# Estruturas de Dados e Algoritmos

## Frequência II

Carlos Menezes

9 de maio de 2021

# Conteúdo

# 1 Análise de Algoritmos

**Definição 1.1 (Algoritmo)** *Sequência finita de ações executáveis que visam obter uma solução para um determinado tipo de problema.*

**Observação 1** *Dois algoritmos que resolvem o mesmo problema podem ter eficiências diferentes. Um algoritmo rápido permite resolver um problema numa máquina lenta.*

Existem formas de avaliar as exigências de um algoritmo:

- Empiricamente: observando a execução do programa;
- Experimentalmente: modelo simplificado do problema, estimando o comportamento futuro;
- Analiticamente: demonstrando a existência de uma função matemática que calcule as exigências do programa em relação aos parâmetros do problema.

**Exemplos**

```
int soma = 0;
for ( int i = 1 ; i <= n ; i++ )
    soma = soma + i;
```

Algoritmo 1: Algoritmo I

```
int soma = 0;
for ( int i = 1 ; i <= n ; i++ )
    for (int j = 1 ; j <= i ; j++ )
        soma = soma + 1;
```

Algoritmo 2: Algoritmo II

```
int soma = n * ( n + 1 ) / 2;
```

Algoritmo 3: Algoritmo III

| | Algoritmo I | Algoritmo II | Algoritmo III |
|---|---|---|---|
| Atribuições | n+2 | $3 + n\frac{n}{2}$ | 1 |
| Adições | 2n | $n(\frac{2n}{2})$ | 1 |
| Multiplicações | | | 1 |
| Divisões | | | 1 |
| Operações | 3n+2 | $\frac{3n^2}{2} + 3$ | 4 |

## 1.1 Crescimento

**Observação 2** *O tempo de execução geralmente dependente de um único parâmetro $N$: medida abstrata do tamanho do problema a considerar, usualmente relacionada com a quantidade de dados a processar.*

## 1.2 Notação Assimptótica

**Observação 3** *A análise do tempo de execução de um algoritmo só é útil para valores de N elevados. É utilizado o* **pior-case** *como valor para complexidade, pois:*

- representa um limite superior no tempo de execução;
- o valor médio é muitas vezes próximo do pior-caso.

**Definição 1.2 (Notação Assimptótica)** *A notação assimptótica permite estabelecer taxas de crescimento dos tempos de execução dos algoritmos em função dos tamanhos das entradas. A complexidade assimptótica é expressa através da ordem de magnitude usando a notação* big O, **O**.

**Definição 1.3 (Ordem de Magnitude)** *A ordem de magnitude de uma função é igual à ordem do seu termo que cresce mais rapidamente.*

**Exemplo**

$$f(n) = n + n^2 + 1$$

A ordem de magnitude de $f(n)$ é $\mathbf{O(n^2)}$.

**Observação 4 (Classes de complexidade comuns)**

- Constante, $\mathbf{O(1)}$
  - o número de instruções de um programa for executado um número limitado/-constante de vezes
- Logarítmica, $\mathbf{O}(\log(\mathbf{n}))$
  - um problema é resolvido através da resolução de um conjunto de subproblemas
- Polinomial, $\mathbf{O(n^p), p \geq 1}$
  - Linear, $\mathbf{O(n)}$
    - quando existe algum processamento para cada elemento de entrada
  - Pseudo-linear, $\mathbf{O}(\mathbf{n}\log(\mathbf{n}))$
    - Quando um problema é resolvido através da resolução de um conjunto de subproblemas, e combinando posteriormente as suas soluções
  - Quadrática, $\mathbf{O(n^2)}$
    - quando a dimensão da entrada duplica, o tempo aumenta 4x
  - Cúbica, $\mathbf{O(n^3)}$
    - quando a dimensão de entrada duplica o tempo aumenta 8x
- Exponencial, $\mathbf{O(p^n)}, n \geq 1$
  - quando a dimensão de entrada duplica o tempo aumenta para o quadrado
- Factorial, $\mathbf{O(n!)}$

**Observação 5 (Garantias, previsões e limitações)** *O tempo de execução dos algoritmos depende criticamente dos dados. O objeto da análise de complexidade é inferir o máximo de informação assumindo o mínimo possível.*

**Observação 6 (Estudo do pior caso)**

O estudo do pior caso permite obter garantias máximas.

- se o resultado no pior caso é aceitável, então a situação é favorável;
- se o resultado no pior caso for mau, pode ser problemático;

**Observação 7 (Estudo do melhor caso)**

Por vezes é relevante obter também limites inferiores. Esta análise pode dar indicações preciosas sobre como e onde melhorar o desempenho de um algoritmo. Se os limites inferiores e superiores forem próximos, é conveniente tentar otimizar a implementação.

- se o resultado no pior caso é aceitável, então a situação é favorável;
- se o resultado no pior caso for mau, pode ser problemático;

# 2 Algoritmos de Ordenação

## 2.1 Insertion Sort

### Observação 8 (Caraterísticas)

Este algoritmo começa tratando a primeiro entrada `A[0]` como um array já ordenado e, em seguida, verifica a segunda entrada `A[1]` e compara-a com a primeiro. Se eles estiverem na ordem errada, é efetuada a troca. Com isto, temos `A[0]` e `A[1]` ordenados. Em seguida, repete o processo para terceira entrada, posicionando-a no lugar certo, deixando `A[0]`, `A[1]` e `A[2]` ordenados. De forma mais geral, no início do $i$-ésimo ciclo, o insertion sort tem o entradas `A[0],...,A[i-1]` ordenadas e inserte `A[i]`, ordenando as entradas `A[0], ..., A[i]`.

- simples implementação;
- requer uma quantidade constante de memória adicional;
- útil para pequenas entradas;
- muitas trocas e menos comparações.
- **Best Case**: $O(n)$
- **Average Case**: $O()\frac{n^2}{4})$
- **Worst Case**: $O(n^2)$

```
for ( i = 1 ; i < n ; i++ ) {
    for( j = i ; j > 0 ; j— )
        if ( a[j] < a[j-1] )
            swap a[j] and a[j-1]
        else break
}
```

Algoritmo 4: Algoritmo Insertion Sort

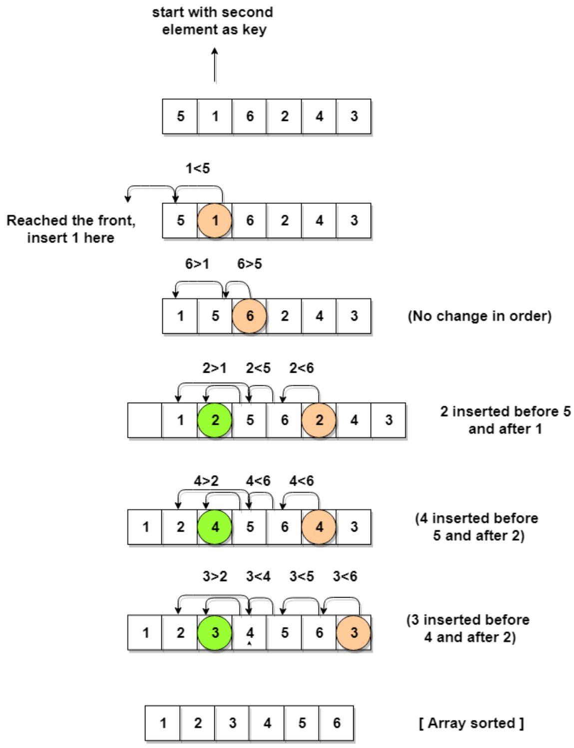



Figura 1: Visualização do algoritmo Insertion Sort.

## 2.2 Selection Sort

**Observação 9 (Caraterísticas)** *Este algoritmo encontra o menor item o coloca-o em* `A[0]`*, trocando-o por qualquer item que já esteja nessa posição. De seguida, encontra o segundo menor item e troca-o pelo item em* `A[1]`*. Este processo é repetido até que todo o array esteja ordenado. De modo mais geral, no i-ésimo ciclo, o Selection Sort encontra o i-ésimo item menor e troca-o com o item em* $A[i-1]$*.*

- simples implementação;
- requer uma quantidade constante de memória adicional;
- útil para pequenas entradas;
- **Best Case**: $O(n^2)$
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$

```
for ( i = 0 ; i < n-1 ; i++ ) {
    k = i
    for ( j = i+1 ; j < n ; j++ )
        if ( a[j] < a[k] )
            k = j
    swap a[i] and a[k]
}
```

Algoritmo 5: Algoritmo Selection Sort

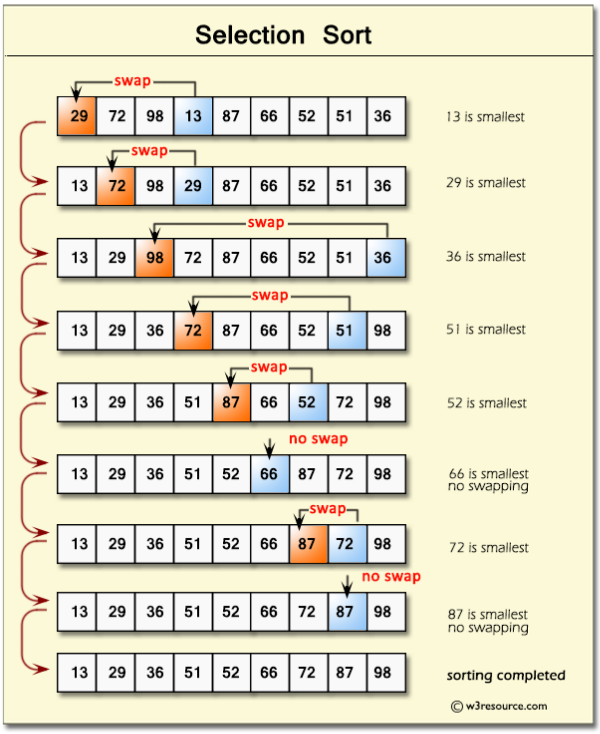



Figura 2: Visualização do algoritmo Selection Sort.

### 2.3 Bubble Sort

**Observação 10 (Caraterísticas)** *Este algoritmo começa comparando* `A[n-1]` *com* `A[n-2]` *e troca-os se estiverem na ordem errada. Em seguida, compara* `A[n-2]` *e* `A[n-3]` *e troca-os, se necessário, e assim por diante. Quando atinge* `A[0]`*, a menor entrada estará no local correto. Em seguida, começa de trás novamente, comparando pares de "vizinhos"a partir de* `A[1]`*. De modo mais geral, no i-ésimo ciclo, o Bubble Sort compara os vizinhos desde trás, trocandoo-os quando necessário. O item com o índice mais baixo que é comprado com o seu vizinho da direita é* `A[i-1]`*. Após o i-ésimo ciclo, as entradas* `A[0],...,A[i-1]` *estão na sua posição final.*

- simples implementação;
- requer uma quantidade constante de memória adicional;
- **Best Case**: $O(n)$
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$

```
for ( i = 1 ; i < n ; i++ )
    for ( j = n-1 ; j >= i ; j-- )
        if ( a[j] < a[j-1] )
            swap a[j] and a[j-1]
```

Algoritmo 6: Algoritmo Bubble Sort

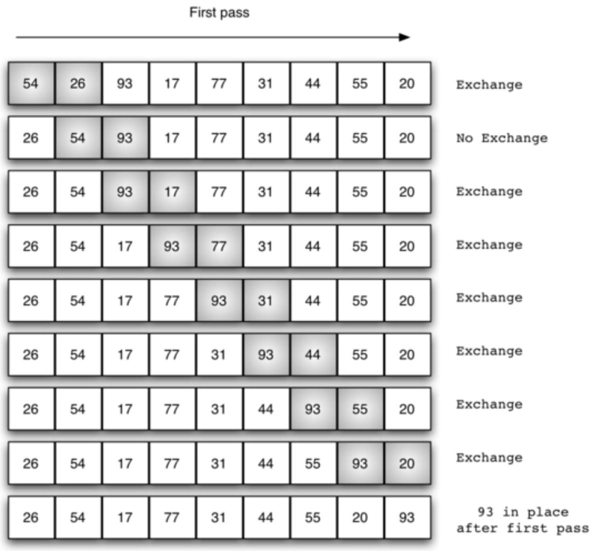



Figura 3: Visualização do algoritmo Bubble Sort.

## 2.4 Shell Sort

**Observação 11 (Caraterísticas)** *Este algoritmo primeiro classifica os elementos distantes uns dos outros e reduz sucessivamente o intervalo entre os elementos a serem classificados. É uma versão generalizada do Insertion Sort. De modo geral, o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores, onde é aplicado o Insertion Sort.*

- **Best Case**: $O(n \log n)$
- **Average Case**: $O(n \log n)$
- **Worst Case**: $O(n^2)$

```
for (int gap = n/2; gap > 0; gap /= 2)
{
    for (int i = gap; i < n; i += 1)
    {
        int temp = arr[i];
        int j;
        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            arr[j] = arr[j - gap];
        arr[j] = temp;
    }
}
```

Algoritmo 7: Algoritmo Shell Sort

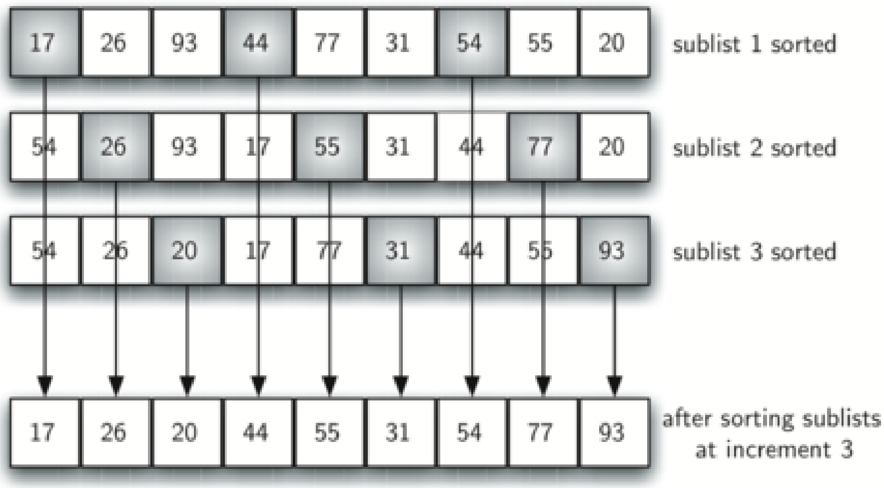



Figura 4: Visualização do algoritmo Shell Sort.

### 2.5 Merge Sort

**Observação 12 (Caraterísticas)** *Este algoritmo utiliza a técnica divisão e conquista: a solução é encontrada à custa da mesma solução mas com argumentos estruturalmente mais simples.*

- se o número de partes é $\leq 1$, terminar;
- dividir o vetor em 2 partes;
- ordenar recursivamente as duas partes;
- fundir as metades ordenadas;
- requer a utilização de um vetor adicional;
- **Best Case**: $O(n \log n)$
- **Average Case**: $O(n \log n)$
- **Worst Case**: $O(n \log n)$

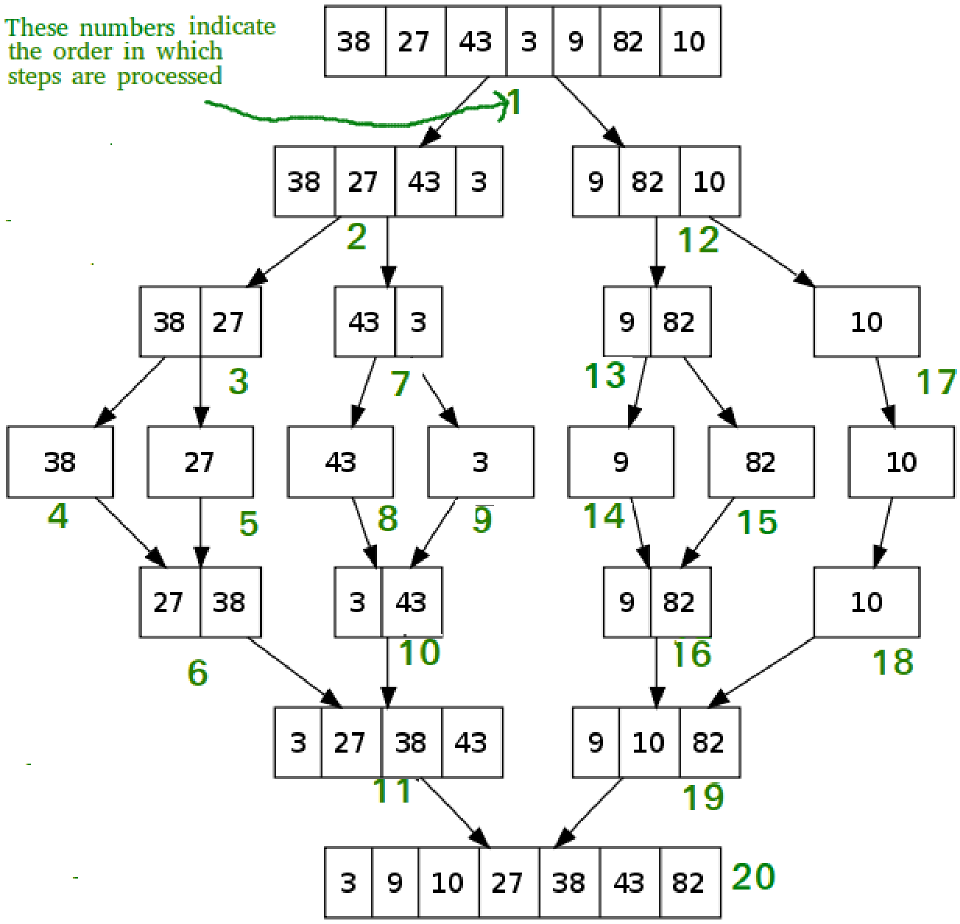



Figura 5: Visualização do algoritmo Merge Sort.

## 2.6 Quick Sort

**Observação 13 (Caraterísticas)** *Este algoritmo utiliza a técnica divisão e conquista: a solução é encontrada à custa da mesma solução mas com argumentos estruturalmente mais simples.*

- se o número de partes é $\leq 1$, terminar;
- escolher um item arbitrário — **pivot** (e.g. o último ou o 1º);
- rearranjar os restantes elementos em dois grupos:
  - elementos com menor valor do que o pivot à esquerda do pivot;
  - elementos com maior valor do que o pivot à direita do pivot;
- repetir processo anterior às listas esquerda e direita até chegar a listas com, no máximo, 1 elemento.
- requer a utilização de um vetor adicional;
- **Best Case**: $O(n \log n)$

- **Average Case**: $O(n \log n)$
- **Worst Case**: $O(n^2)$

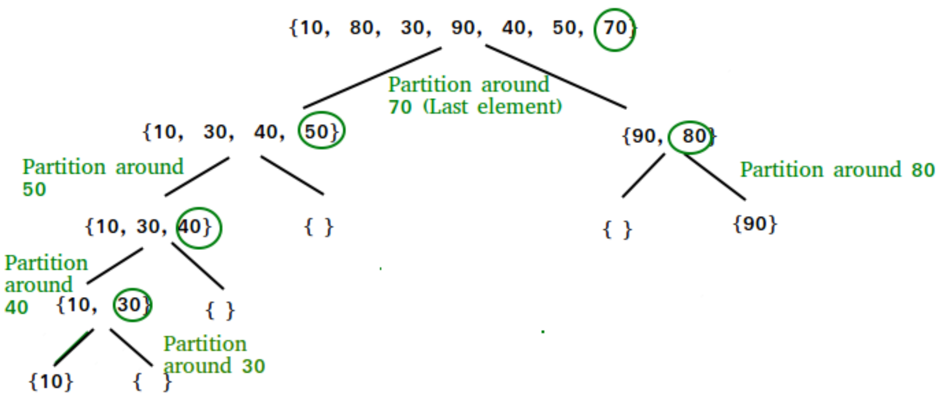



Figura 6: Visualização do algoritmo Quick Sort.

## 2.7 Sumário

**Array Sorting Algorithms**

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| Radix Sort | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| Counting Sort | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| Cubesort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

Figura 7: Sumário de algoritmos de ordenação.

# 3 Recursividade

**Definição 3.1 (Recursividade.)** *A recursão é um método de resolver um problema em que a solução depende de soluções para instâncias menores do mesmo problema.*

**Observação 14** *Um programa recursivo não se pode chamar a si próprio infinitas vezes, caso contrário nunca pararia. Deste modo, é essencial existir uma* ***condição de paragem****.*

- apesar das vantagens das implementações recursivas, é relativamente fácil escrever funções recursivas que são extremamente ineficientes.
- características básicas de um qualquer programa recursivo:
  - chama-se a si próprio para valores menores do seu argumento;
  - possui uma condição de paragem em que calcula o resultado diretamente.

**Observação 15 (Divisão e Conquista.)**

Este problemas consistem em:

- divisão em partes de tamanhos variáveis;
- divisão em mais que duas partes;
- divisão em partes que se sobrepõem;
- realizar diversas quantidades de cálculos na componente não recursiva do algoritmo.

Geralmente, estes algoritmos realizam cálculos para:

- dividir a entrada em partes;
- combinar os resultados obtidos em cada parte;
- facilitar os cálculos entre as duas chamadas.

**Observação 16 (Estratégias.)**

- divisão e conquista:
  - problema é partido em sub-problemas que se resolvem separadamente;
  - solução obtida por combinações das soluções;
- Programação dinâmica:
  - resolvem-se os problemas de pequena dimensão e guardam-se as soluções;
  - solução de um problema é obtida combinando as de problemas de menor dimensão.

# 4 Árvores

**Definição 4.1 (Árvore.)** *Uma árvore é um tipo de dados abstrato amplamente usado que simula uma estrutura de árvore hierárquica, com um valor raiz e subárvores de filhos com um nó pai, representados como um conjunto de nós vinculados.*

**Observação 17 (*Nodes*)**

- cada *node* pode possuir um número variável de nós descendentes diretos;
- *nodes* com o mesmo ascendente direto designam-se por *siblings*;
- um *node* sem ascendente designa-se por *root* (é único numa árvore).

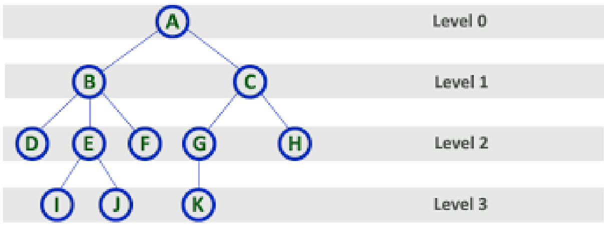



Figura 8: Níveis de uma árvore.

**Observação 18 (Terminologia)**

- **caminho**: sequência de *nodes* desde a *root* até a uma *leaf*;
  - a **dimensão** de um caminho corresponde ao seu número de nós.
- a **altura/profundidade** de uma árvore é a maior dimensão do maior caminho (número de níveis).
- **Grau** ou **aridade** de:
  - um *node*: número de *children nodes*;
  - uma árvore: maior grau dos seus *nodes*.
- **subárvore** de um *node* é a árvore com *root* nesse *node*;
- **árvores degeneradas** são árvores de grau 1 (e.g. listas).

## 4.1 Árvores Binárias

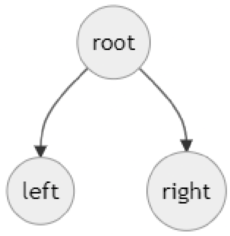
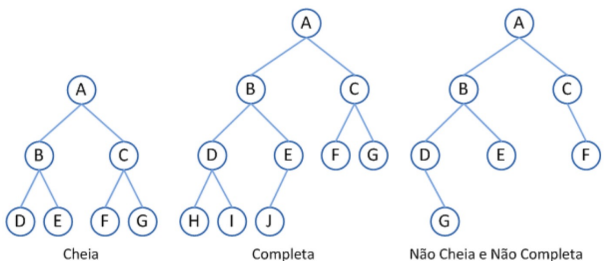



Figura 9: Árvore binária.

- árvores de grau 2;
  - cada *node* pode ter até dois descendentes;
  - contêm duas subárvores (que podem ser vazias);
  - é usual considerar a notação infixa: $< A_1; x; A_2 >$ (esquerda; *node*; direita);
  - **árvore estritamente binária**: todos os nós têm grau 0 ou 2;
  - **árvore binária equilibrada**: a diferença entre as alturas das subárvores não é superior a 1 e todas as subárvores são equilibradas;
  - uma árvore binária está **cheia** se:
    * for vazia, ou;
    * as subárvores tiverem a mesma altura e estiverem cheias (se $d$ é a altura da árvore, então esta é formada por $2^d - 1$ *nodes*);
  - uma árvore binária de altura $h$ está completa se estiver cheia até ao nível $h-1$ e todos os nós do último nível estão o mais à esquerda possível.




**Observação 19 (Pesquisa de um elemento)**

- realização de travessias em profundidade:
  - prefixa;
  - infixa;
  - sufixa;
- realização de travessias em largura.

**Travessia Prefixa, RLR**

1. visitar o *root node*;
2. travessia prefixa da subárvore esquerda;
3. travessia prefixa da subárvore direita;

**Travessia Infixa, LRR**

1. travessia infixa da subárvore esquerda;
2. visitar o *root node*;
3. travessia infixa da subárvore direita;

**Travessia Sufixa, LRR**

1. travessia sufixa da subárvore esquerda;
2. travessia sufixa da subárvore direita;
3. visitar o *root node*;

**Travessia em Largura**

1. visitar o *root node*;
2. visitar os *nodes* de cada nível, da esquerda para a direita;

## 4.2 Árvores de Pesquisa Binária

Numa árvore de pesquisa binária, os valores são inseridos após serem comparados com o valor raiz:

- valores maiores à direita;
- valores menores à esquerda;
- utiliza-se a recursão para inserir novos valores;
- a inserção é feita nos *leaf nodes*.

**Observação 20 (Remoção de Elementos)** *Existem dois métodos para a remoção de* nodes*: por **fusão**; por **cópia**.*

**Observação 21 (Remoção por cópia.)**

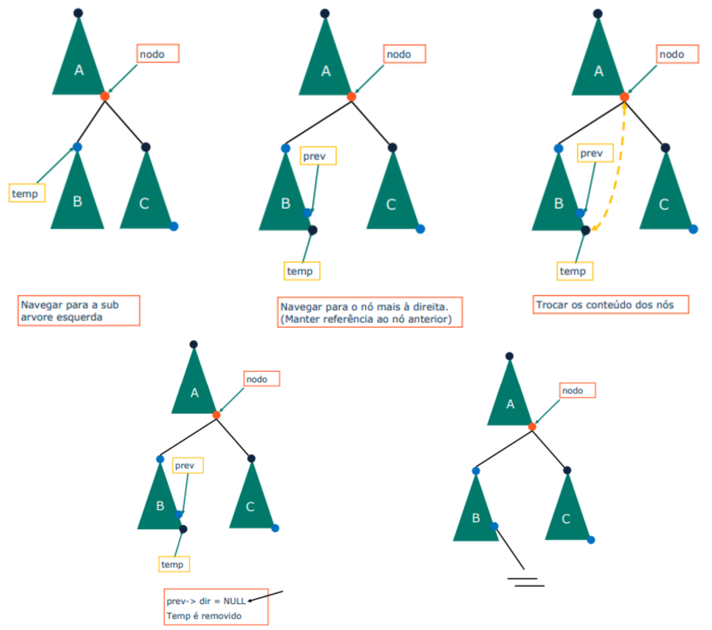



Figura 10: Remoção de *node* por cópia.

**Observação 22 (Remoção por fusão.)**

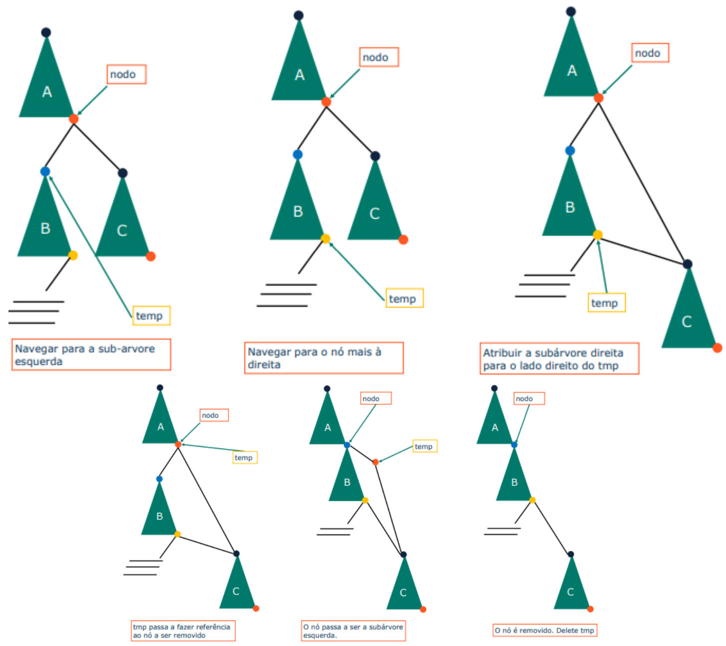



Figura 11: Remoção de *node* por fusão.