Quick Sort

Prof. Dr. Eleandro Maschio
Tecnologia em Sistemas para Internet
Câmpus Guarapuava
Universidade Tecnológica Federal do Paraná (UTFPR)

Quick Sort

- Criado por **Sir Charles Antony Richard Hoare** (C. A. R. Hoare) em 1960 em visita à Universidade de Moscou.
- Necessidade ordenar palavras a fim de traduzir um dicionário de inglês para russo.
- Objetivo de **reduzir o problema original** em subproblemas que possam ser resolvidos mais fácil e rapidamente.

Quick Sort

- É considerado o **melhor algoritmo de ordenação** atualmente disponível para propósito geral.
- Baseia-se na **Ordenação por Troca**.
- Surpreende-nos ao considerarmos o terrível desempenho do ***Bubble Sort***.

Quick Sort

Lógica de Implementação

- As implementações são fortemente influenciadas pelos **recursos das linguagens** utilizadas.

- Determinadas lógicas de implementação são mais **típicas** (e favorecidas) por certas linguagens.

Quick Sort

Lógica de Implementação (1)

(**1**) Escolha um elemento arbitrário da sequência e chame-o de **pivô**.

(**2**) Divida todos os demais elementos (exceto o pivô) em **duas partições**.

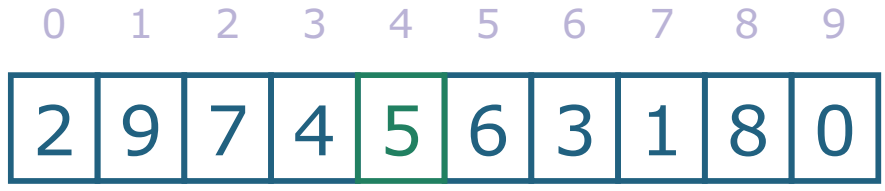
- Todos os elementos **menores** do que o pivô devem estar na primeira partição (**lado esquerdo**).
- Todos os elementos **maiores** do que o pivô devem estar na segunda partição (**lado direito**).

(**3**) Use a **recursividade** para ordenar cada uma das partições. A base da recursão são as sequências de tamanho zero ou um, consideradas em ordem.

(**4**) Concatene, respectivamente, a **primeira partição**, o **pivô** e a **segunda partição**. Note que as partições estão ordenadas.

Dinâmica de Funcionamento

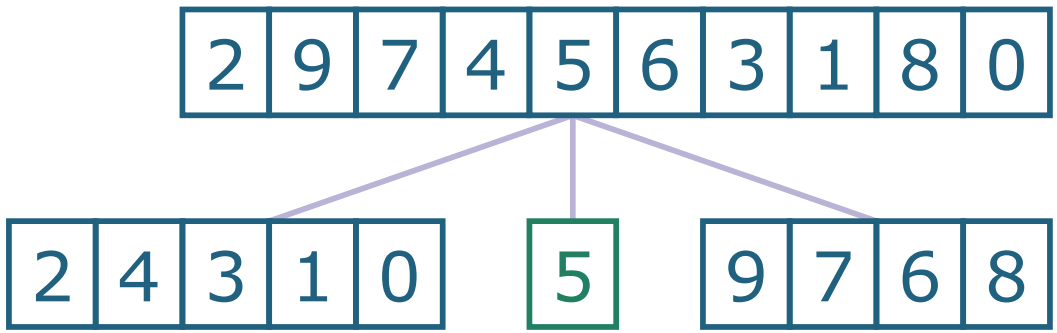
Considere a matriz unidimensional.



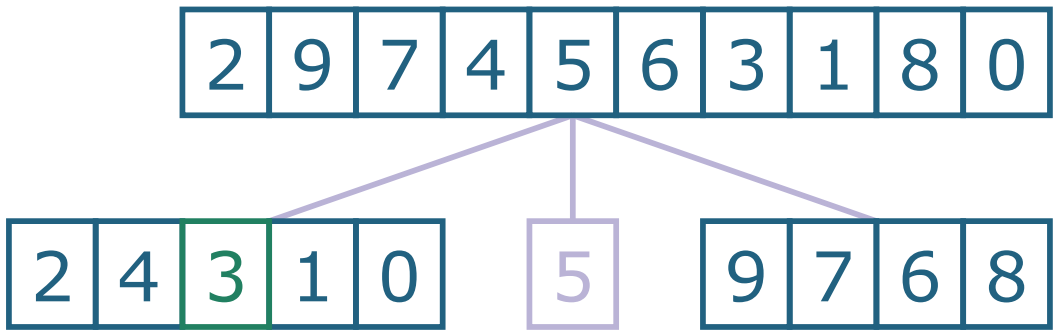

```
int meio = (inicio + fim) / 2 = 4;

int pivo = v[meio] = 5;
```

Dinâmica de Funcionamento

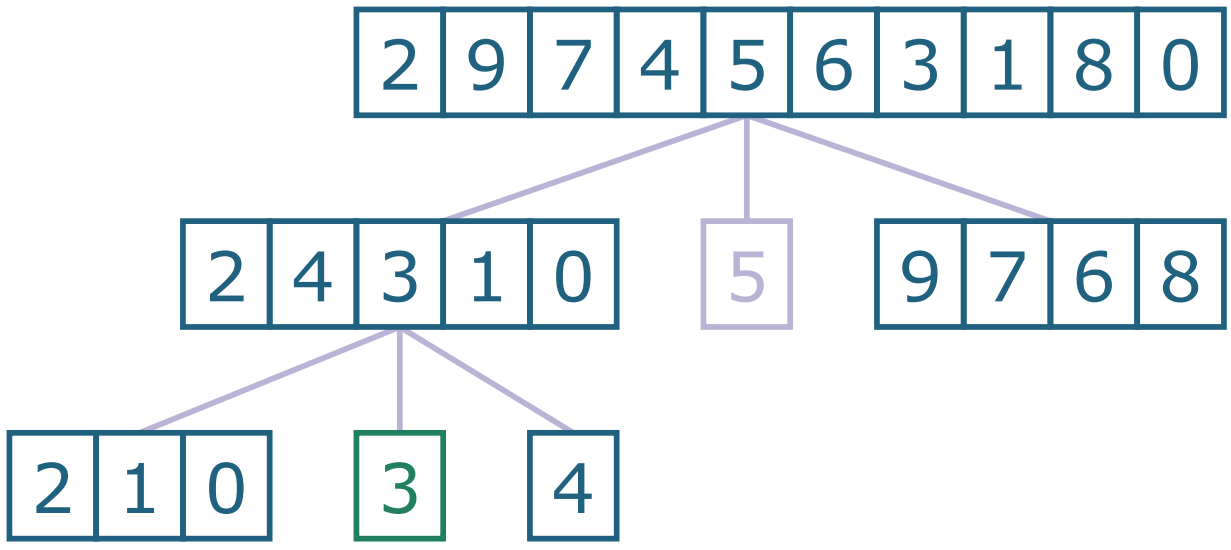
Dinâmica de Funcionamento



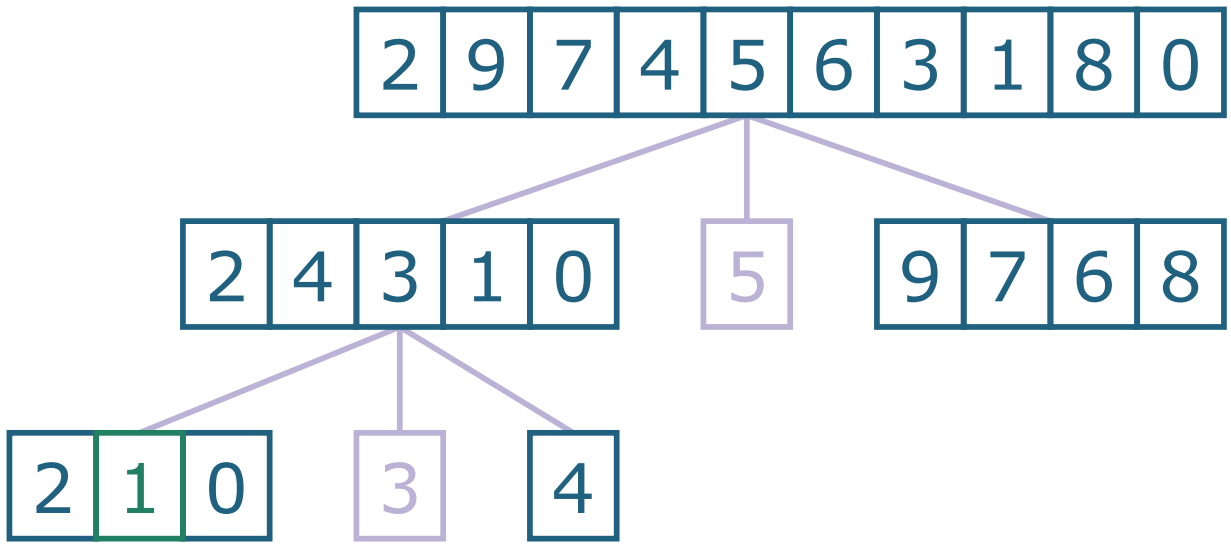
Dinâmica de Funcionamento



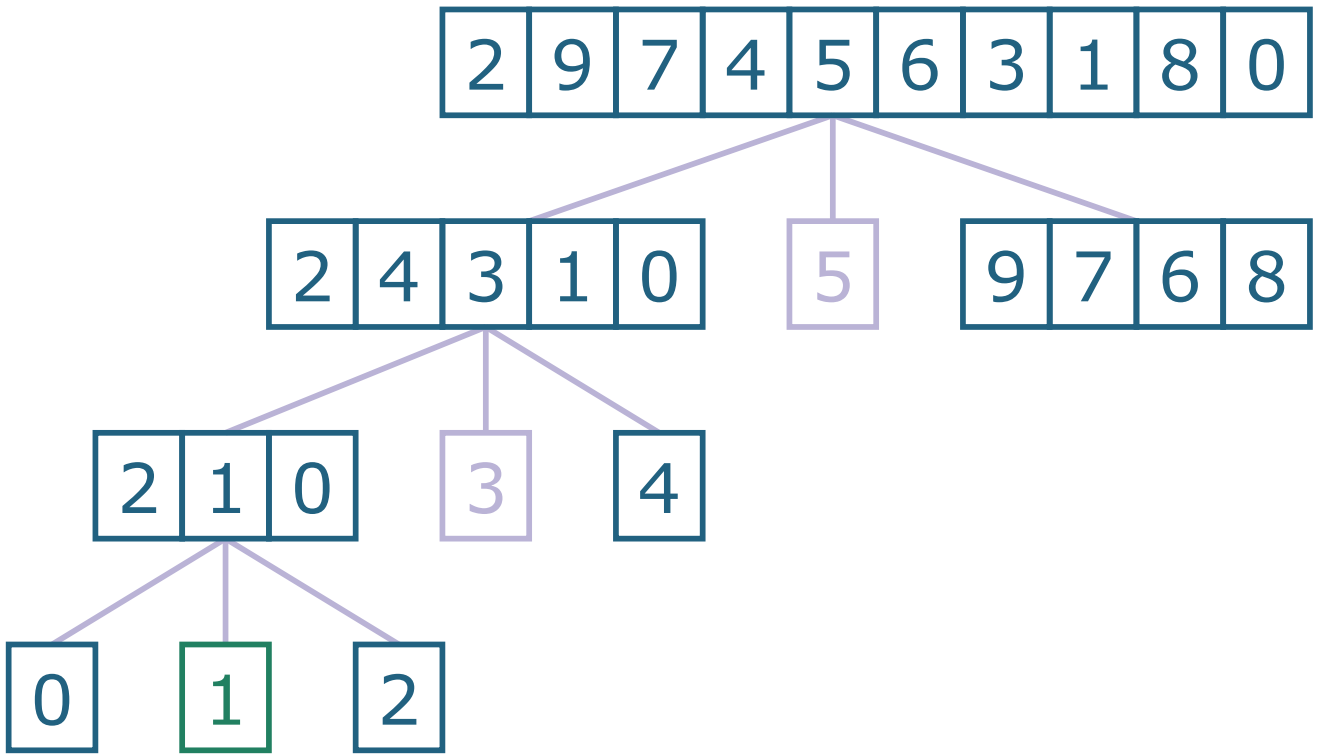
Dinâmica de Funcionamento



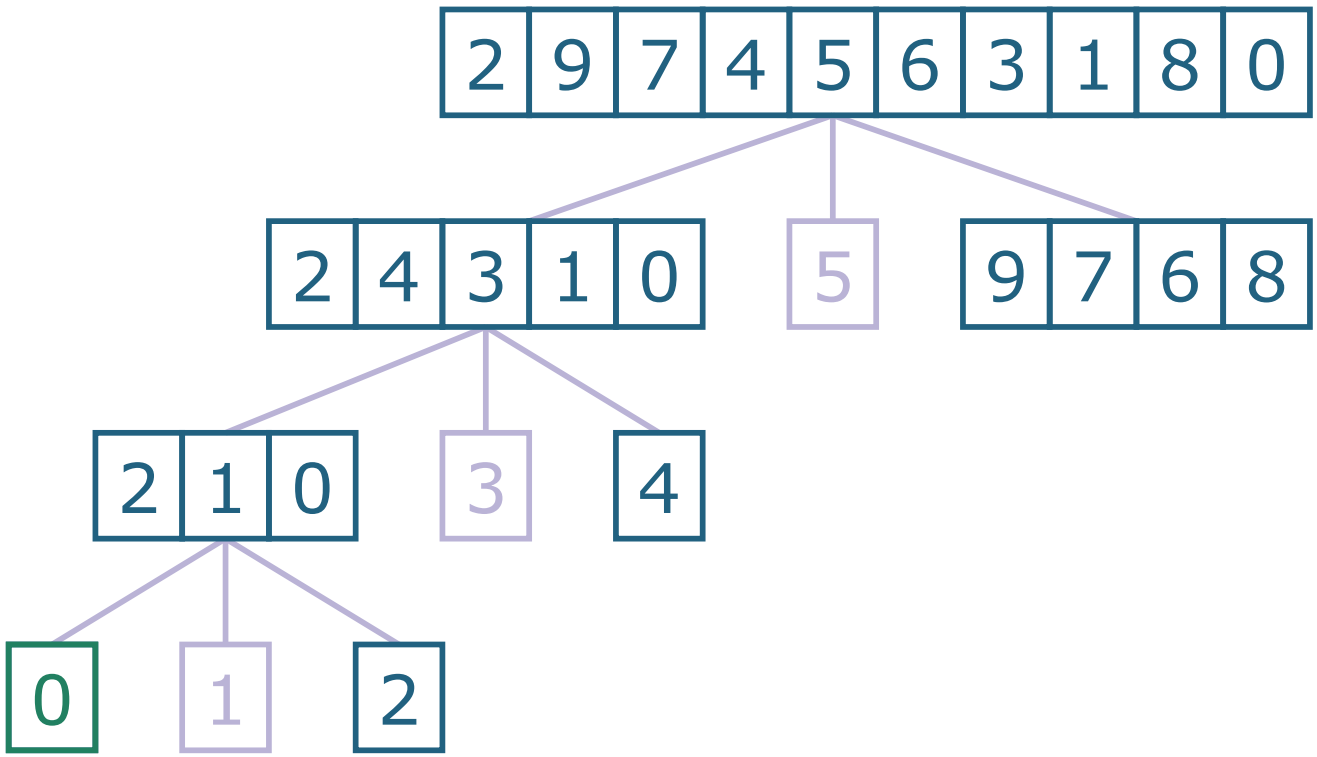
Dinâmica de Funcionamento



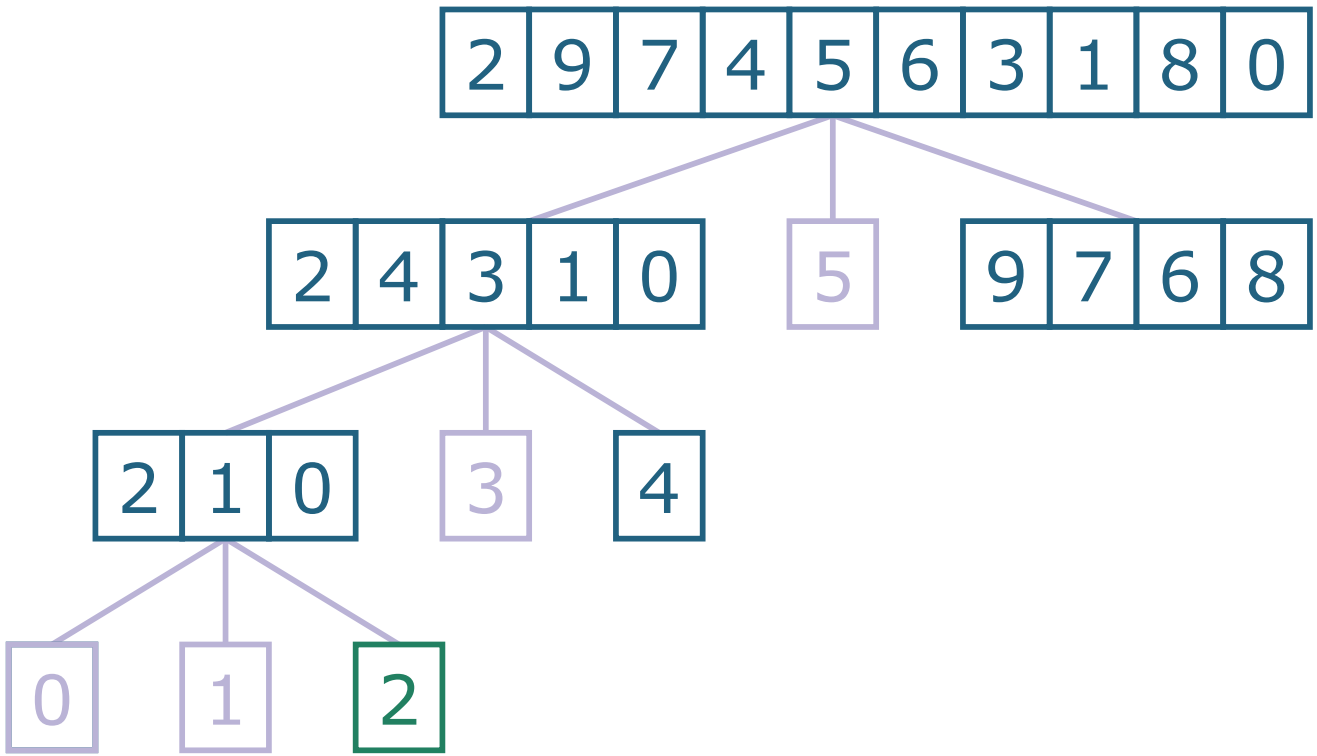
Dinâmica de Funcionamento



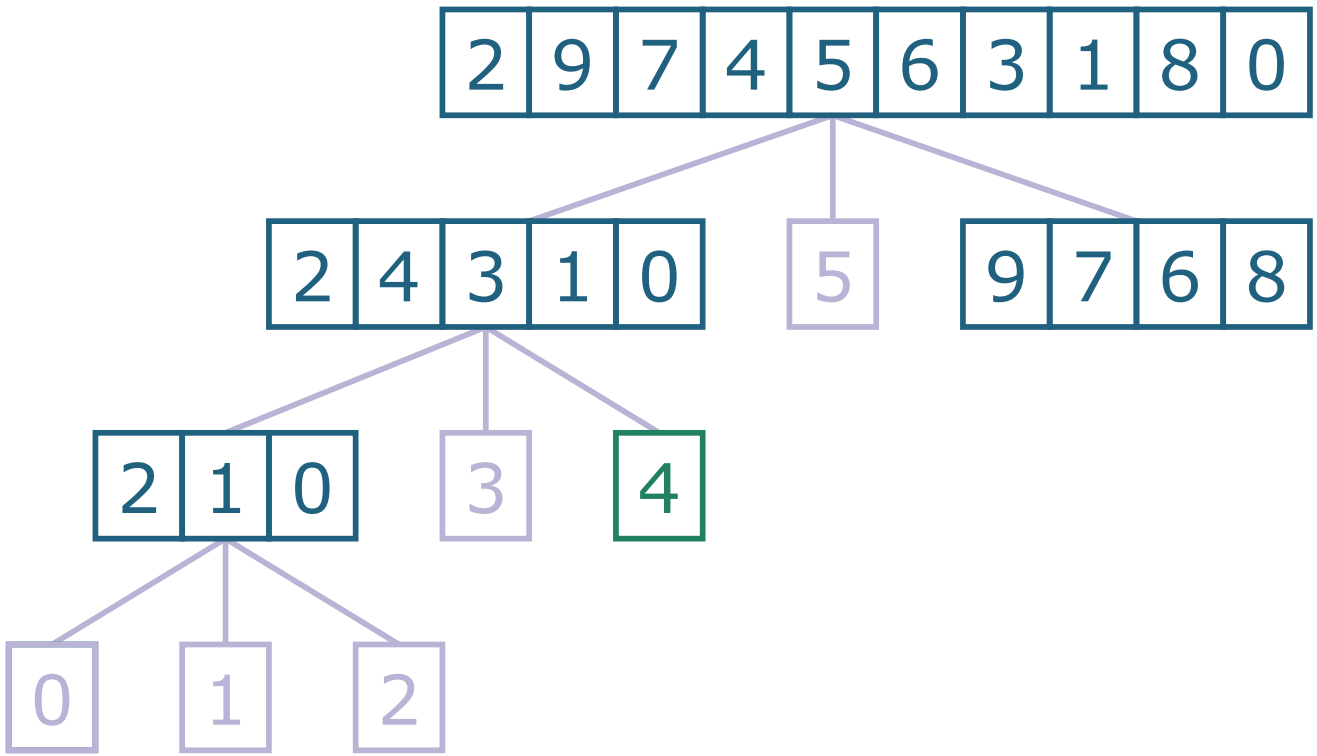
Dinâmica de Funcionamento



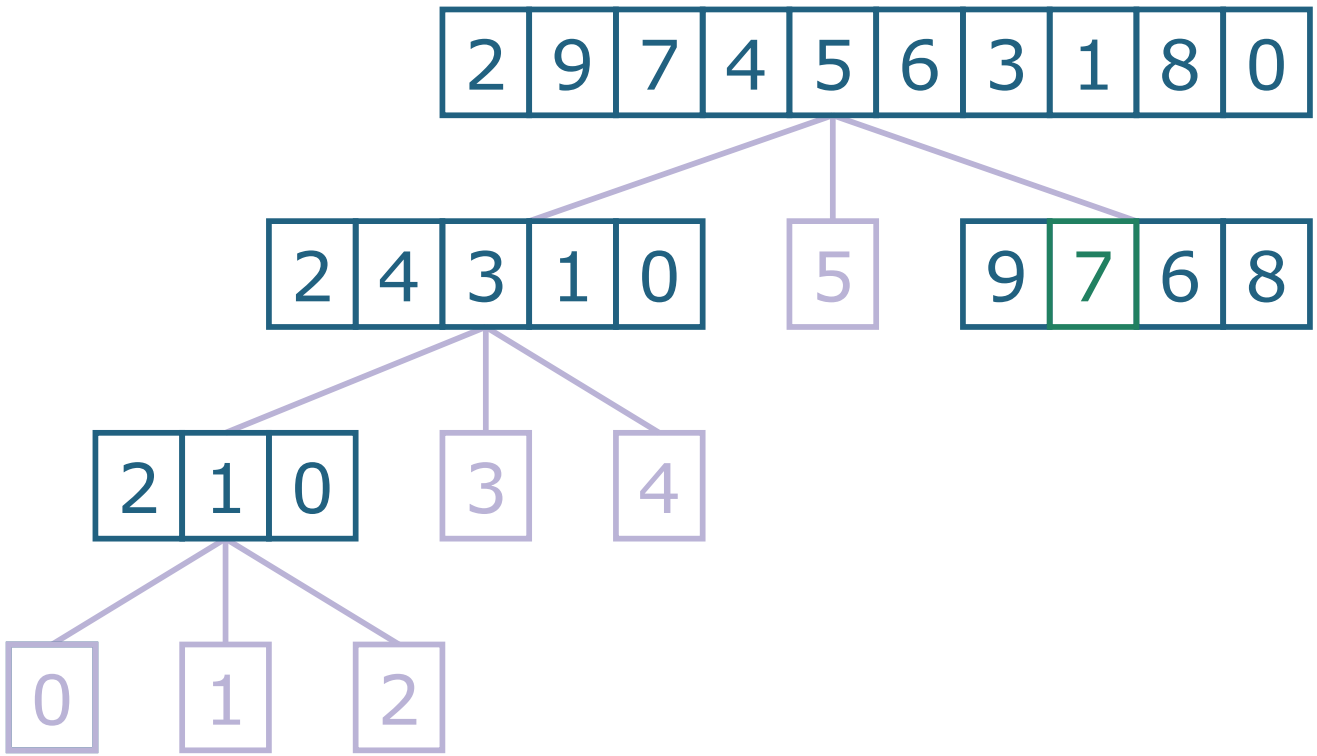
Dinâmica de Funcionamento



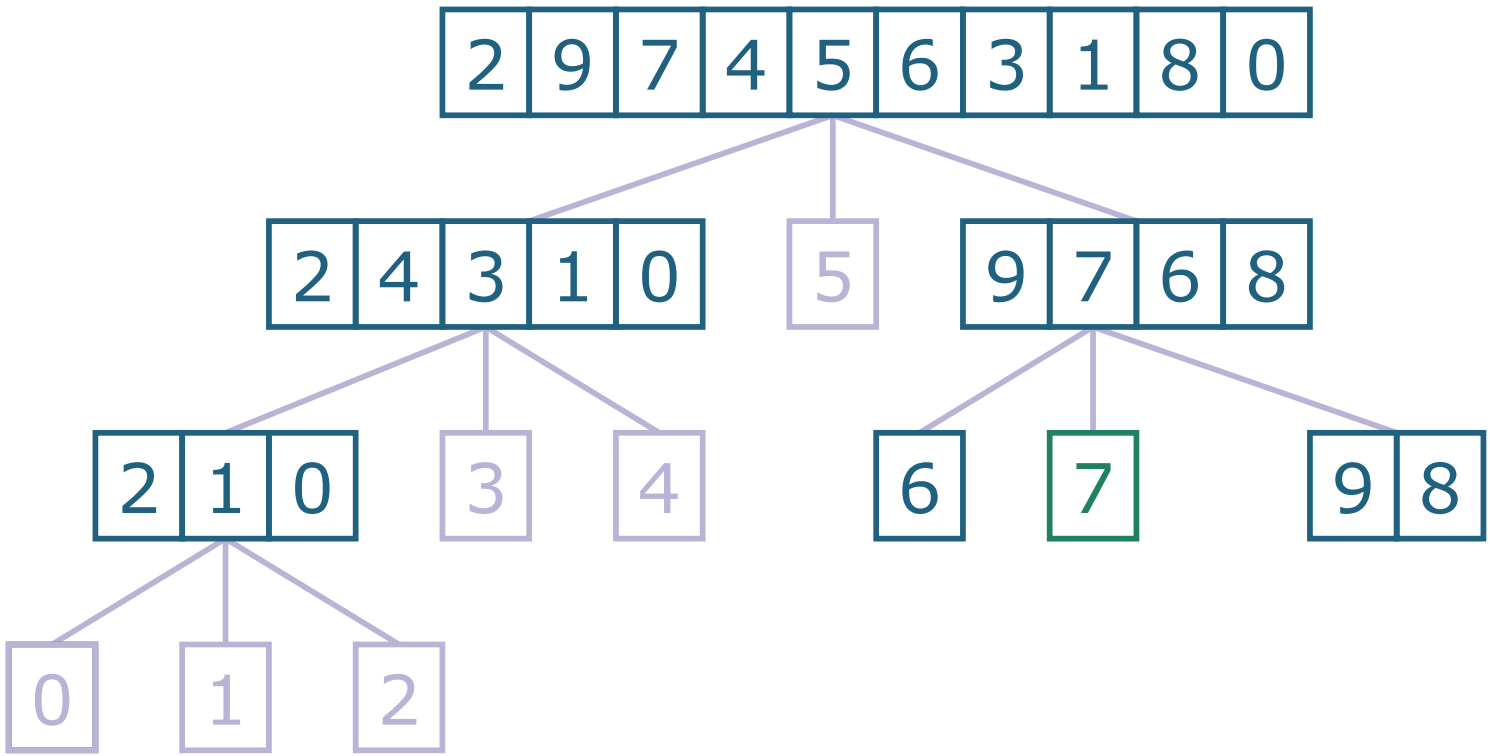
Dinâmica de Funcionamento



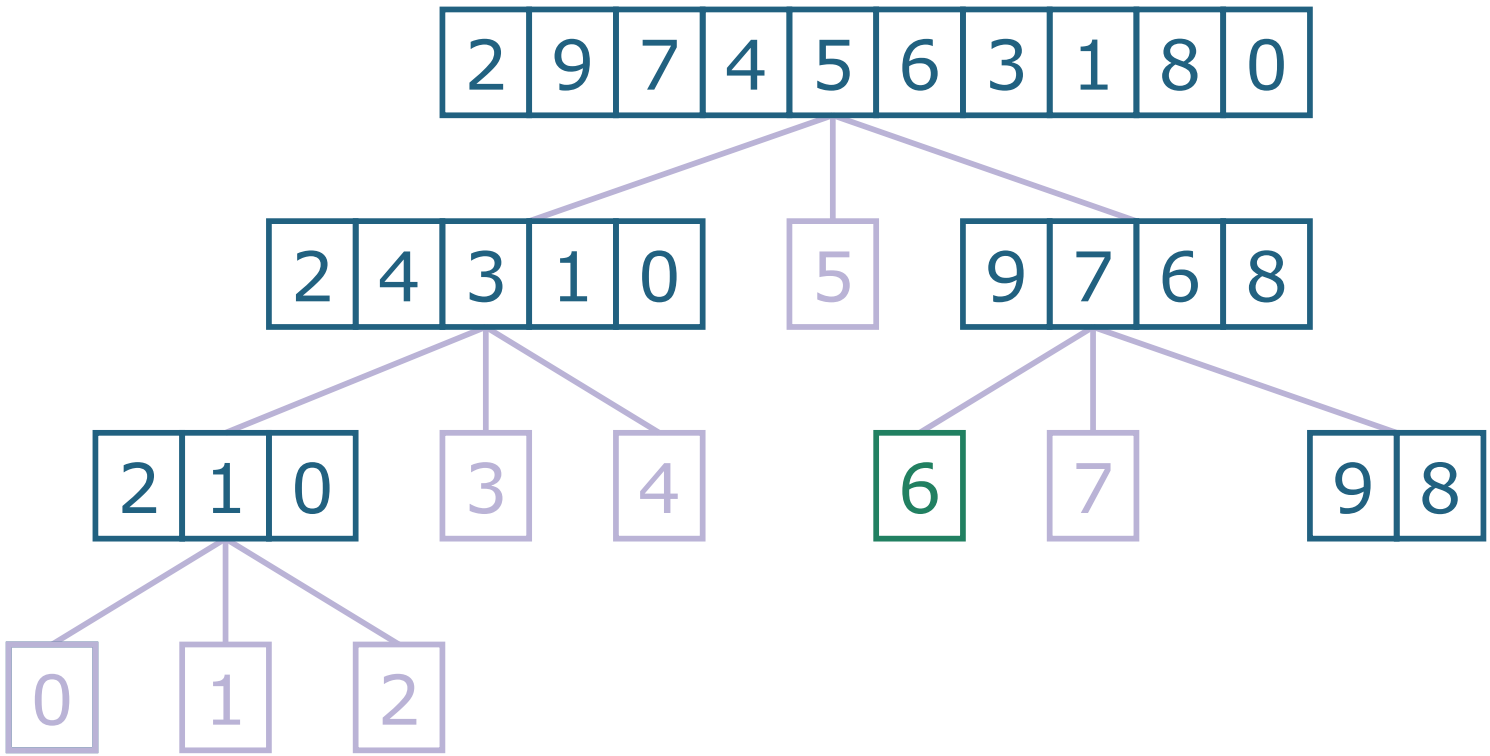
Dinâmica de Funcionamento



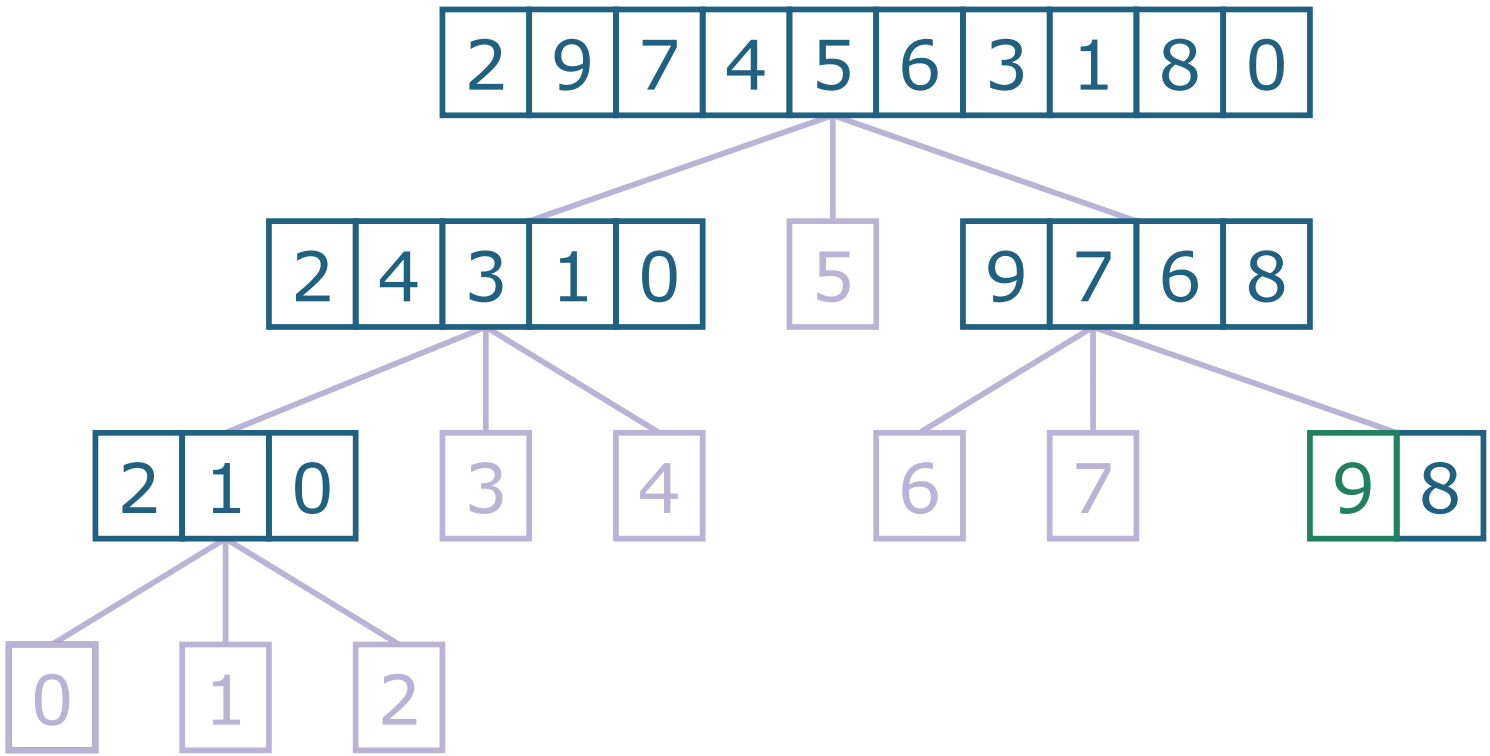
Dinâmica de Funcionamento



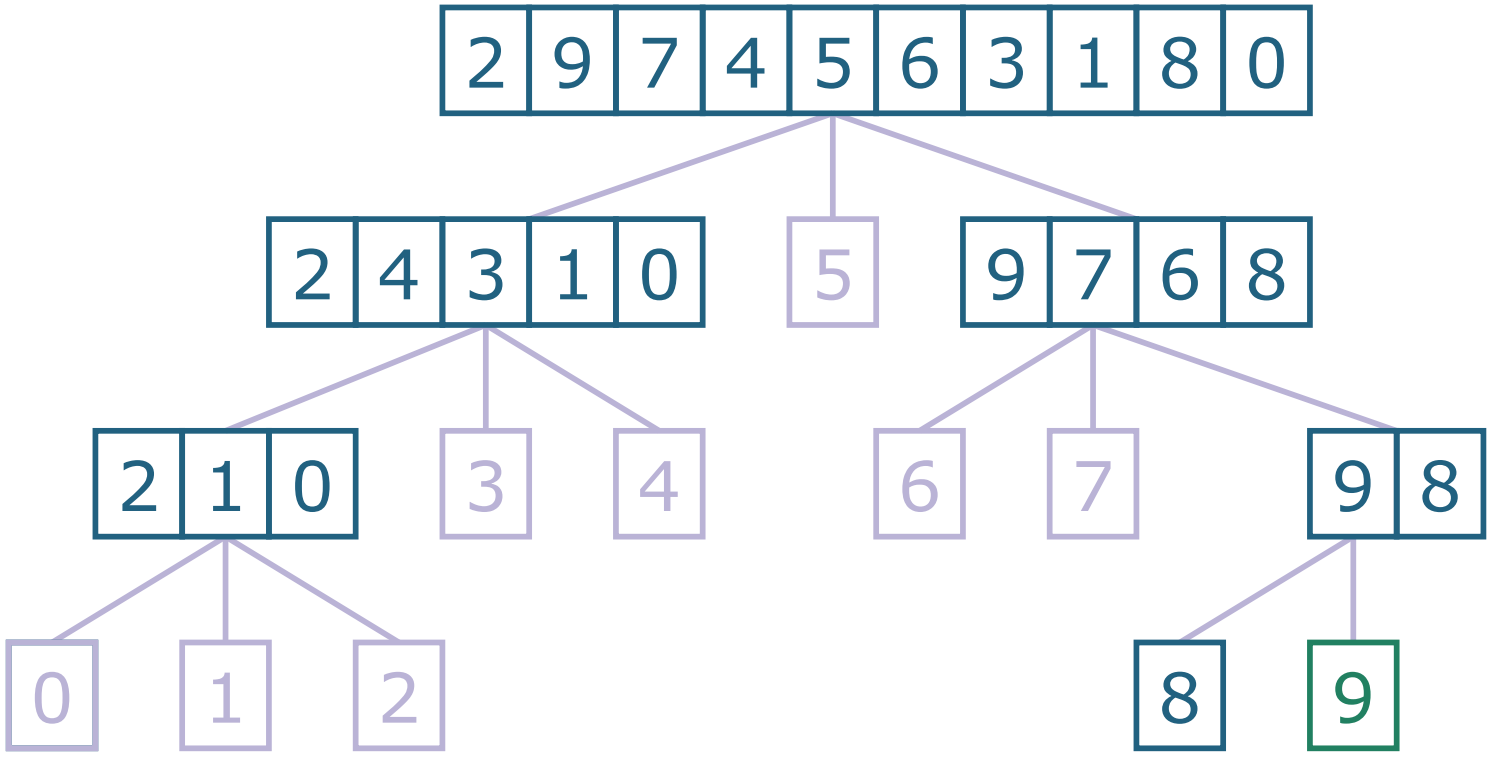
Dinâmica de Funcionamento



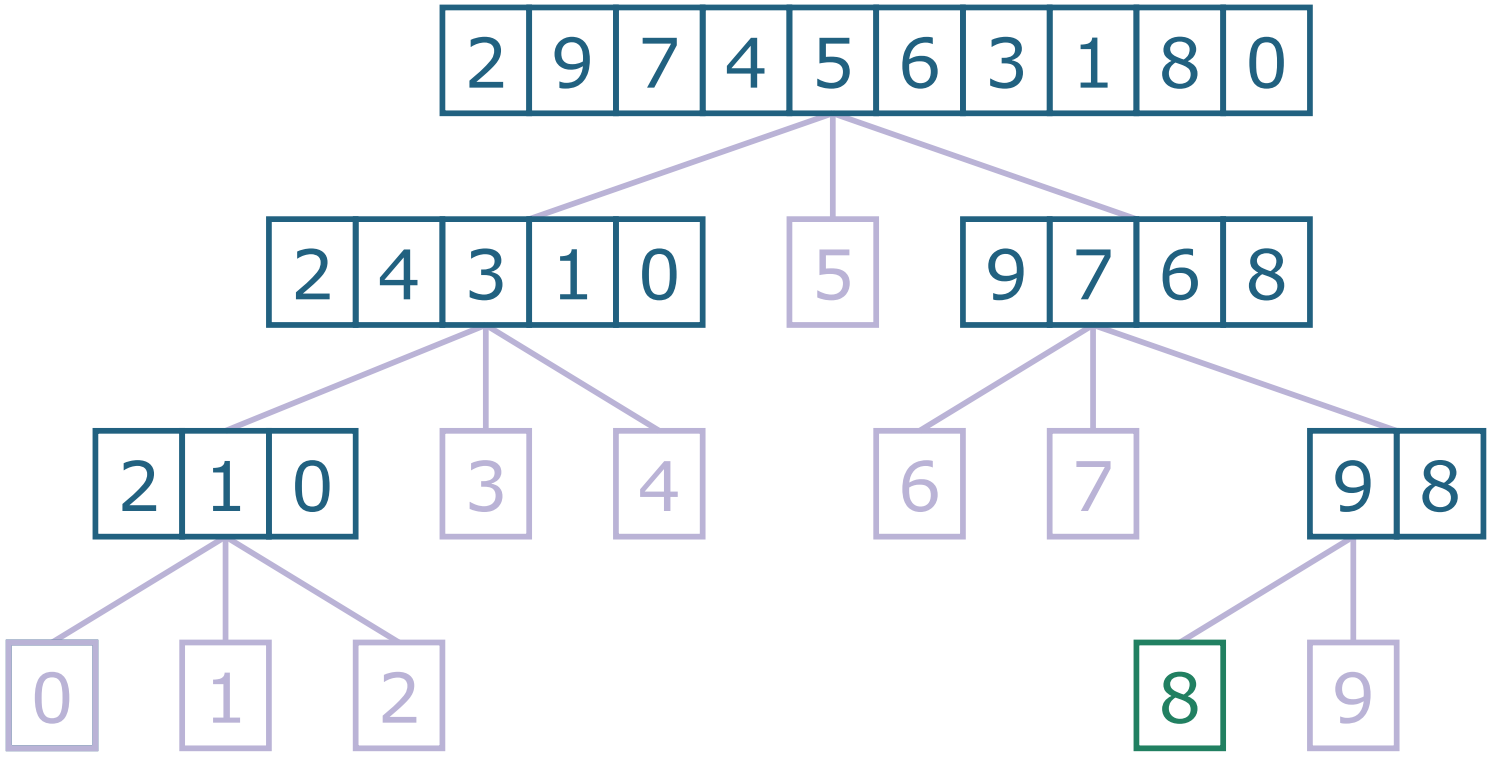
Dinâmica de Funcionamento



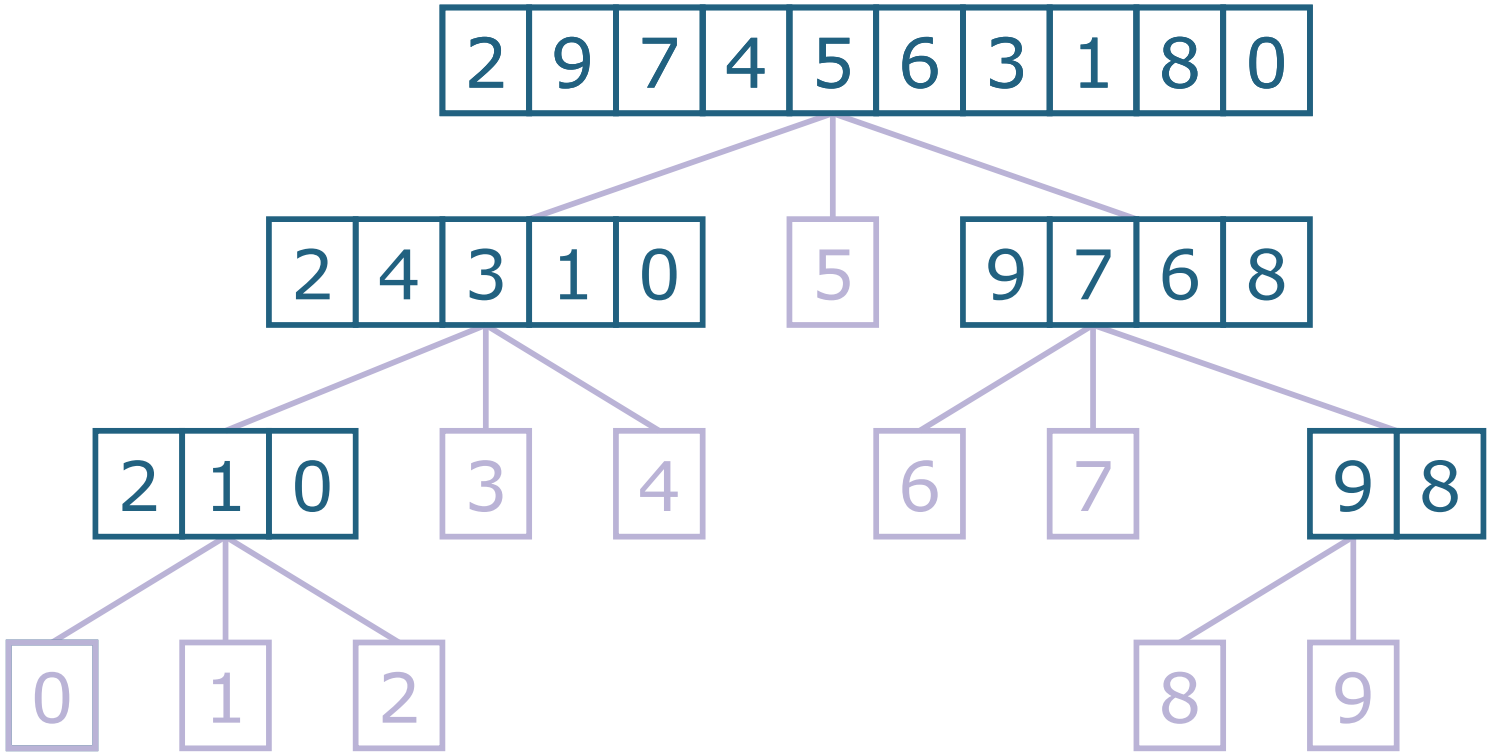
Dinâmica de Funcionamento



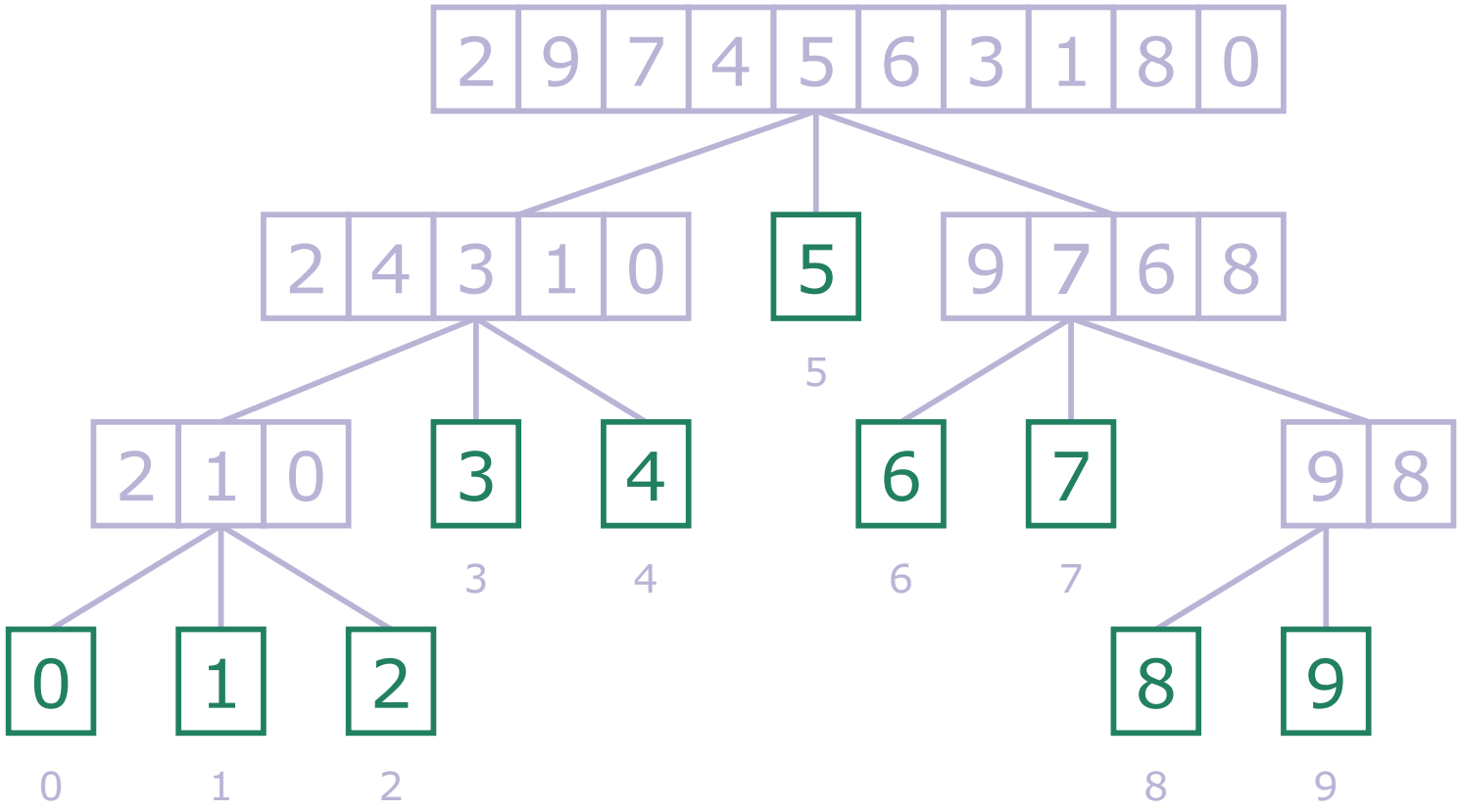
Dinâmica de Funcionamento



Dinâmica de Funcionamento



Dinâmica de Funcionamento



Dinâmica de Funcionamento

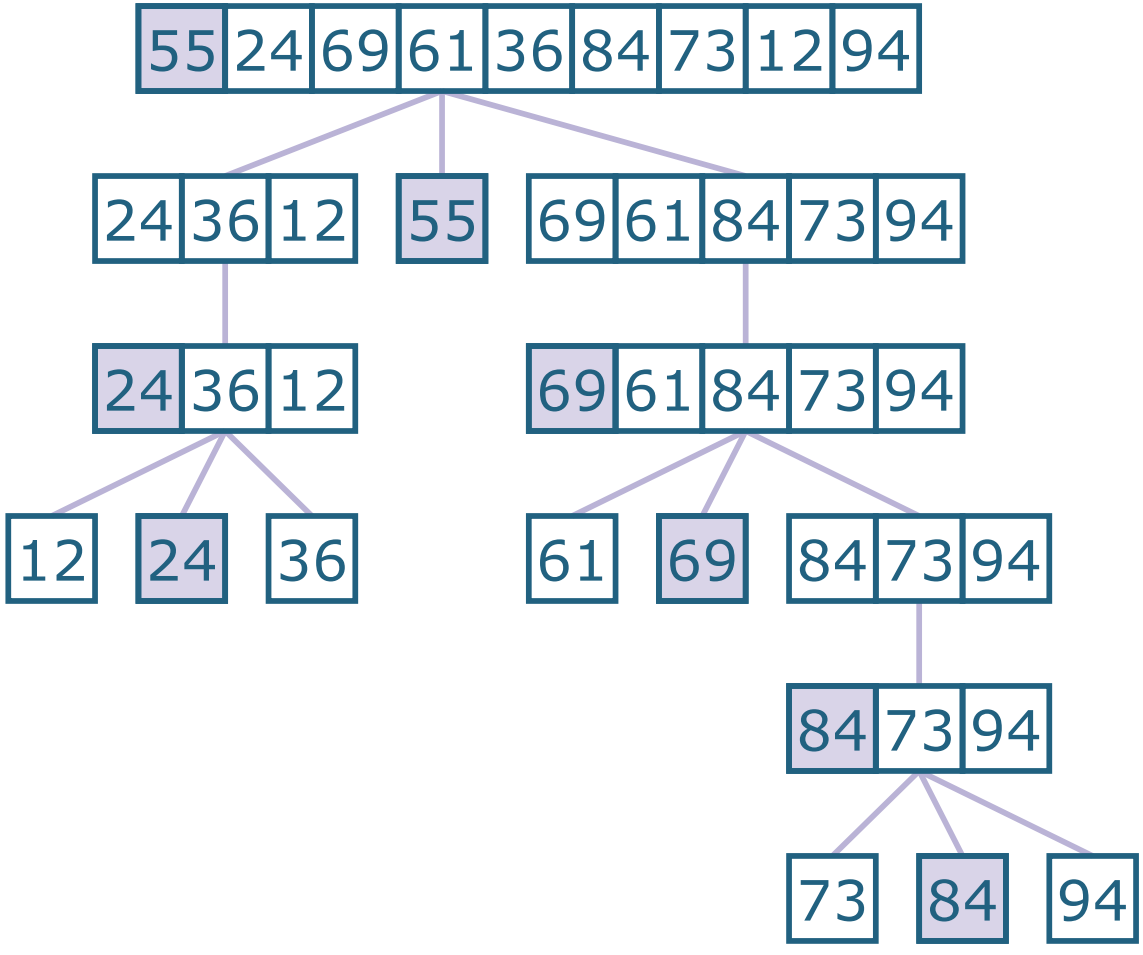
- A cada iteração pelo menos um elemento é alocado em sua **posição final** e não será mais manipulado nas iterações seguintes.
- O pivô de cada iteração **sempre é alocado** em sua posição final.
- **Cuidado:** a ordem dos elementos das partições geradas varia conforme a implementação.

Escolha dos Pivôs

- Qualquer **elemento da sequência** pode ser utilizado como pivô.
- O **melhor pivô** garante partições de mesmo tamanho (ou tamanho diferindo em 1).
- O **pior pivô** cria uma partição vazia. Por exemplo, se o pivô é o primeiro ou último elemento de uma sequência já ordenada.
- Possíveis escolhas de pivôs:
  - **Primeiro** elemento.
  - **Último** elemento.
  - Elemento do **meio**.
  - **Mediana** entre primeio, último e elemento do meio.
  - Um elemento **aleatório** da sequência.

Escolha dos Pivôs

Simulação admitindo o primeiro elemento como pivô.



Quick Sort

Implementação em Haskell (funcionalista)

```
qsort [] = []
qsort (p:l) = qsort(filter(< p) l) ++ [p] ++ qsort(filter(>= p) l)
```

- **Base recursiva:** lista vazia.
- **Recursão:**
  - chame a cabeça da lista (primeiro elemento) de ***p***.
  - filtre todos os elementos menores que *p* para a **primeira partição** e os maiores que (ou iguais a) *p* para a **segunda partição**.
  - aplique o método em **cada partição** em separado.
  - **concatene** a primeira partição, *p* e a segunda partição.

Lógica de Implementação (2)

- A maioria das implementações não tratam repetições do pivô na sequência, pois consideram as **chaves de ordenação** como **exclusivas**.
- Chaves não exclusivas podem demandar uma partição para os pivôs.
- Implementação facilitada com o uso de **listas**.

Quick Sort

Pseudocódigo Considerando uma Partição para os Pivôs

```
function quicksort(array)
    less, equal, greater := three empty arrays
    if length(array) > 1
        pivot := select any element of array
        for each x in array
            if x < pivot then add x to less
            if x = pivot then add x to equal
            if x > pivot then add x to greater
        quicksort(less)
        quicksort(greater)
        array := concatenate(less, equal, greater)
```

Quick Sort

Lógica de Implementação (3)

- Também se baseia na ideia de **partições**.
- Selecione um elemento arbitrário, chamado de **pivô**, faça a **partição** da matriz em duas seções. Aloque todos os **elementos maiores** (ou iguais) ao pivô de um lado (direito) e todos os elementos **menores** (ou também iguais) do que o pivô do outro lado (esquerdo).
- **Repita** o passo anterior em cada partição até que a matriz esteja ordenada.
- É a implementação mais comum para as linguagens Imperativistas.

Quick Sort

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define TAM 20

int main()
{
    srand(time(NULL));

    Printf("Quick Sort\n");
    printf("==========\n\n\n");

    int dados[TAM], i = 0;
    printf("Sequencia Gerada\n");
    printf("----------------\n");
    for (i = 0; i < TAM; i++)
    {
        dados[i] = rand() % 100;
        printf("%2d ", dados[i]);
    }
```

Quick Sort

```
    // Método de ordenação
    quicksort(dados, 0, TAM-1);

    printf("\n\n");
    printf("Sequencia Ordenada\n");
    printf("------------------\n");
    for (i = 0; i < TAM; i++)
    {
        printf("%2d ", dados[i]);
    }

    return(0);
}
```

Quick Sort

```
void quicksort(int v[TAM], int inicio, int fim)
{
   int meio, i, j, pivo, aux;
   meio = (inicio + fim) / 2;
   pivo = v[meio];
   i = inicio;
   j = fim;

   while (i < j)
   {
      while (v[i] < pivo)
         i = i + 1;

      while (v[j] > pivo)
         j = j - 1;
```

Quick Sort

```
        if (i <= j)
        {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
            i = i + 1;
            j = j - 1;
        }
    }

    if (j > inicio)
        quicksort(v, inicio, j);

    if (i < fim)
        quicksort(v, i, fim);
}
```

Técnica da Divisão e Conquista

- **Divida**, **conquiste** e **combine**.

- **Quick Sort**
  - O maior esforço ocorre no particionamento (**divisão**).
  - A concatenação das subsequências é trivial (**combinação**).

- **Merge Sort**
  - A repartição das subsequências é trivial (**divisão**).
  - A intercalação exige maior esforço (**combinação**)

Outros Métodos de Ordenação

- Na literatura e na Internet há outros métodos, geralmente detalhados e **comparados** em termos das respectivas complexidades:
  - **Temporal** (velocidade no caso médio, pior e melhor).
  - **Espacial** (consumo de memória).

- Sorthing Algorithm

  http://en.wikipedia.org/wiki/Sorting_algorithm