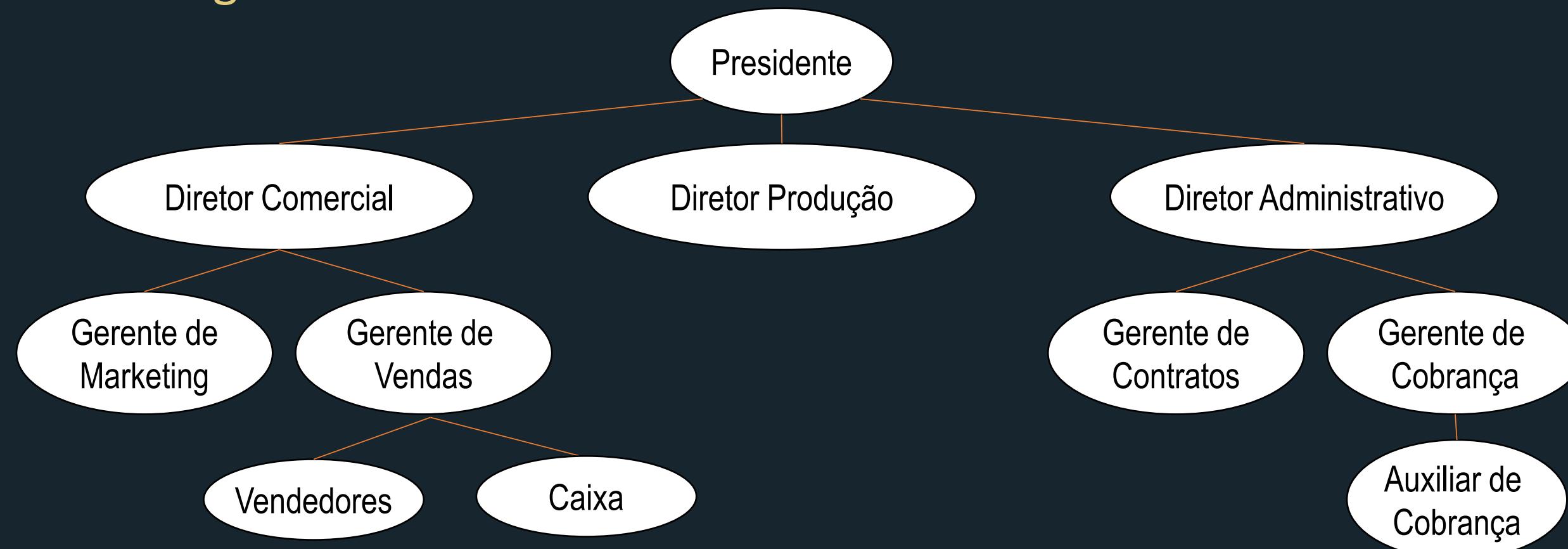
- Fundamentos de computação e Algoritmos
- Estrutura de dados padrão da linguagem
- Complexidade algorítmica
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Árvores

- Características
  - Estruturas de dados não lineares
  - Permitem a implementação de vários algoritmos mais rápidos do que no uso de estruturas de dados lineares como as listas
  - Fornecem uma forma natural de organizar os dados
    - Sistemas de arquivos
    - Bancos de dados
    - Sites da Web

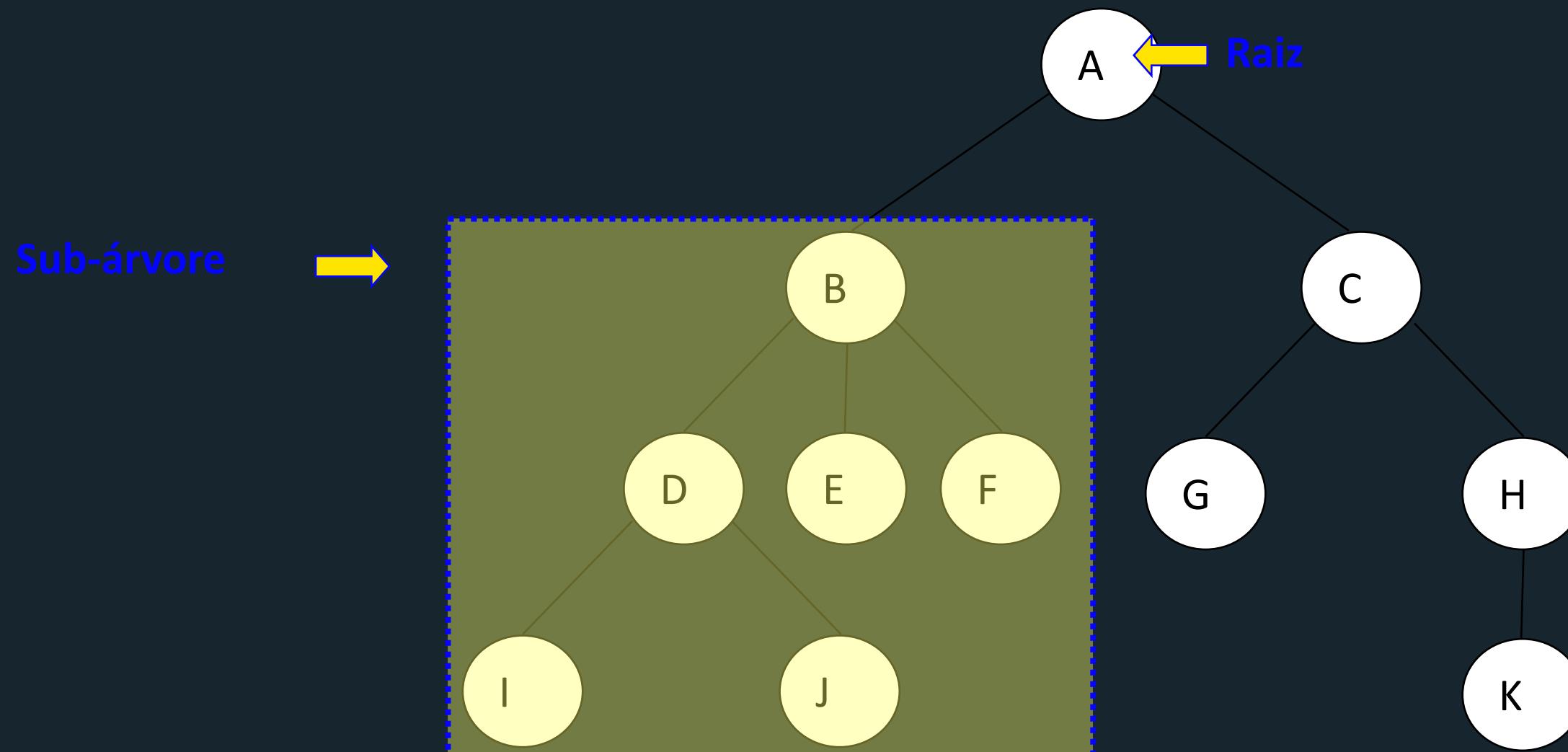
# Árvores

- Conceitos e terminologia
  - Tipo abstrato de dados que armazena elementos de maneira hierárquica
  - Normalmente, são desenhadas colocando-se os elementos dentro de elipses ou retângulos e conectando-os com linhas retas



# Árvore

- Conceitos e terminologias
  - Uma árvore pode ser representada da seguinte forma:



# Árvores

- Outra propriedade de uma árvore T:
  - Cada nodo v de T diferente da raiz tem um único nodo pai, w
  - Todo nodo com pai w é filho de w
- Pela definição
  - Uma árvore pode ser vazia, isto é, não possui nodos
  - Esta convenção permite que se defina uma árvore **recursivamente**
  - Uma árvore T ou está vazia, ou consiste de um nodo r, chamado de raiz de T, e um conjunto de árvores cujas raízes são filhas de r

# Árvores

- Conceitos e terminologias
- Outros relacionamentos entre nodos
  - Dois nodos que são filhos de um mesmo pai são **irmãos**
  - Um nodo **v** é **externo** se **v** não tem filhos
  - Um nodo **v** é **interno** se tem um ou mais filhos
- Nodo **interno** também é conhecido como **galho**
- Nodo externo também é conhecido como **folha**

# Recursão

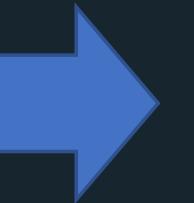


- “Para entender a recursão, é preciso entender antes a recursão.”
- Recursão
  - Método para resolução de problemas
  - Consiste em solucionar partes menores do mesmo problema
  - Finaliza quando o problema original foi resolvido
- Recursão é alcançada chamando-se a própria função.
- Um método ou função será recursivo se ele puder chamar a si mesmo diretamente

# Recursão

- Exemplo básico de recursão
  - Cálculo de fatorial pode ser calculado por recursão

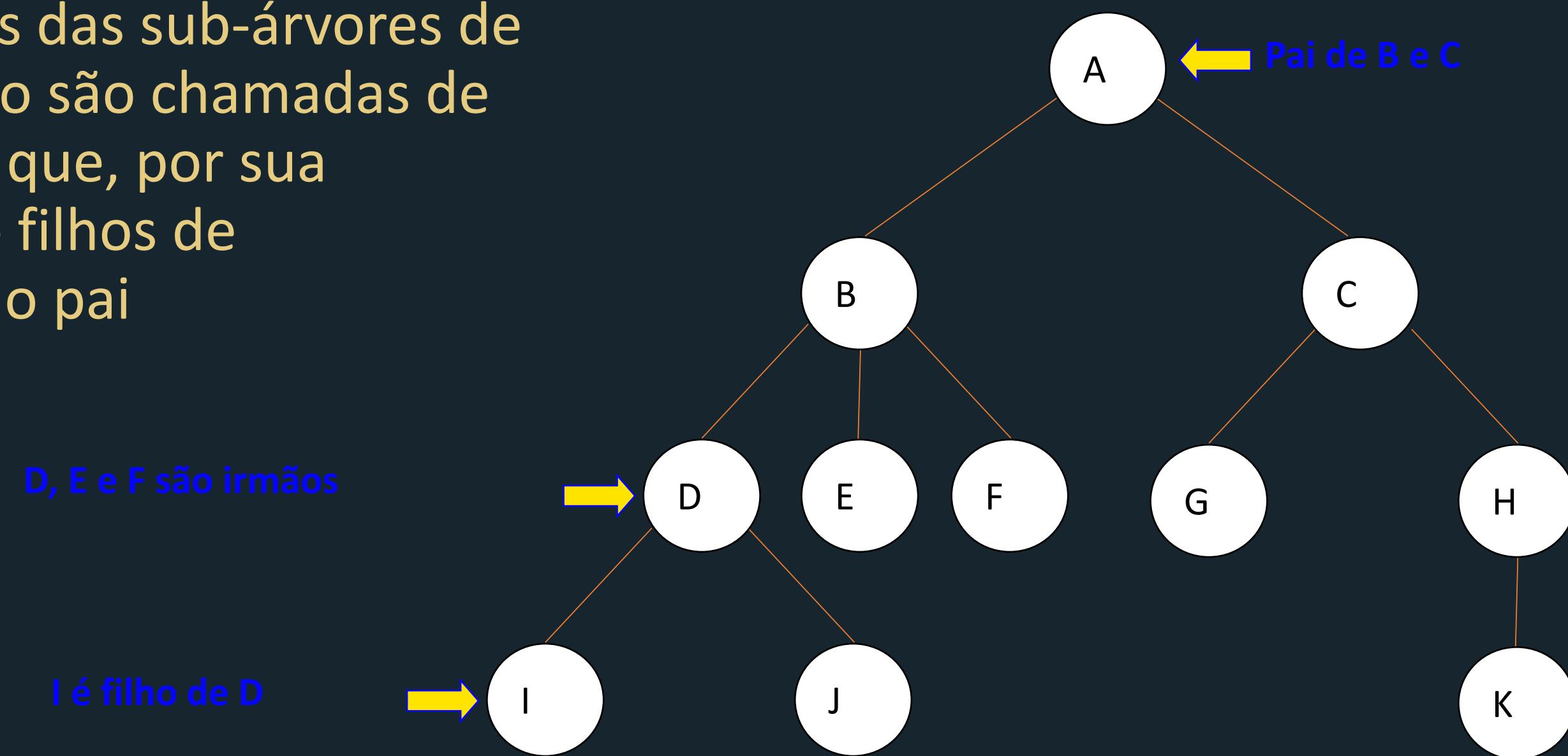
```
1 function factorial(number){  
2     if (number < 0)  
3         return undefined;  
4     let total = 1;  
5     for (let n = number; n > 1; n--)  
6         total = total * n;  
7     return total;  
8 }  
9  
10 console.log(factorial(5)); // 120
```



```
1 function factorialRecursivo(n) {  
2     if (n === 1 || n === 0)  
3         return 1;  
4     return n * factorialRecursivo(n - 1);  
5 }  
6  
7  
8 console.log(factorialRecursivo(5));
```

# Árvores

- A raiz de uma árvore é chamada de pai de suas sub-árvore
- As raízes das sub-árvore de um nodo são chamadas de irmãos, que, por sua vez, são filhos de seu nodo pai



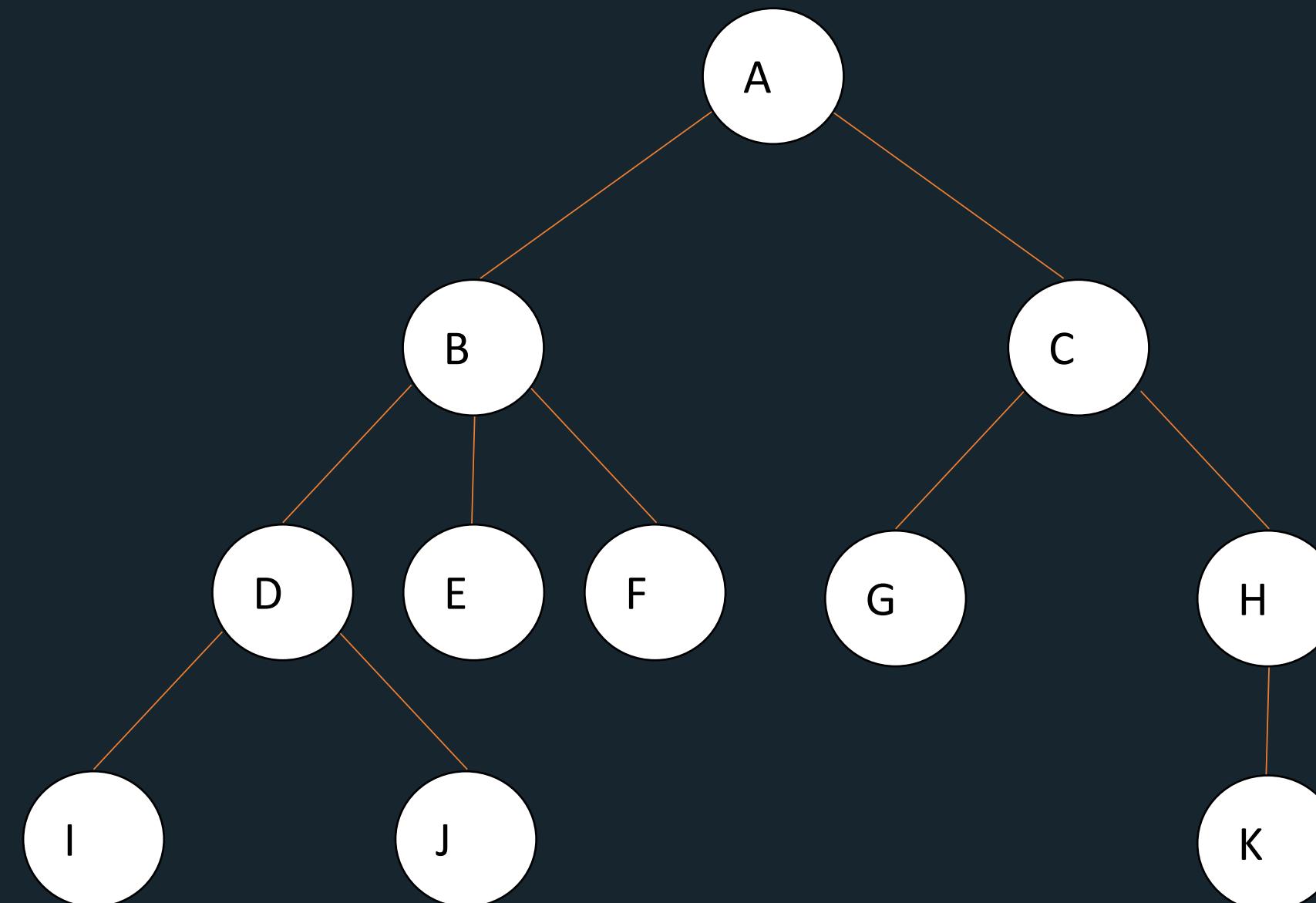
# Árvore

- Grau
  - É o número de sub-árvore de um nodo
  - Quando o grau é zero, ou seja, o nodo não possui filhos, ele é **folha**
- Nível de um nodo
  - É o número de linhas que liga o nodo à raiz, sabendo que a raiz é o nível zero
- Altura
  - É definida como sendo o nível mais alto da árvore

# Árvore

- Conceitos e Terminologia - Exemplo

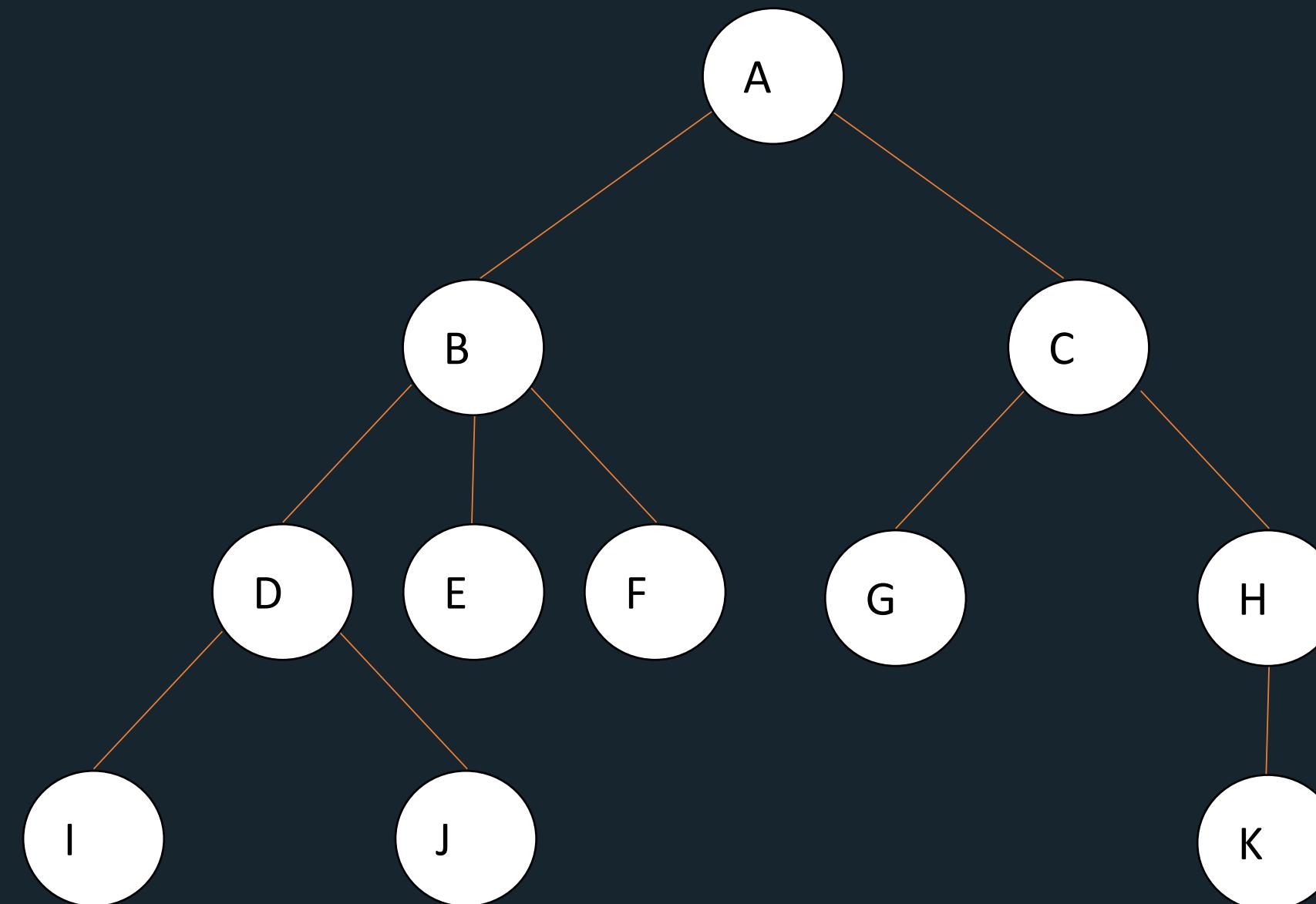
- Grau do nodo A: 2
- Grau do nodo B: 3
- Grau do nodo H: 1
- Grau do nodo J: 0



# Árvores

- Conceitos e Terminologia - Exemplo:

- Nível do nodo A: 0
- Nível do nodo C: 1
- Nível do nodo E: 2
- Nível do nodo G: 2
- Nível do nodo I: 3
- Altura da árvore: 3



# Árvores

- Conceitos e Terminologia - Exemplo:

- Raiz:

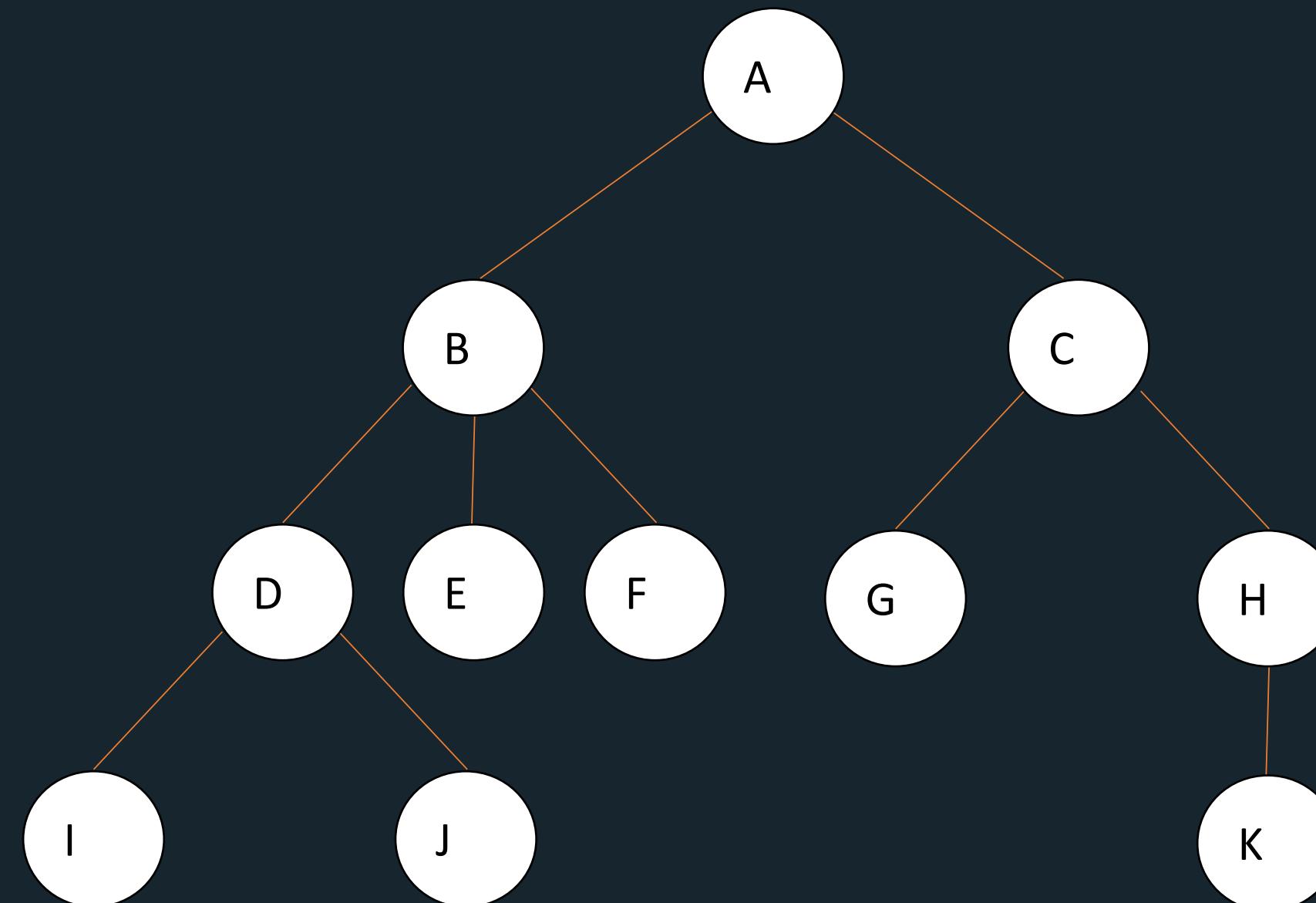
- A

- Nodos internos:

- B, C, D e H

- Folhas:

- I, J, E, F, G e K



# Árvores

- Métodos esperados
- Uma árvore deve disponibilizar métodos de acesso que retornam e aceitam posições:
  - *root()*: retorna a raiz da árvore;
  - *parent(v)*: retorna o nodo pai de  $v$ , ocorrendo um erro se for a raiz;
  - *children(v)*: retorna os filhos do nodo  $v$ ;

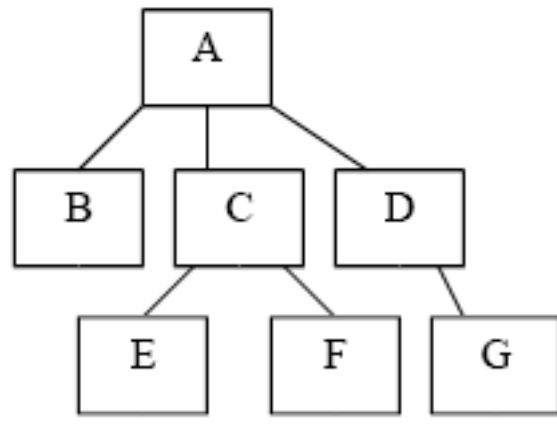
# Árvores

- Métodos esperados
  - Métodos de consulta:
    - $isInternal(v)$ : testa se um nodo v é interno e retorna *true* ou *false*;
    - $isExternal(v)$ : testa se um nodo v é externo e retorna *true* ou *false*;
    - $isRoot(v)$ : testa se um nodo v é raiz e retorna *true* ou *false*;

# Árvores

- Métodos esperados
  - Métodos “genéricos” (não estão necessariamente relacionados com sua estrutura):
    - *size()*: retorna o número de nodos na árvore;
    - *isEmpty()*: testa se a árvore tem ou não tem algum nodo;
    - *positions()*: retorna uma coleção com todos os nodos da árvore;
    - *replaceElement(v,e)*: retorna o elemento armazenado em *v* e o substitui por *e*;

# Árvores



- Representação em memória

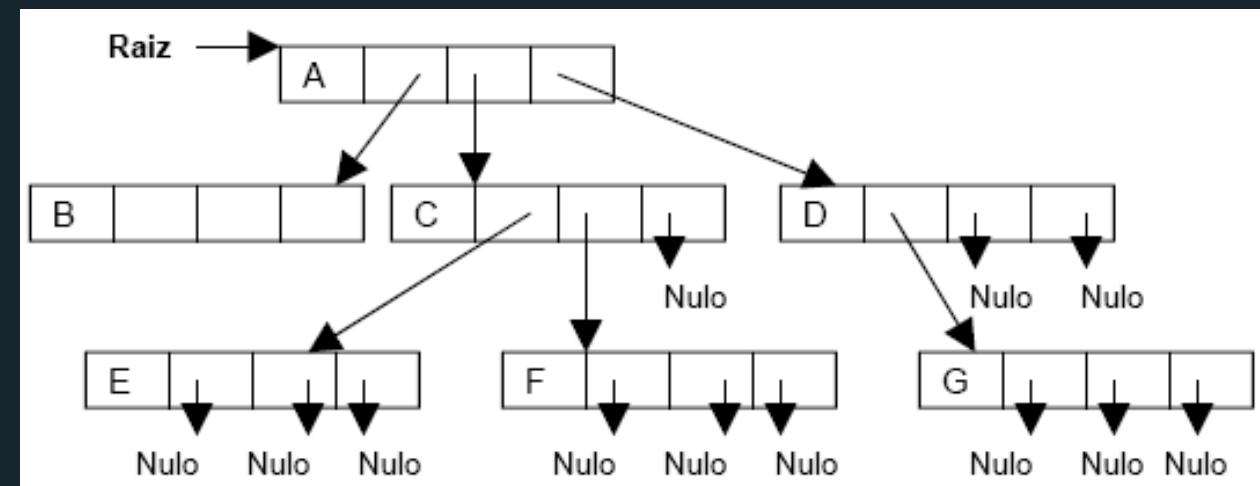
- Da mesma forma que as estruturas de dados lineares, podemos alocar as árvores de duas maneiras

- Por contiguidade
  - Espaço comum de memória

Possível  
representação

	0		1		2		3		4		5		6														
A	1	2	3	B	-1	-1	-1	C	4	5	-1	D	6	-1	-1	E	-1	-1	-1	F	-1	-1	-1	G	-1	-1	-1

- Por encadeamento
  - Diferentes espaços de memória



# Árvore binária

- Uma árvore binária é uma árvore ordenada com as seguintes propriedades:
  - Todos os nodos tem no máximo dois filhos
  - Cada nodo filho é rotulado como sendo um *filho da esquerda* ou um *filho da direita*
  - O filho da esquerda precede o filho da direita na ordenação dos filhos de um nodo

# Árvore binária

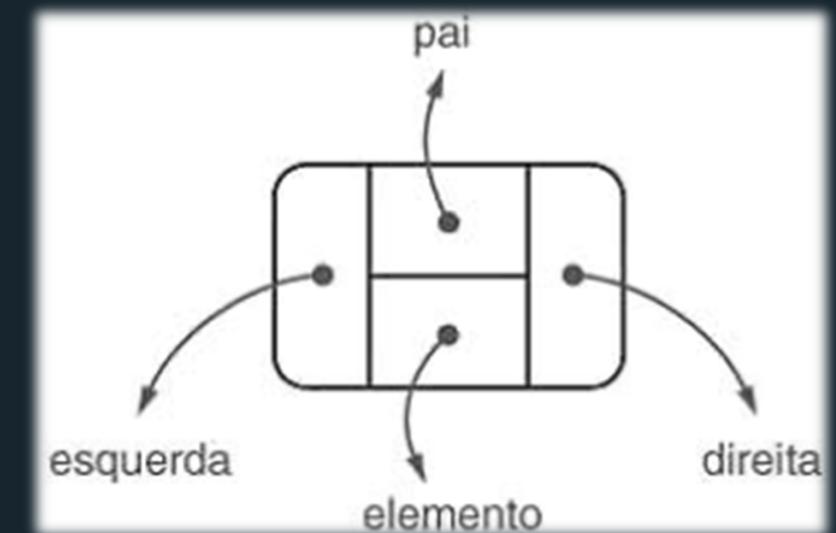
- Subárvores:
  - O filho da esquerda de um nodo interno  $v$  é chamado de **subárvore da esquerda**
  - O filho da direita de um nodo interno  $v$  é chamado de **subárvore da direita**
- Árvore binária própria (ou cheia)
  - Cada nodo tem 0 ou 2 filhos
- As estruturas de árvores binárias são muito utilizadas na computação

# Árvore binária

- Vários métodos são necessários para manipular uma árvore binária
  - Criar a árvore
  - Inserir nodos na árvore
  - Pesquisar nodos na árvore
  - Excluir nodos da árvore
  - Determinar a altura da árvore e o nível de um nodo
  - Caminhar em uma árvore, visitando seus nodos
  - ...

# Árvore binária

- Forma mais usual de implementação
- Estruturas encadeadas (alocação dinâmica)
- Cada nodo conterá
  - A informação
  - Uma referência para o nodo pai
  - Uma referências para a subárvore da esquerda
  - Uma referências para a subárvore da direita



```
1 class Node {  
2     constructor(element) {  
3         this.element = element;  
4         this.parent = null;  
5         this.left = null;  
6         this.right = null;  
7     }  
8 }
```

# Árvore binária

- A estrutura da árvore armazena
  - referência para a raiz da árvore
  - número de elementos já inseridos
- Suporta, por exemplo, os seguintes métodos (1/3):
  - boolean addRoot(element): retorna falso se a árvore não está vazia, senão insere o elemento e como raiz
  - boolean addLeft(element, father): insere o elemento e como filho de father, na subárvore da esquerda; retorna falso se não encontrar father
  - boolean addRight(element, father): insere o elemento e como filho de father, na subárvore da direita ; retorna falso se não encontrar father
  - boolean hasRight(element): retorna verdadeiro se possui subárvore à direita
  - boolean hasLeft(element): retorna verdadeiro se possui subárvore à esquerda

# Árvore binária

- A estrutura da árvore armazena
  - referência para a raiz da árvore
  - número de elementos já inseridos
- Suporta, por exemplo, os seguintes métodos (2/3):
  - Integer getParent(element): retorna o pai do elemento e
  - boolean isInternal(element): retorna true se o elemento está armazenado em um nodo interno
  - boolean isExternal(element): retorna true se o elemento está armazenado em um nodo externo
  - boolean removeBranch(element): remove o elemento e e seus filhos
  - boolean contains(element): retorna true se a árvore contém o elemento e
  - boolean isEmpty(): retorna true se a árvore está vazia
  - void clear(): remove todos os elementos da árvore
  - int size(): retorna o número de elementos armazenados na árvore

# Árvore binária

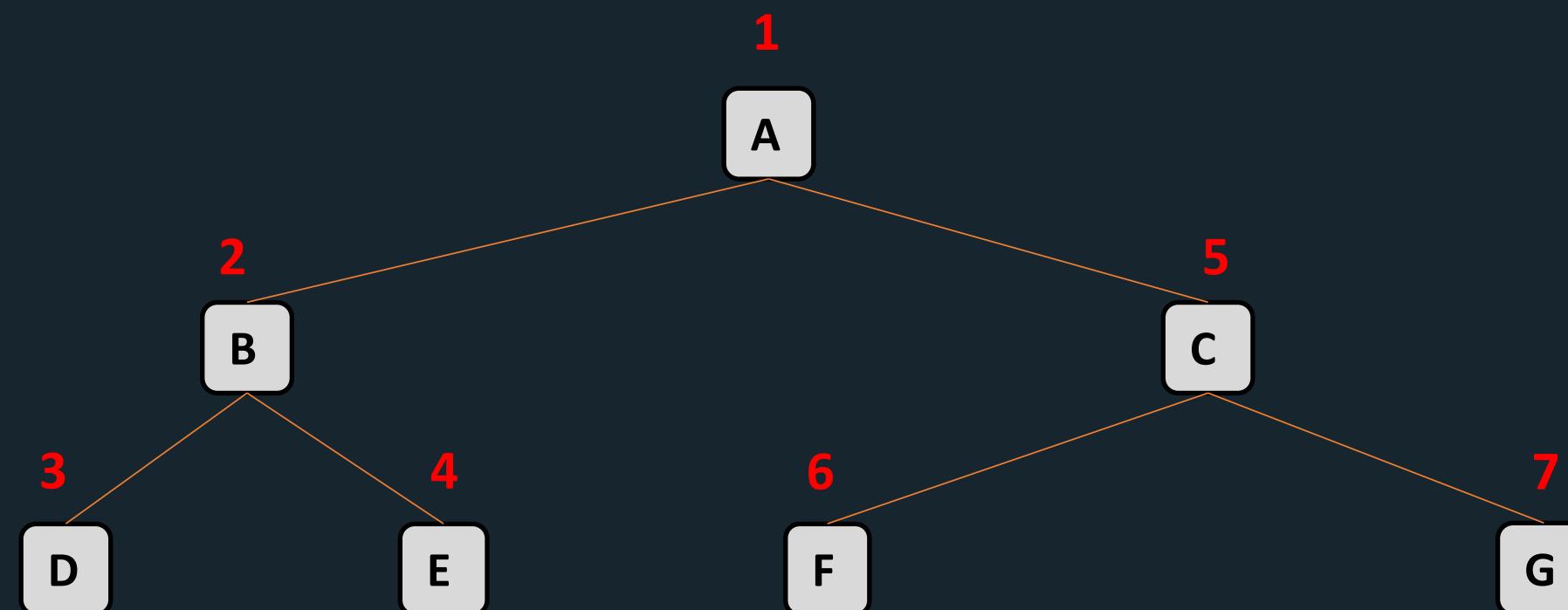
- A estrutura da árvore armazena
  - referência para a raiz da árvore
  - número de elementos já inseridos
- Suporta, por exemplo, os seguintes métodos (3/3):
  - Integer getRoot(): retorna o elemento armazenado na raiz
  - void setRoot(element): altera o elemento armazenado na raiz
  - Integer getLeft(element): retorna o filho à esquerda de e
  - Integer getRight(element): retorna o filho à direita de e
  - LinkedList<Integer> positionsPre(): retorna uma lista com todos os elementos da árvore na ordem pré-fixada
  - LinkedList<Integer> positionsCentral(): retorna uma lista com todos os elementos da árvore na ordem central
  - LinkedList<Integer> positionsPos(): retorna uma lista com todos os elementos da árvore na ordem pos-fixada
  - LinkedList<Integer> positionsWidth(): retorna uma lista com todos os elementos da árvore com um caminhamento em largura

# Árvore Binária

- Formas diferentes de caminhamento:
  - Percurso em profundidade
    - Percurso pré-fixado ou em pré-ordem
    - Percurso pós-fixado ou em pós-ordem
    - Percurso central ou em ordem central
  - Percurso em largura

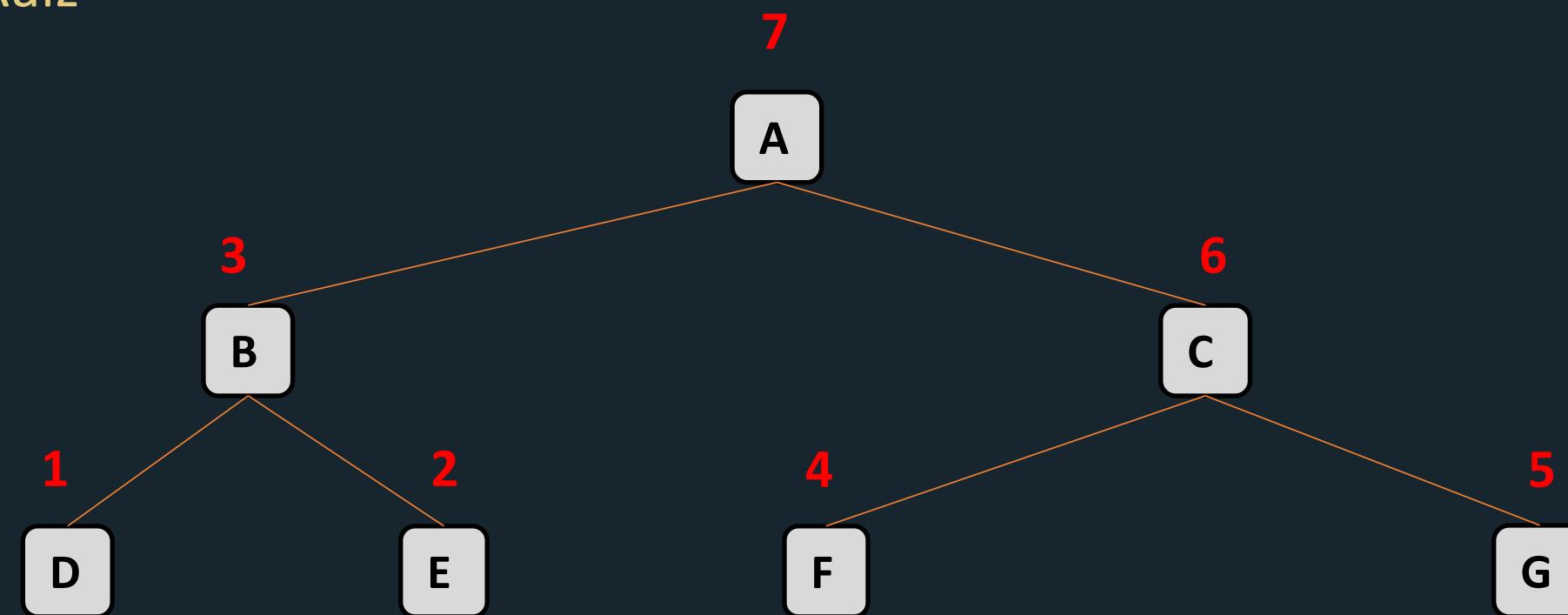
# Árvore Binária

- No caminhamento pré-fixado
  - Nodo é visitado antes de seus descendentes
  - Segue a ordem:
    - Visita Raiz
    - Percorre subárvore da esquerda
    - Percorre subárvore da direita
  - Exemplo:



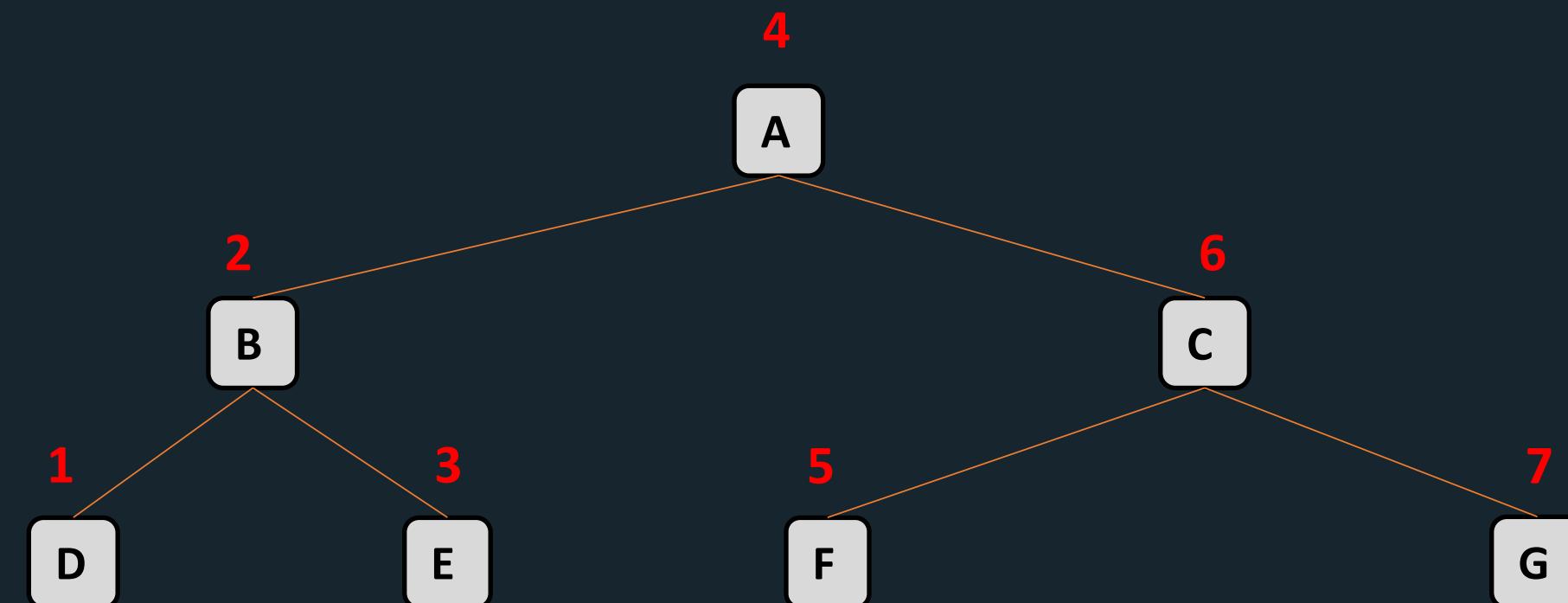
# Árvore Binária

- No caminhamento pós-fixado
  - Nodo é visitado depois de seus descendentes
  - Segue a ordem:
    - Percorre subárvore da esquerda
    - Percorre subárvore da direita
    - Visita Raiz
- Exemplo:



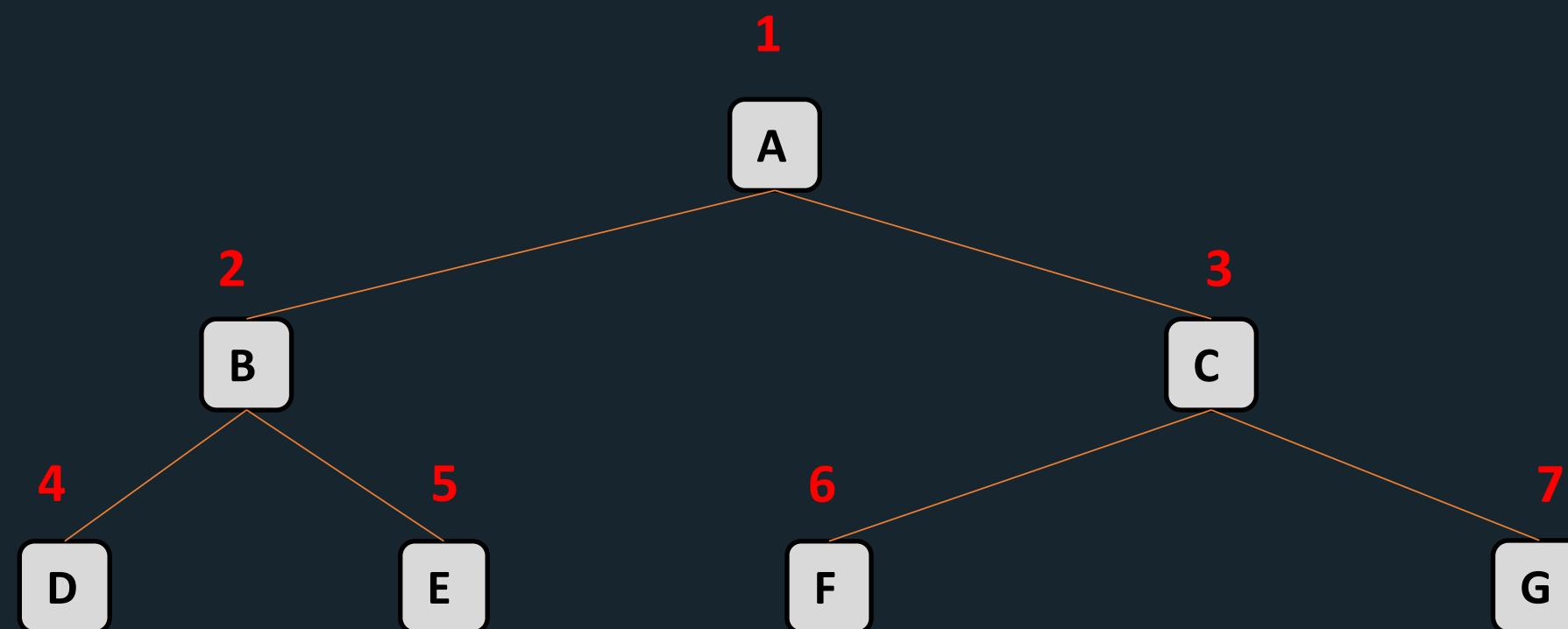
# Árvore Binária

- No caminhamento central
  - Segue a ordem:
    - Percorre subárvore da esquerda
    - Visita Raiz
    - Percorre subárvore da direita
  - Exemplo:



# Árvore Binária

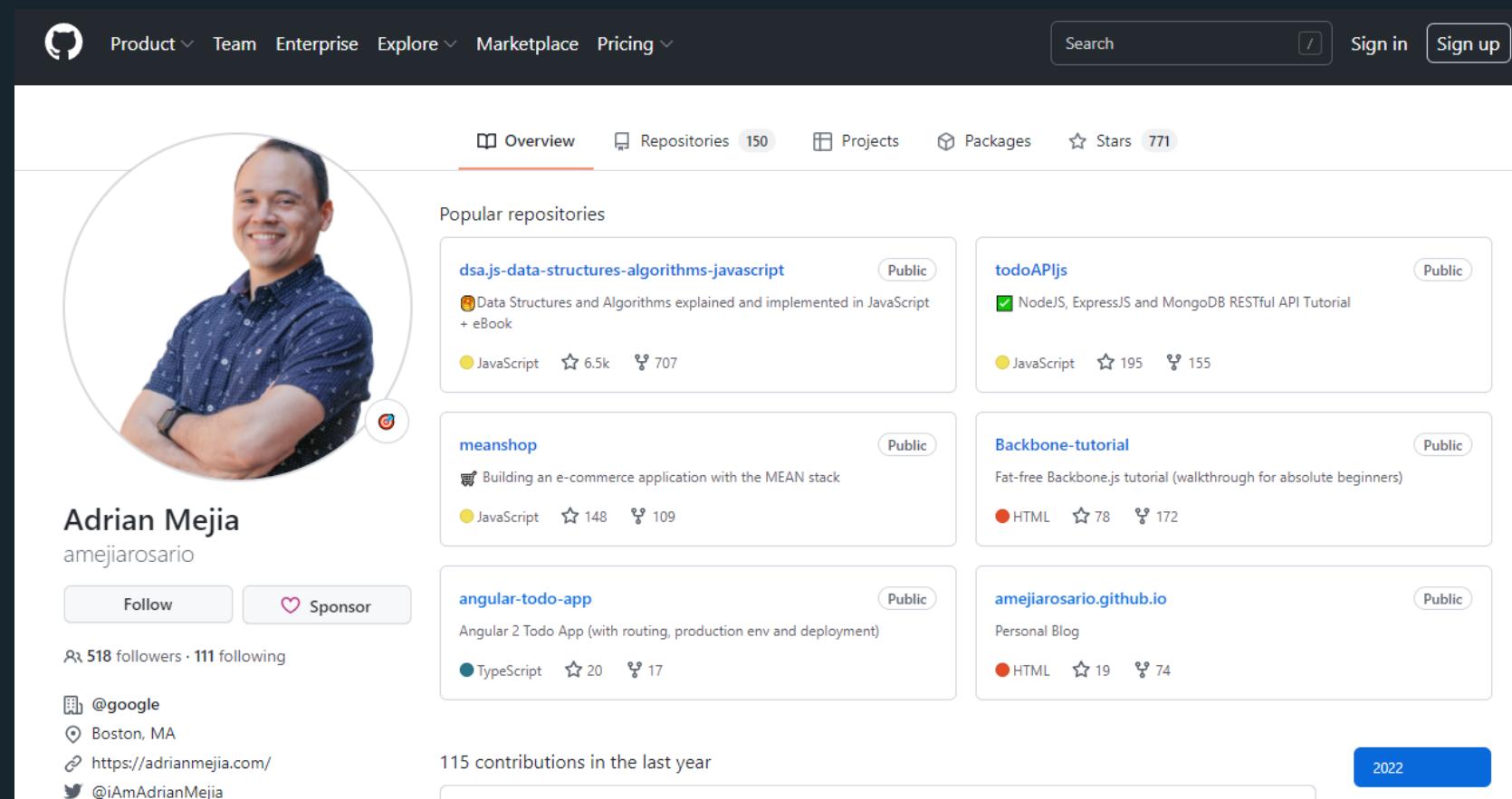
- Percurso em largura
  - Visita os nodos na ordem dos níveis da árvore, da esquerda para a direita
    - Visite os nodos de nível 0;
    - Visite os nodos de nível 1;
    - ...
  - Exemplo:



# Árvore binária

- Código de exemplo

- <https://github.com/loiane/javascript-datastructures-algorithms/blob/main/src/js/data-structures/binary-search-tree.js>
- <https://github.com/amejiarosario/dsa.js-data-structures-algorithms-javascript/blob/master/src/data-structures/trees/binary-tree-node.js>



# Organização da aula

- Fundamentos de computação e Algoritmos
- Estrutura de dados padrão da linguagem
- Complexidade algorítmica
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Técnicas e Projeto de algoritmos
- Considerações finais

# O vasto mundo dos algoritmos

- Tipos
  - <https://github.com/loiane/javascript-datastructures-algorithms/tree/main/src/js/algorithms>
  - Divisão e conquista
    - Quebra o problema em partes menores e independentes (e.g. busca binária)
    - Combina os resultados
  - Programação dinâmica
    - Quebra o problema em partes menores mas independentes (e.g. fibonacci)
  - Gulosos (greedy)
    - Considera os melhoramentos locais (não local) para definir caminhos
  - Algoritmo de Backtracking
    - Considera rever ponto ótimo no passado caso os melhoramento atuais não sejam bons
  - etc

# Casos específicos - Algoritmos de ordenamento

- Motivação
  - A presença de dados desordenados impactará no desempenho da solução
    - Caso de exemplificação
      - Você é responsável por coletar o contato da pessoas (nome, idade e telefone) em um encontro onde o material utilizado foi post-it e caneta
      - Todos preenchem e depositam os papeis em uma urna
      - Em seguida, alguém lhe pede o contato de um participante em específico, lhe fornecendo o nome a ser procurado
      - Qual procedimento?
      - A ausência de uma organização de armazenamento da informação lhe consumirá tempo

# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento

- Bubble sort

- Compara dois elementos por vez, passando por todo vetor  $n^2$  vezes

- Selection sort

- Insertion Sort

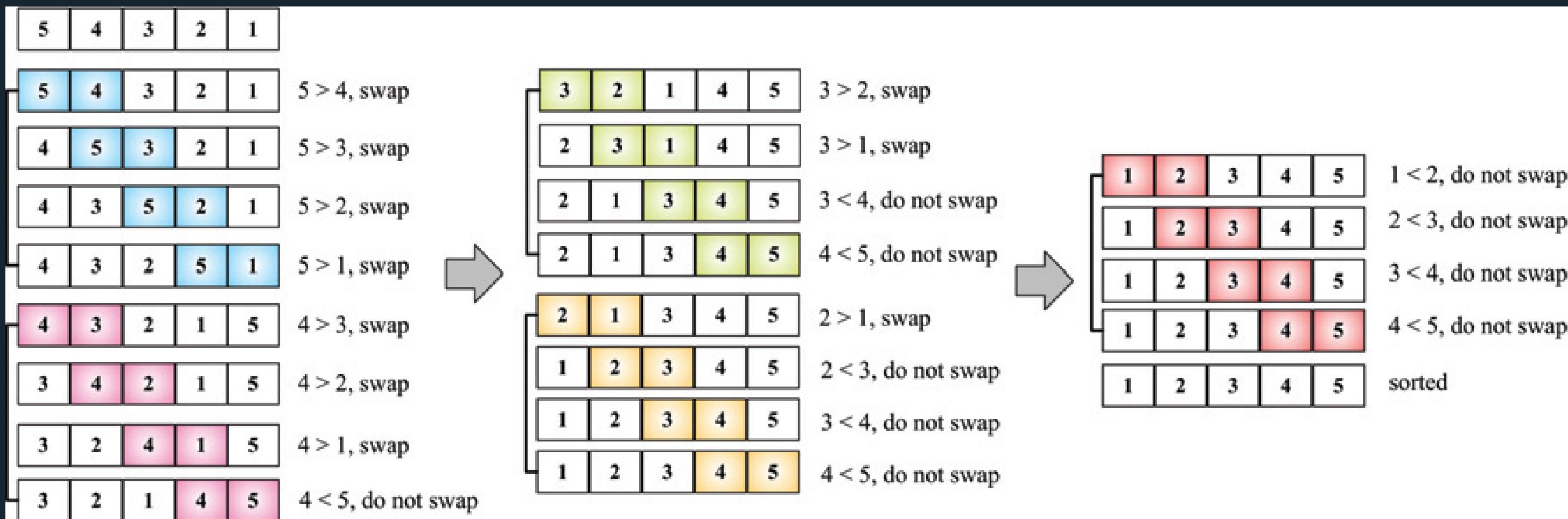
- Merge Sort

- Quick sort

```
1  function bubbleSort(array, compareFn = defaultCompare) {
2    const { length } = array; // {1}
3    for (let i = 0; i < length; i++) { // {2}
4      for (let j = 0; j < length - 1; j++) { // {3}
5        if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) { // {4}
6          swap(array, j, j + 1); // {5}
7        }
8      }
9    }
10   return array;
```

# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento
  - Bubble sort



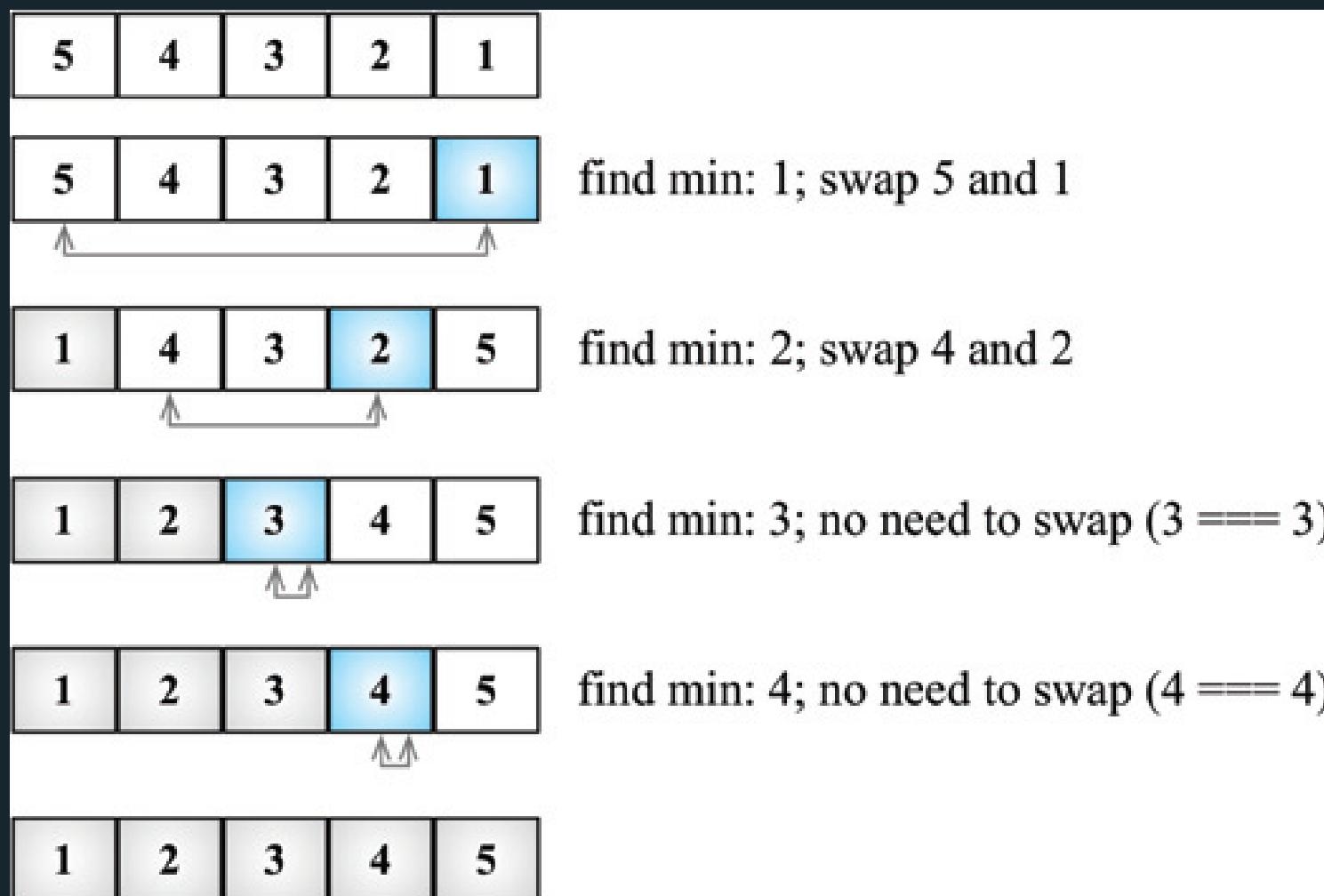
# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento
  - Selection sort
    - Restringe o número de trocas realizadas ao longo da ordenação
  - Insertion Sort
  - Merge Sort
  - Quick sort

```
1  function selectionSort(array, compareFn = defaultCompare) {  
2      const { length } = array; // {1}  
3      let indexMin;  
4      for (let i = 0; i < length - 1; i++) { // {2}  
5          indexMin = i; // {3}  
6          for (let j = i; j < length; j++) { // {4}  
7              if (compareFn(array[indexMin], array[j]) === Compare.BIGGER_THAN) { // {5}  
8                  indexMin = j; // {6}  
9              }  
10         }  
11         if (i !== indexMin) { // {7}  
12             swap(array, i, indexMin);  
13         }  
14     }  
15     return array;  
16 };
```

# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento
  - Selection sort



# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento

- Insertion Sort

- Abre espaço para inserção, restringindo a área de comparação

- Merge Sort

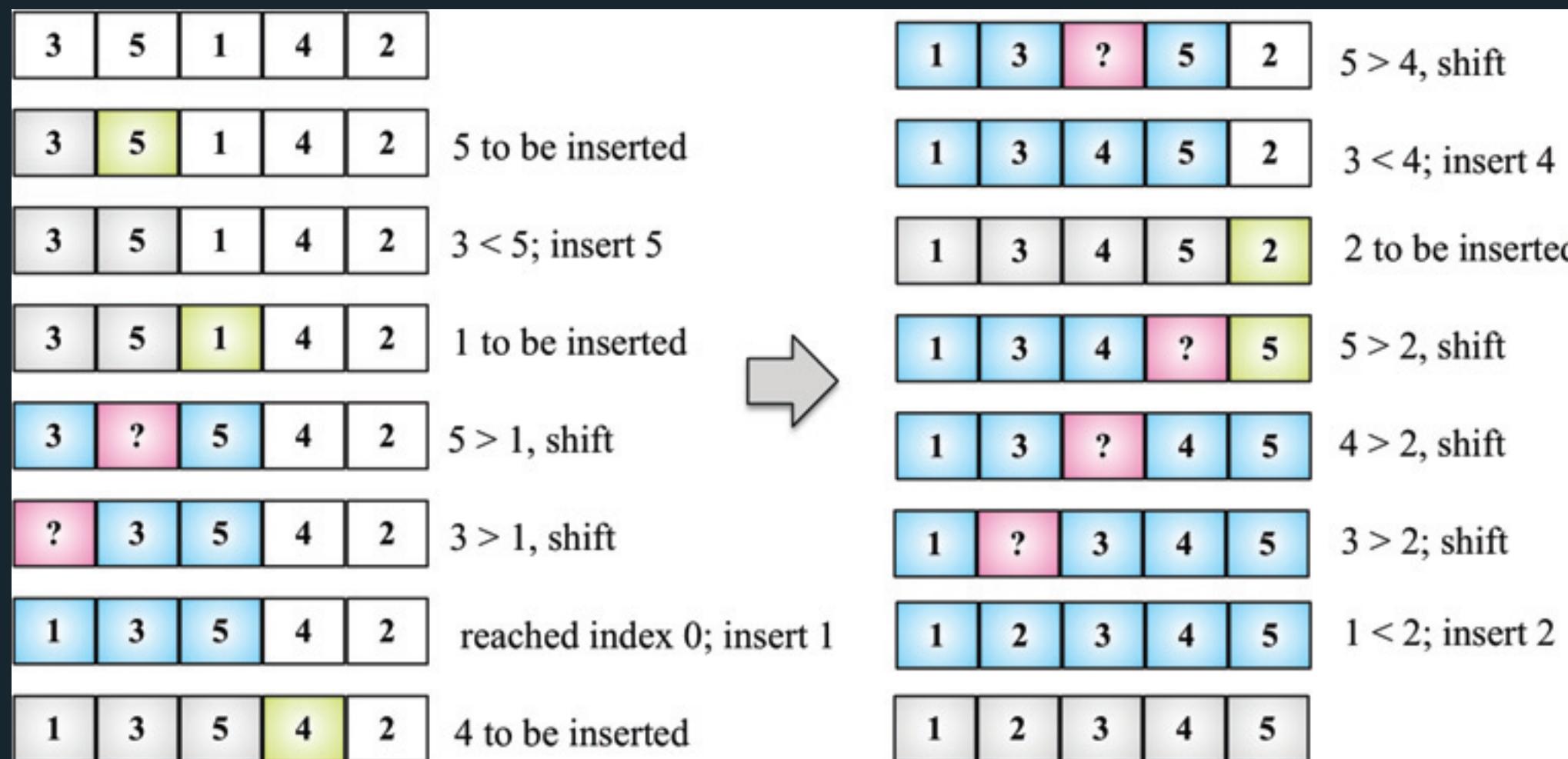
- Quick sort

```
1  function insertionSort(array, compareFn = defaultCompare) {  
2      const { length } = array; // {1}  
3      let temp;  
4      for (let i = 1; i < length; i++) { // {2}  
5          let j = i; // {3}  
6          temp = array[i]; // {4}  
7          while (j > 0 && compareFn(array[j - 1], temp) === Compare.BIGGER_THAN) { // {5}  
8              array[j] = array[j - 1]; // {6}  
9              j--;  
10         }  
11         array[j] = temp; // {7}  
12     }  
13     return array;  
14 }
```

# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento

- Insertion Sort



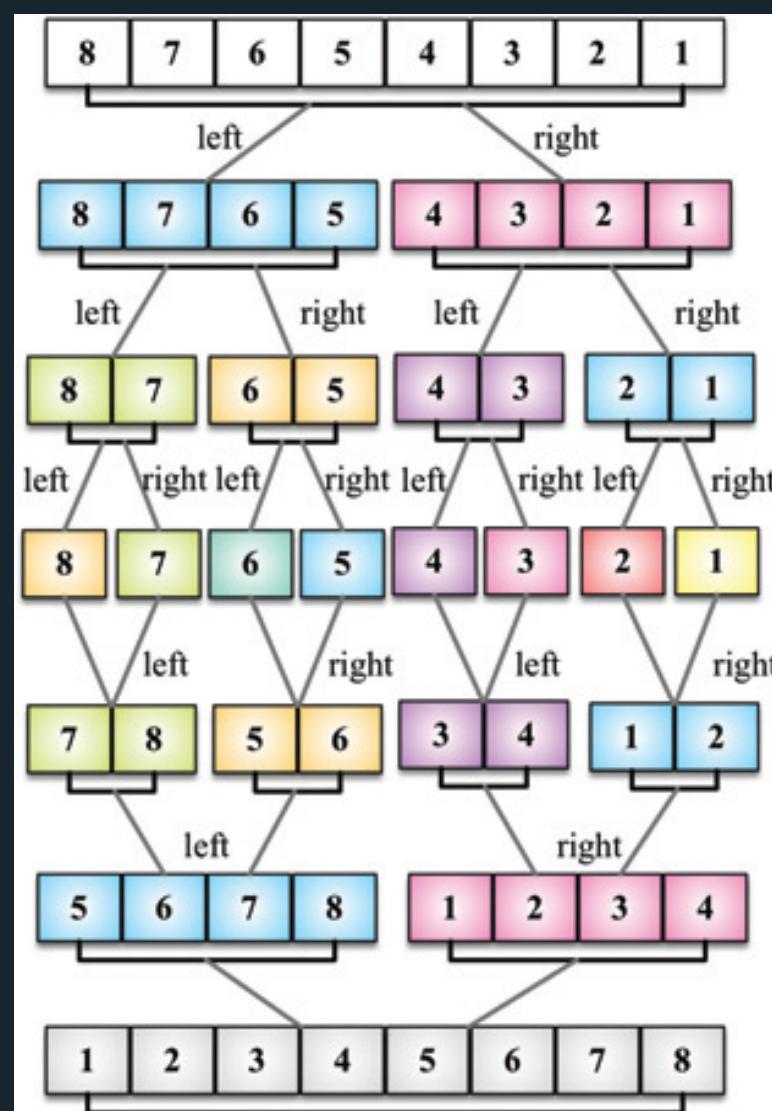
# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento
  - Merge Sort
    - Explore recursividade
  - Quick sort

```
1  function mergeSort(array, compareFn = defaultCompare) {  
2      if (array.length > 1) { // {1}  
3          const { length } = array;  
4          const middle = Math.floor(length / 2); // {2}  
5          const left = mergeSort(array.slice(0, middle), compareFn); // {3}  
6          const right = mergeSort(array.slice(middle, length), compareFn); // {4}  
7          array = merge(left, right, compareFn); // {5}  
8      }  
9      return array;  
10 }
```

# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamiento
  - Merge Sort



# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamento

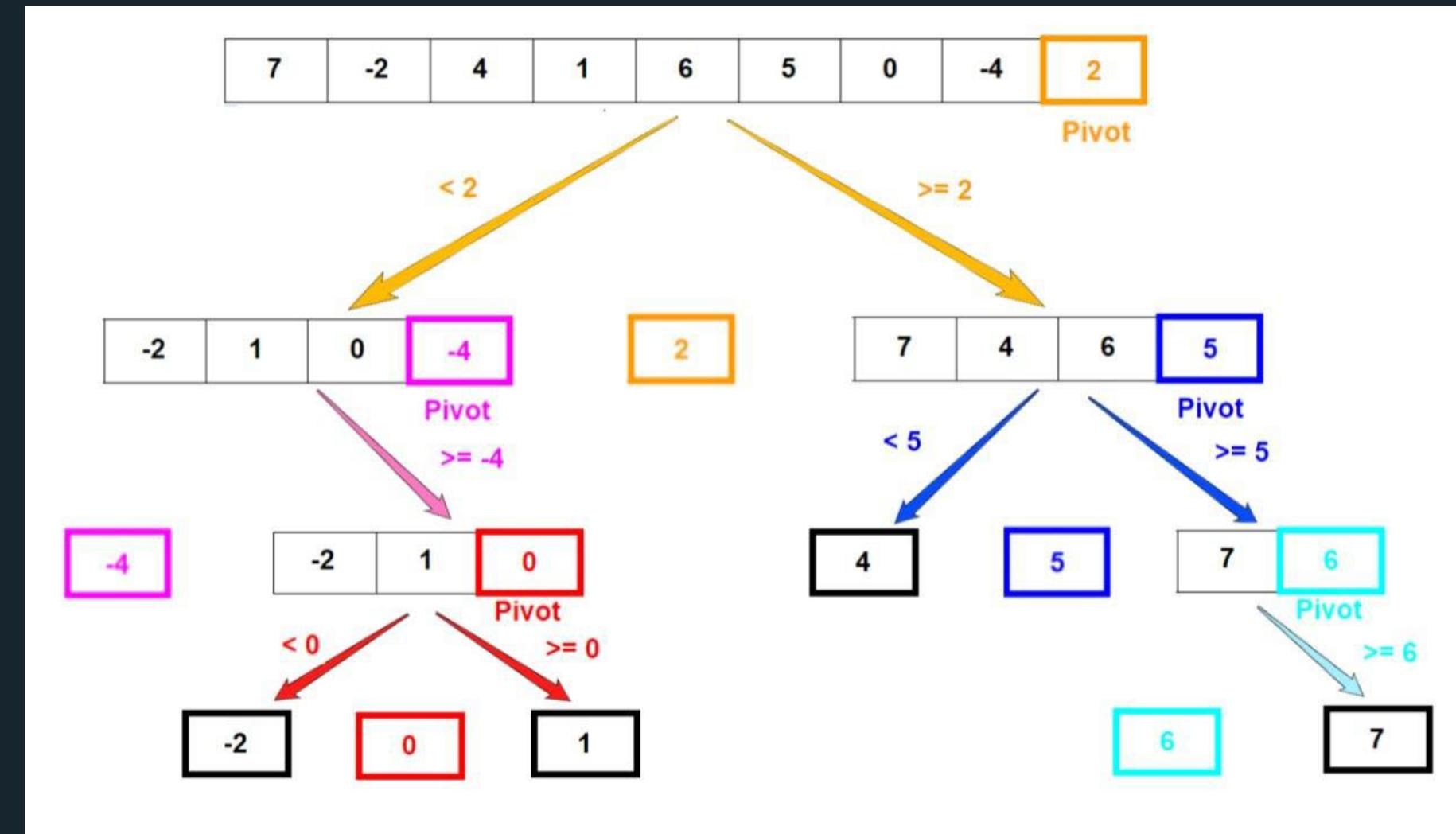
- Quick sort

- Define um pivô
  - Quebra o problema no pivô
  - Redefine um pivô
    - Para cada segmento
  - Ao final
    - Retorna ordenando

```
1  function quickSort(array, compareFn = defaultCompare) {  
2    return quick(array, 0, array.length - 1, compareFn);  
3  };  
4  
5  function quick(array, left, right, compareFn) {  
6    let index; // {1}  
7    if (array.length > 1) { // {2}  
8      index = partition(array, left, right, compareFn); // {3}  
9      if (left < index - 1) { // {4}  
10        quick(array, left, index - 1, compareFn); // {5}  
11      }  
12      if (index < right) { // {6}  
13        quick(array, index, right, compareFn); // {7}  
14      }  
15    }  
16    return array;  
17  };
```

# Casos específicos - Algoritmos de ordenamento

- Algoritmos de ordenamiento
  - Quick sort



# Casos específicos - Algoritmos de ordenamento

- Comparação entre algoritmos de ordenamento
- <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>
  - <https://www.toptal.com/developers/sorting-algorithms>

# Casos específicos - Algoritmos de busca

- Busca linear/sequencial
- Busca binária

8   7   6   5   4   3   2   1	searching for #2 mid = 9/2 = 4
1   2   3   4   5   6   7   8	mid = 9/2 = 4 $4 < 2$
1   2   3   4   5   6   7   8	mid = 4/2 = 2 $2 == 2$

# Casos específicos - Algoritmos de busca

- **Busca linear/sequencial**

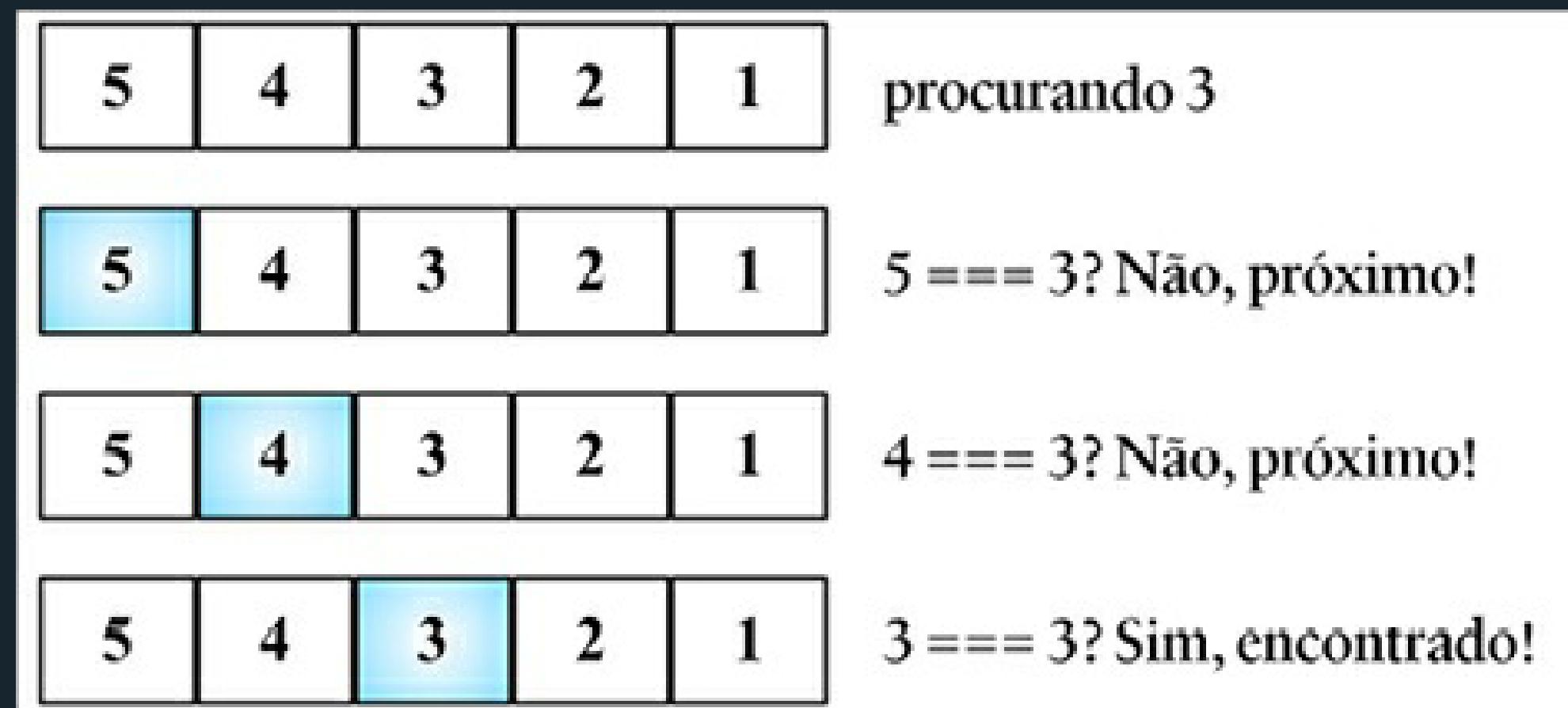
- Investiga cada posição da estrutura de dados
- Retorna o item solicitado (e.g. chave, posição)

```
1 const DOES_NOT_EXIST = -1;
2 function sequentialSearch(array, value, equalsFn = defaultEquals) {
3     for (let i = 0; i < array.length; i++) { // {1}
4         if (equalsFn(value, array[i])) { // {2}
5             return i; // {3}
6         }
7     }
8     return DOES_NOT_EXIST; // {4}
9 }
```

# Casos específicos - Algoritmos de busca

- **Busca linear/sequencial**

- Investiga cada posição da estrutura de dados
- Retorna o item solicitado (e.g. chave, posição)



# Casos específicos - Algoritmos de busca

- **Busca binária**

- Requer que a estrutura de dados esteja ordenada
- “Sorteia uma posição”
  - Se o valor contido na posição é MAIOR do que o procurado então procura sorteia a esquerda
  - Se o valor contido na posição é MENOR do que o procurado então procura sorteia a esquerda

- A cada progresso

- Redefine espaço de busca

```
1 function binarySearch(array, value, compareFn = defaultCompare) {  
2     const sortedArray = quickSort(array); // {1}  
3     let low = 0; // {2}  
4     let high = sortedArray.length - 1; // {3}  
5     while (lesserOrEquals(low, high, compareFn)) { // {4}  
6         const mid = Math.floor((low + high) / 2); // {5}  
7         const element = sortedArray[mid]; // {6}  
8         if (compareFn(element, value) === Compare.LESS_THAN) { // {7}  
9             low = mid + 1; // {8}  
10        } else if (compareFn(element, value) === Compare.BIGGER_THAN) { // {9}  
11            high = mid - 1; // {10}  
12        } else {  
13            return mid; // {11}  
14        }  
15    }  
16    return DOES_NOT_EXIST; // {12}  
17}
```

# Casos específicos - Algoritmos de busca

## • Busca binária

- Requer que o array esteja ordenado
- "Sorteia um intervalo":
  - Se o valor é menor que o elemento central, procura na metade esquerda
  - Se o valor é maior que o elemento central, procura na metade direita
- A cada passo:
  - Redefinir o intervalo

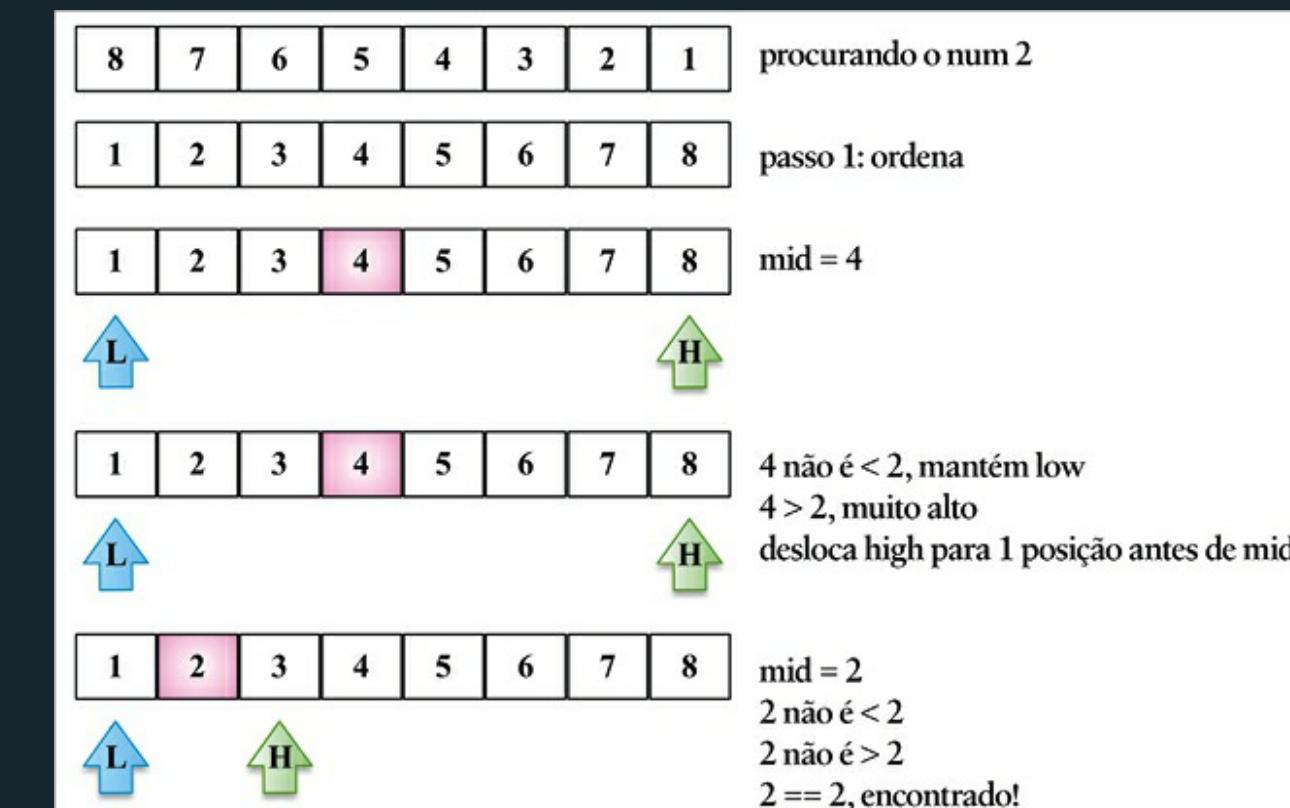
```
1  function binarySearch(array, value, compareFn = defaultCompare) {
2    const sortedArray = quickSort(array); // {1}
3    let low = 0; // {2}
4    let high = sortedArray.length - 1; // {3}
5    while (lesserOrEquals(low, high, compareFn)) { // {4}
6      const mid = Math.floor((low + high) / 2); // {5}
7      const element = sortedArray[mid]; // {6}
8      if (compareFn(element, value) === Compare.LESS_THAN) { // {7}
9        low = mid + 1; // {8}
10     } else if (compareFn(element, value) === Compare.BIGGER_THAN) { // {9}
11       high = mid - 1; // {10}
12     } else {
13       return mid; // {11}
14     }
15   }
16   return DOES_NOT_EXIST; // {12}
17 }
```

# Casos específicos - Algoritmos de busca

- **Busca binária**

- Requer que a estrutura de dados esteja ordenada
- “Sorteia uma posição”
  - Se o valor contido na posição é MAIOR do que o procurado então procura sorteia a esquerda
  - Se o valor contido na posição é MENOR do que o procurado então procura sorteia a esquerda

- A cada progresso
  - Redefine espaço de busca



# Organização da aula

- Fundamentos de computação e Algoritmos
- Estrutura de dados padrão da linguagem
- Complexidade algorítmica
- Listas encadeadas como estrutura de dados
- Árvores como estrutura de dados
- Algoritmos de ordenamento e seleção
- Considerações finais

# Considerações finais

- Use as referências bibliográficas
  - GRONER, Loiane. Learning JavaScript Data Structures and Algorithms. Third Edition. Birmingham: Packt, 2018.
  - CORMEN, Thomas H.; et al. Algoritmos: teoria e prática. Terceira Edição. Rio de Janeiro : Elsevier, 2012.\*
  - PIVA JR, Dilermando et al. Estrutura de dados e técnicas de programação. 2a. Edição. Rio de Janeiro: Elsevier, 2019.
- Explore o github
- Crie seu repositório/portifólio

E ao final...

SUCCESS

⇒ Go get it ⇒

PUCRS online  uol edtech