Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

Ordenação – QuickSort

Prof. Mateus Ferreira Satler

Índice

1. Introdução

- Anteriormente, introduziu-se a ideia de uso da **recursão** nos algoritmos de ordenação.
  - A recursão parte do princípio que é mais fácil resolver problemas menores.
  - Abordagem **Dividir-para-Conquistar**.
- O algoritmo **QuickSort** baseia-se no fato de que as permutações devem ser preferencialmente empregadas para pares de elementos que guardem entre si distâncias grandes, com a finalidade de se conseguir uma maior eficiência.

2. Algoritmo QuickSort

- Proposto por Hoare em 1960 e publicado em 1962.
  - Também conhecido como ordenação por partição.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
  - Provavelmente um dos mais utilizados.

2. Algoritmo QuickSort

- A ideia básica é dividir o problema de ordenar um conjunto com **n** itens em dois problemas menores.
  - Os problemas menores são ordenados independentemente.
  - Os resultados são combinados para produzir a solução final.
  - Mesma ideia do algoritmo **MergeSort**.

2. Algoritmo QuickSort

- A parte mais delicada do método é o processo de **partição**.
  - O vetor **`v[esq..dir]`** é rearranjado por meio da escolha arbitrária de um **pivô x**.
  - O vetor **v** é particionado em duas partes:
    - Parte esquerda: **chaves** $\leq$ **x**.
    - Parte direita: **chaves** $\geq$ **x**.

2. Algoritmo QuickSort

- Idealmente, o **pivô x** deveria ser selecionado de modo que aproximadamente metade dos elementos ficassem a esquerda do **pivô** e a outra metade do lado direito do **pivô.**

- Considere o caso onde o menor ou o maior elemento é escolhido como **pivô.**
  - Nesse caso um conjunto da **`esquerda`** ficaria vazio e o outro da **`direita`** com **`n-1`** elementos.

2. Algoritmo QuickSort

- A escolha do **pivô** depende da implementação.
  - **Usar o primeiro elemento do vetor:**
    - Aceitável se a entrada é aleatória.
    - Se a entrada for pré-ordenada ou estiver na ordem inversa é uma péssima escolha.
  - **Escolha aleatória:**
    - Estratégia segura de um modo geral.
    - A geração de números aleatórios não implica em um desempenho melhor no restante do algoritmo.
  - **Média das entradas:**
    - Difícil de calcular.
    - Prejudicaria o desempenho.
    - Pegar 3 elementos e calcular a média.

2. Algoritmo QuickSort

- **Funcionamento:**
  - Um elemento é escolhido como **pivô.**
  - Valores menores do que o **pivô** são colocados antes dele e os maiores, depois.
    - Supondo o **pivô** na posição `p`, esse processo cria duas partições: `[0,...,p-1]` e `[p+1,...,n-1]`.
  - Aplicar recursivamente a cada partição.
    - Até que cada partição contenha um único elemento.

2. Algoritmo QuickSort

- Implementação: **quicksort**

```
void quicksort (Item *v, int inicio, int fim) {
  if (inicio < fim) {
    int pivo = partition (v, inicio, fim);
    quicksort (v, inicio, pivo-1);
    quicksort (v, pivo+1, fim);
  }
}
```

2. Algoritmo QuickSort

- Implementação: **partition**

```
int partition(Item* v, int inicio,
              int fim) {
  int esq, dir;
  Item pivo, aux;
  esq = inicio;
  dir = fim;
  pivo = v[inicio];

  while (esq < dir) {
    while (esq <= fim &&
           v[esq].chave <= pivo.chave)
      esq++;
    while (dir >= inicio &&
           v[dir].chave > pivo.chave)
      dir--;
    if (esq < dir) {
      aux = v[esq];
      v[esq] = v[dir];
      v[dir] = aux;
    } //Fim if
  } //Fim while
  v[inicio] = v[dir];
  v[dir] = pivo;
  return dir;
}
```

3. Funcionamento

- Funcionamento da função **`partition`**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10										

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10	esq									dir

```
inicio = 0
   fim = 9
  pivô = 4
```

```
int partition (Item* v, int inicio, int fim) {
  int esq, dir;
  Item pivo, aux;
  esq = inicio;
  dir = fim;
  pivo = v[inicio];
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10	esq									dir

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10		esq								dir

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10		esq							dir	

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10		esq						dir		

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	8	2	7	6	3	5	1	9	10
n = 10		esq						dir		

```
inicio = 0
   fim = 9
  pivô = 4
```

```
if (esq < dir) {
  aux = v[esq];
  v[esq] = v[dir];
  v[dir] = aux; }
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10		esq						dir		

```
inicio = 0
   fim = 9
  pivô = 4
```

```
if (esq < dir) {
  aux = v[esq];
  v[esq] = v[dir];
  v[dir] = aux; }
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10		esq						dir		

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10			esq					dir		

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10				esq				dir		

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10				esq			dir			

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10				esq		dir				

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	7	6	3	5	8	9	10
n = 10				esq		dir				

```
inicio = 0
   fim = 9
  pivô = 4
```

```
if (esq < dir) {
  aux = v[esq];
  v[esq] = v[dir];
  v[dir] = aux; }
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	**3**	6	**7**	5	8	9	10
n = 10				esq		dir				

```
inicio = 0
   fim = 9
  pivô = 4
```

```
if (esq < dir) {
  aux = v[esq];
  v[esq] = v[dir];
  v[dir] = aux; }
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	3	6	7	5	8	9	10
n = 10				esq		dir				

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	3	6	7	5	8	9	10
n = 10					esq	dir				

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	3	6	7	5	8	9	10
n = 10					esq dir					

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	3	6	7	5	8	9	10
n = 10				dir	esq					

```
inicio = 0
   fim = 9
  pivô = 4
```

```
while (esq < dir) {
  while (esq <= fim && v[esq].chave <= pivo.chave)
    esq++;
  while (dir >= inicio && v[dir].chave > pivo.chave)
    dir--;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	3	6	7	5	8	9	10
n = 10				dir	esq					

```
inicio = 0
   fim = 9
  pivô = 4
```

```
if (esq < dir) {
  aux = v[esq];
  v[esq] = v[dir];
  v[dir] = aux; }
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	4	1	2	3	6	7	5	8	9	10
n = 10				dir	esq					

```
inicio = 0
   fim = 9
  pivô = 4
```

```
v[inicio] = v[dir];
v[dir] = pivo;
return dir;
```

3. Funcionamento

- Funcionamento da função **partition**

Posição	0	1	2	3	4	5	6	7	8	9
Chave	3	1	2	3	6	7	5	8	9	10
n = 10				dir	esq					

```
inicio = 0
   fim = 9
  pivô = 4
```

```
v[inicio] = v[dir];
v[dir] = pivo;
return dir;
```

3. Funcionamento

- Funcionamento da função **partition**

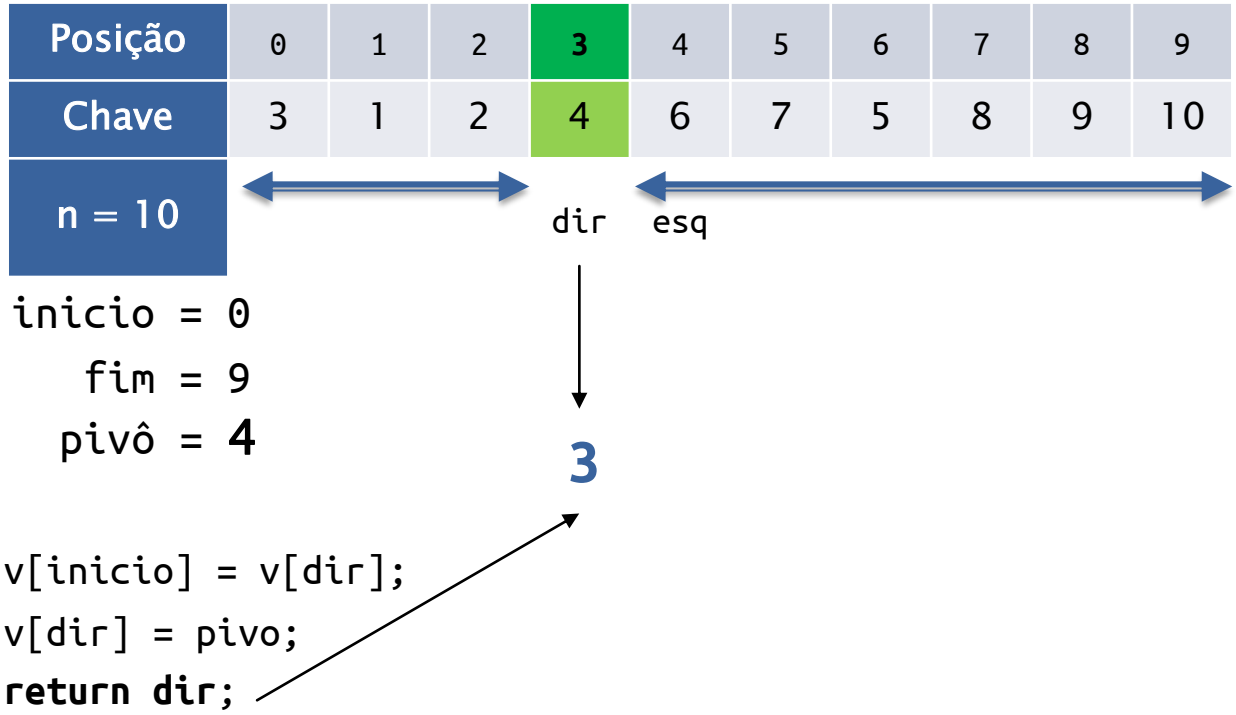
Posição	0	1	2	3	4	5	6	7	8	9
Chave	3	1	2	**4**	6	7	5	8	9	10
n = 10				dir	esq					

```
inicio = 0
   fim = 9
  pivô = 4

v[inicio] = v[dir];
v[dir] = pivo;
return dir;
```

3. Funcionamento

- Funcionamento da função **partition**



3. Funcionamento

- Funcionamento do **QuickSort**

4	8	2	7	6	3	5	1	9	10

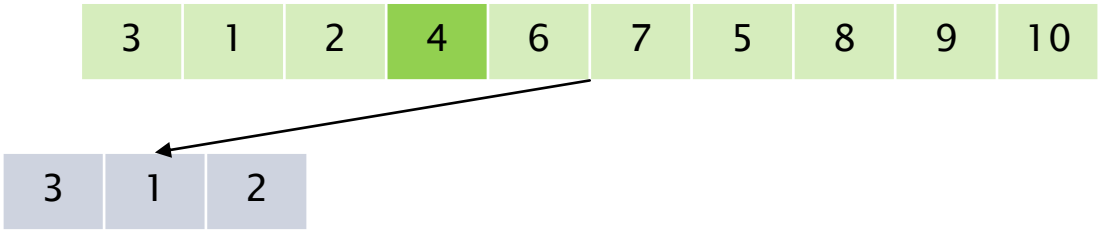
3. Funcionamento

- Funcionamento do **QuickSort**

3	1	2	4	6	7	5	8	9	10

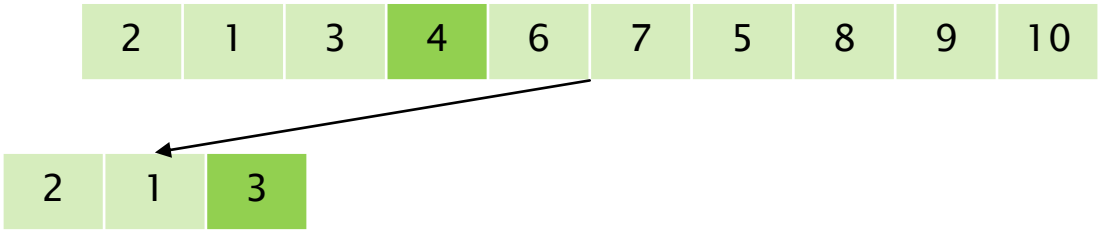
3. Funcionamento

- Funcionamento do **QuickSort**



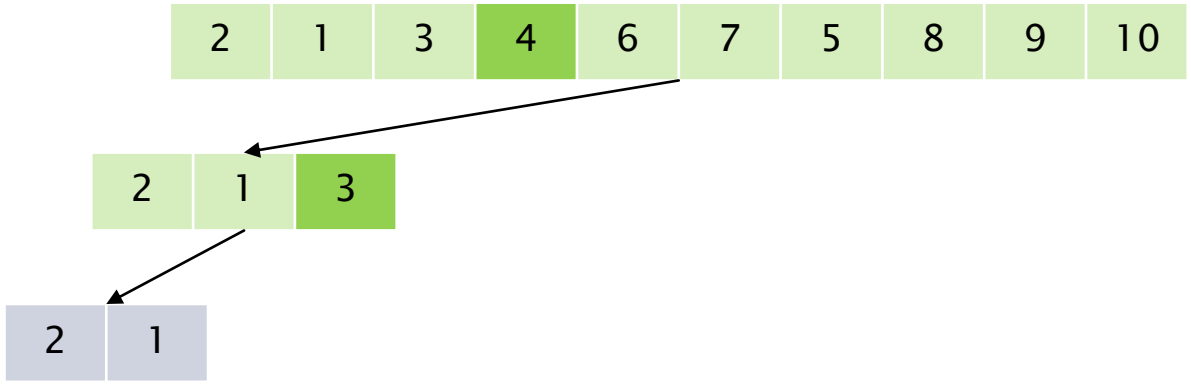
3. Funcionamento

- Funcionamento do **QuickSort**



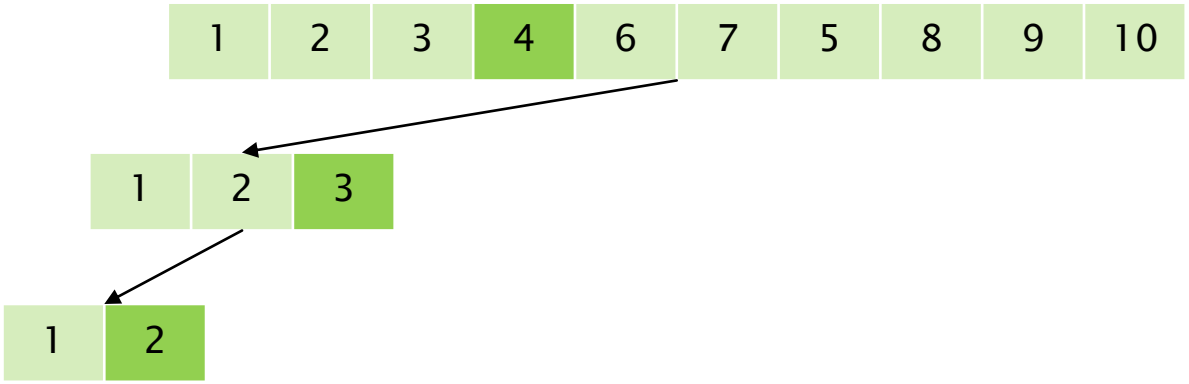
3. Funcionamento

- Funcionamento do **QuickSort**



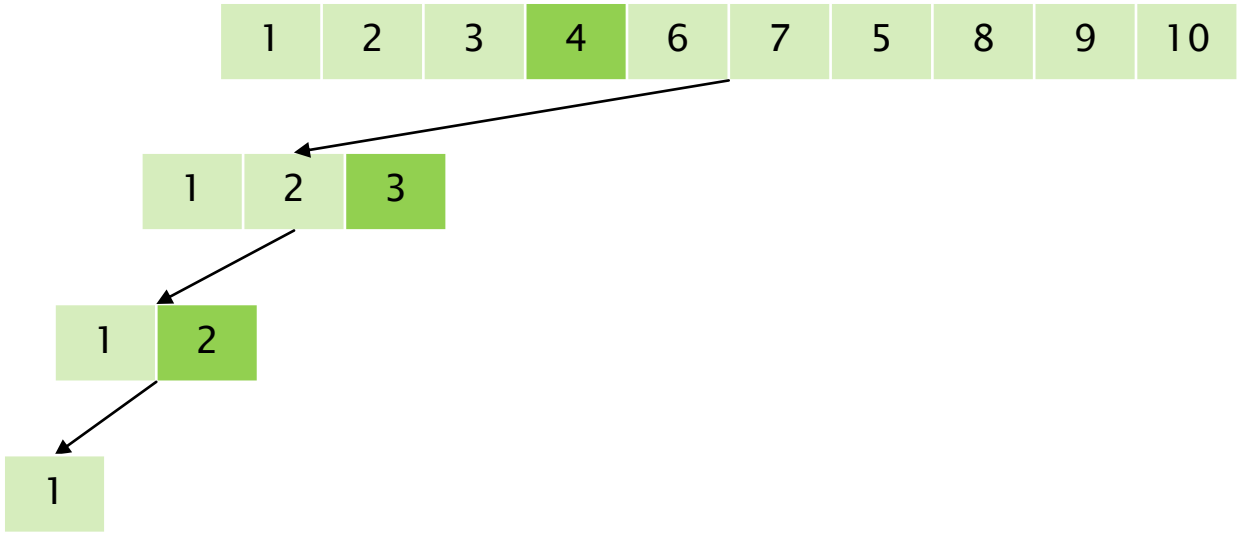
3. Funcionamento

- Funcionamento do **QuickSort**



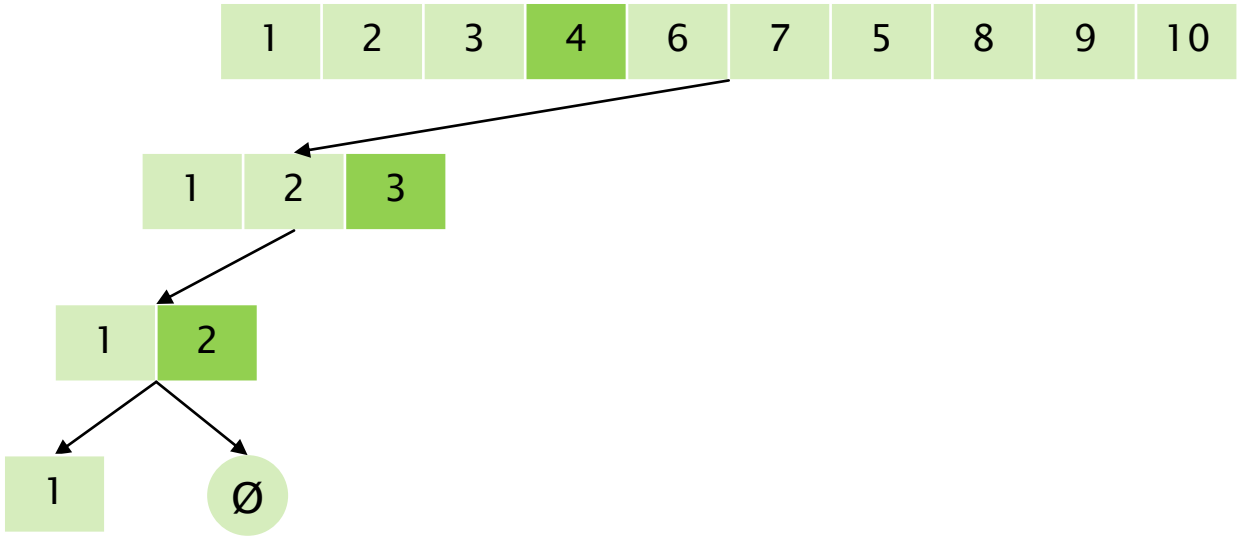
3. Funcionamento

- Funcionamento do **QuickSort**



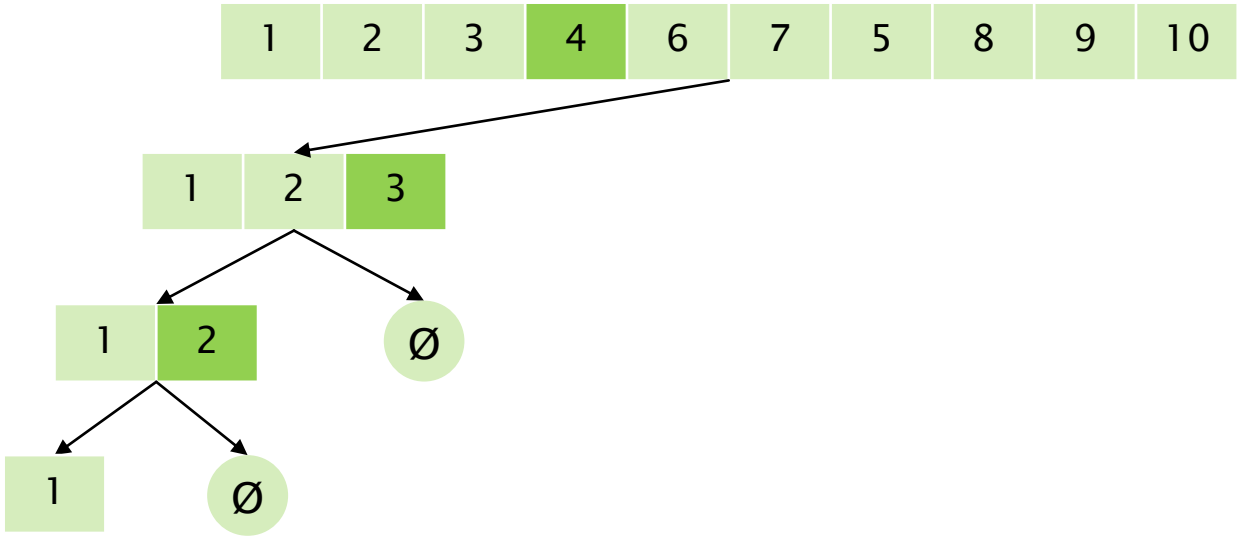
3. Funcionamento

- Funcionamento do **QuickSort**



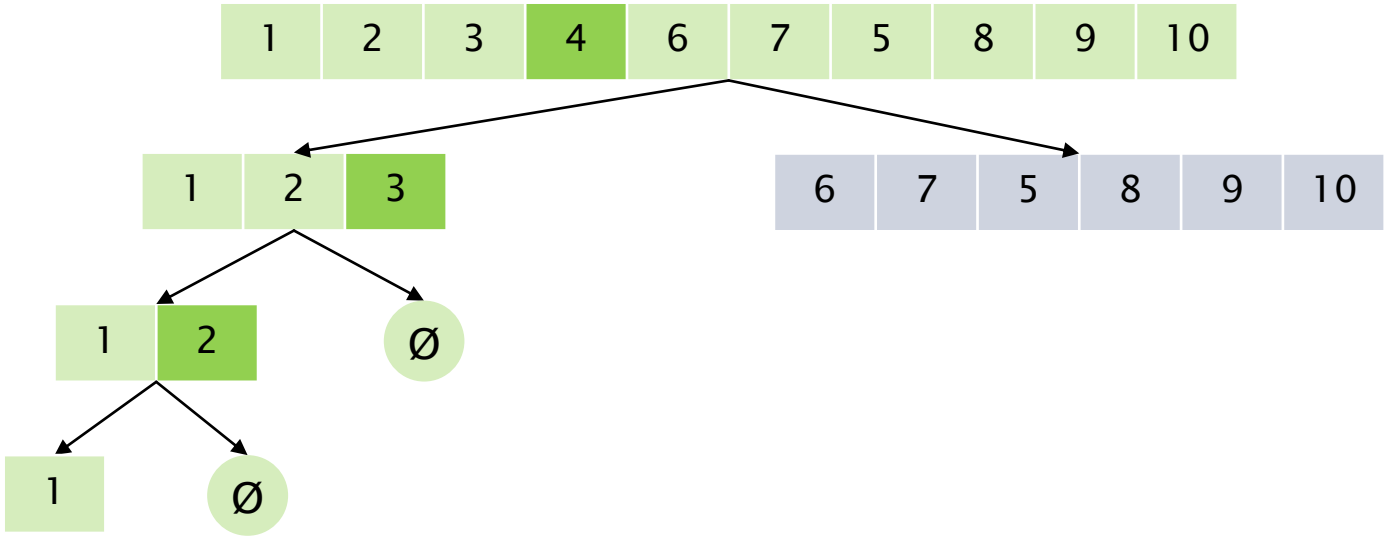
3. Funcionamento

- Funcionamento do **QuickSort**



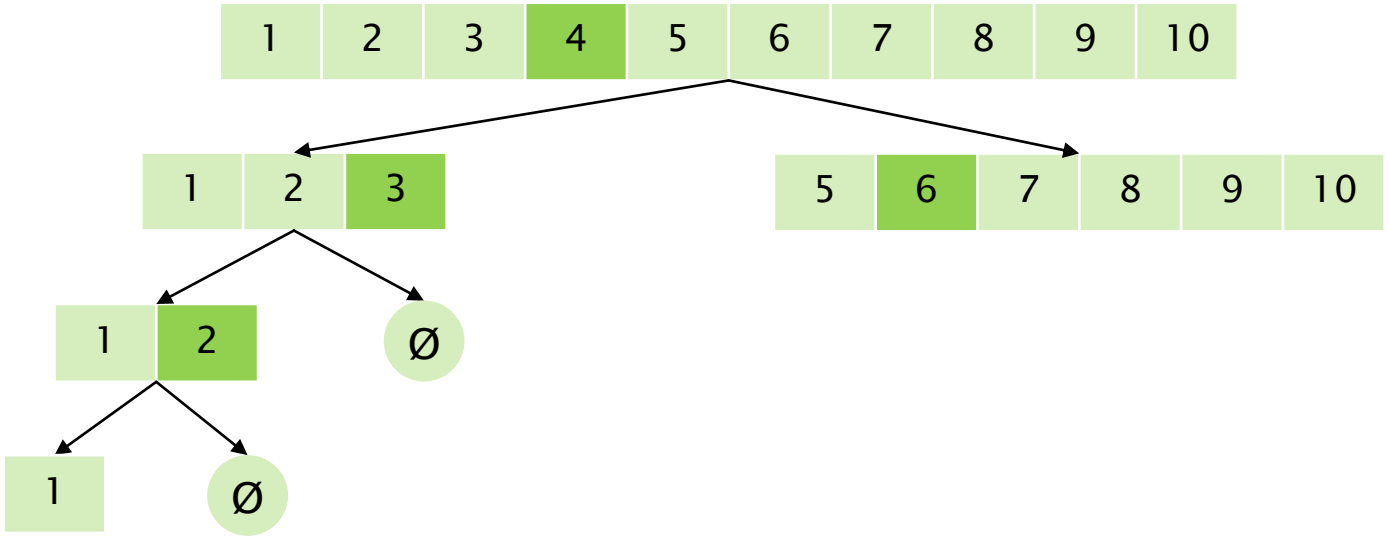
3. Funcionamento

- Funcionamento do **QuickSort**



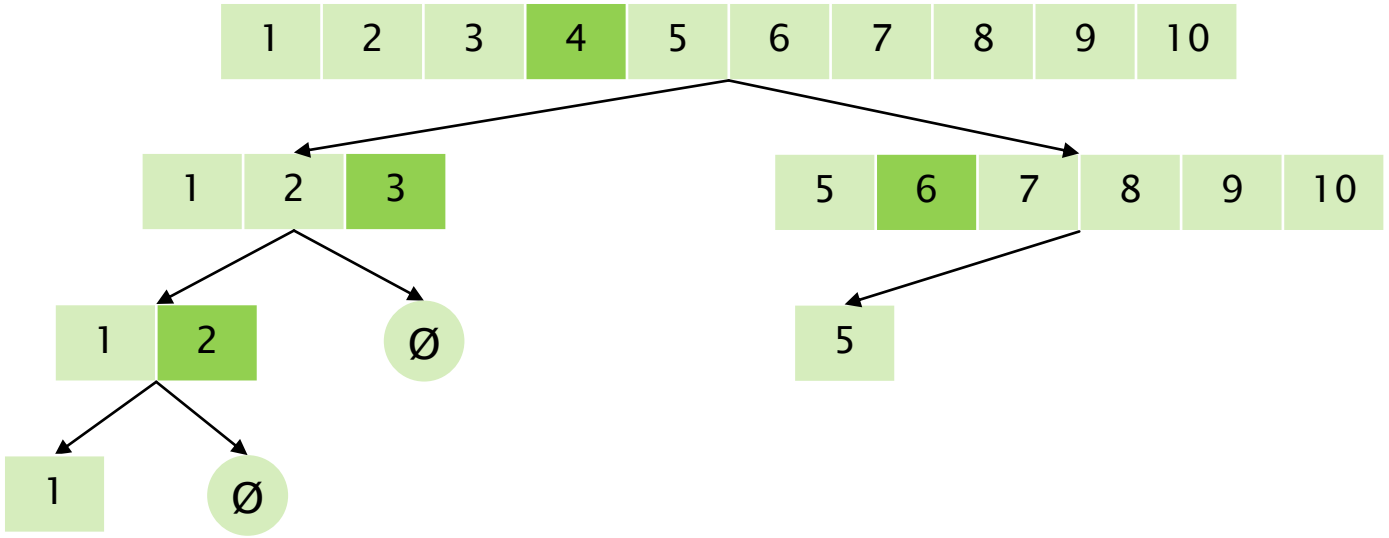
3. Funcionamento

- Funcionamento do **QuickSort**



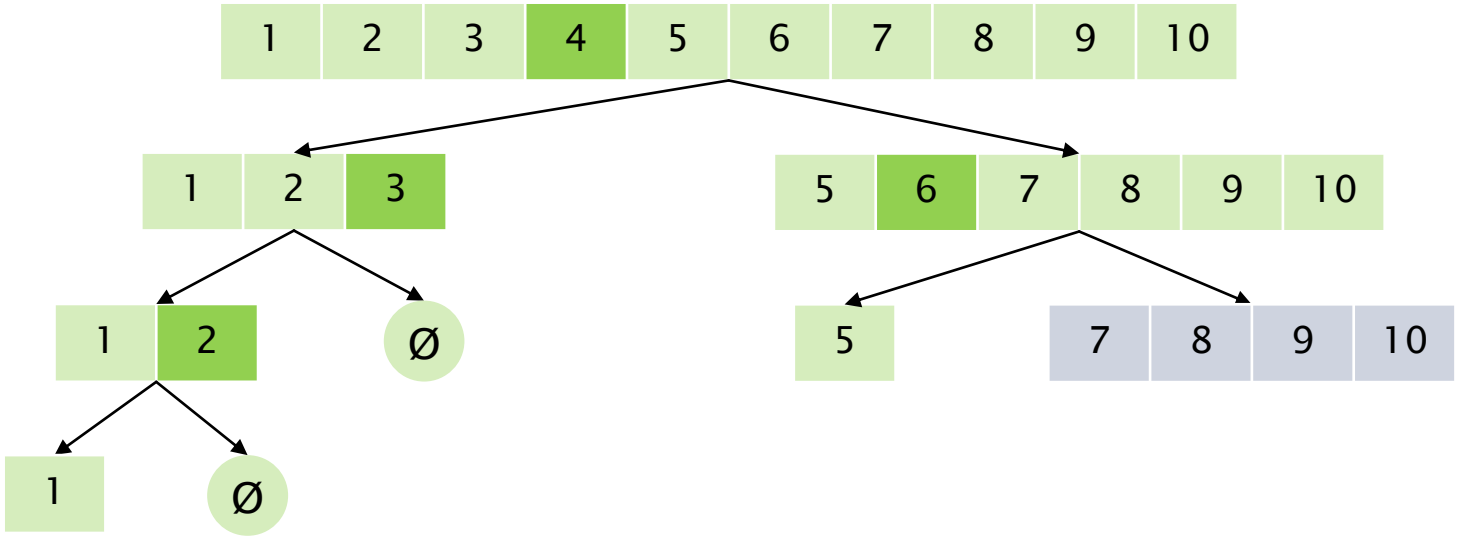
3. Funcionamento

- Funcionamento do **QuickSort**



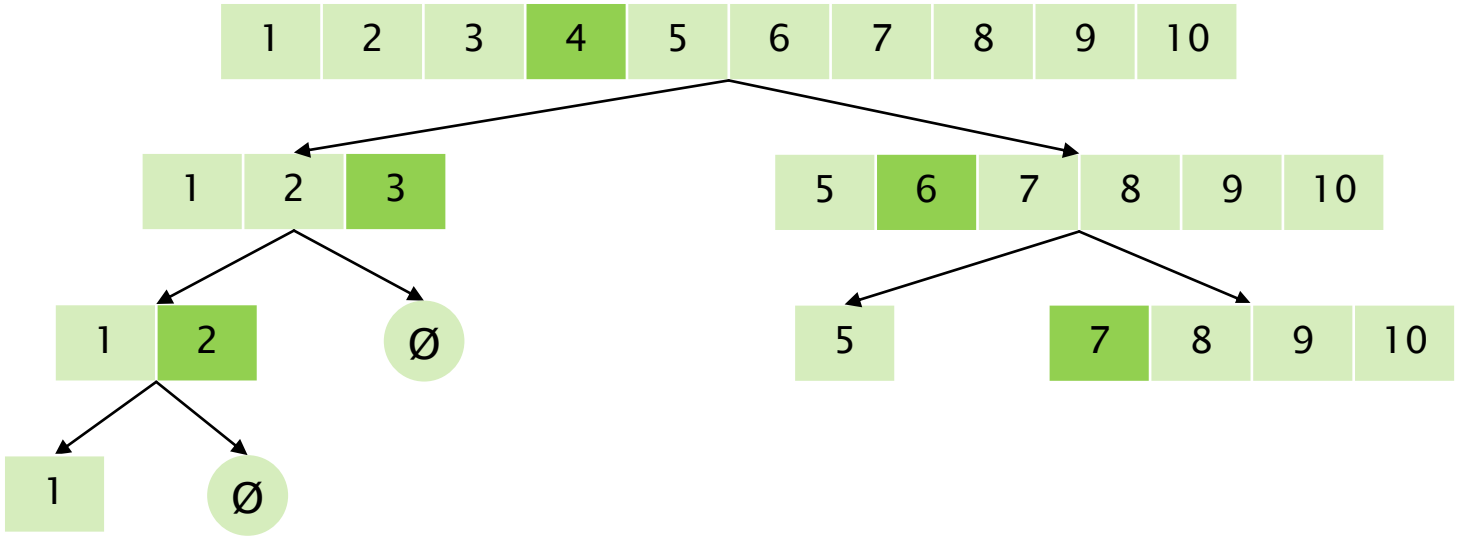
3. Funcionamento

- Funcionamento do **QuickSort**



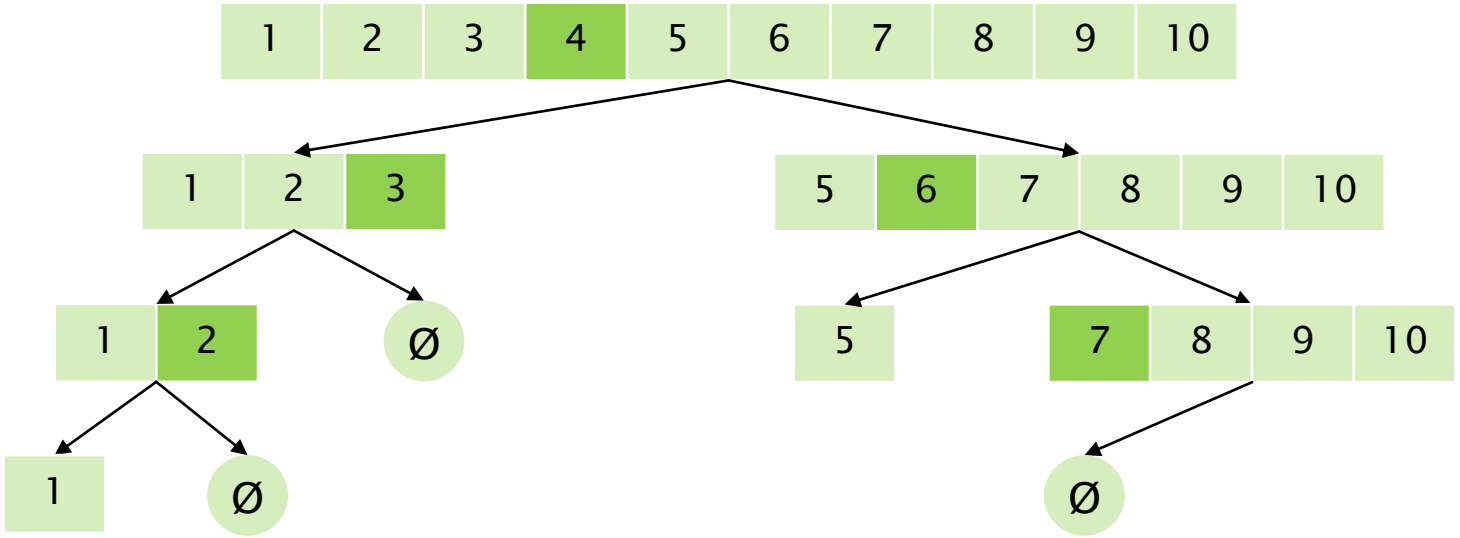
3. Funcionamento

- Funcionamento do **QuickSort**



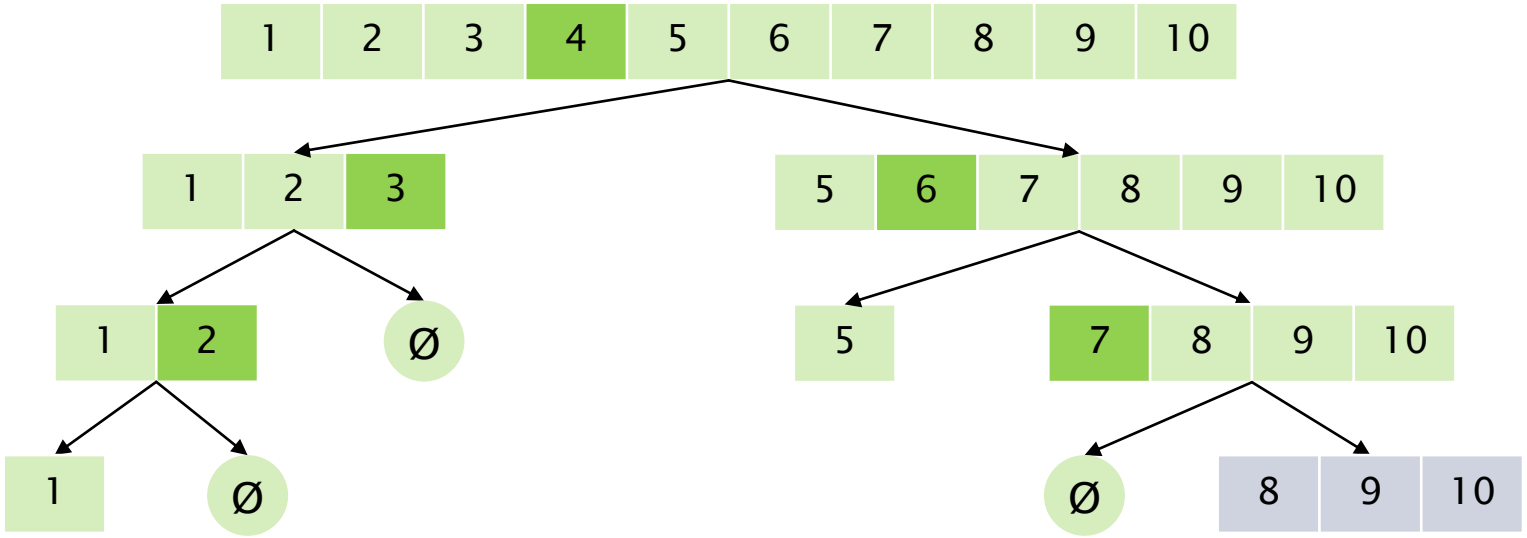
3. Funcionamento

- Funcionamento do **QuickSort**



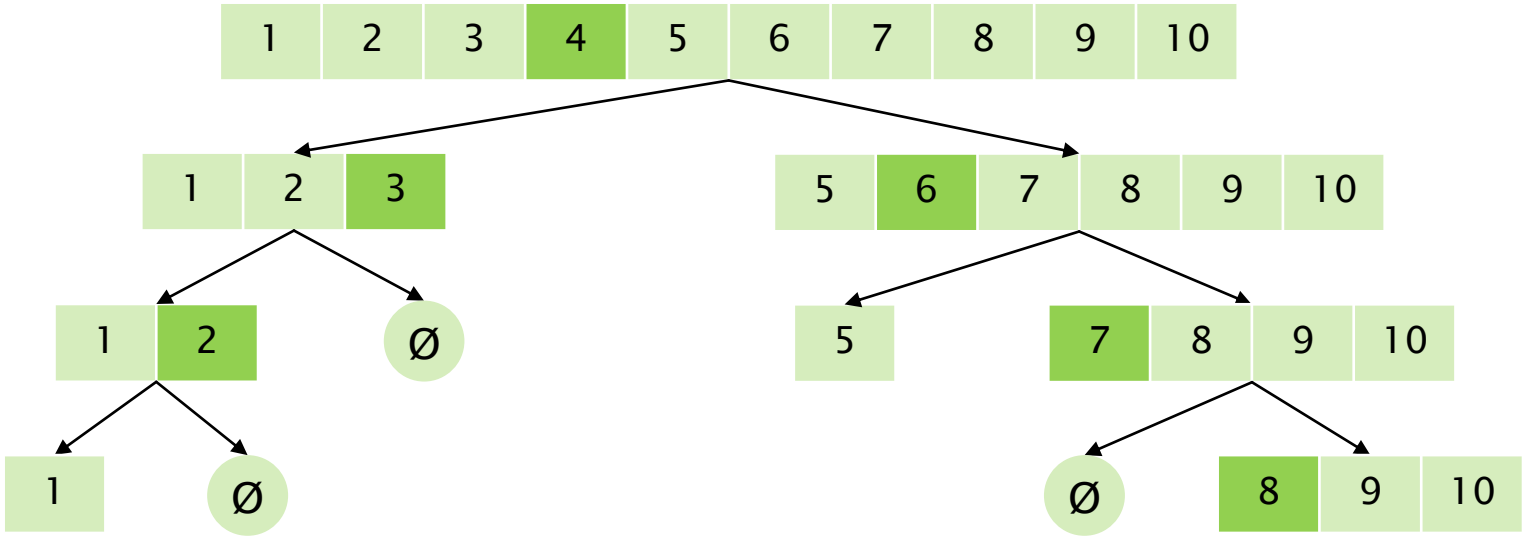
3. Funcionamento

- Funcionamento do **QuickSort**



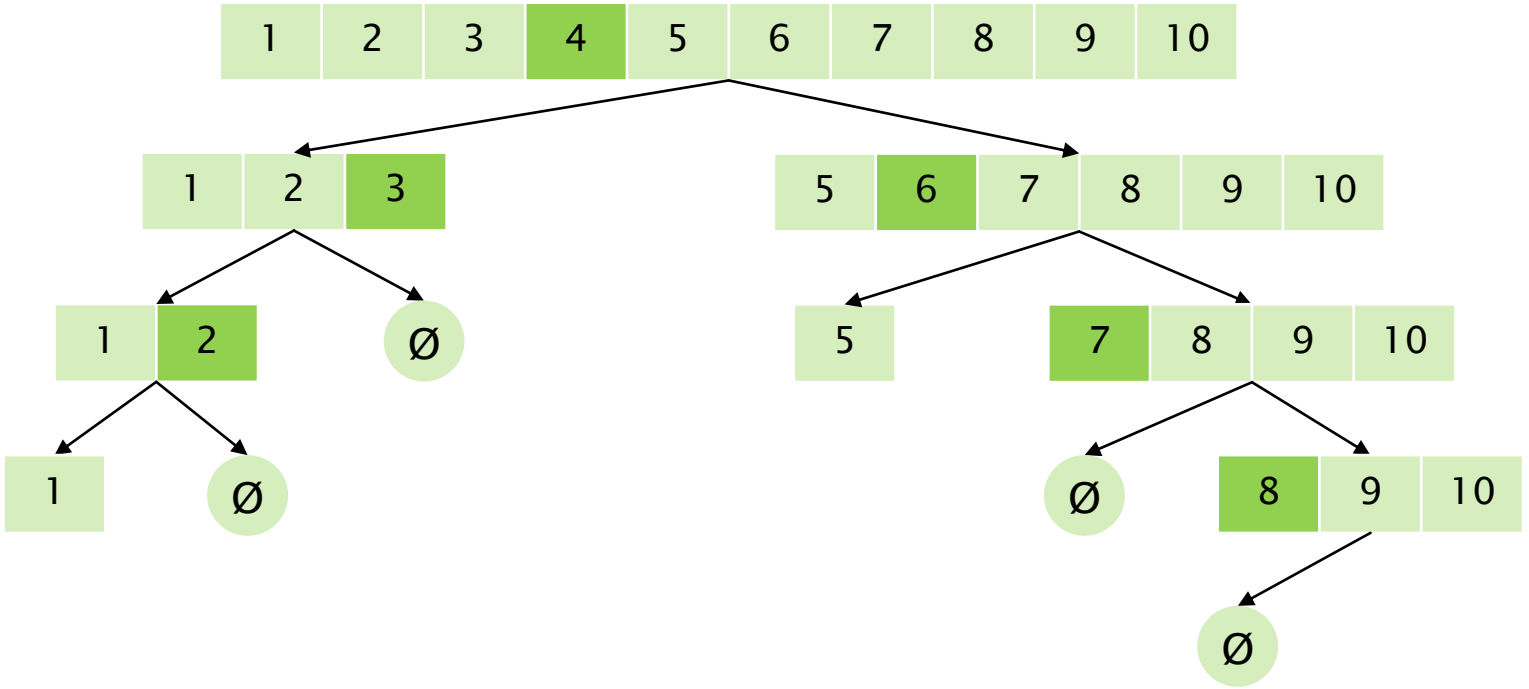
3. Funcionamento

- Funcionamento do **QuickSort**



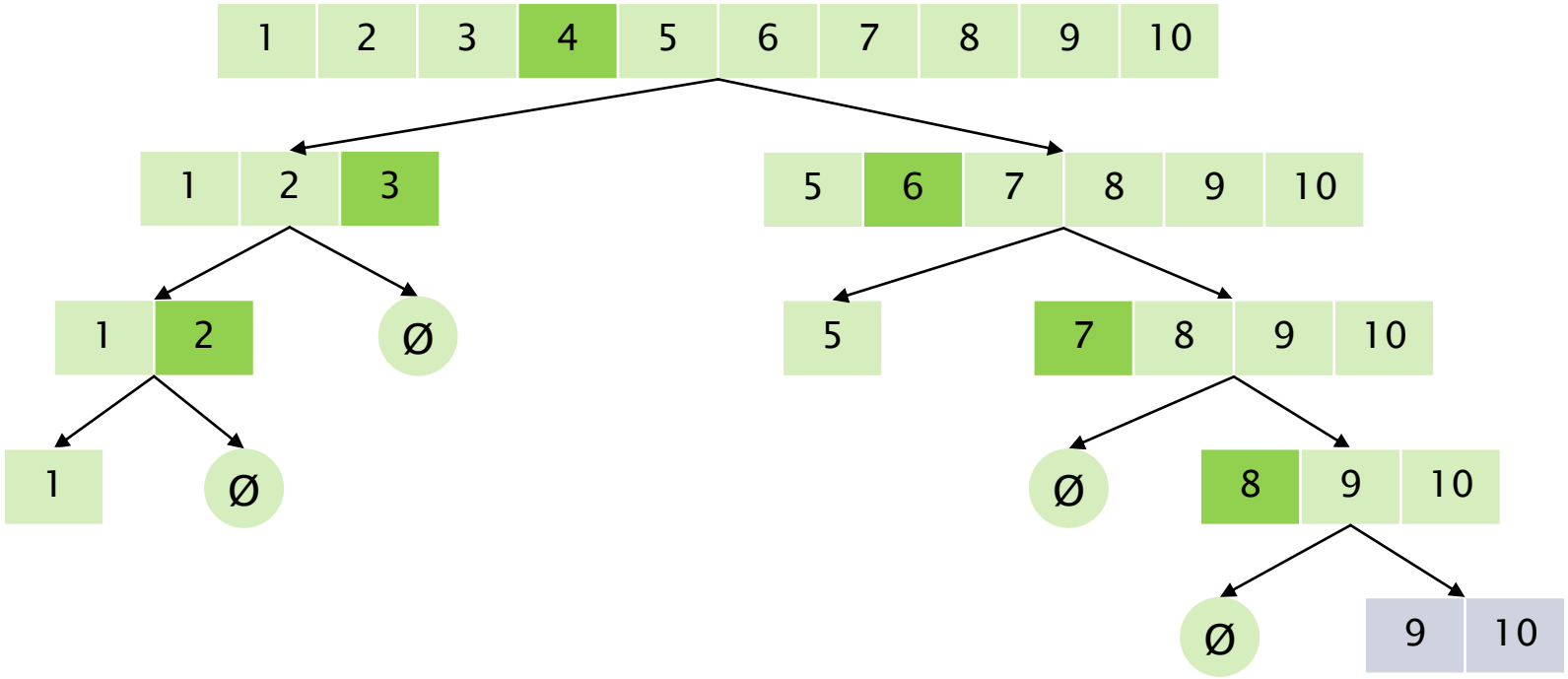
3. Funcionamento

- Funcionamento do **QuickSort**



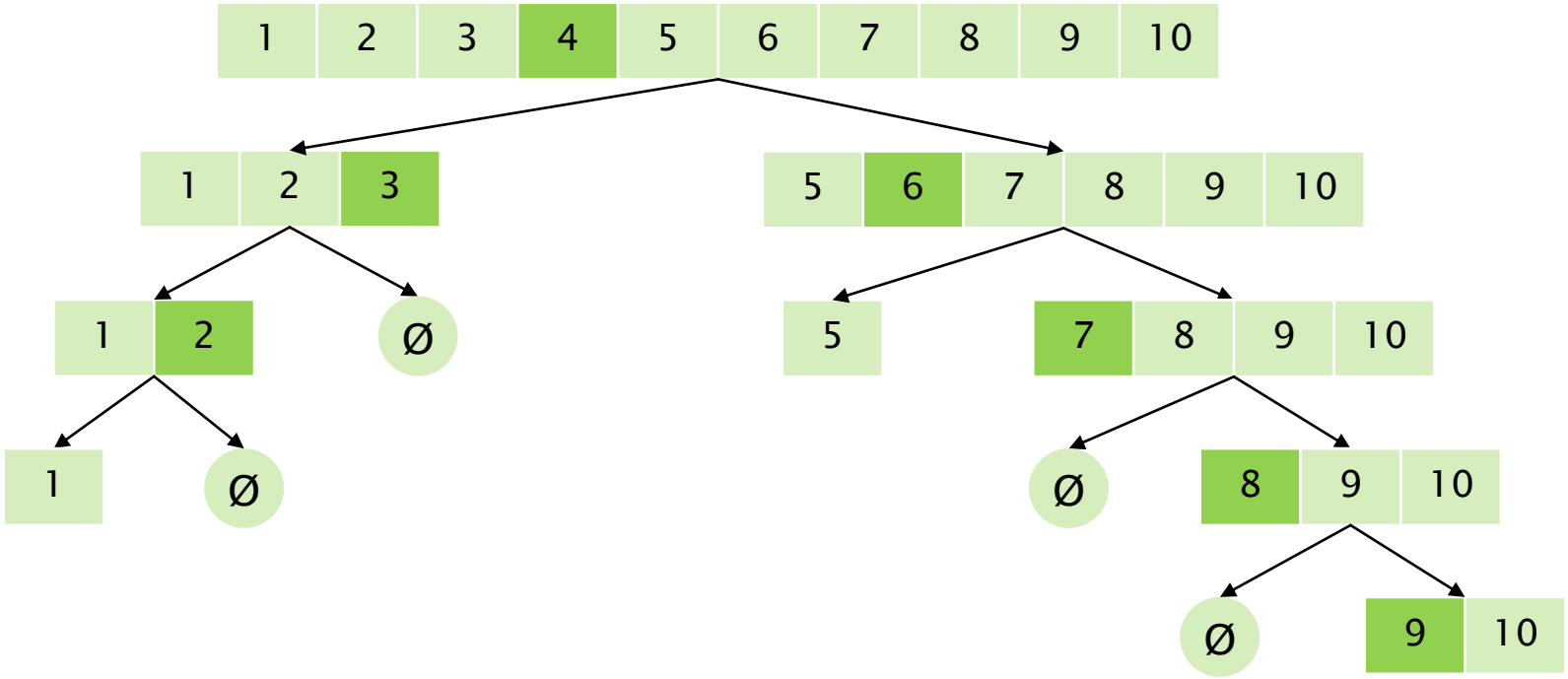
3. Funcionamento

- Funcionamento do **QuickSort**



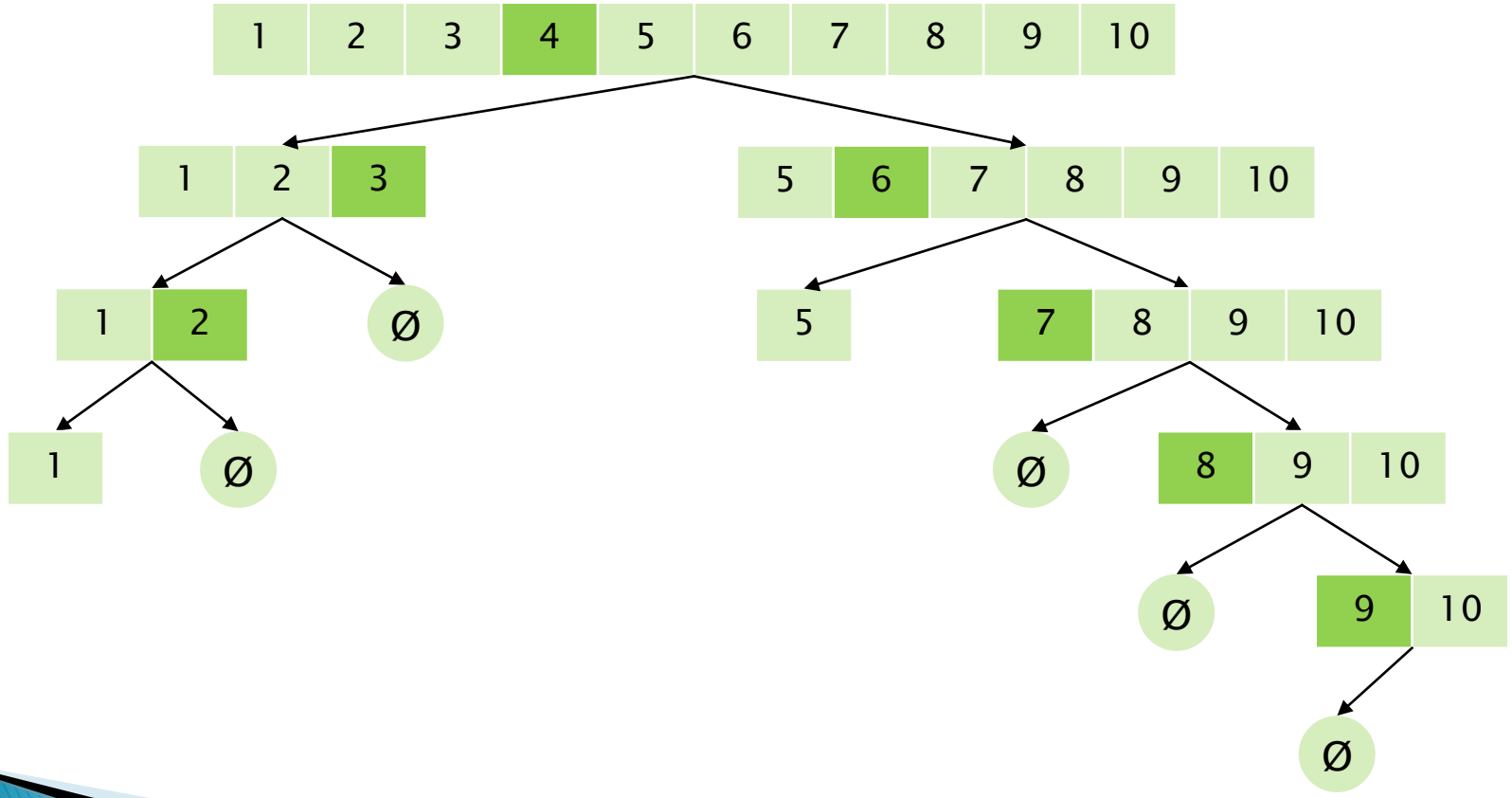
3. Funcionamento

- Funcionamento do **QuickSort**



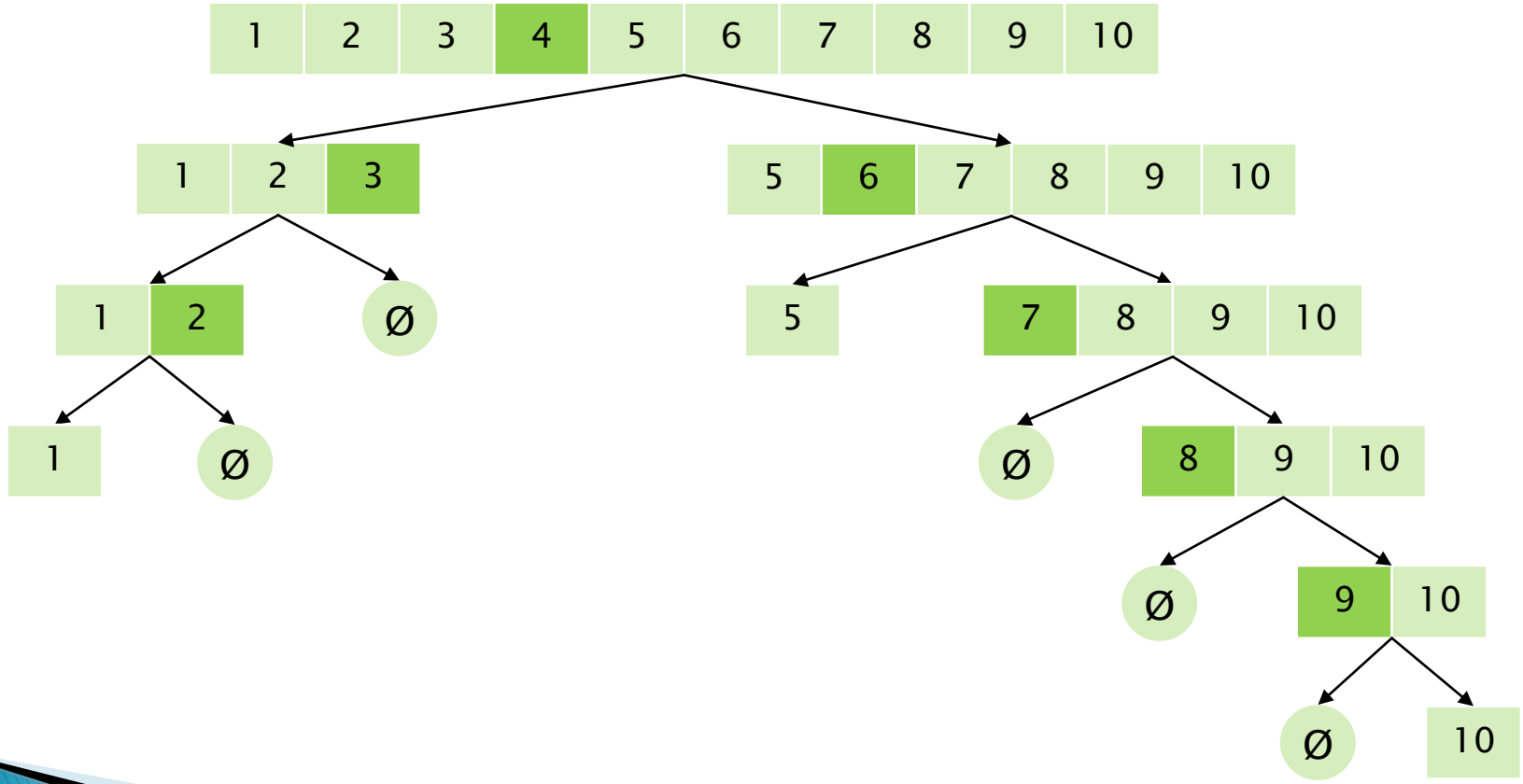
3. Funcionamento

- Funcionamento do **QuickSort**



3. Funcionamento

- Funcionamento do **QuickSort**



4. Análise

- Seja **C(n)** a função que conta o número de comparações.
  - Pior caso: **C(n) = O(n²)**
    - O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
    - Isto faz com que o procedimento Ordena seja chamado recursivamente **n** vezes, eliminando apenas um item em cada chamada.
  - O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
    - Para isso basta escolher três itens quaisquer do vetor e usar a mediana dos três como pivô.

4. Análise

- Seja **C(n)** a função que conta o número de comparações.
  - Melhor caso:
    - $C(n) = 2C(n/2) + n =$ **n log n**
    - Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
  - Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):
    - $C(n) \approx 1{,}386n \log n - 0{,}846n$,
    - Isso significa que em média o tempo de execução do **quicksort** é **O(n log n)**.

4. Análise

- **Vantagens:**
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de **n log n** comparações em média para ordenar **n** itens.
- **Desvantagens:**
  - Tem um pior caso **O(n²)** comparações.
  - Sua implementação é muito delicada e difícil:
    - Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é estável.

5. Referências

- Material de aula dos Profs. Luiz Chaimowicz e Raquel O. Prates, da UFMG: https://homepages.dcc.ufmg.br/~glpappa/aeds2/AEDS2.1%20Conceitos%20Basicos%20TAD.pdf
- Horowitz, E. & Sahni, S.; Fundamentos de Estruturas de Dados, Editora Campus, 1984.
- Wirth, N.; Algoritmos e Estruturas de Dados, Prentice/Hall do Brasil, 1989.
- Material de aula do Prof. José Augusto Baranauskas, da USP: https://dcm.ffclrp.usp.br/~augusto/teaching.htm
- Material de aula do Prof. Rafael C. S. Schouery, da Unicamp: https://www.ic.unicamp.br/~rafael/cursos/2s2019/mc202/index.html