Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

Ordenação – MergeSort

Prof. Mateus Ferreira Satler

Índice

1. Introdução

- Anteriormente, foram apresentados três algoritmos de ordenação com complexidade $O(n^2)$:
  - *Bubblesort*
  - *SelectionSort*
  - *InsertionSort*
- E um algoritmo um pouco melhor:
  - *ShellSort*

1. Introdução

- É possível também utilizar a ideia da **recursão** para realizar a ordenação.
  - A recursão parte do princípio que é mais fácil resolver problemas menores.
  - Para certos problemas, podemos dividi-lo em duas ou mais partes.

1. Introdução

- Abordagem **Dividir-para-Conquistar**
  - Método em Computação que consiste em:
    - Dividir a entrada em conjuntos menores.
    - Resolver cada instância menor de maneira recursiva.
    - Reunir as soluções parciais para compor a solução do problema original.

2. Algoritmo MergeSort

- Também conhecido como ordenação por intercalação.
  - Algoritmo recursivo que usa a ideia de dividir para conquistar para ordenar os dados.
  - Parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um com muitos.
  - O algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combina-los por meio de intercalação (merge).

2. Algoritmo MergeSort

- **Funcionamento:**
  - Divide, recursivamente, o vetor em duas partes.
    - Continua até cada parte ter apenas um elemento.
  - Em seguida, combina dois vetores de forma a obter um vetor maior e ordenado.
    - A combinação é feita intercalando os elementos de acordo com o sentido da ordenação (crescente ou decrescente).
  - Este processo se repete até que exista apenas um vetor.

2. Algoritmo MergeSort

- Algoritmo usa 2 funções:
  - **mergesort**: divide os dados em vetores cada vez menores.
  - **merge**: intercala os dados de forma ordenada em um vetor maior.

2. Algoritmo MergeSort

- Ordenação: **`mergesort`**
  - Recebemos um vetor de tamanho **n** com limites:
    - O vetor começa na posição **`v[inicio]`**
    - O vetor termina na posição **`v[fim]`**
  - Dividimos o vetor em dois sub-vetores de tamanho **n/2**.
  - O caso base é um vetor de tamanho **0** ou **1**.

2. Algoritmo MergeSort

- Implementação: **mergesort**

```
void mergesort (Item *v, int inicio, int fim) {

  if (inicio < fim) {
    int meio = (inicio + fim) / 2;

    /* divisão */
    mergesort(v, inicio, meio);
    mergesort(v, meio + 1, fim);

    /* conquista */
    merge(v, inicio, meio, fim);
  }
}
```

2. Algoritmo MergeSort

- Intercalação: **`merge`**
  - Os dois sub-vetores estão armazenados em v:
    - O primeiro nas posições de **`início`** até **`meio`**
    - O segundo nas posições de **`meio + 1`** até **`fim`**
  - Precisamos de um vetor auxiliar do tamanho do vetor.

2. Algoritmo MergeSort

- Implementação: **merge**

```
void merge (Item *v, int inicio, int meio, int fim) {

  Item *aux = (Item*) malloc((fim-inicio+1)*sizeof(Item));
  int i = inicio, j = meio + 1, k = 0;

  /* intercala */
  while (i <= meio && j <= fim)
    if (v[i].chave <= v[j].chave)
      aux[k++] = v[i++];
    else
      aux[k++] = v[j++];
```

2. Algoritmo MergeSort

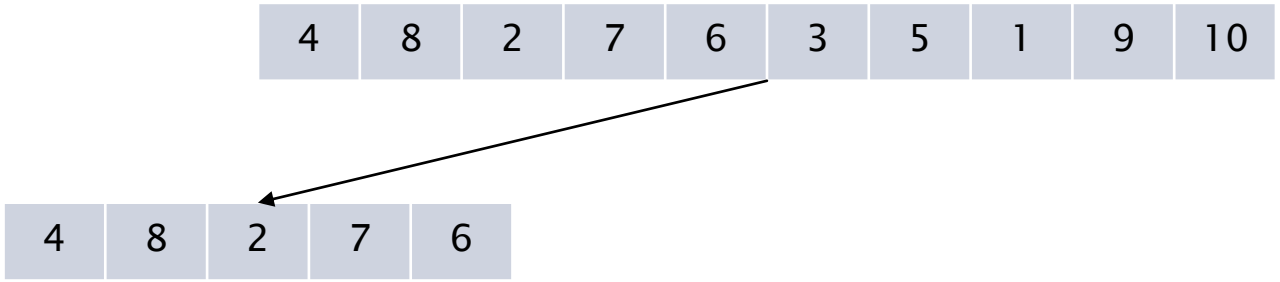
- Implementação: **`merge`**

```
  /* copia o resto do sub-vetor que não terminou */
  while (i <= meio)
    aux[k++] = v[i++];
  while (j <= fim)
    aux[k++] = v[j++];

  /* copia de volta para v */
    for (i = inicio, k = 0; i <= fim; i++, k++)
      v[i] = aux[k];

    free(aux);
}
```

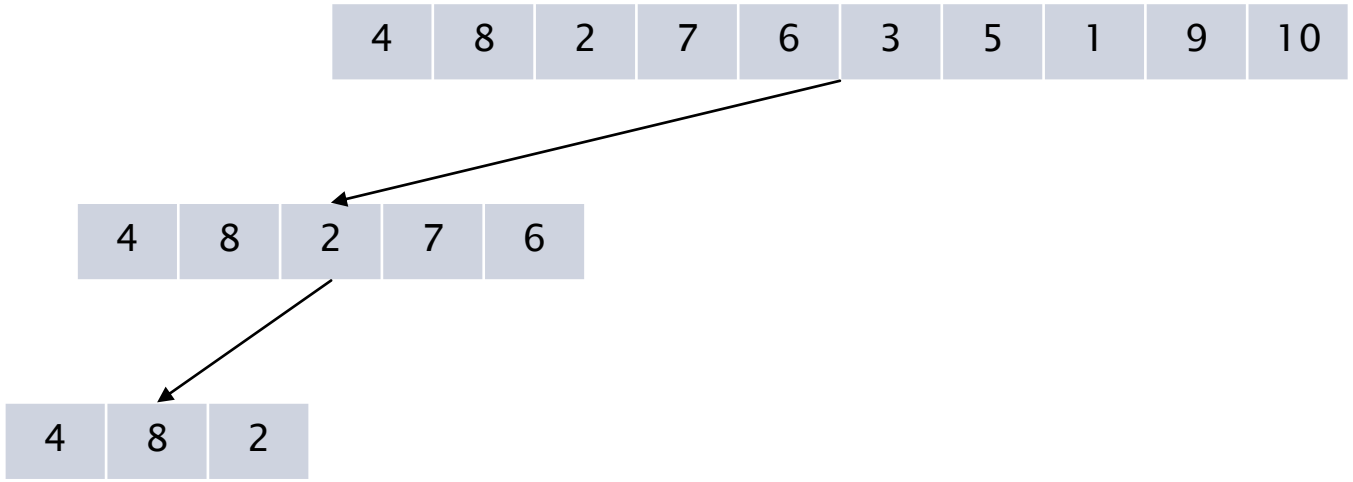
3. Funcionamento

4	8	2	7	6	3	5	1	9	10

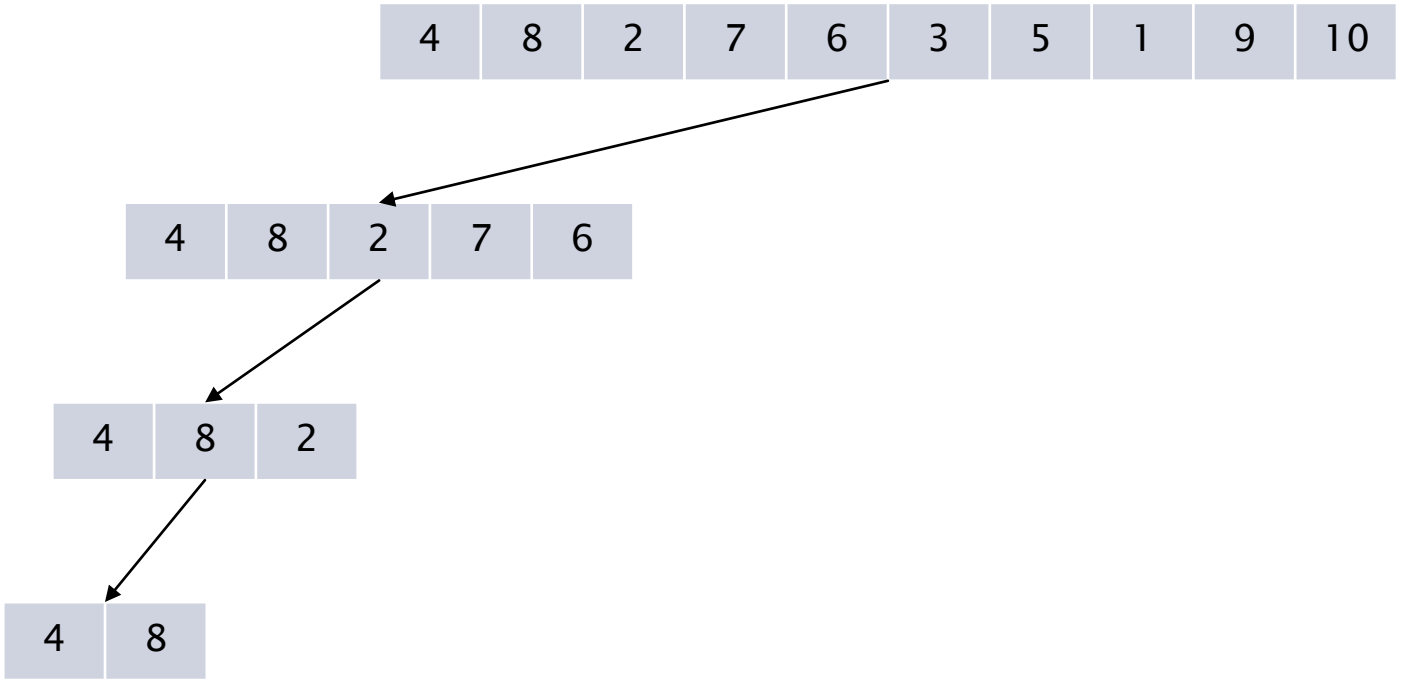
3. Funcionamento



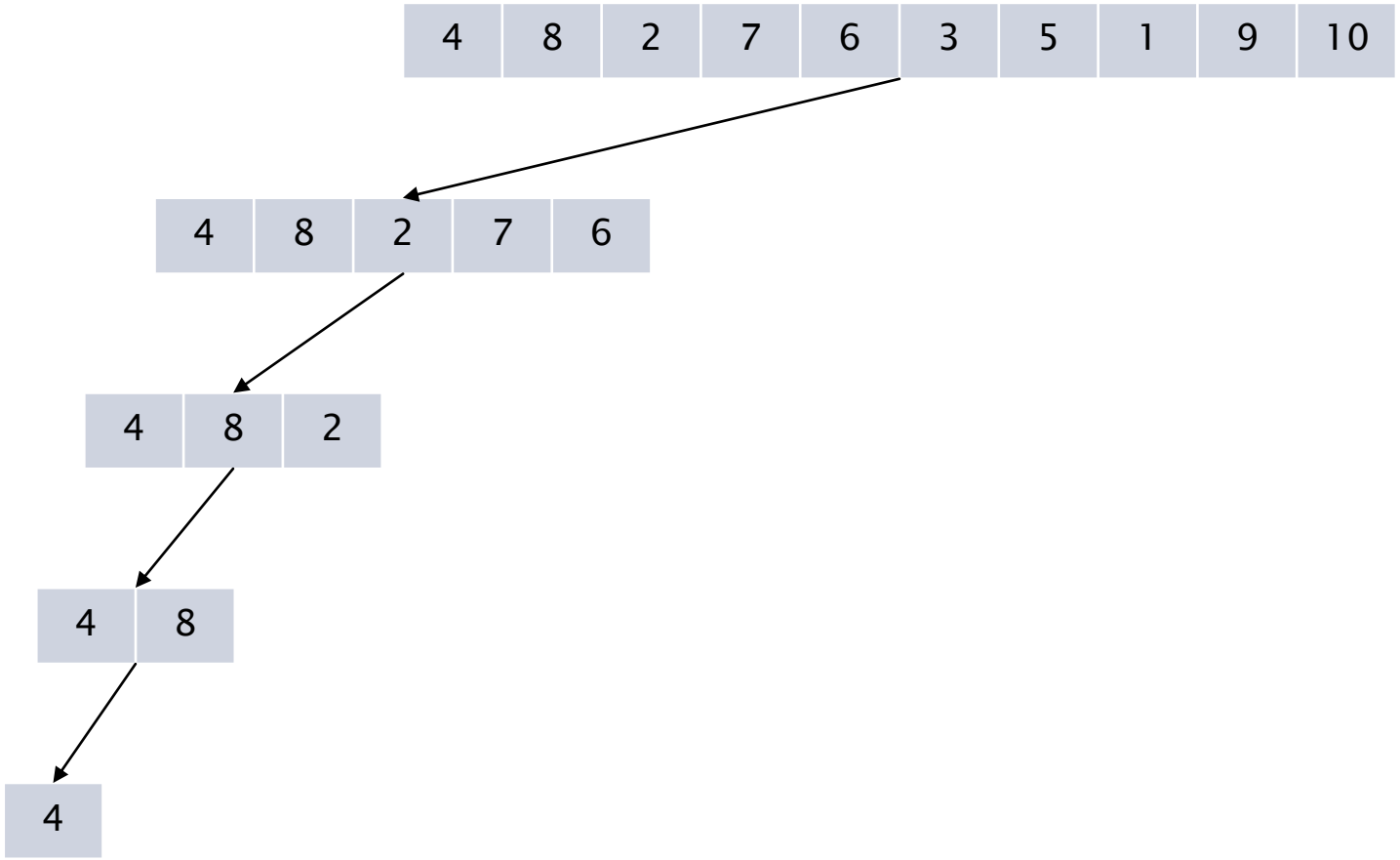
3. Funcionamento



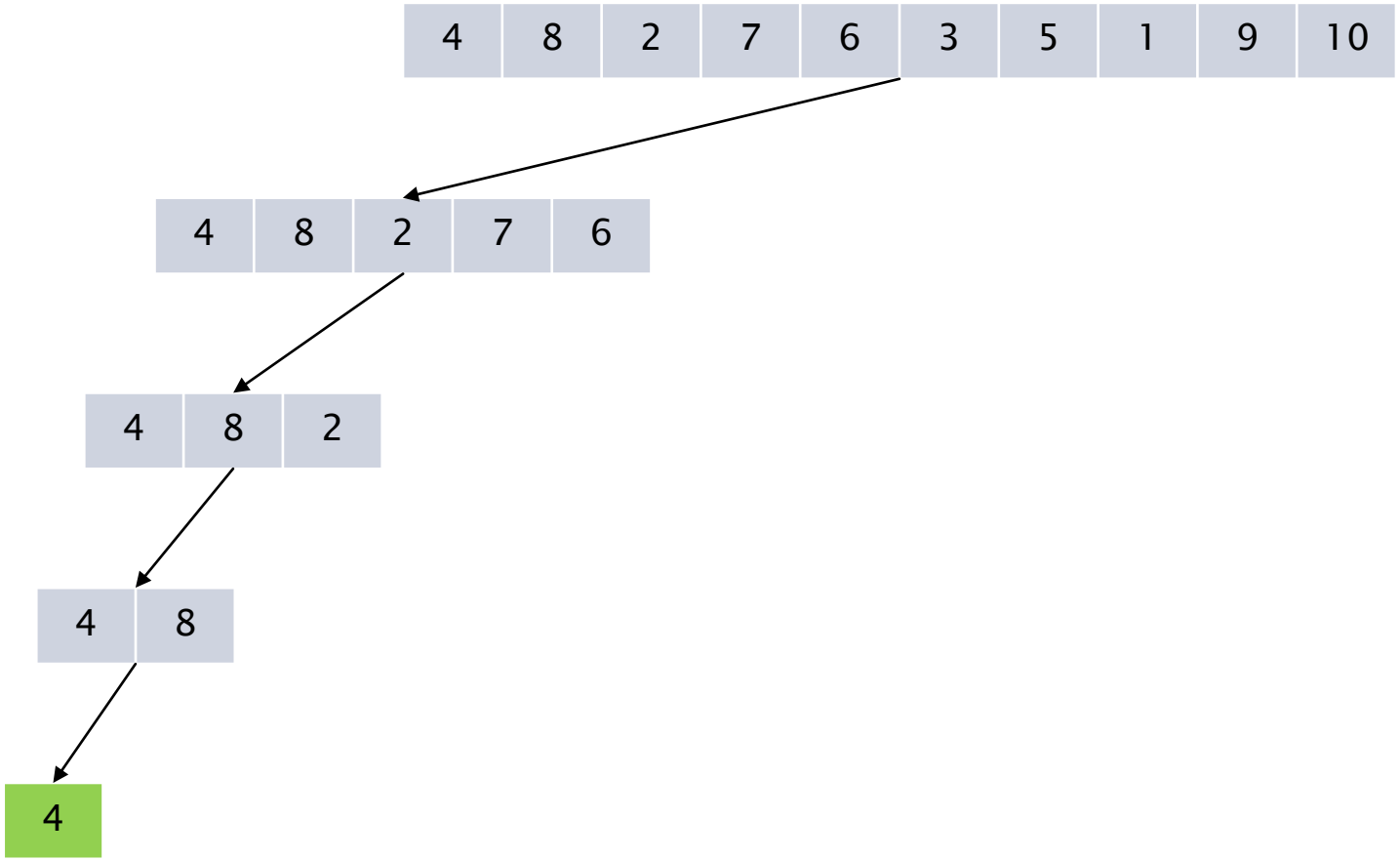
3. Funcionamento



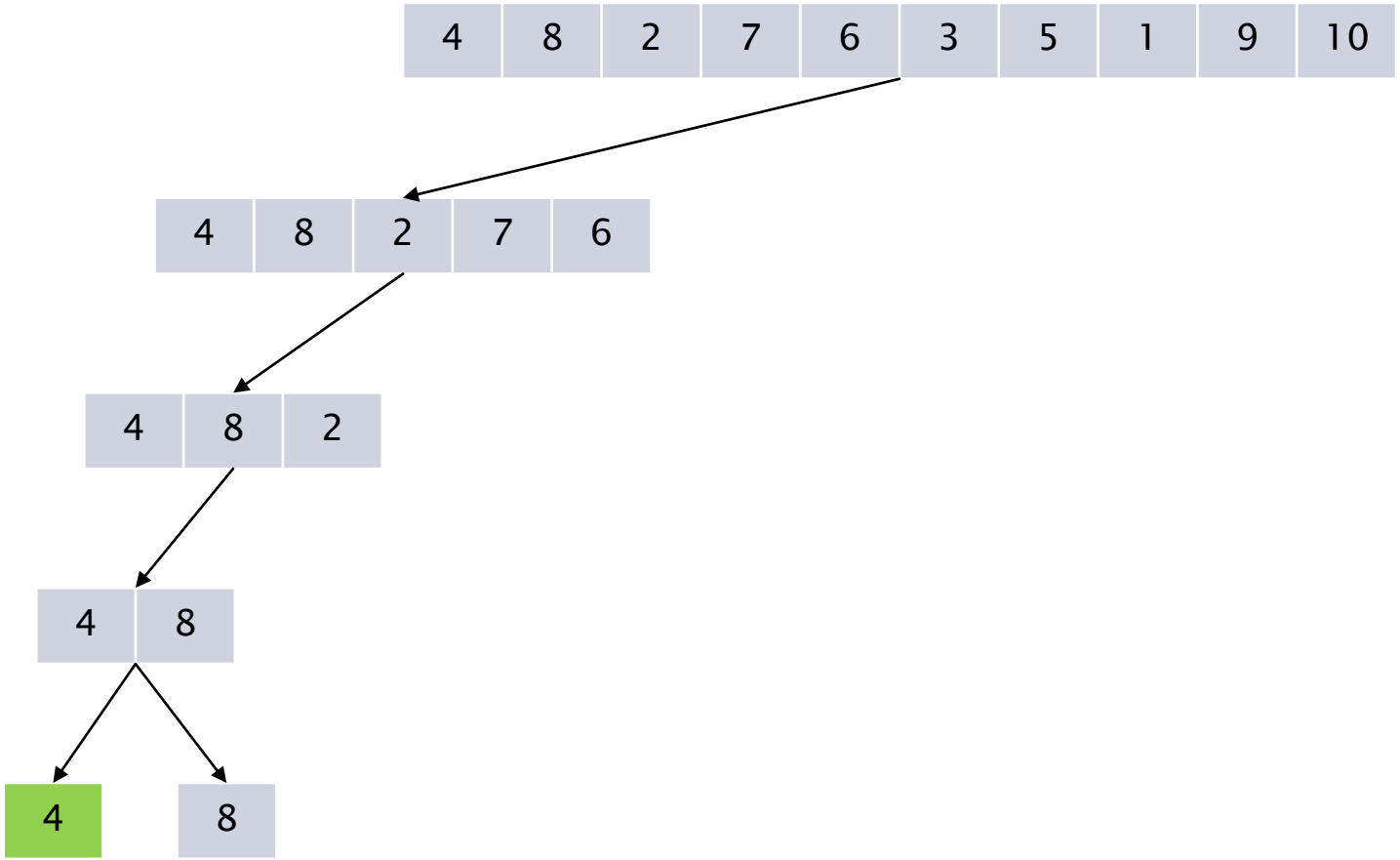
3. Funcionamento



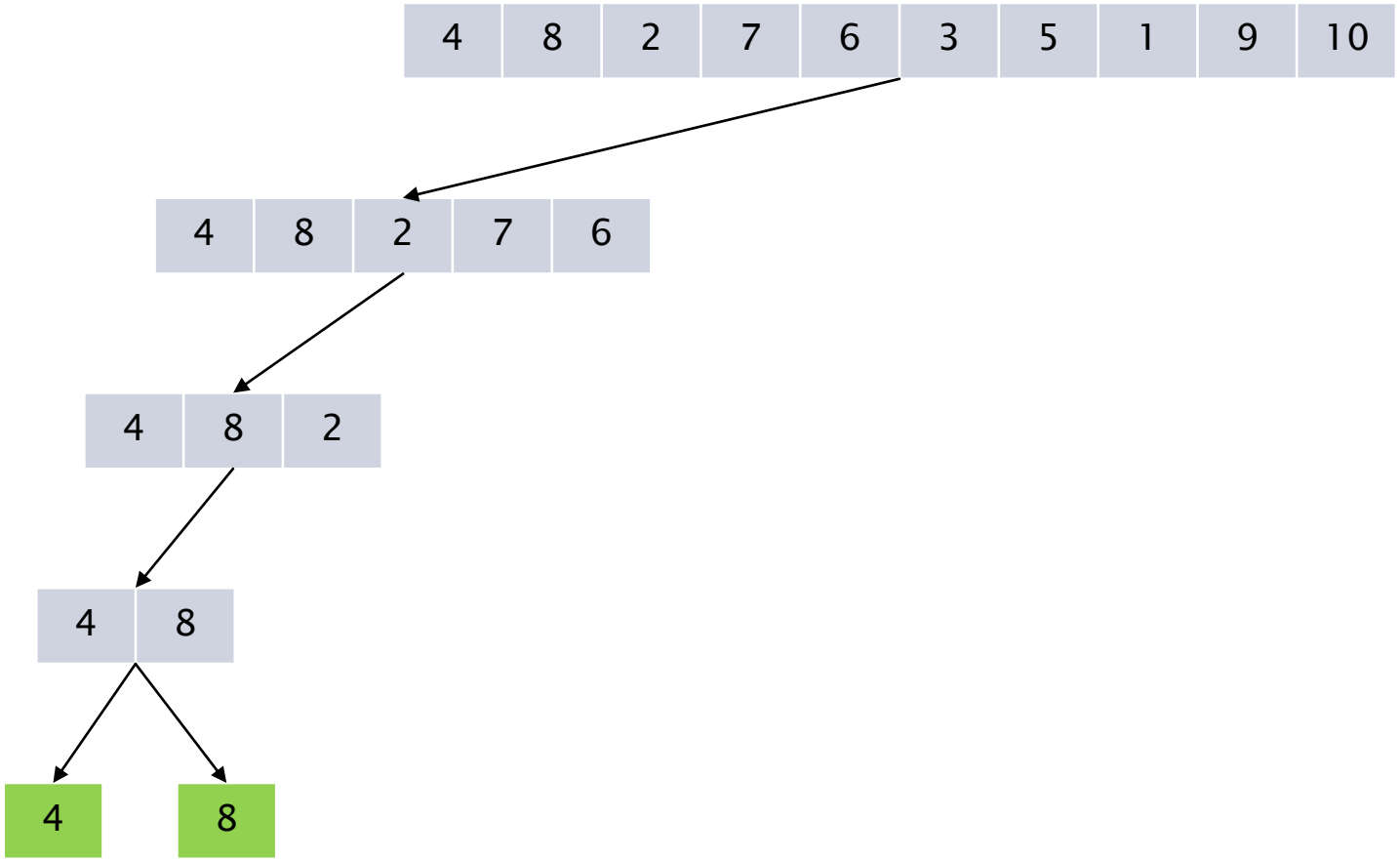
3. Funcionamento



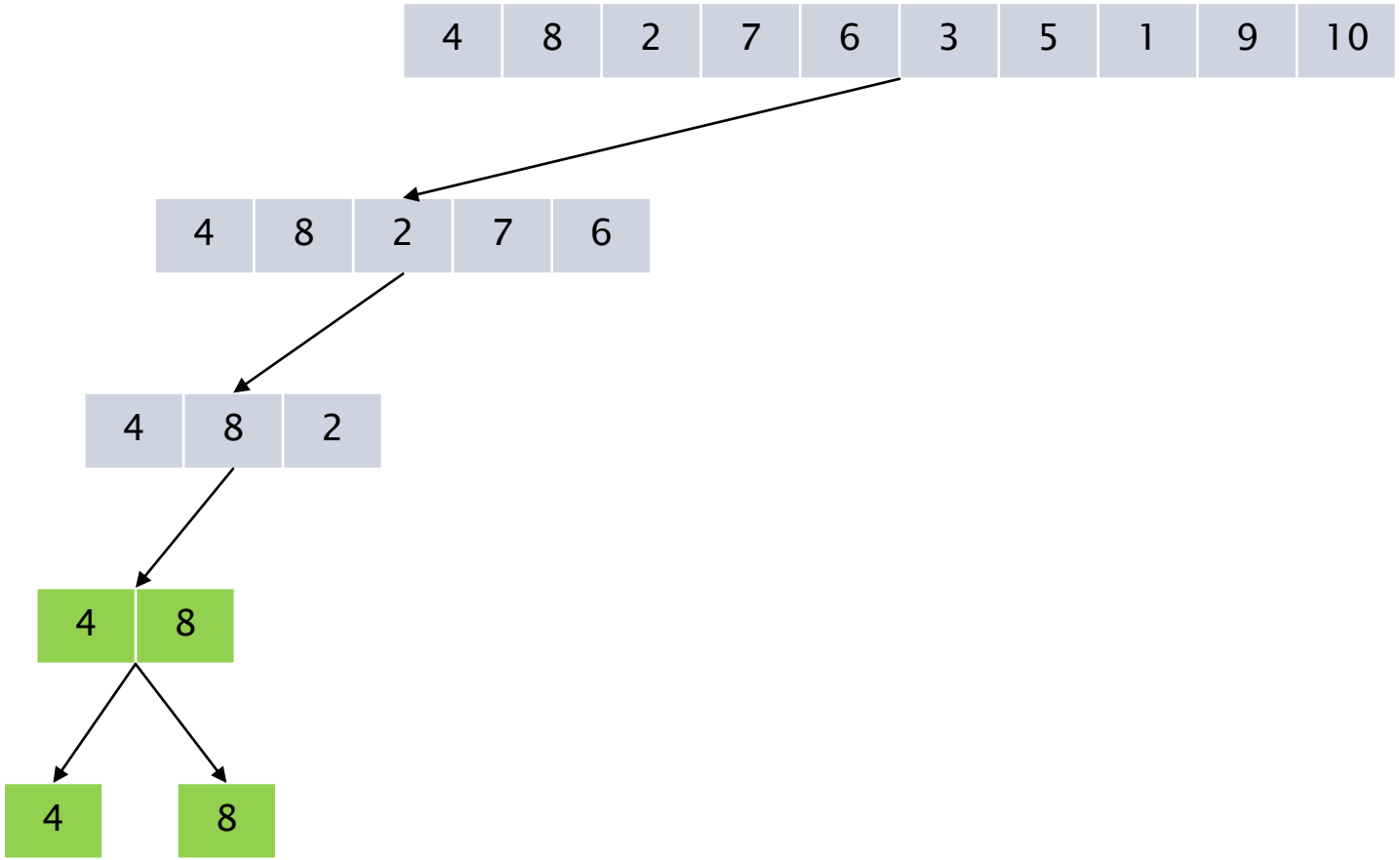
3. Funcionamento



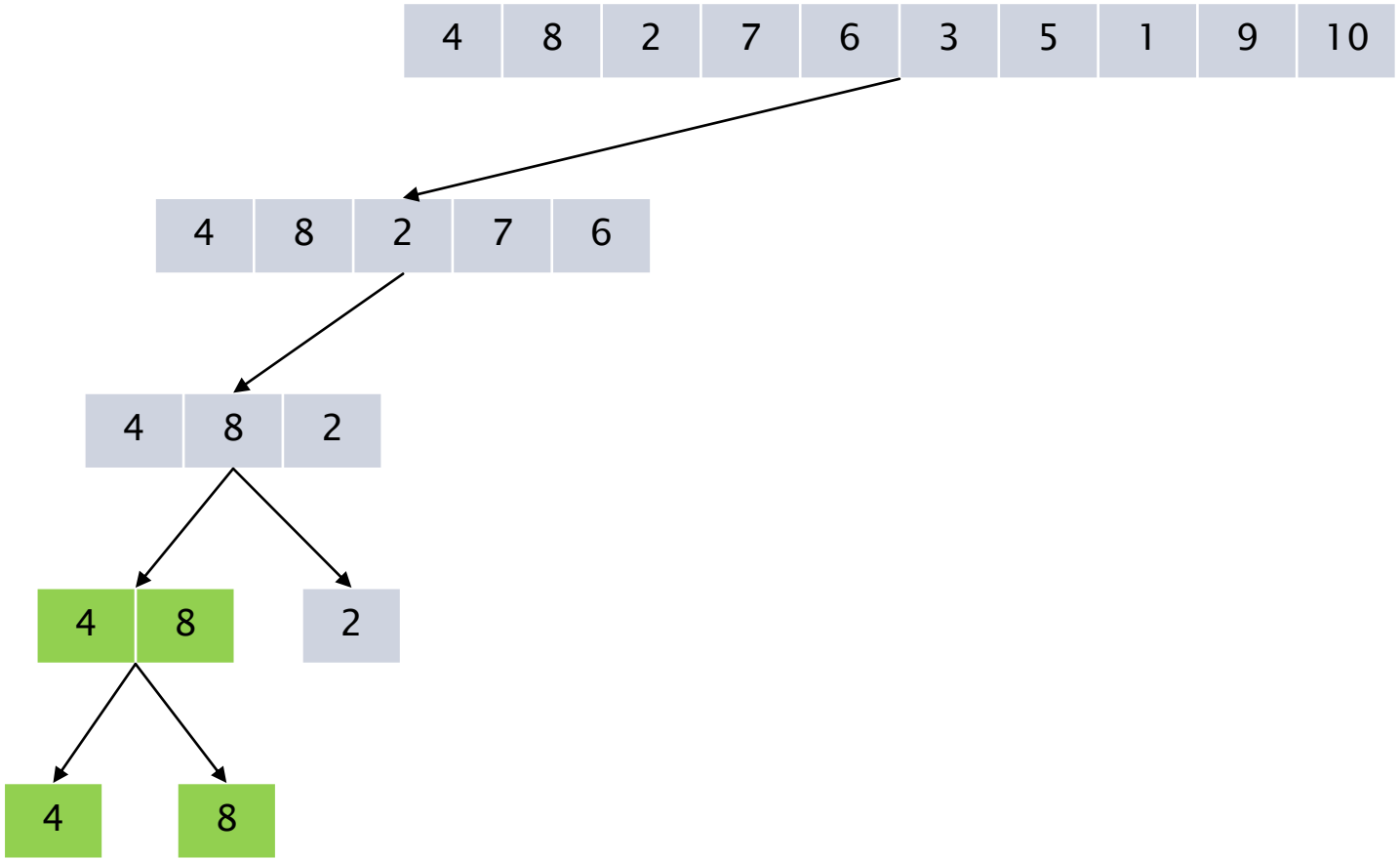
3. Funcionamento



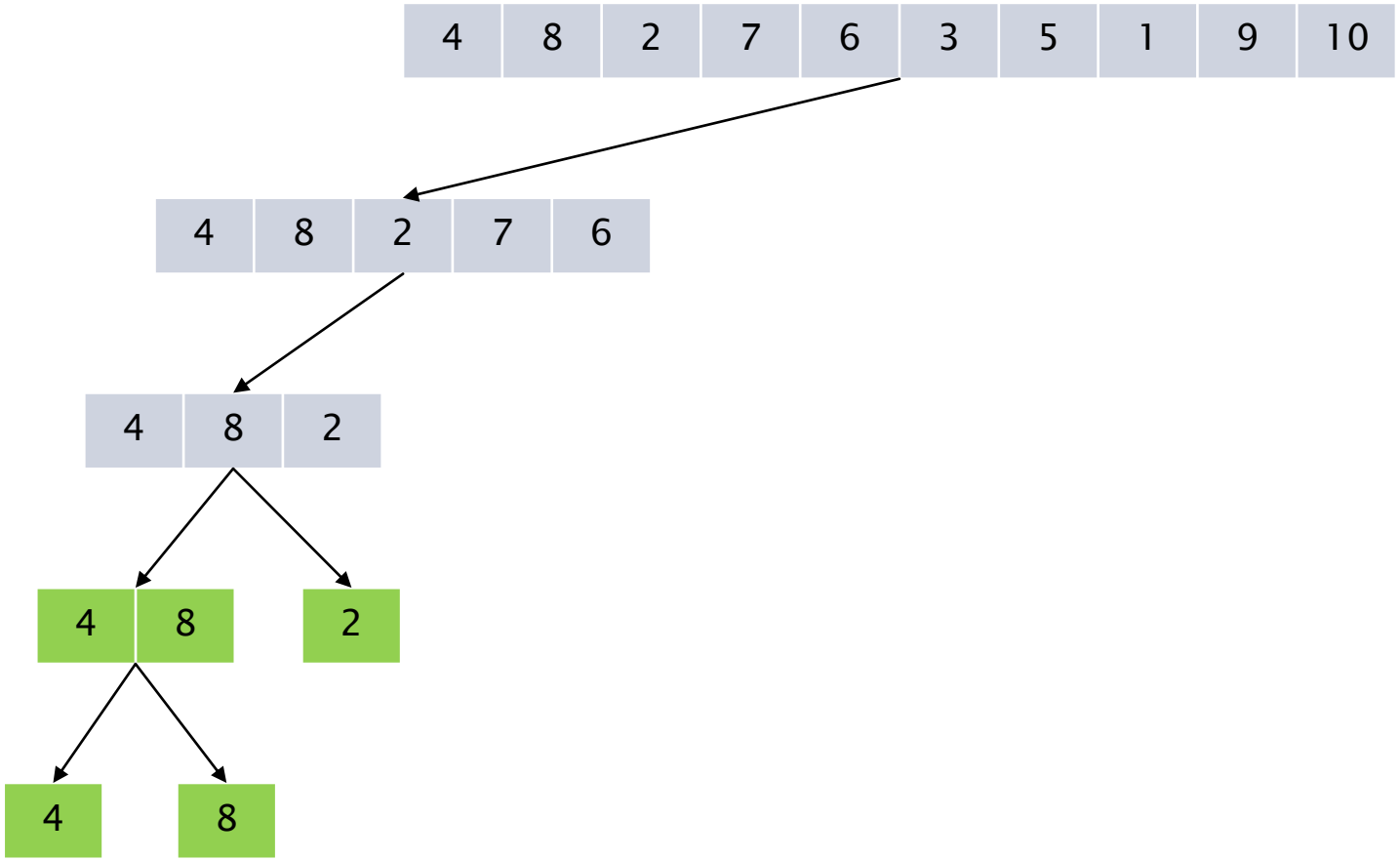
3. Funcionamento



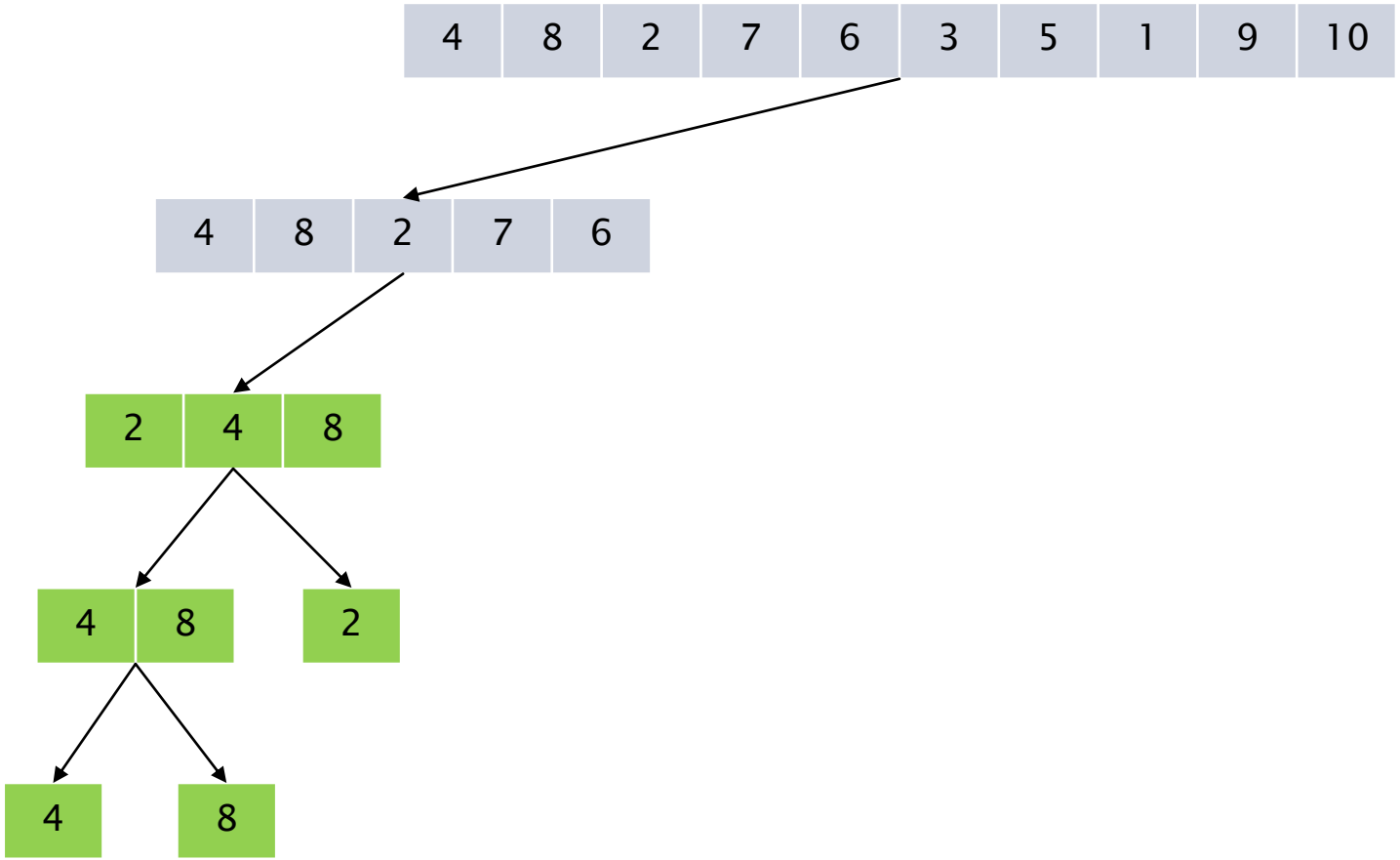
3. Funcionamento



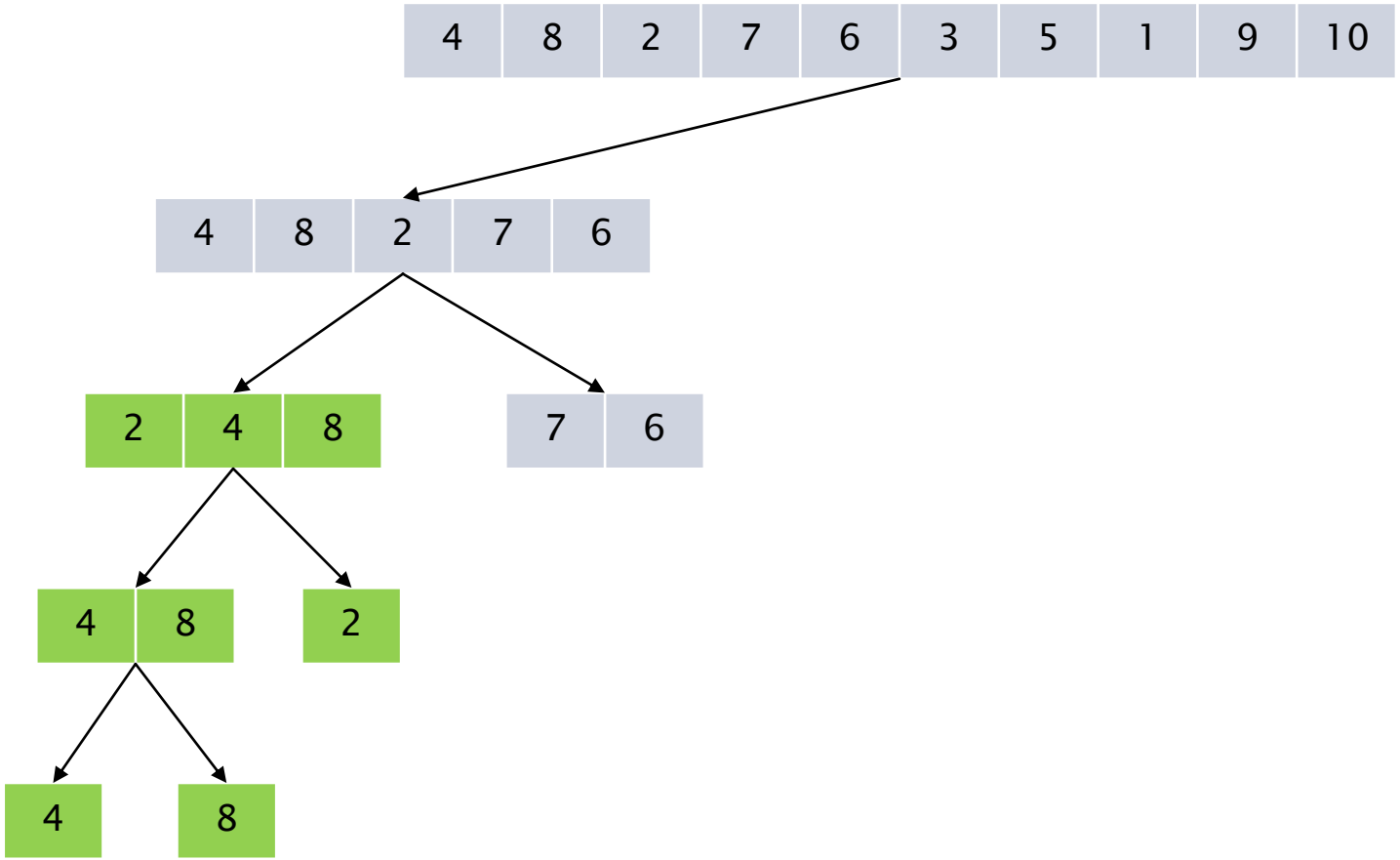
3. Funcionamento



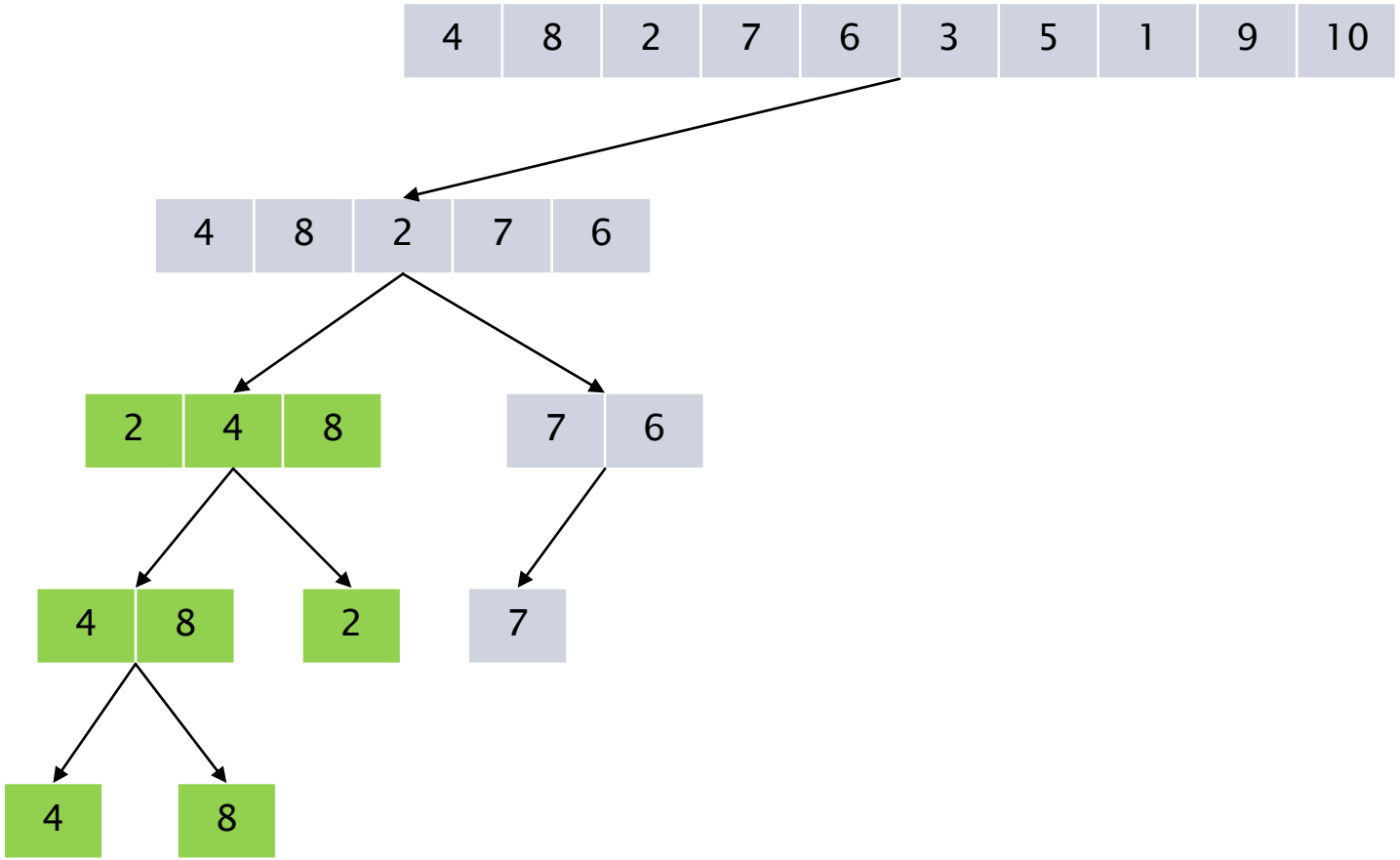
3. Funcionamento



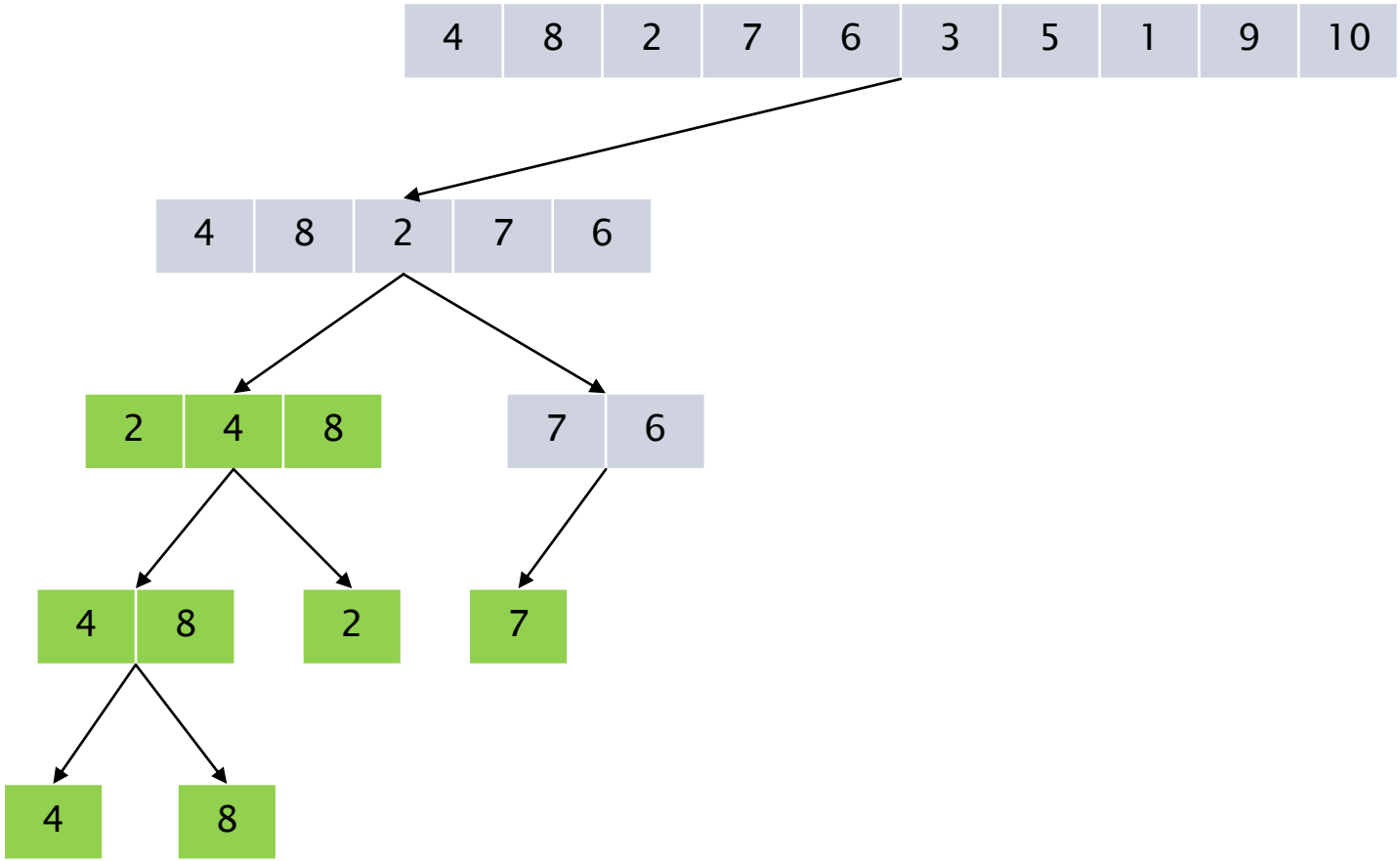
3. Funcionamento



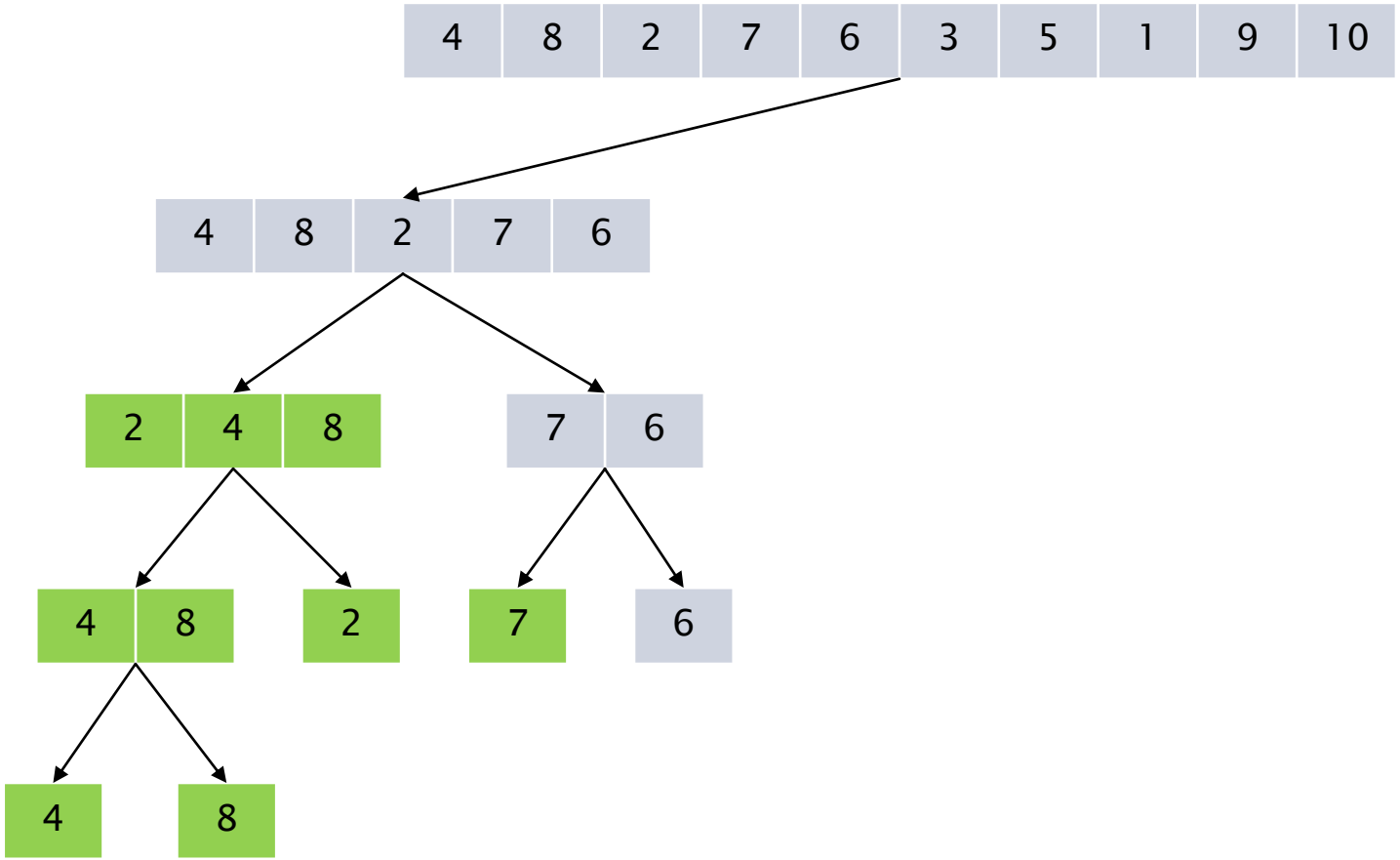
3. Funcionamento



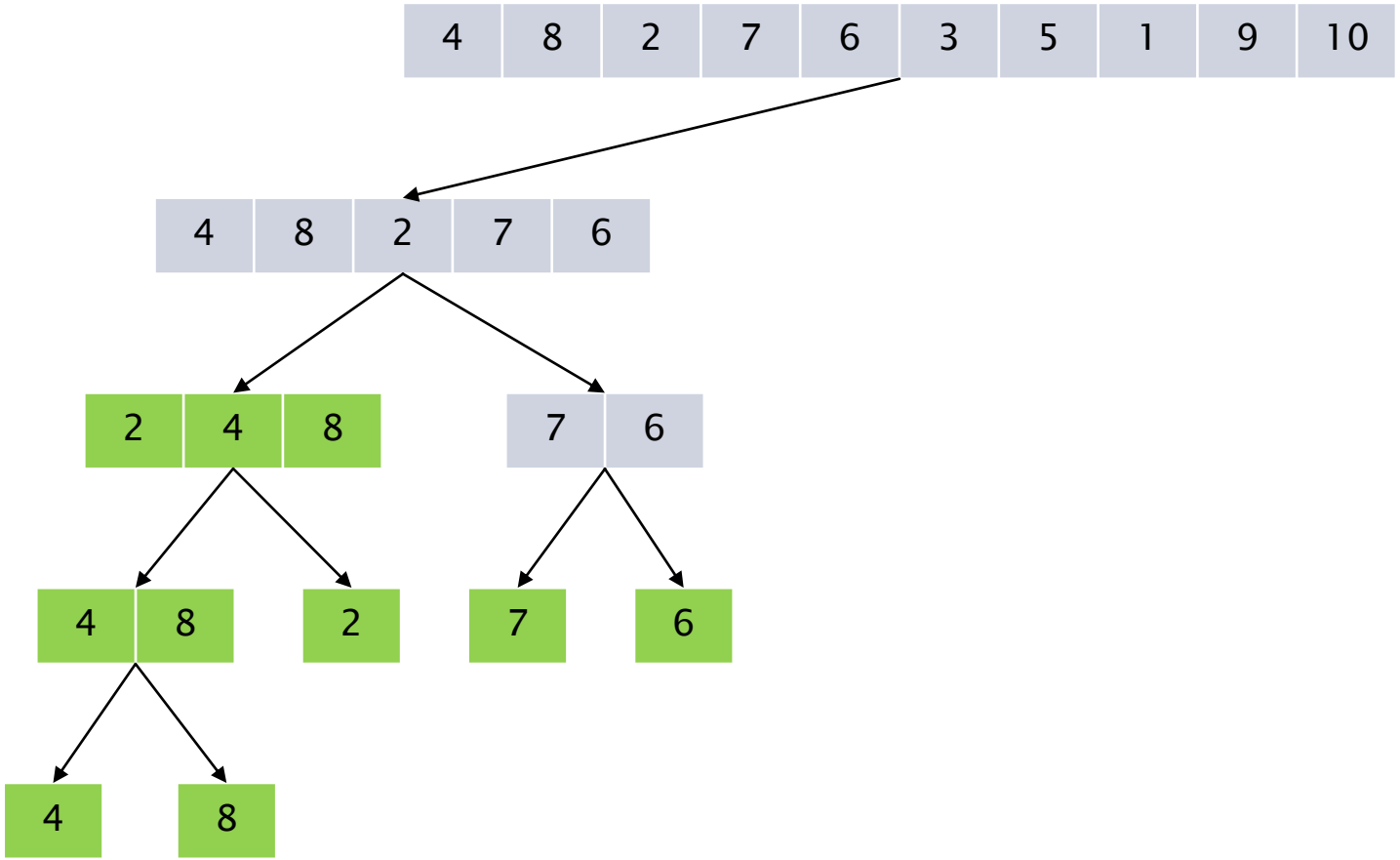
3. Funcionamento



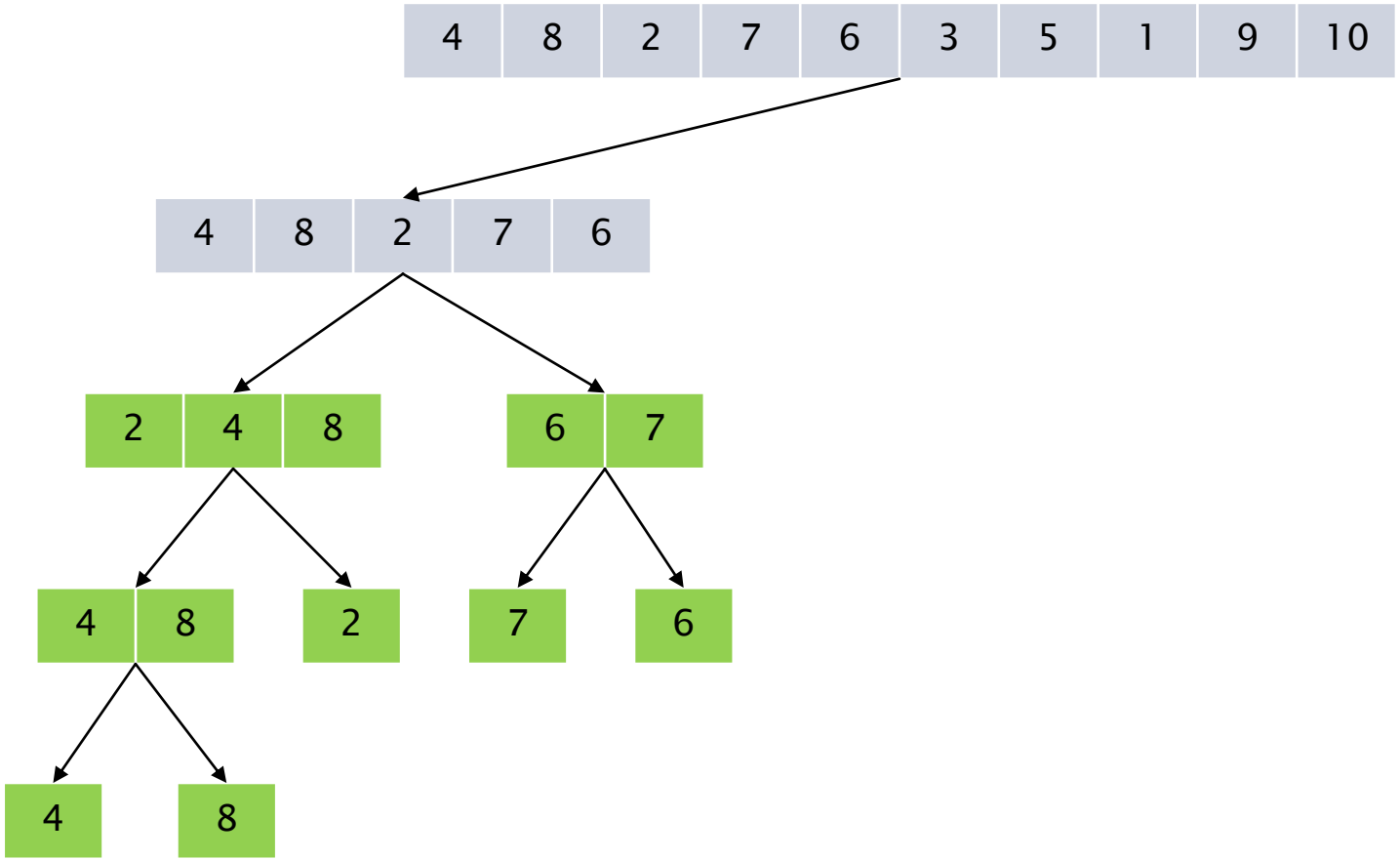
3. Funcionamento



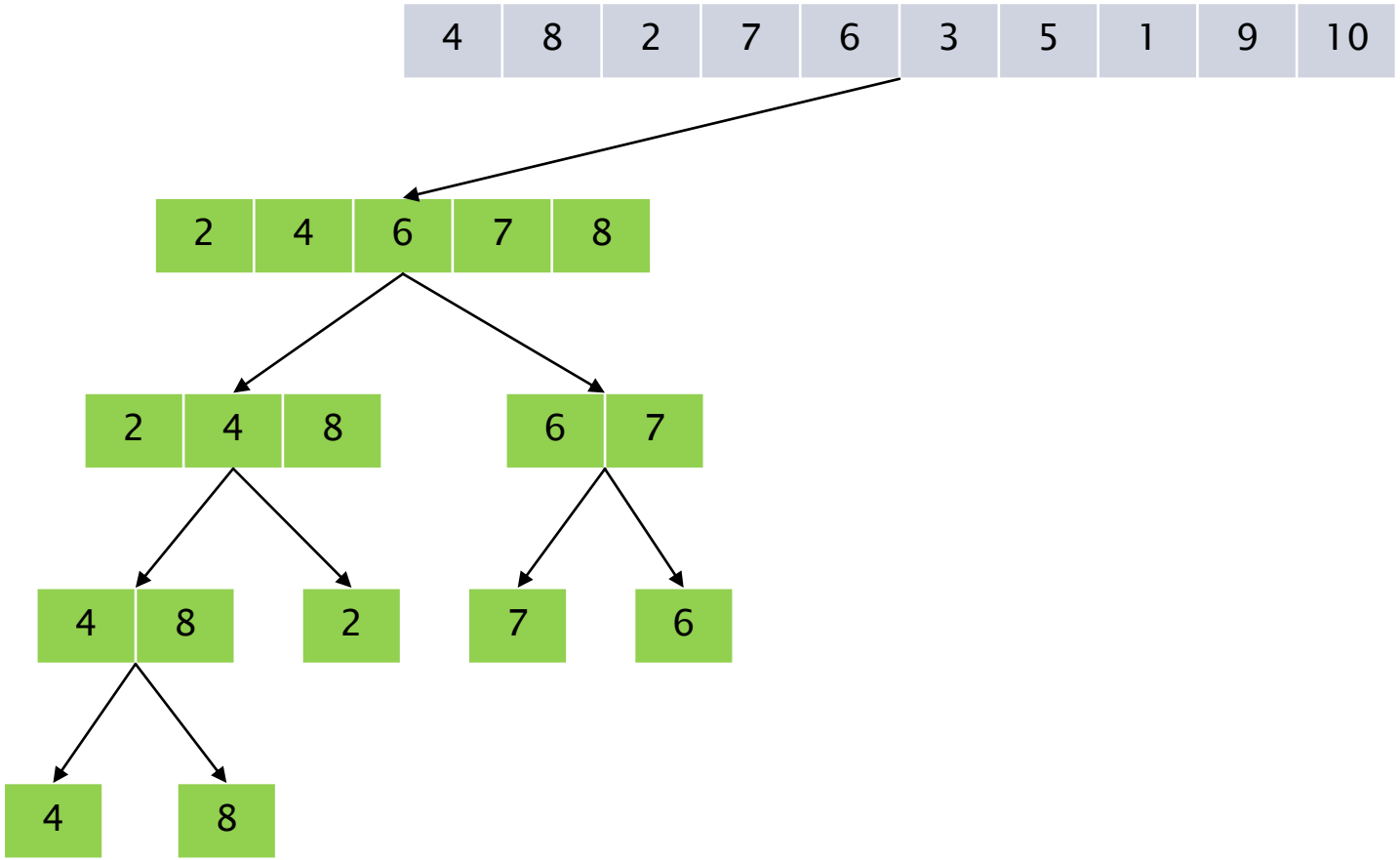
3. Funcionamento



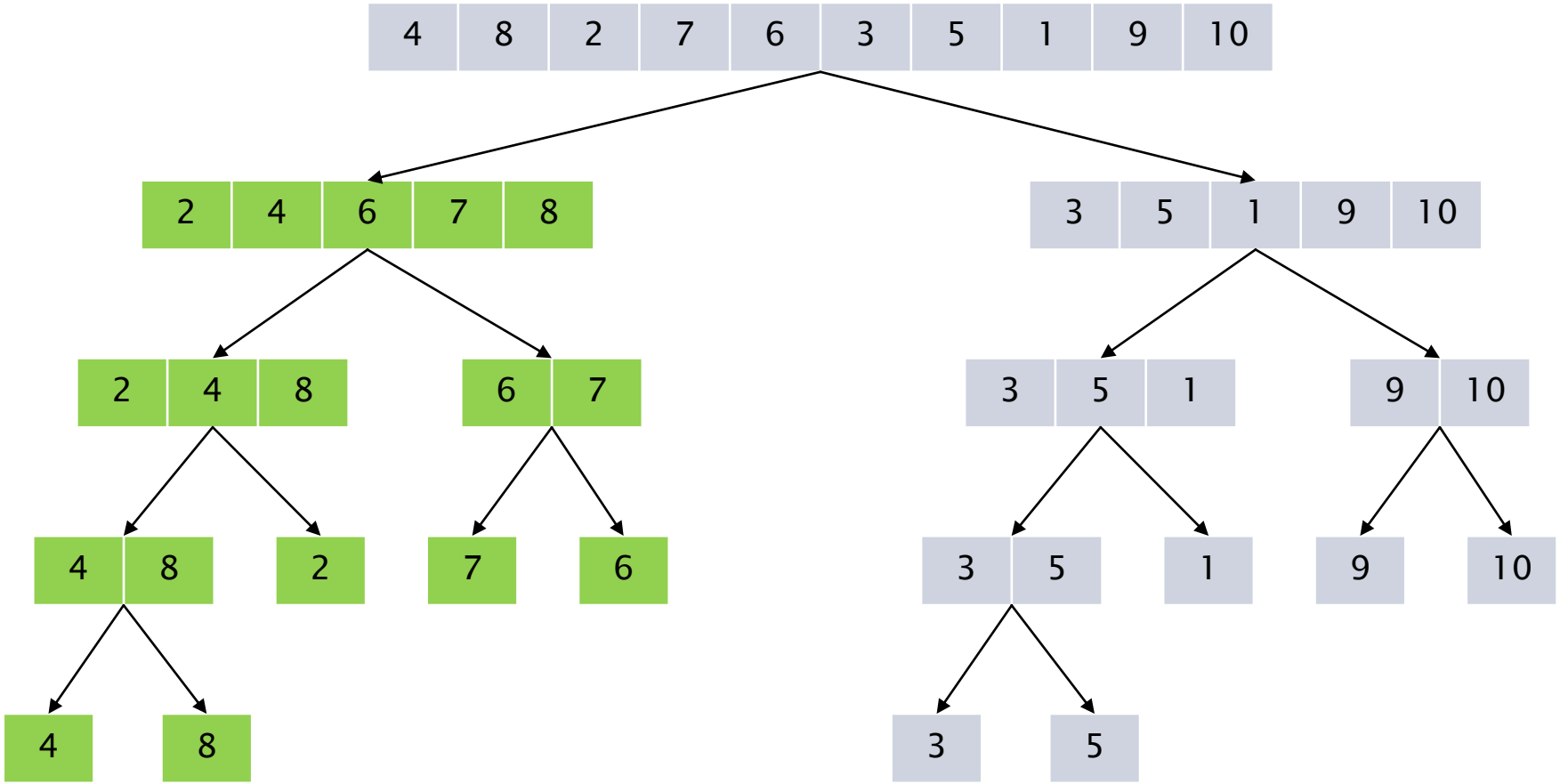
3. Funcionamento



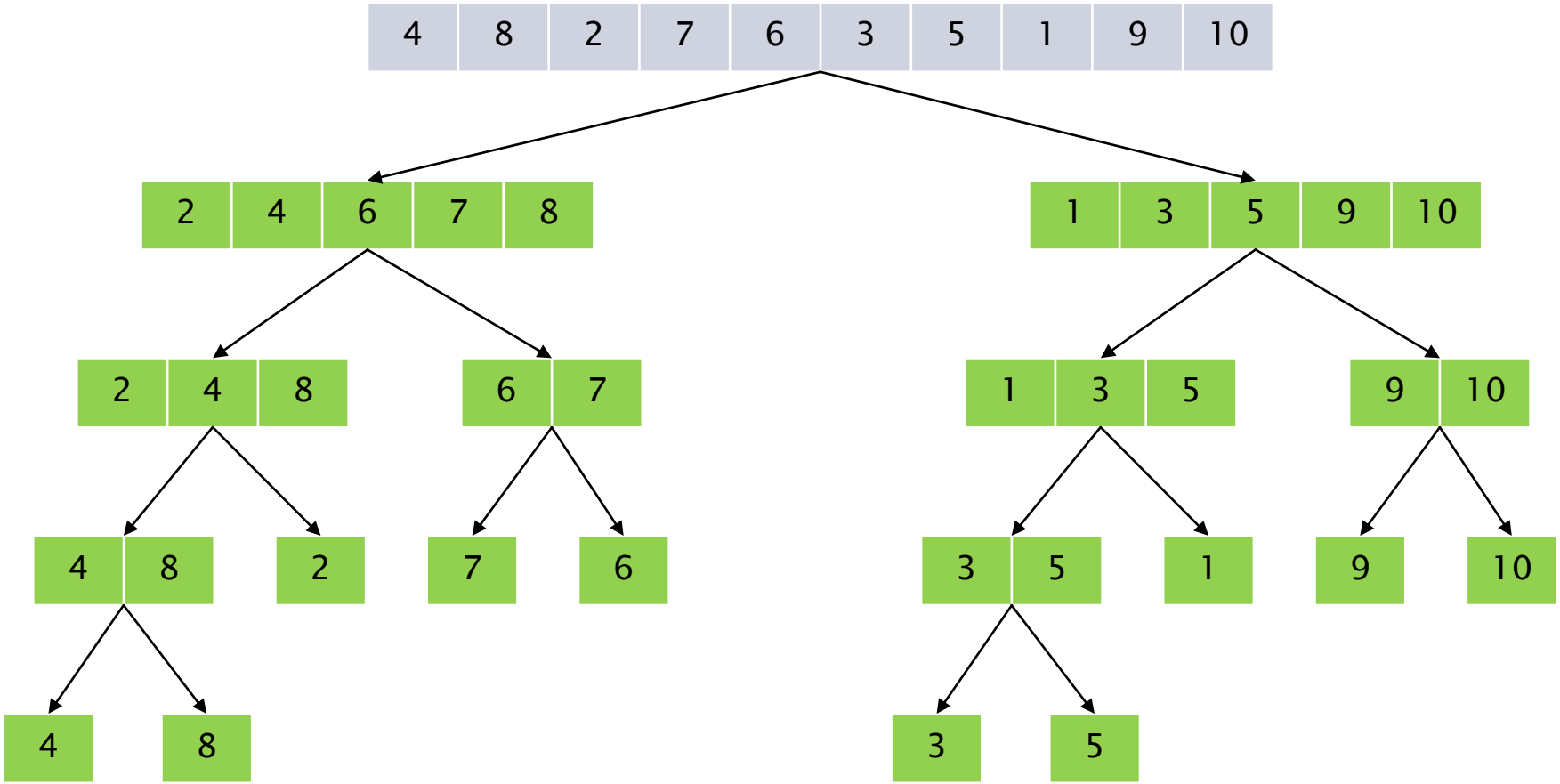
3. Funcionamento



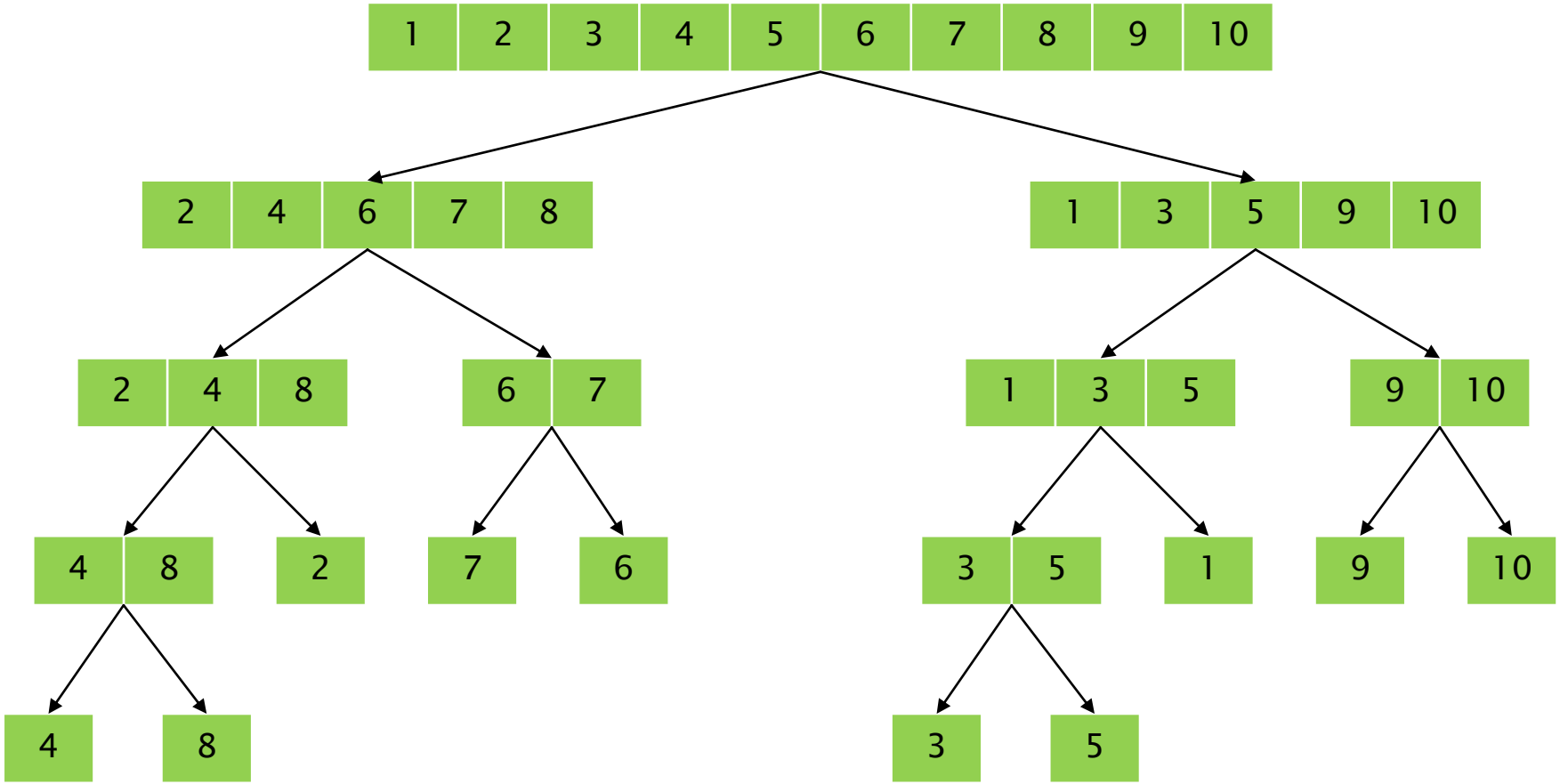
3. Funcionamento



3. Funcionamento



3. Funcionamento



4. Análise

- Complexidade
  - Considerando um vetor com **n** elementos, o tempo de execução é de ordem $\mathbf{O(n \times \log_2 n)}$ em todos os casos.
  - Sua eficiência não depende da ordem inicial dos elementos.

4. Análise

- **Vantagens:**
  - **Estável**: não altera a ordem dos dados com mesma chave.
  - Fácil Implementação
- **Desvantagens:**
  - Possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação.
    - Tanto para o vetor auxiliar – **O(n)**
    - Quanto para a pilha de recursão – **$O(\log_2 n)$**

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