Ordenação com Heapsort

Projeto e Análise de Algoritmos

Bruno Prado

Departamento de Computação / UFS

Introdução

- O que é o Heapsort?
  - É um algoritmo de ordenação sem estabilidade desenvolvido por J. W. J. Williams em 1964

Introdução

- O que é o Heapsort?
  - É um algoritmo de ordenação sem estabilidade desenvolvido por J. W. J. Williams em 1964
  - Emprega a estratégia de Transformação e Conquista que converte os dados para uma estrutura de *heap*

Introdução

- Estratégia de Transformação e Conquista
  1. Etapa de transformação do problema
     - Estruturação dos dados

Introdução

- Estratégia de Transformação e Conquista
  1. Etapa de transformação do problema
     - Estruturação dos dados
  2. Aplicação de propriedades das estruturas
     - A organização dos dados permitem simplificações

Introdução

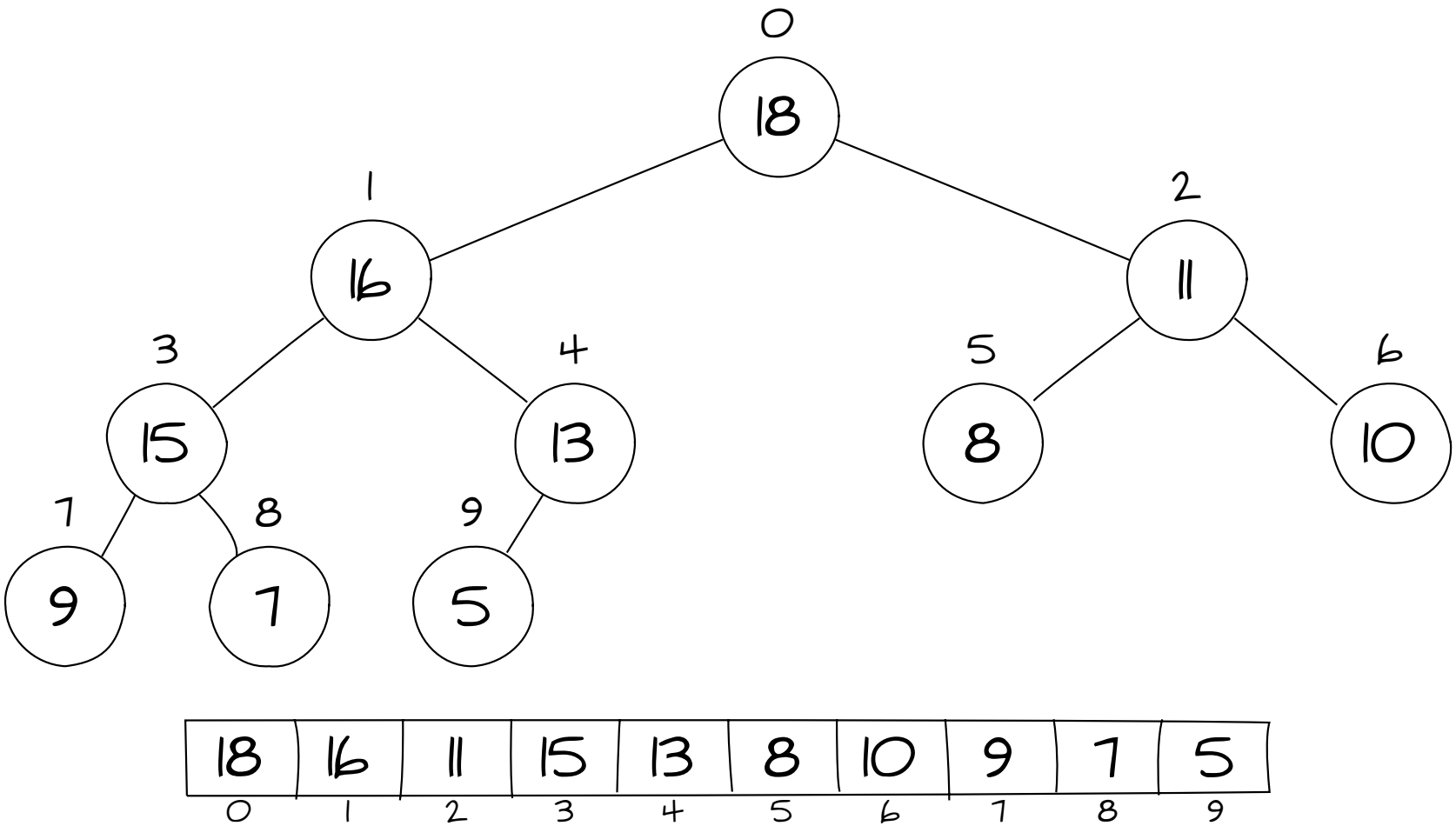
- Estratégia de Transformação e Conquista
  1. Etapa de transformação do problema
     - Estruturação dos dados
  2. Aplicação de propriedades das estruturas
     - A organização dos dados permitem simplificações
  3. Etapa de conquista da solução completa
     - Os resultados parciais são combinados

Árvore *heap*

- O que é uma árvore *heap*?
  - Árvore binária de prioridade
  - Representação implícita em vetor
  - Percursos por indexação dos nós

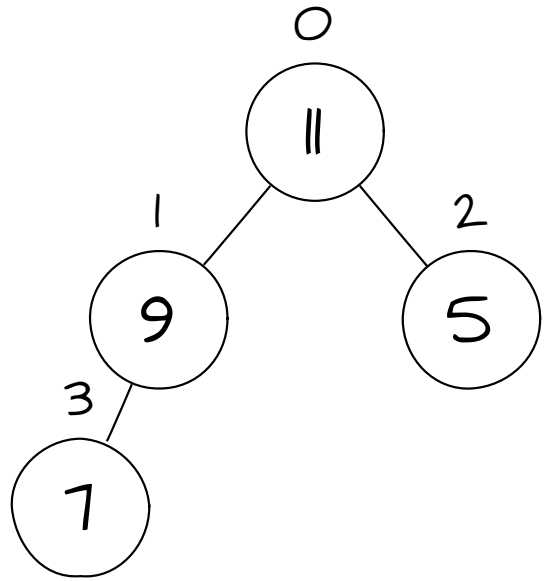
Árvore *heap*

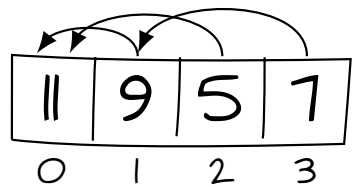
- Árvore binária *heap*



Árvore *heap*

- Representação e indexação
  - Nó pai

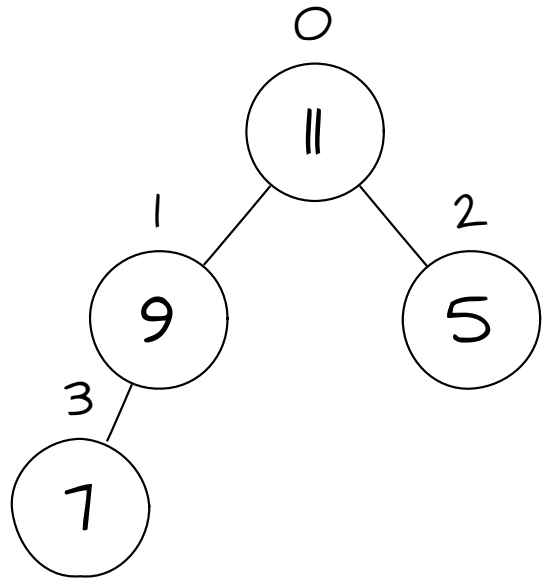


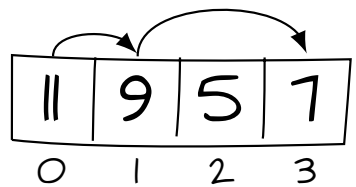


$$Pai(i) = \frac{i-1}{2}$$

Árvore *heap*

- Representação e indexação
  - Nó filho esquerdo

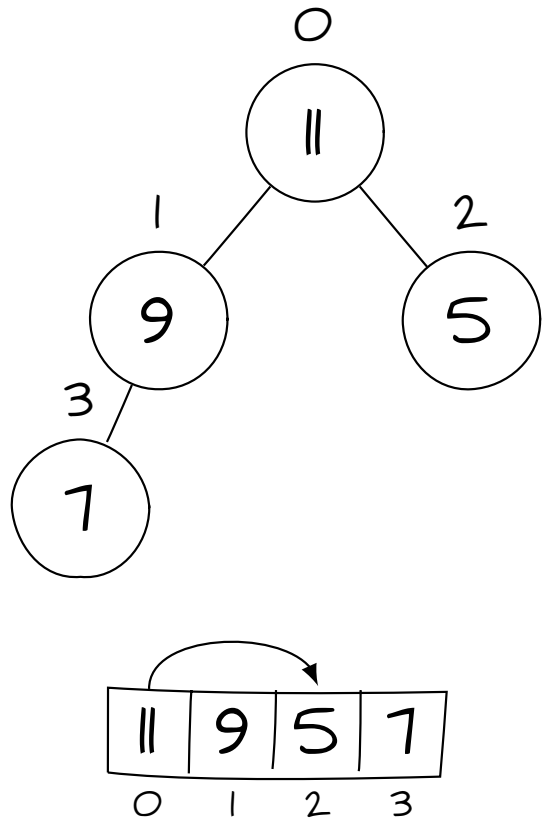




$$Esquerdo(i) = 2i + 1$$

Árvore *heap*

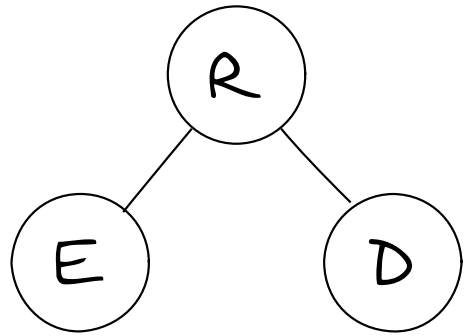
- Representação e indexação
  - Nó filho direito



$$Direito(i) = 2i + 2$$

Árvore *heap*

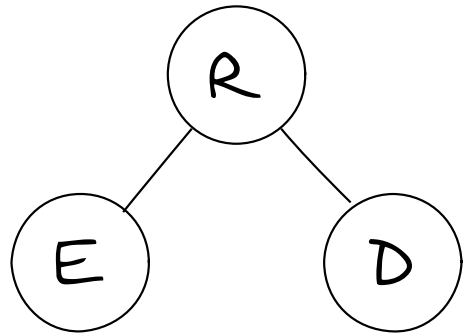
- Tipos de árvores *heap*
  - *Heap* mínimo



Propriedade $R \leq E$ e $R \leq D$

Árvore *heap*

- Tipos de árvores *heap*
  - *Heap* máximo



Propriedade $R \geq E$ e $R \geq D$

Árvore *heap*

- Aplicação da propriedade de *heap*

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Heapify recursivo
4   void heapify(int32_t* V, uint32_t T, uint32_t i) {
5       // Declaração dos índices
6       uint32_t P = i, E = esquerdo(i), D = direito(i);
7       // Filho da esquerda é maior
8       if(E < T && V[E] > V[P])
9           P = E;
10      // Filho da direita é maior
11      if(D < T && V[D] > V[P])
12          P = D;
13      // Troca e chamada recursiva
14      if(P != i) {
15          trocar(V, P, i);
16          heapify(V, T, P);
17      }
18  }
... ...
```

Heapsort

- Etapa de transformação

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
... ...
17  // Heapsort iterativo
18  void heapsort(int32_t V[], uint32_t n) {
19      // Construção do heap máximo ou mínimo
20      construir_heap(V, n);
21      // Iterando sobre os nós do heap
22      for(int32_t i = n - 1; i > 0; i--) {
23          // Trocando a raiz pelo i-ésimo
24          trocar(&V[0], &V[i]);
25          // Aplicando heapify na raiz
26          heapify(V, 0, i);
27      }
28  }
```

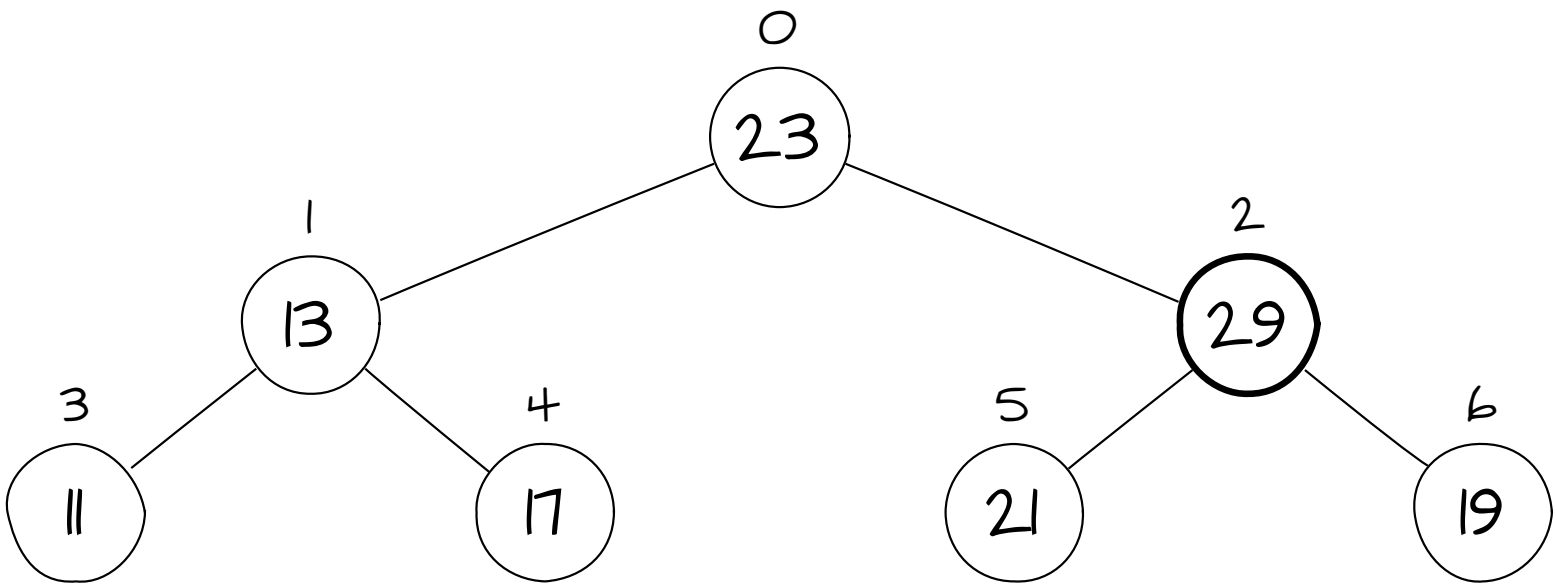
Heapsort

- Etapa de transformação

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
... ...
17  // Heapsort iterativo
18  void heapsort(int32_t V[], uint32_t n) {
19      // Construção do heap máximo ou mínimo
20      construir_heap(V, n);
21      // Iterando sobre os nós do heap
22      for(int32_t i = n - 1; i > 0; i--) {
23          // Trocando a raiz pelo i-ésimo
24          trocar(&V[0], &V[i]);
25          // Aplicando heapify na raiz
26          heapify(V, 0, i);
27      }
28  }
```

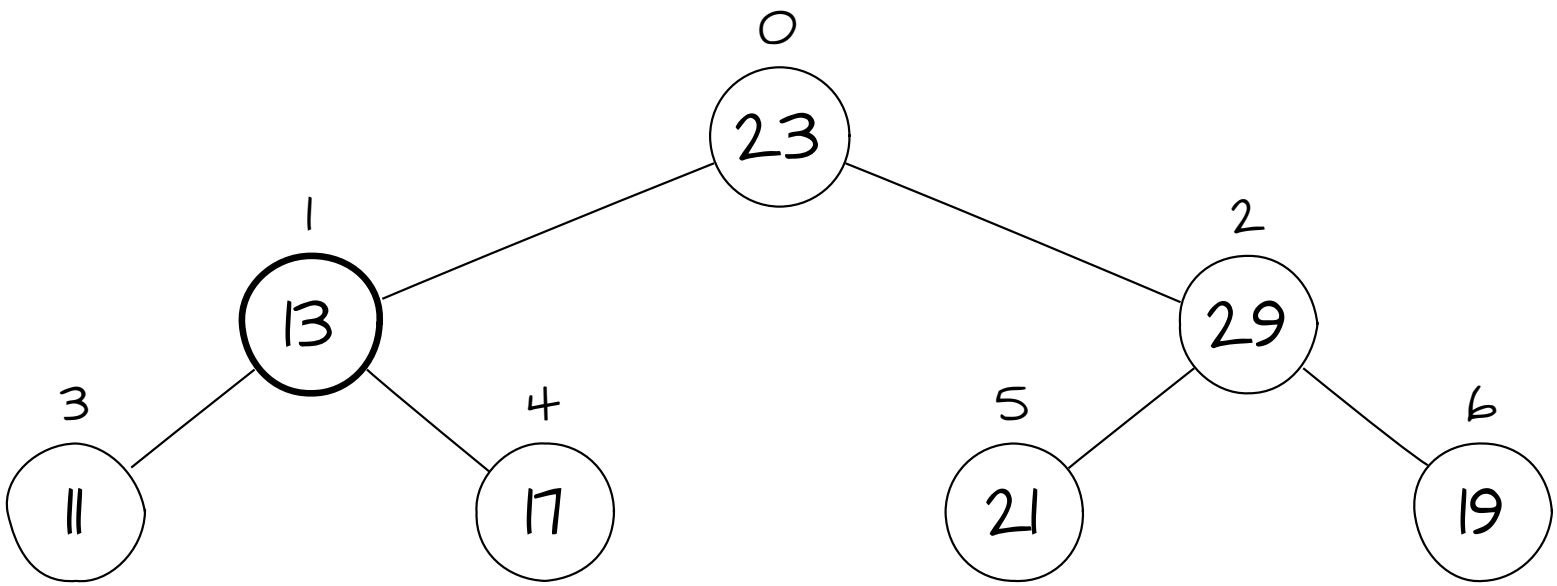
Heapsort

- Etapa de transformação
  - Começa pelo último nó com filhos
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



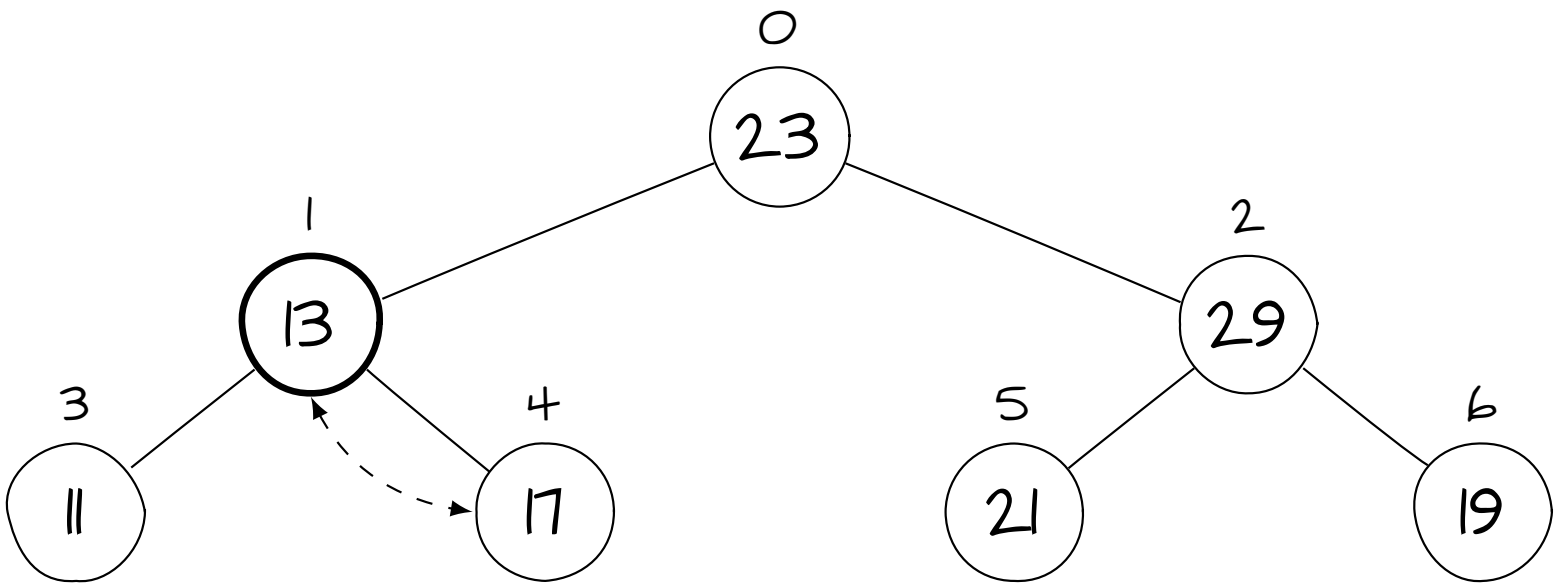
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



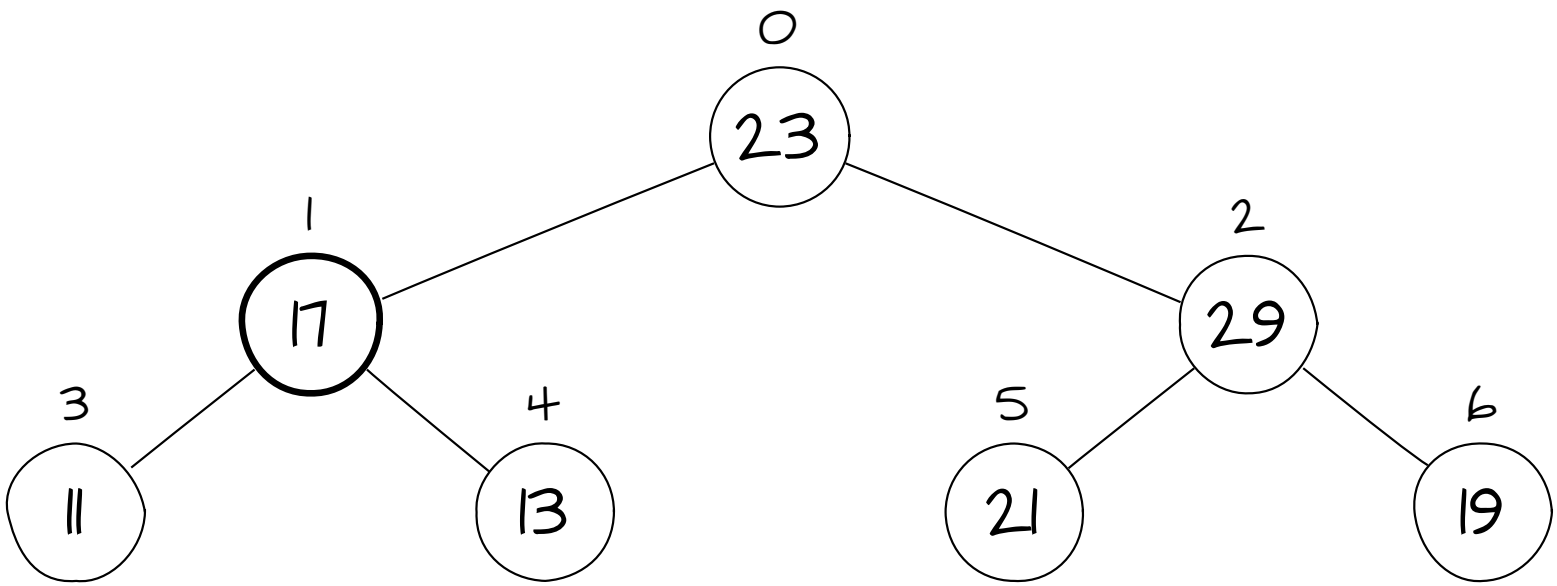
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



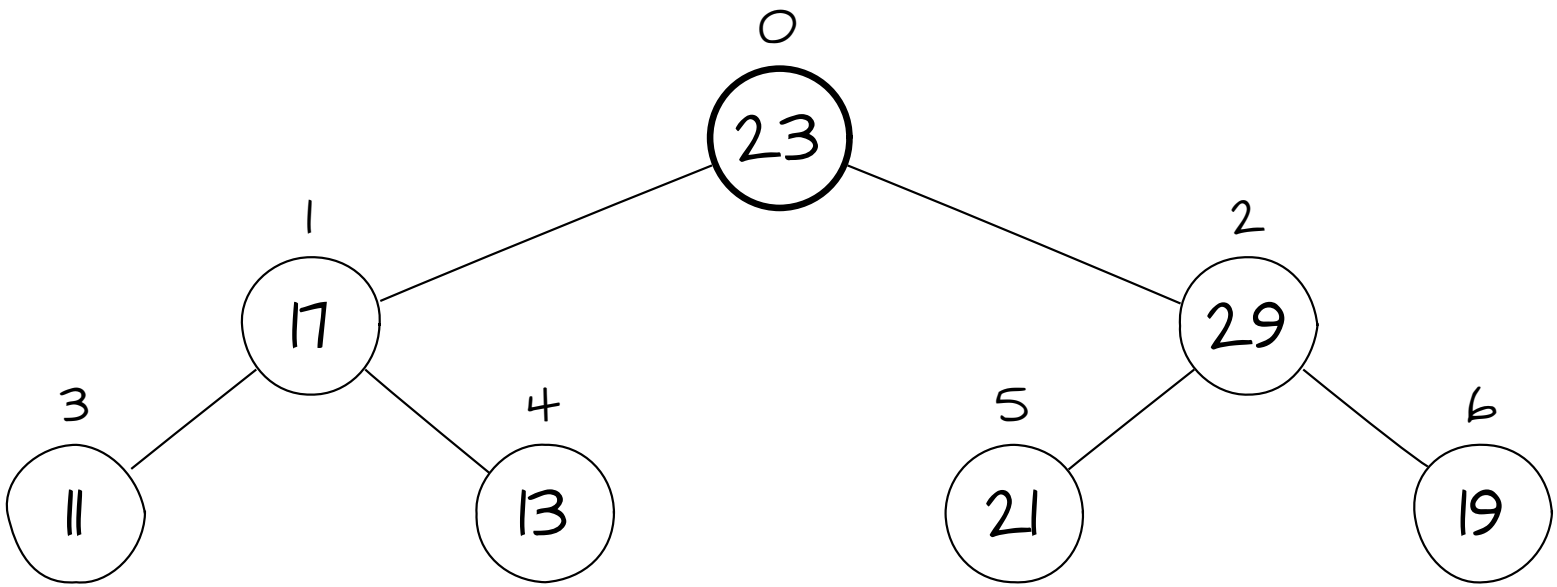
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



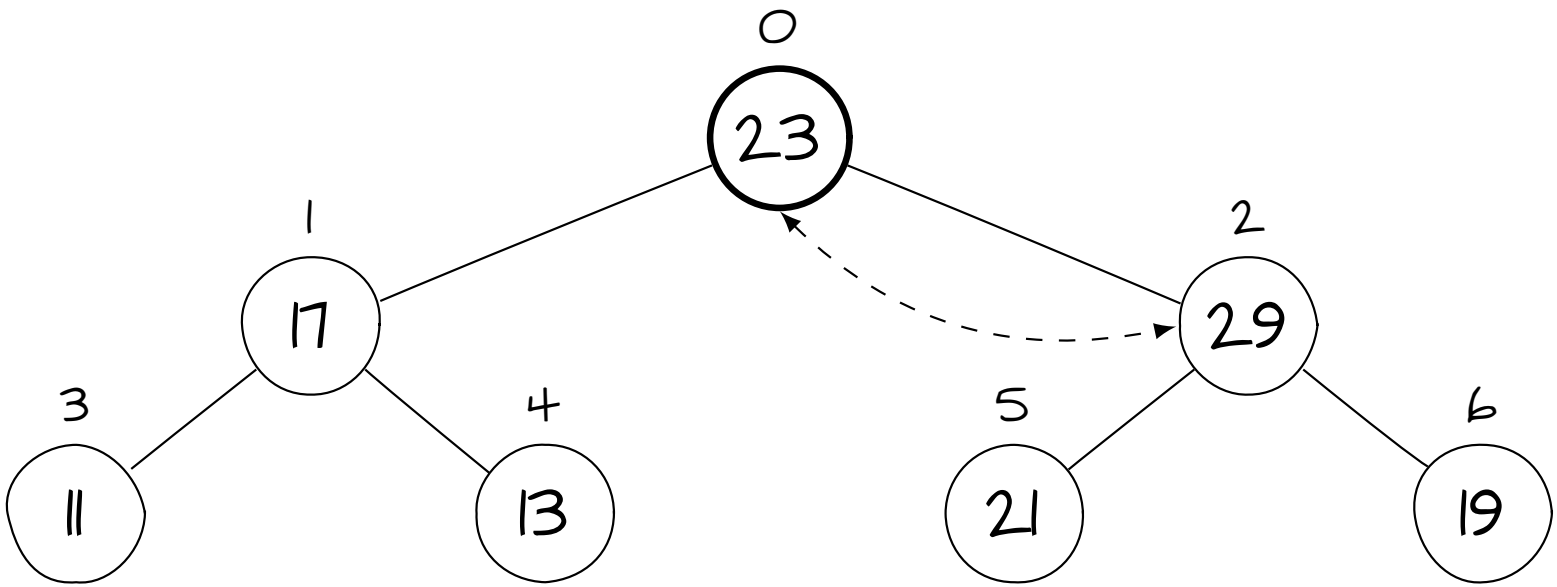
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



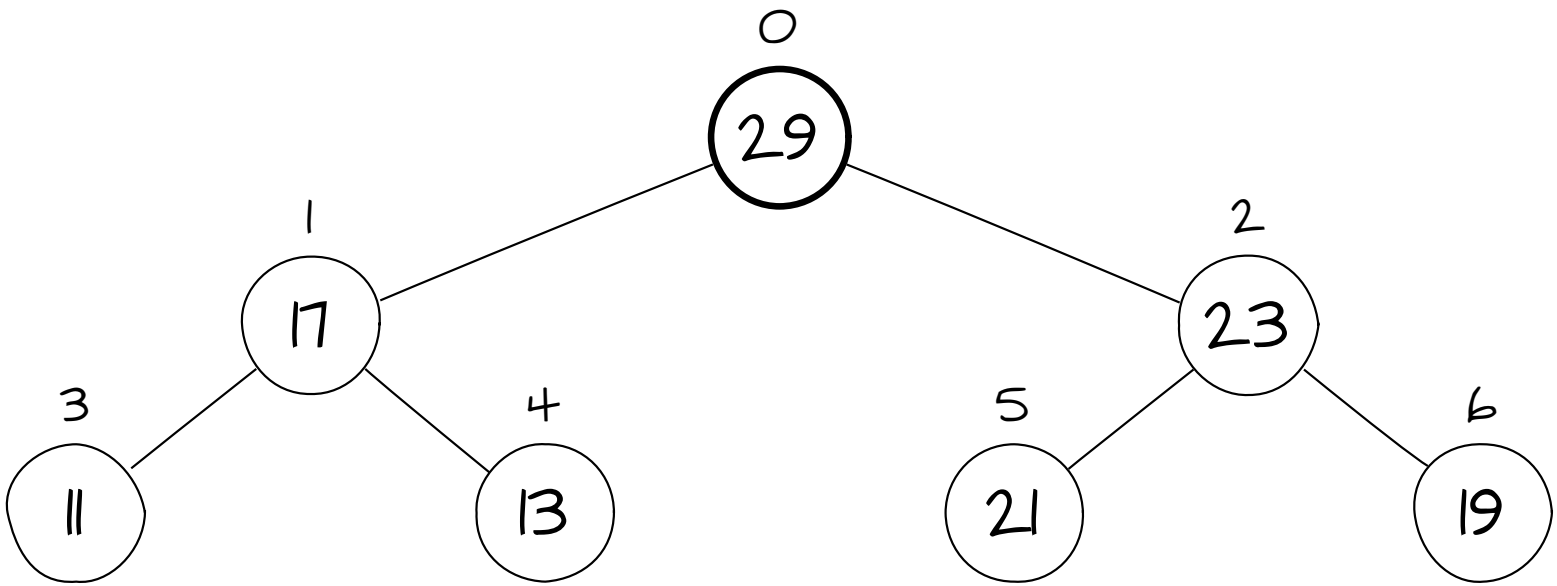
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



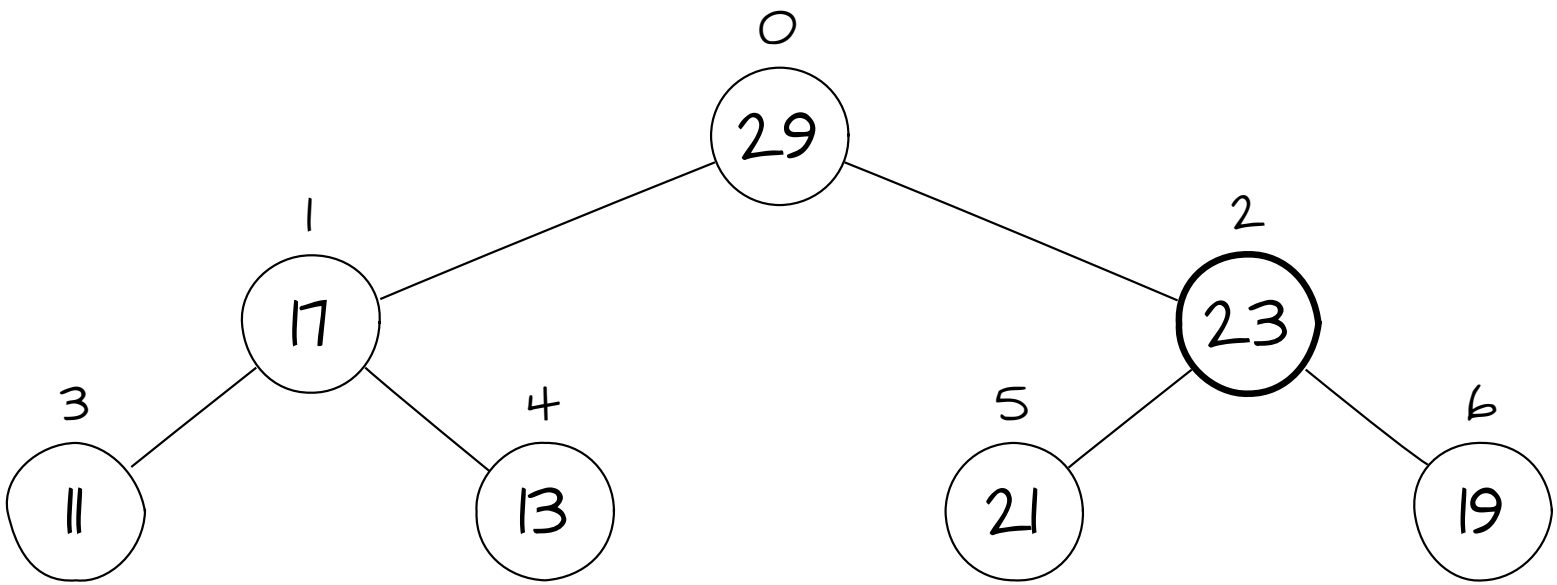
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



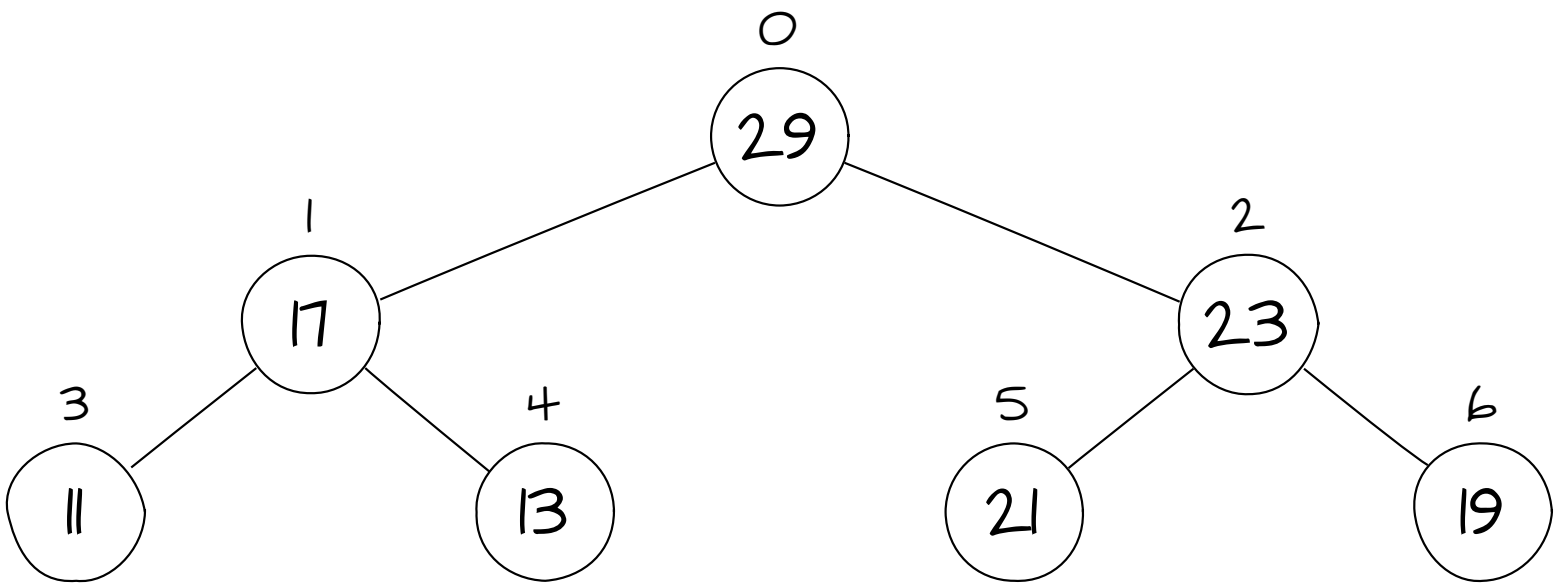
Heapsort

- Etapa de transformação
  - O índice é decrementado até atingir a raiz
  - *Heapify* no índice $i = \frac{(Tamanho-1)-1}{2}$



Heapsort

- Etapa de transformação
  - A construção do *heap* máximo foi finalizada



Heapsort

- Etapa de conquista

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
... ...
17  // Heapsort iterativo
18  void heapsort(int32_t V[], uint32_t n) {
19      // Construção do heap máximo ou mínimo
20      construir_heap(V, n);
21      // Iterando sobre os nós do heap
22      for(int32_t i = n - 1; i > 0; i--) {
23          // Trocando a raiz pelo i-ésimo
24          trocar(&V[0], &V[i]);
25          // Aplicando heapify na raiz
26          heapify(V, 0, i);
27      }
28  }
```

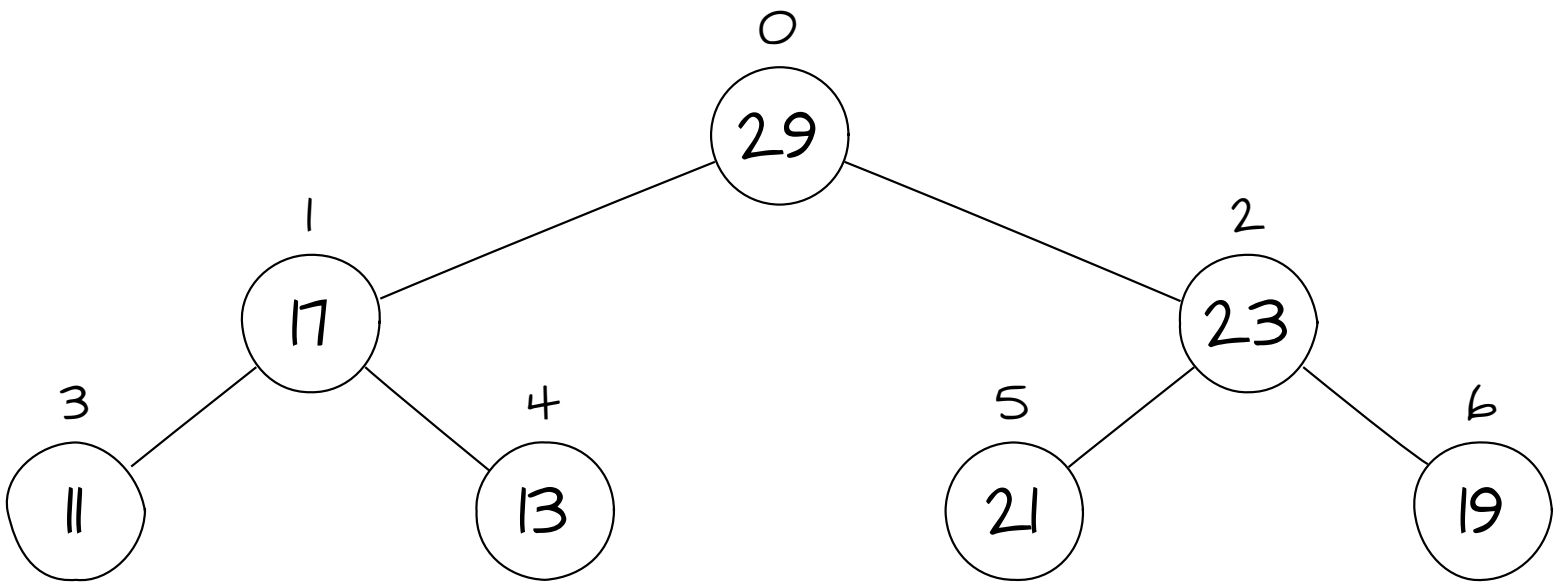
Heapsort

- Etapa de conquista

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
... ...
17  // Heapsort iterativo
18  void heapsort(int32_t V[], uint32_t n) {
19      // Construção do heap máximo ou mínimo
20      construir_heap(V, n);
21      // Iterando sobre os nós do heap
22      for(int32_t i = n - 1; i > 0; i--) {
23          // Trocando a raiz pelo i-ésimo
24          trocar(&V[0], &V[i]);
25          // Aplicando heapify na raiz
26          heapify(V, 0, i);
27      }
28  }
```

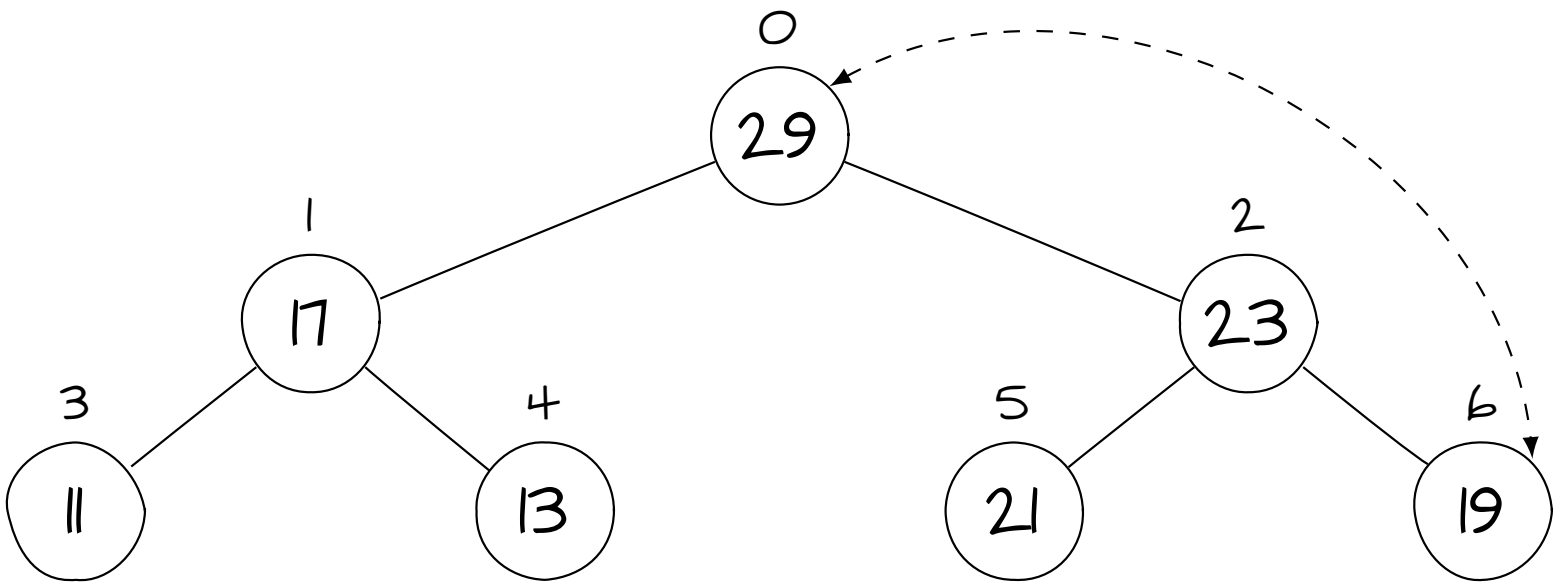
Heapsort

- Etapa de conquista



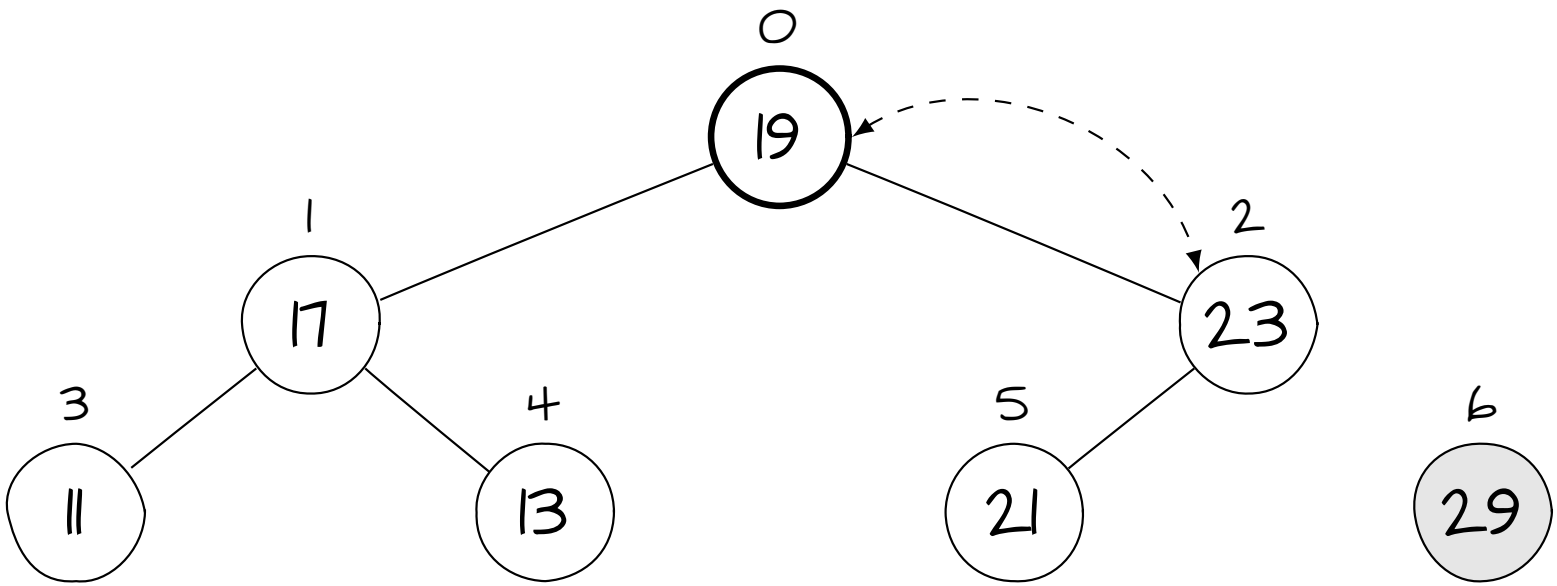
Heapsort

- Etapa de conquista



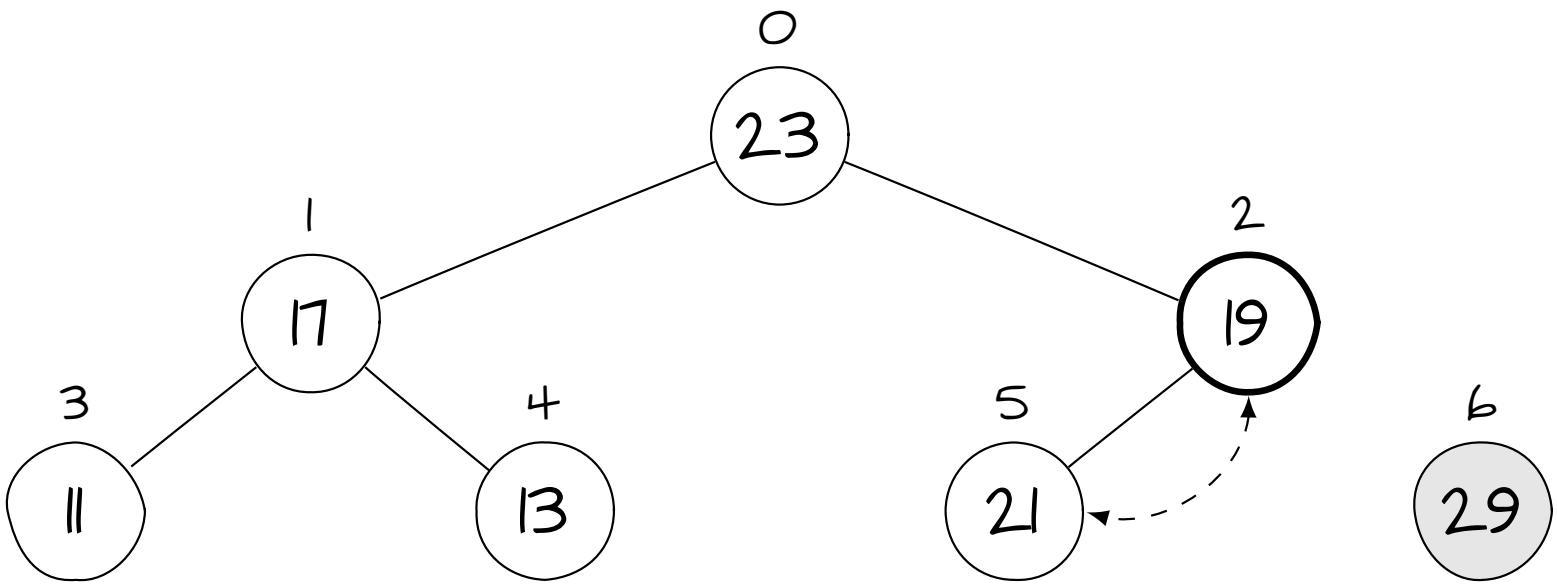
Heapsort

- Etapa de conquista



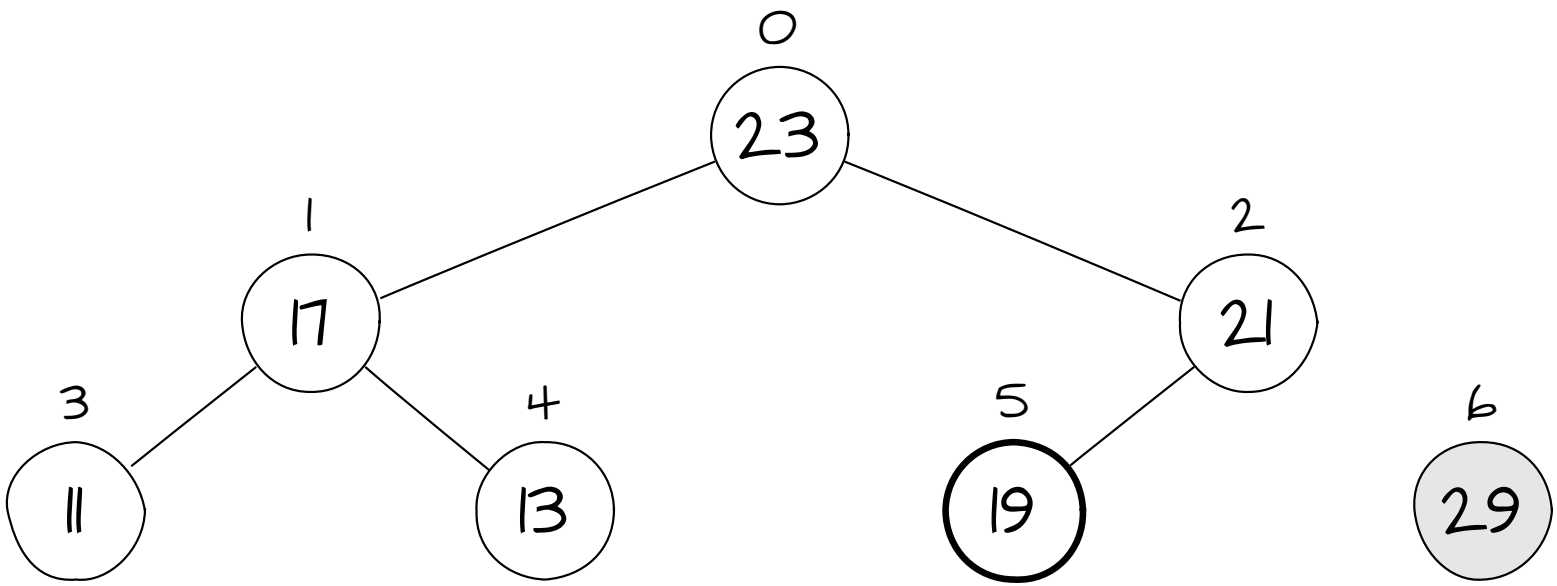
Heapsort

- Etapa de conquista



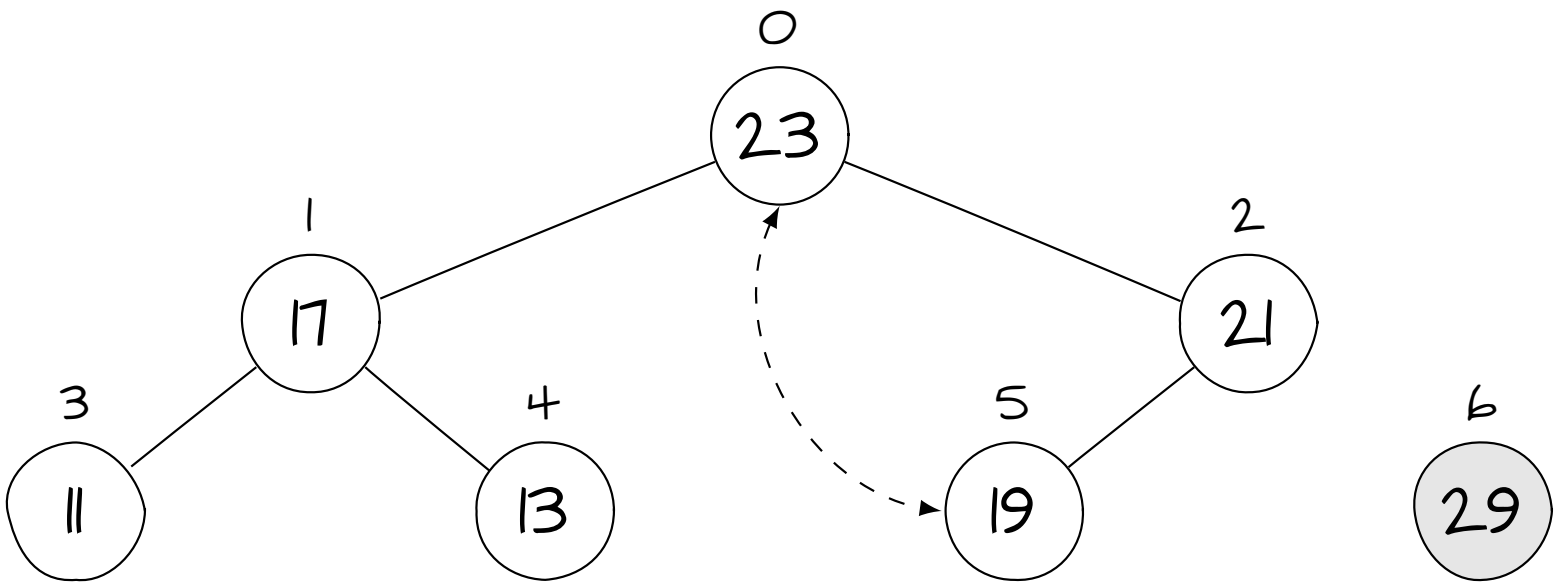
Heapsort

- Etapa de conquista



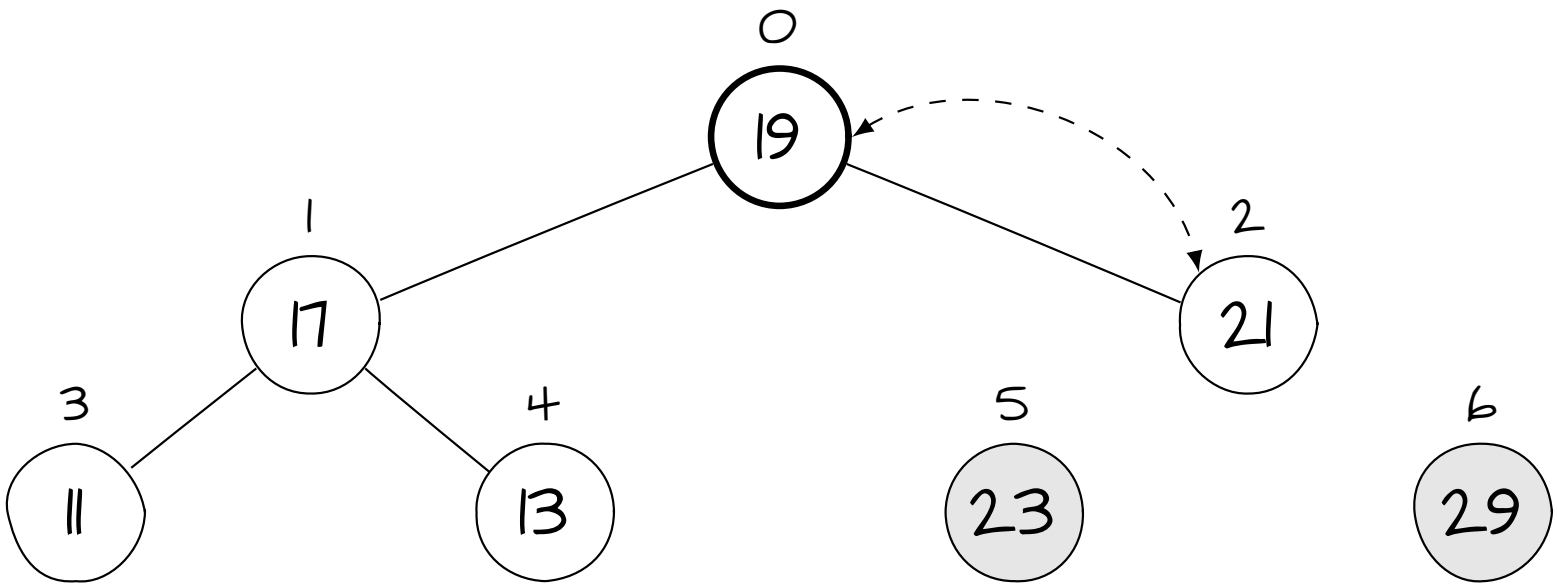
Heapsort

- Etapa de conquista



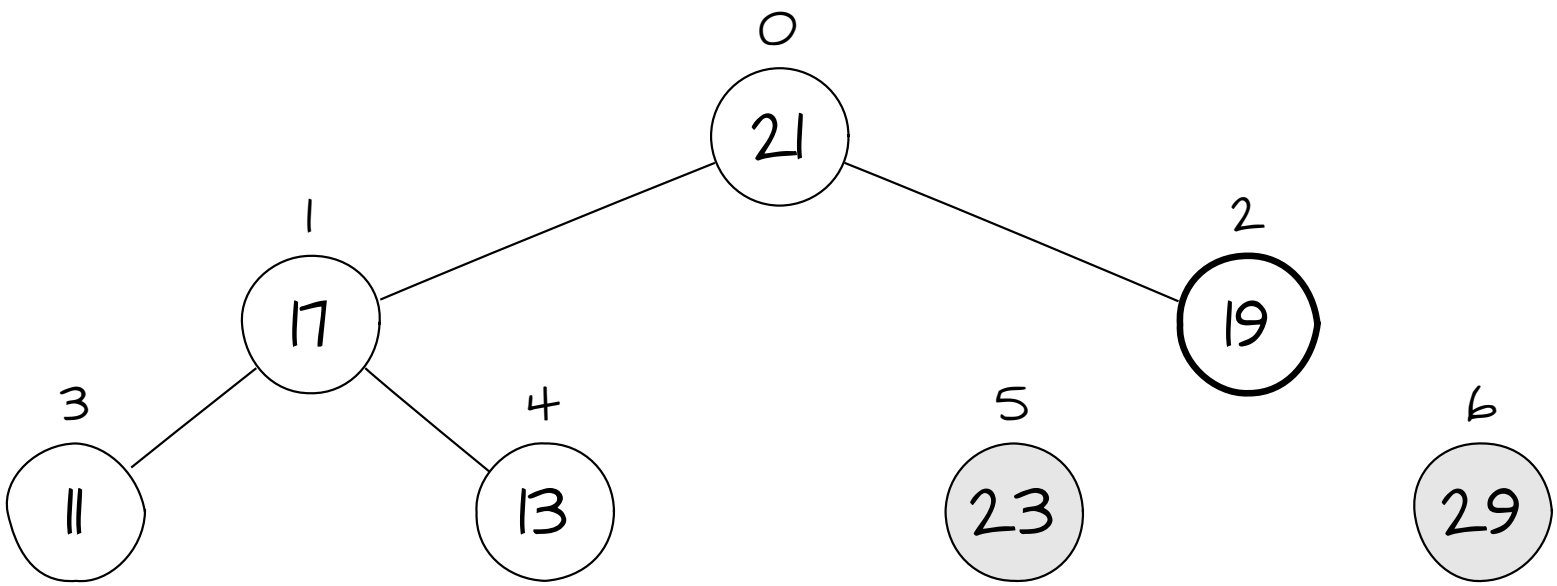
Heapsort

- Etapa de conquista



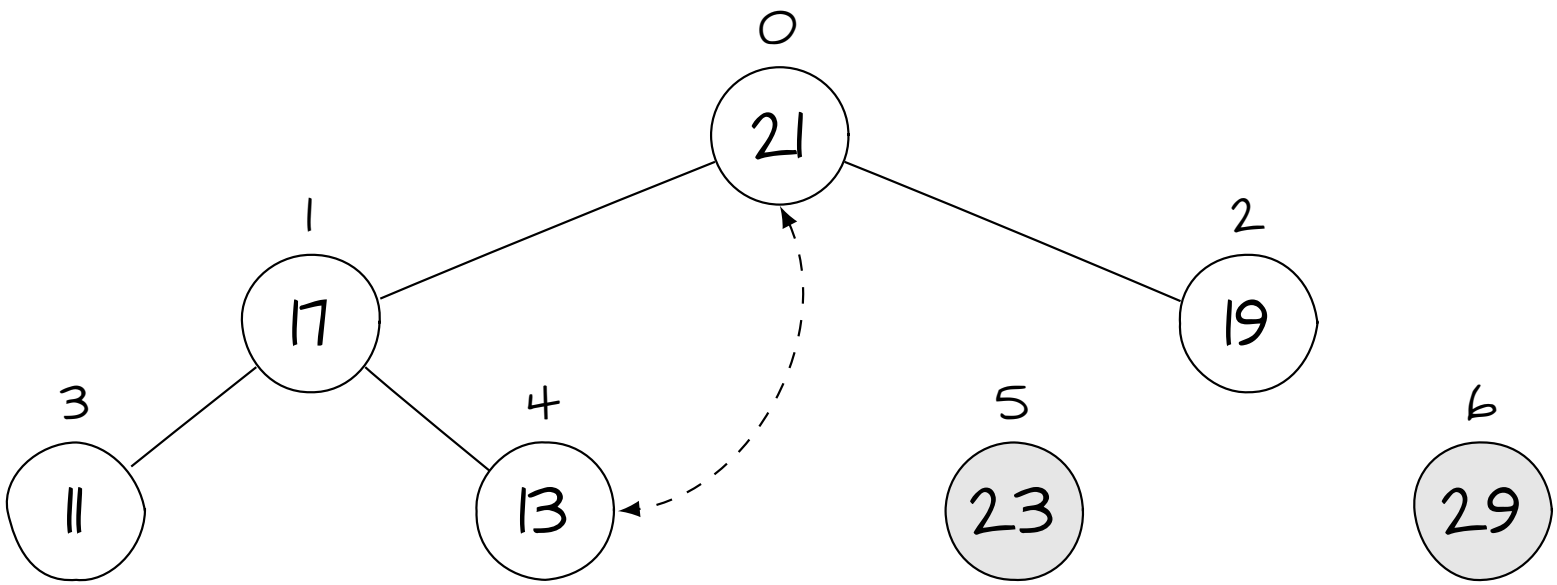
Heapsort

- Etapa de conquista



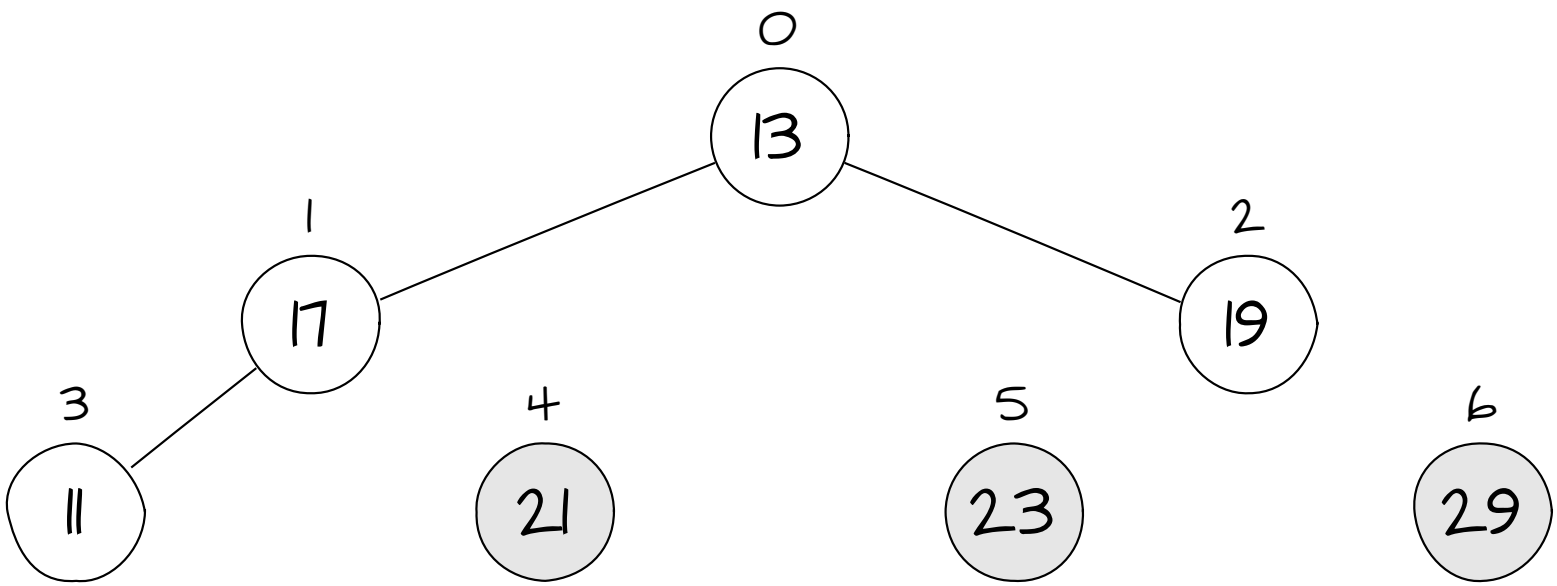
Heapsort

- Etapa de conquista



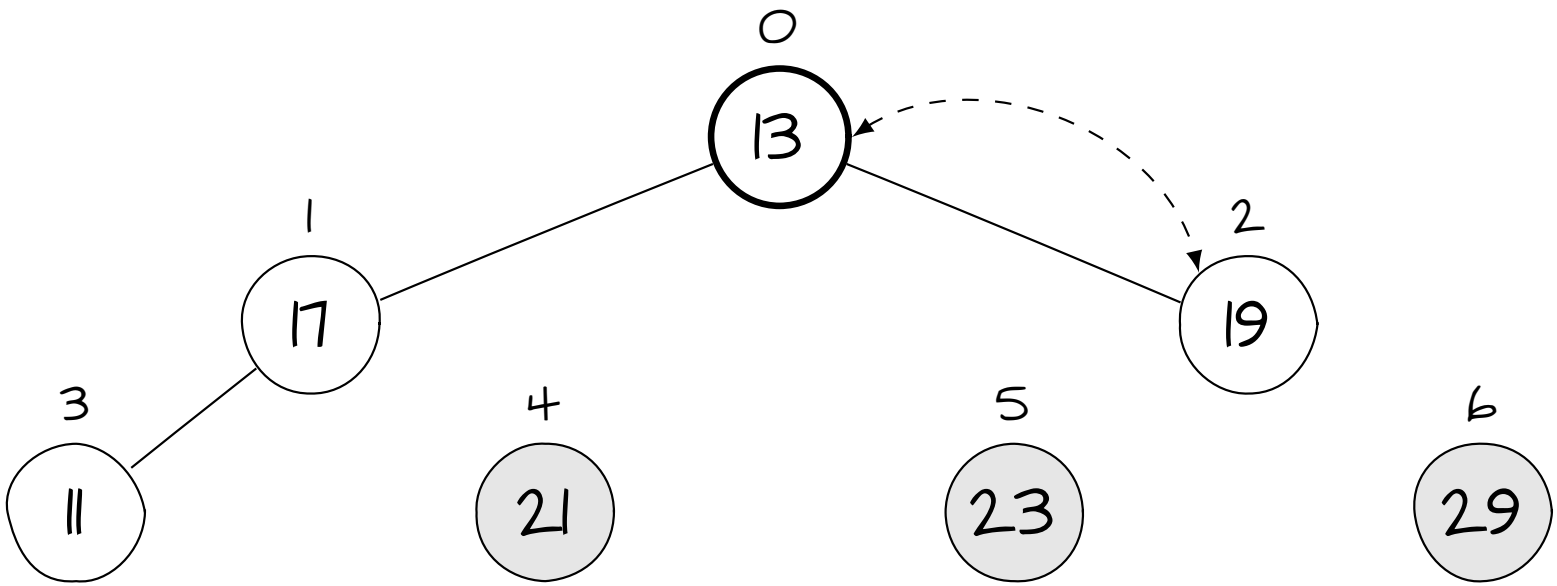
Heapsort

- Etapa de conquista



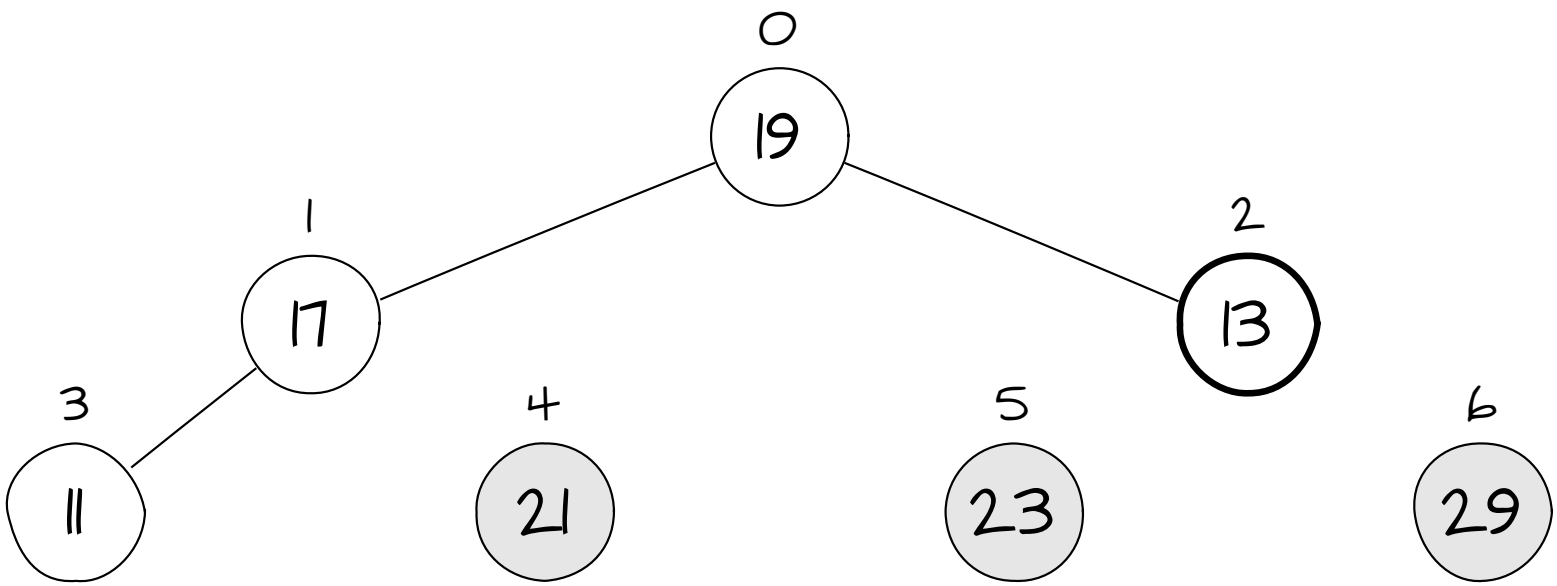
Heapsort

- Etapa de conquista



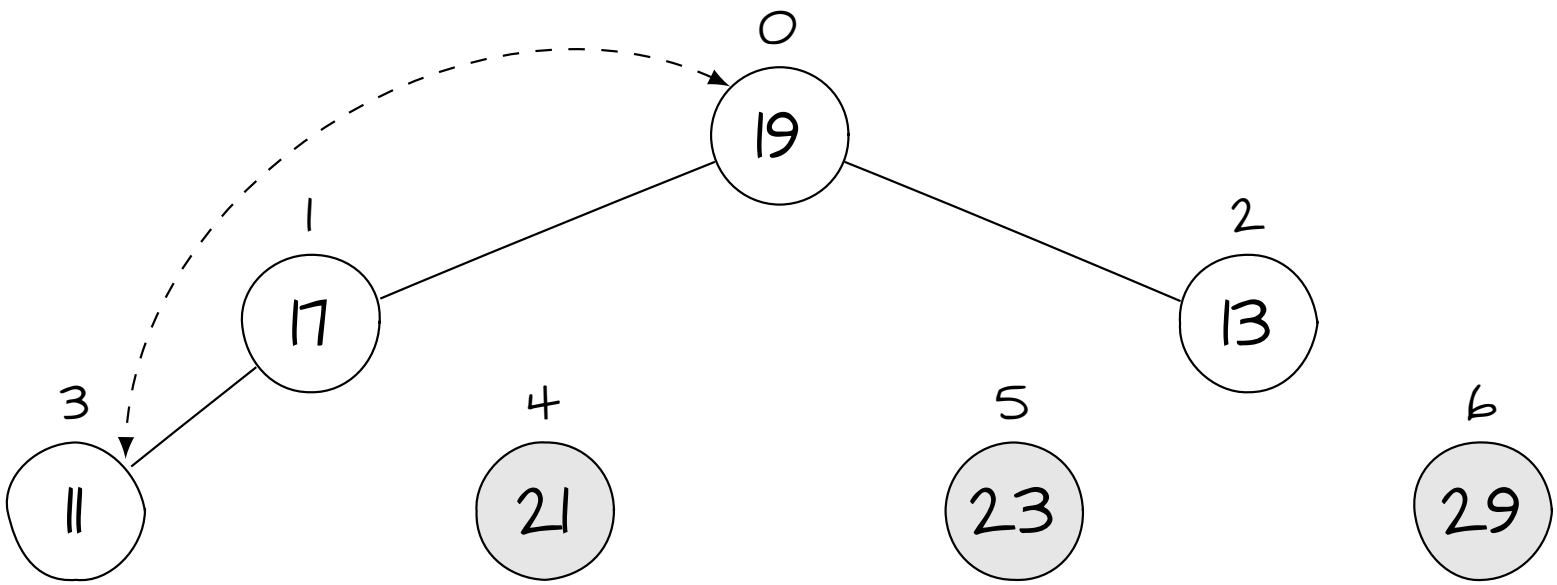
Heapsort

- Etapa de conquista



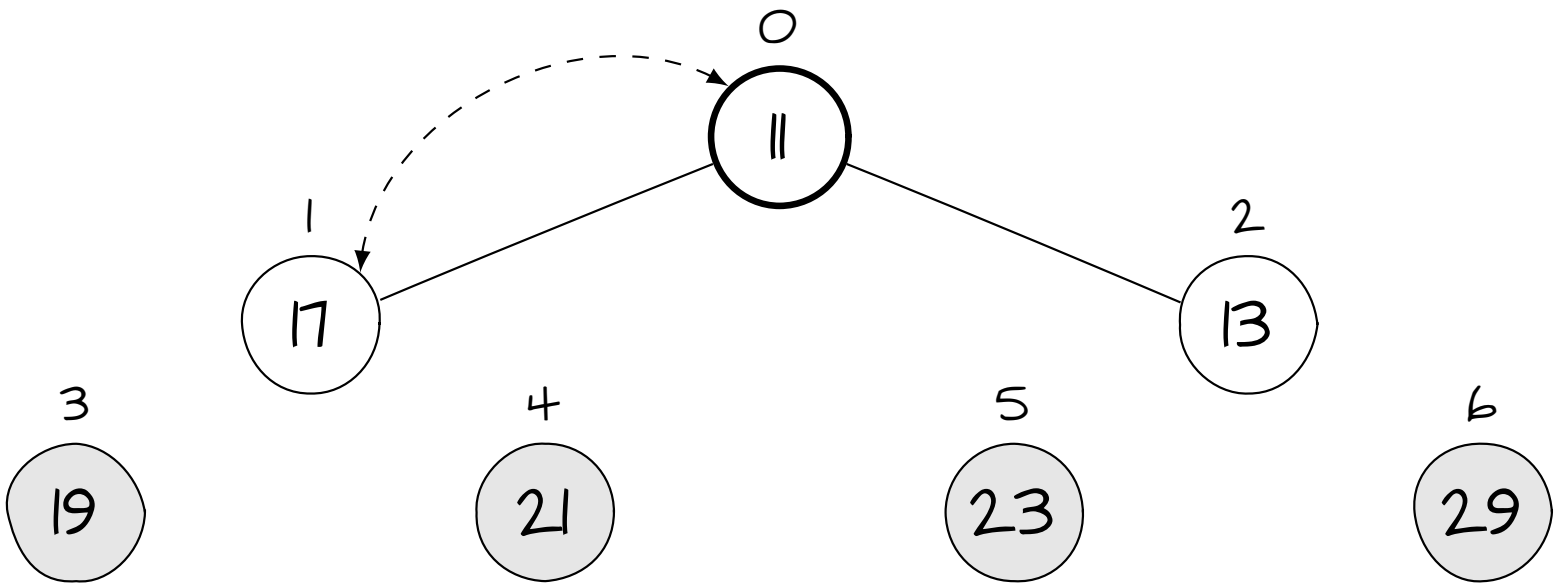
Heapsort

- Etapa de conquista



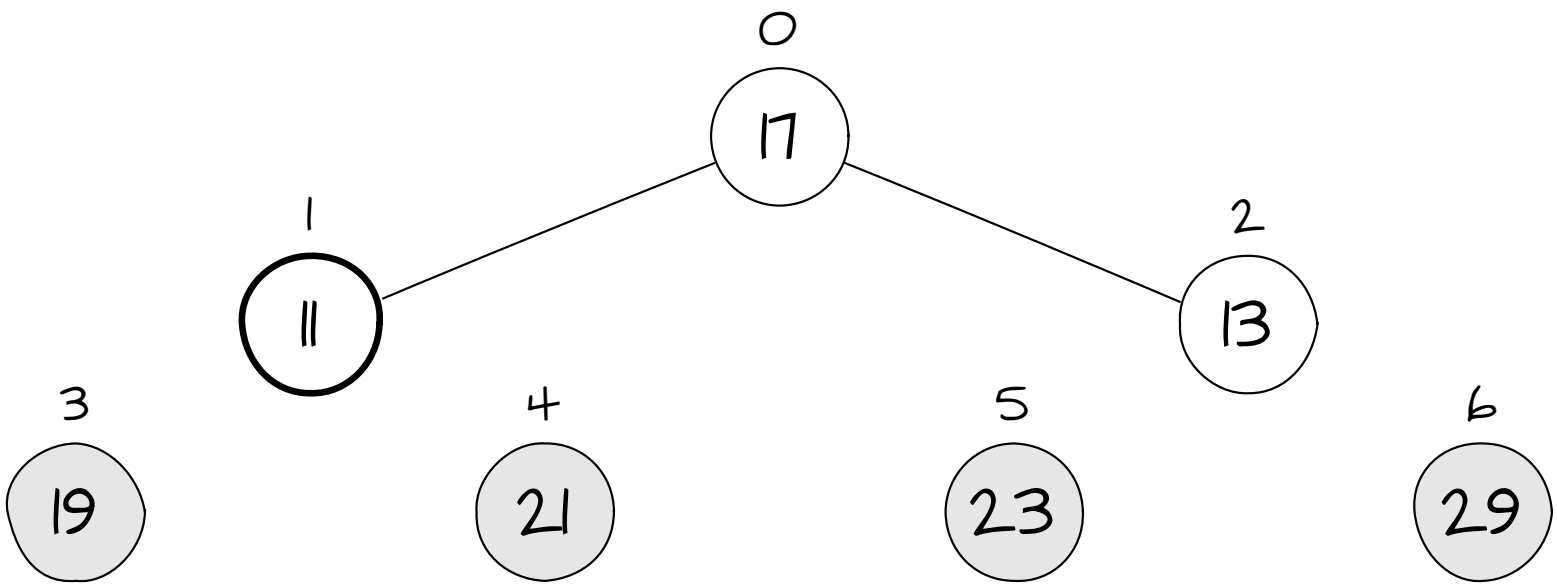
Heapsort

- Etapa de conquista



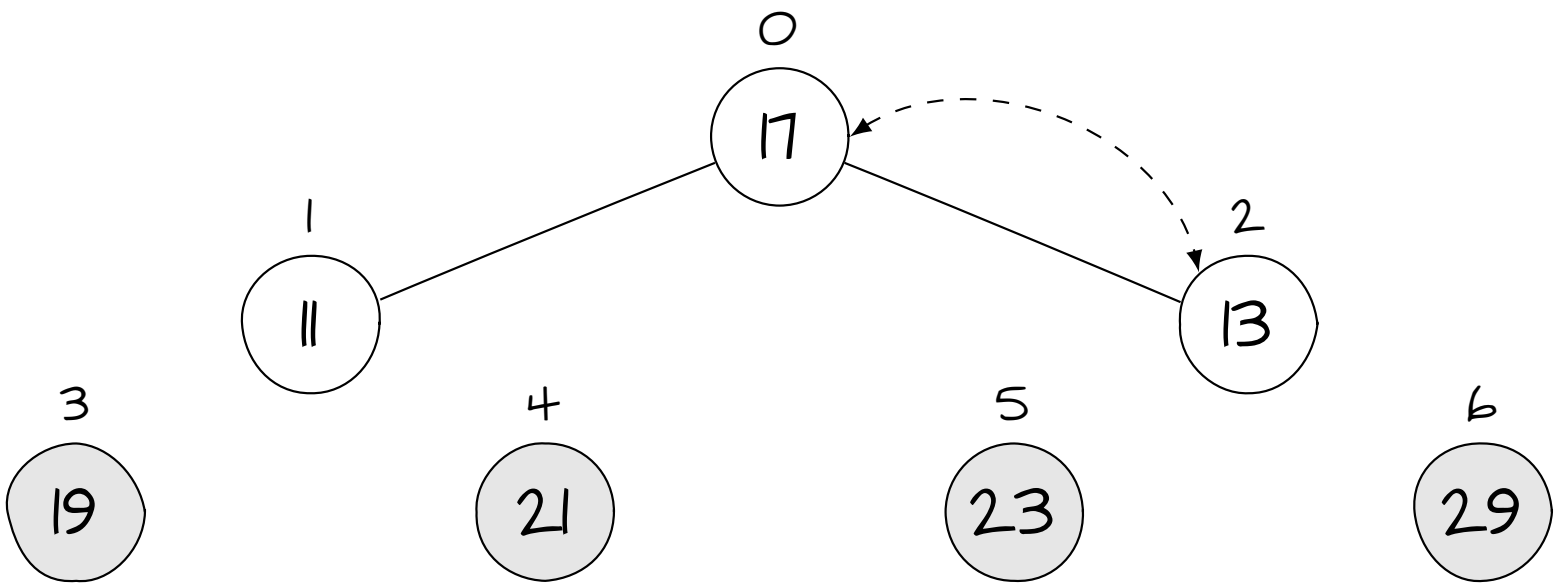
Heapsort

- Etapa de conquista



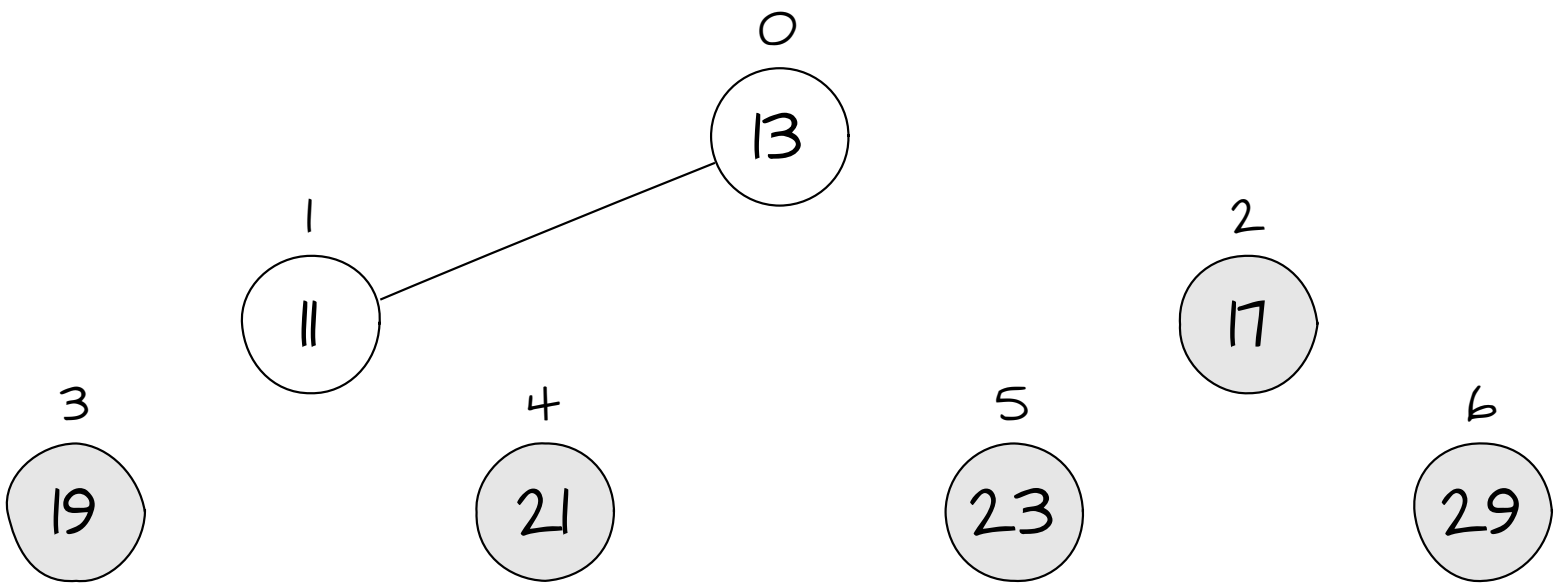
Heapsort

- Etapa de conquista



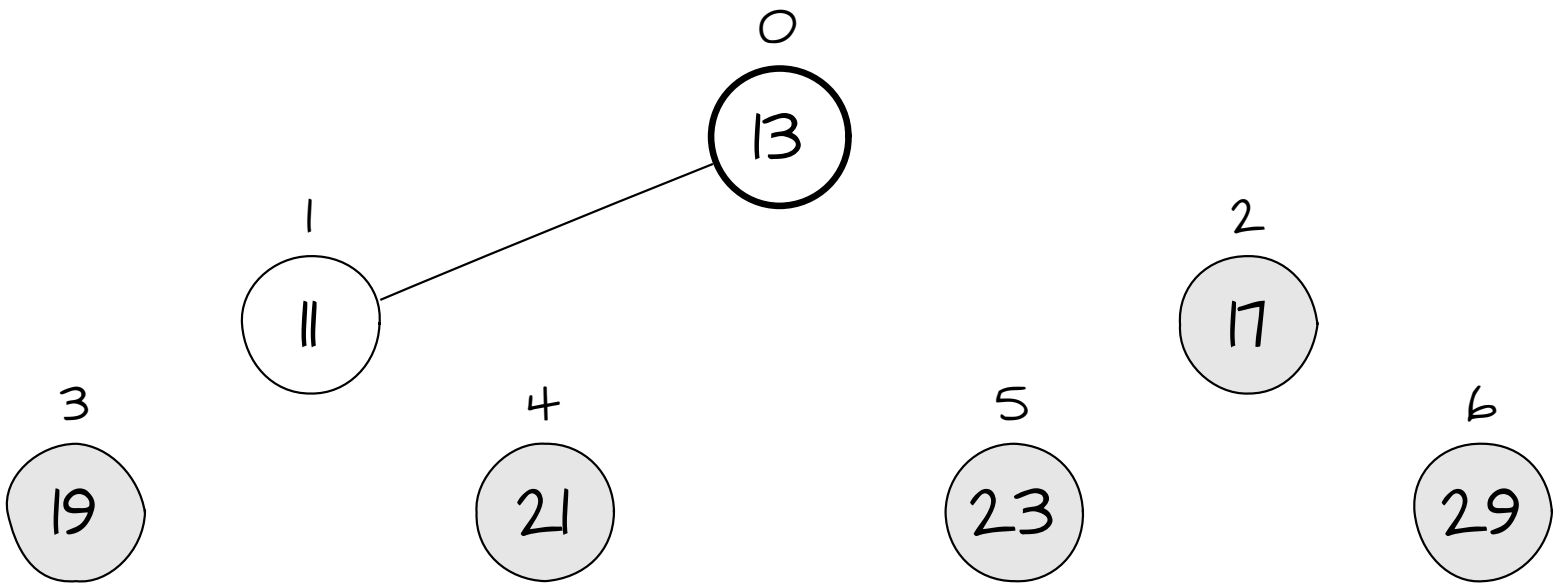
Heapsort

- Etapa de conquista



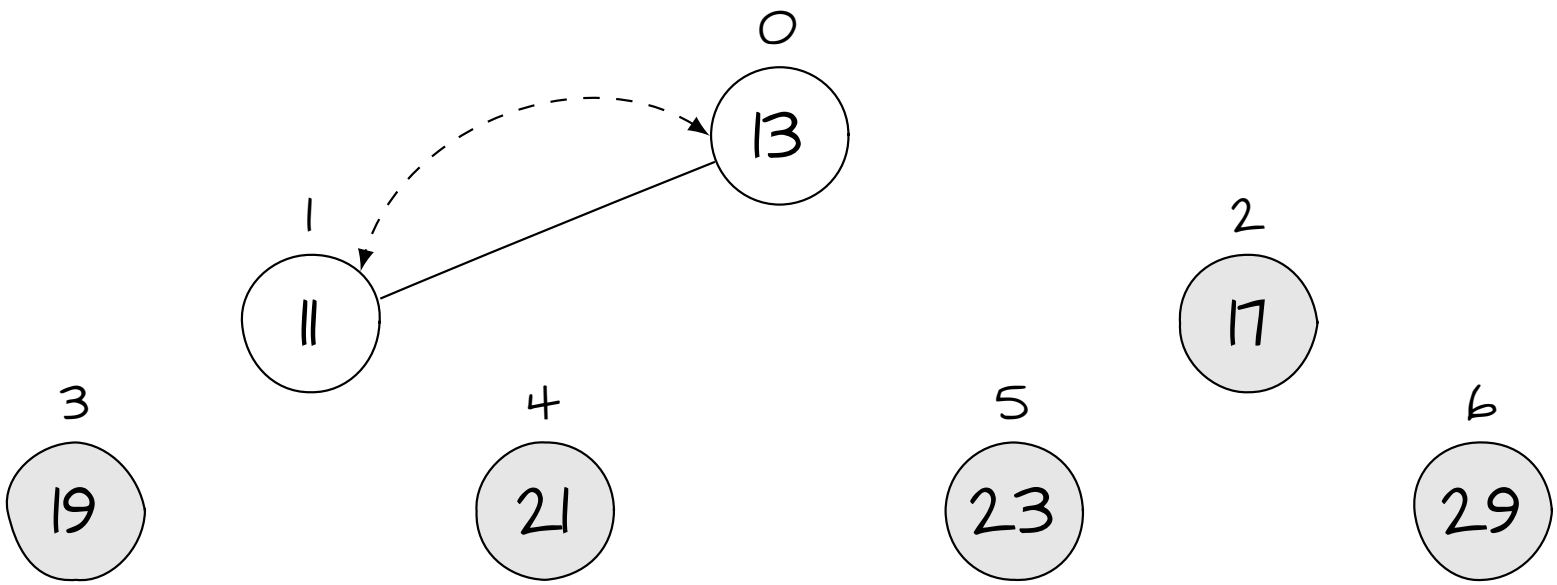
Heapsort

- Etapa de conquista



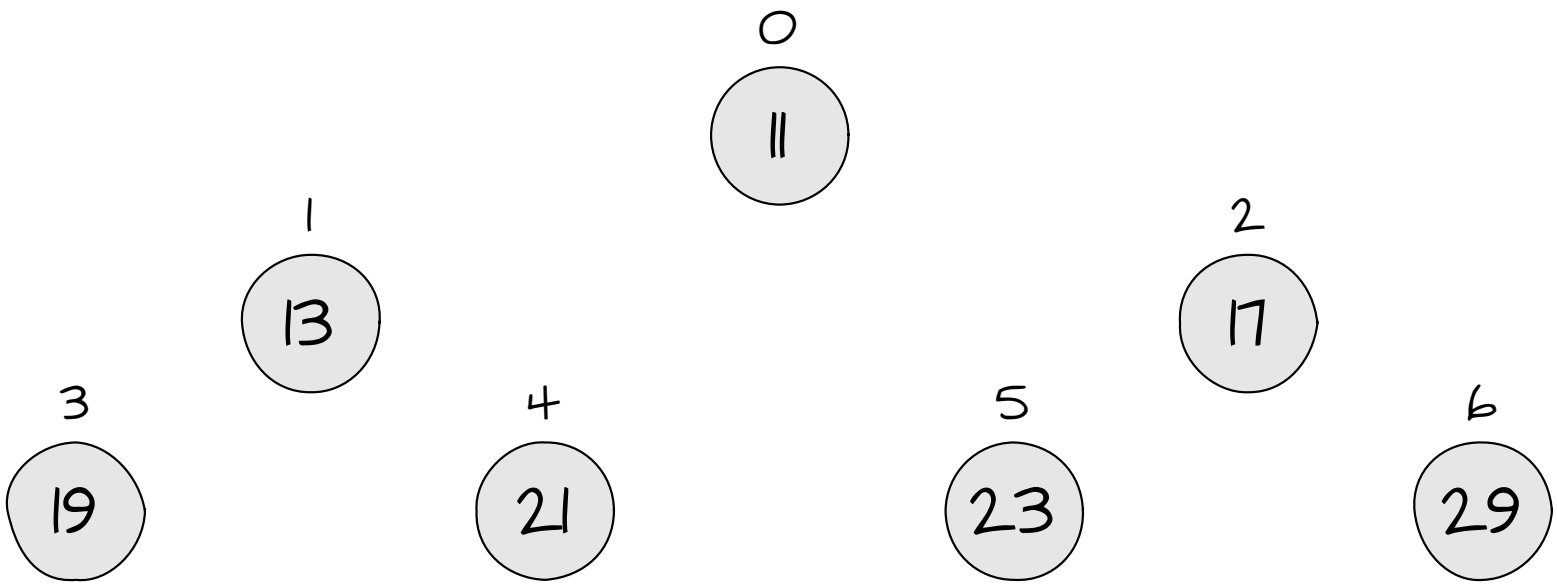
Heapsort

- Etapa de conquista



Heapsort

- Etapa de conquista



Heapsort

- Análise de complexidade
  - Espaço $\Theta(n + \log n) = \Theta(n)$
  - Tempo $\Theta(n) + \Theta(n \log n) = \Theta(n \log_2 n)$
    - A construção custa $\Theta(n)$
    - A manutenção com *heapify* é $O(\log_2 n)$

Heapsort

- Características do Heapsort
  ✓ *In-place*: não utiliza espaço adicional, utilizando o próprio vetor de entrada

Heapsort

- Características do Heapsort
  - ✓ *In-place*: não utiliza espaço adicional, utilizando o próprio vetor de entrada
  - ✓ Espaço $\Theta(n)$ e tempo $\Theta(n \log n)$

Heapsort

- Características do Heapsort
  - ✓ *In-place*: não utiliza espaço adicional, utilizando o próprio vetor de entrada
  - ✓ Espaço $\Theta(n)$ e tempo $\Theta(n \log n)$
  - ✓ É adequado para aplicações de tempo real

Heapsort

- Características do Heapsort
  - ✗ É mais lento que o Quicksort na prática

Heapsort

- Características do Heapsort
  - ✗ É mais lento que o Quicksort na prática
  - ✗ Não explora a localidade espacial dos dados

Heapsort

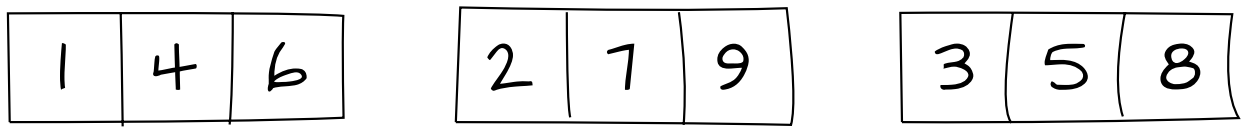
- Características do Heapsort
  - ✗ É mais lento que o Quicksort na prática
  - ✗ Não explora a localidade espacial dos dados
  - ✗ É instável, ignorando a ordem relativa dos elementos

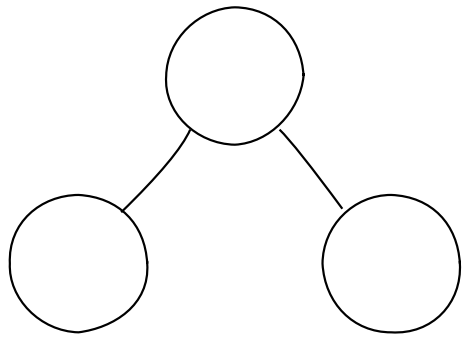
Heapsort

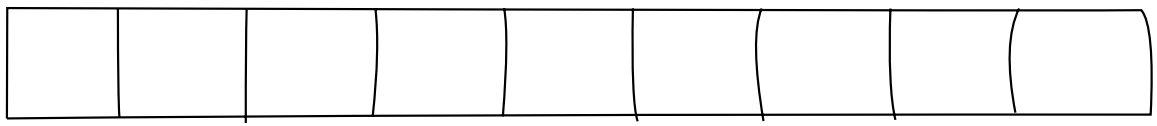
- Características do Heapsort
  - ✗ É mais lento que o Quicksort na prática
  - ✗ Não explora a localidade espacial dos dados
  - ✗ É instável, ignorando a ordem relativa dos elementos
  - ✗ A paralelização não é direta

Heapsort

- Intercalando k vetores de tamanho n (k-*way merge*)



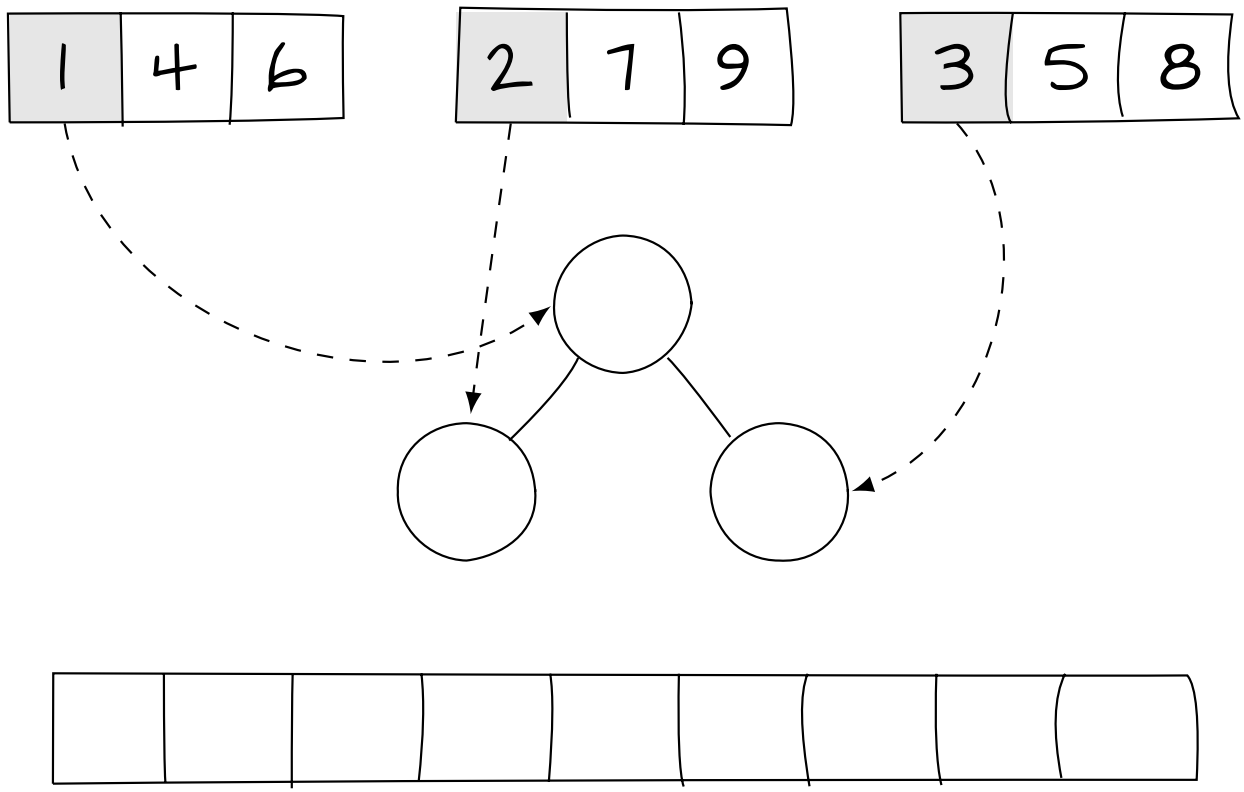




Os vetores estão ordenados

Heapsort

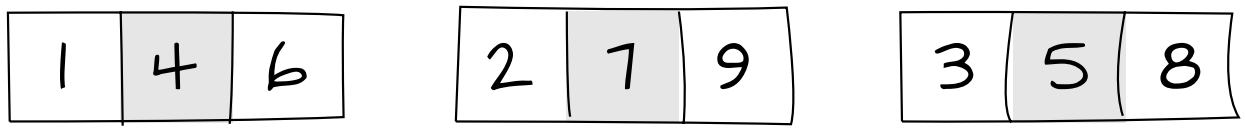
- Intercalando k vetores de tamanho n (*k-way merge*)

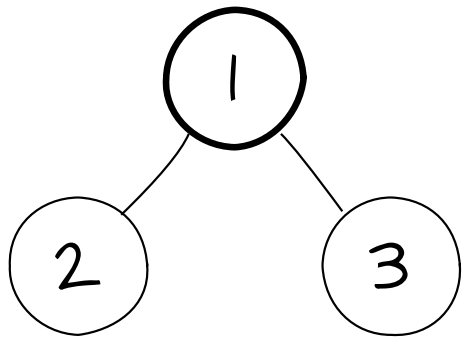


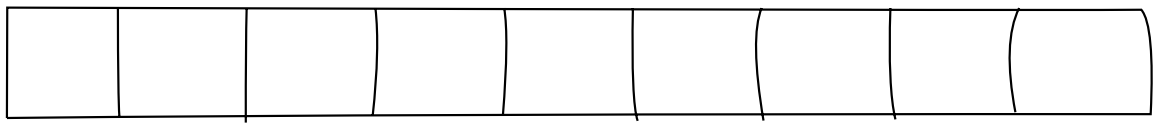
Cada vetor possui um índice para acesso em $O(1)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



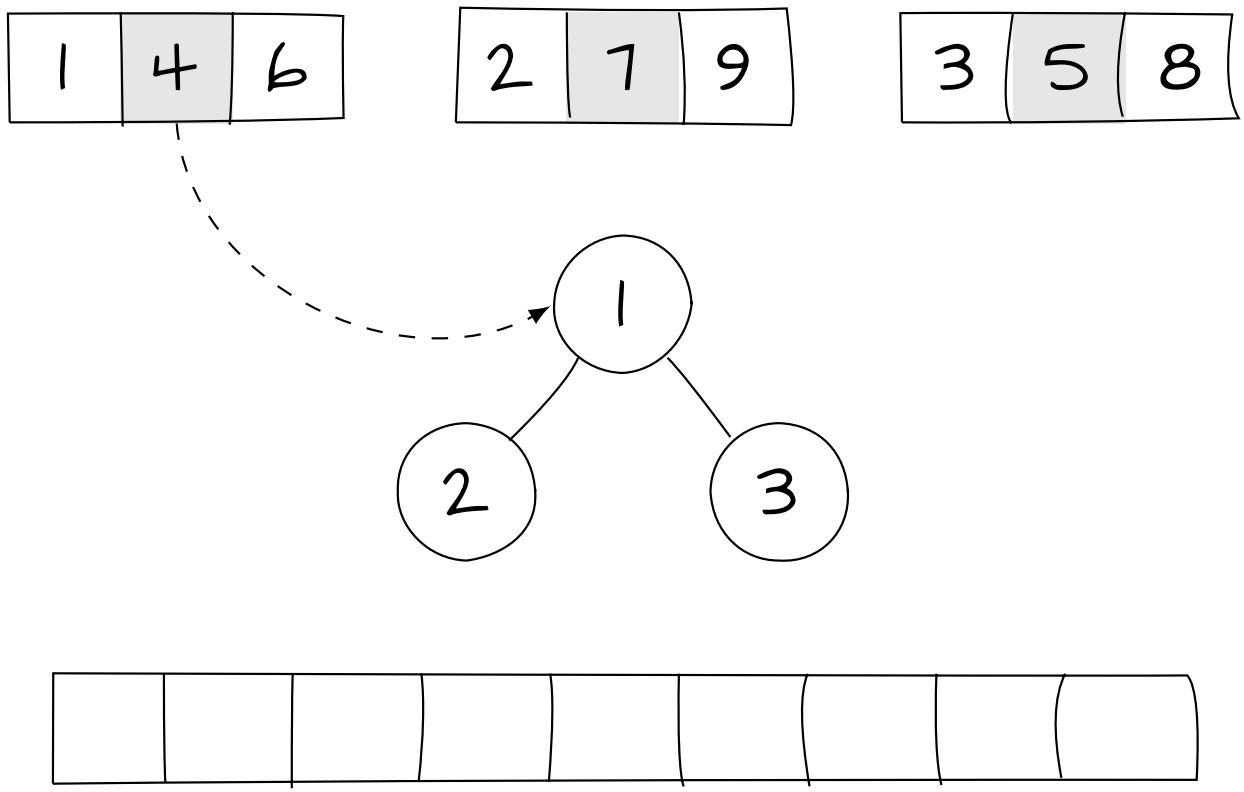




A construção do *heap* mínimo é $\Theta(k)$

Heapsort

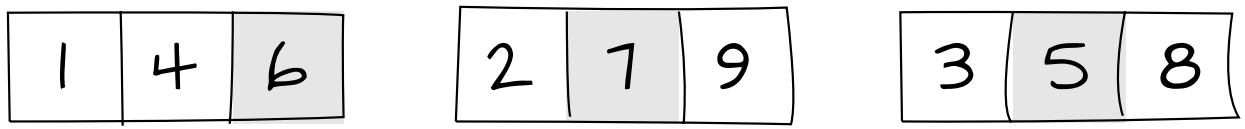
- Intercalando k vetores de tamanho n (*k-way merge*)

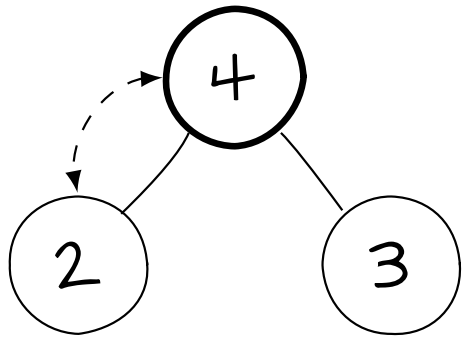


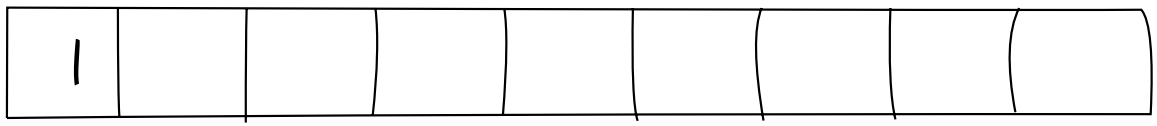
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



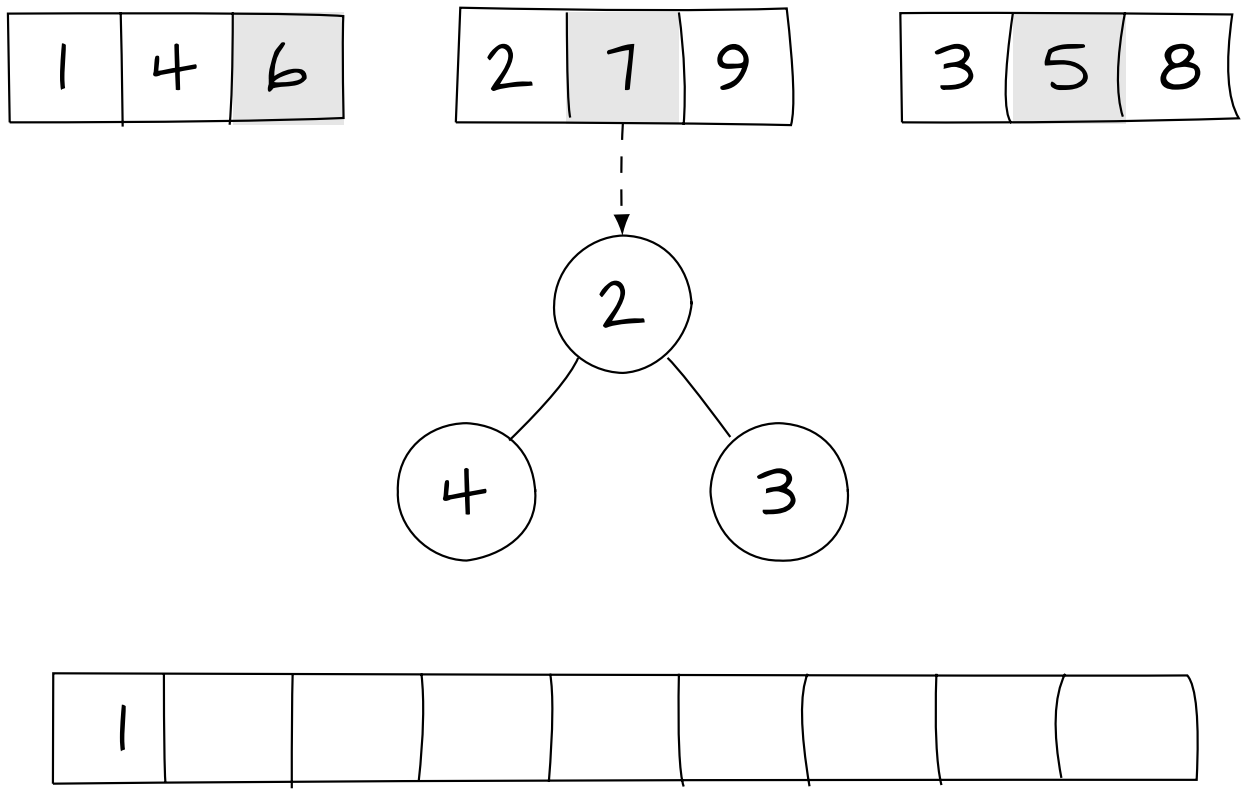




São realizadas kn operações com custo $O(\log k)$

Heapsort

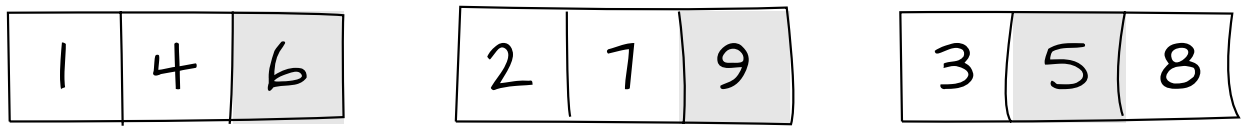
- Intercalando k vetores de tamanho n (*k-way merge*)

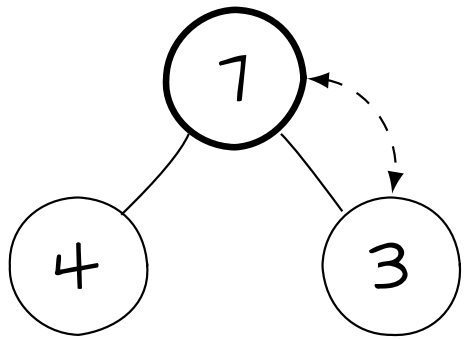


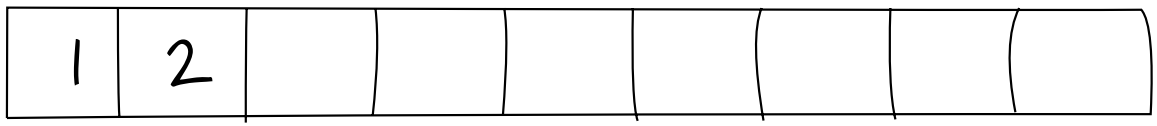
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



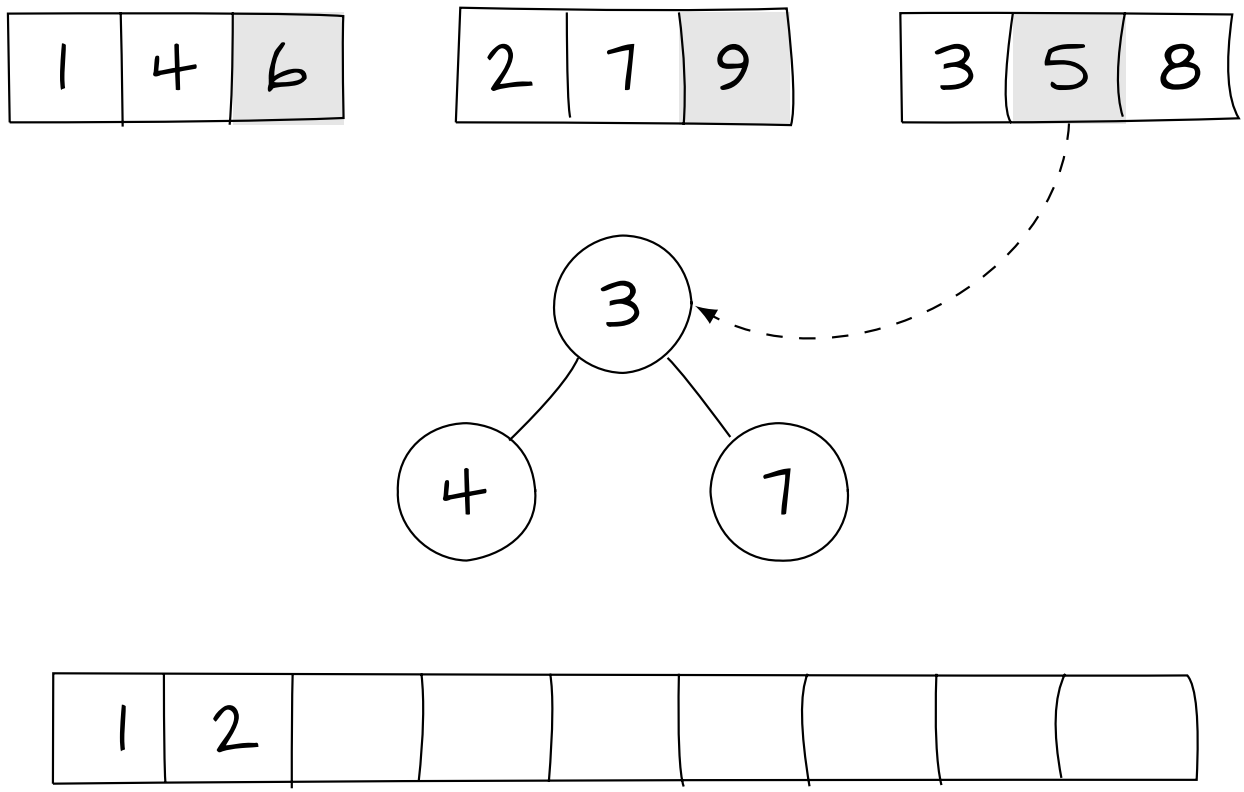




São realizadas kn operações com custo $O(\log k)$

Heapsort

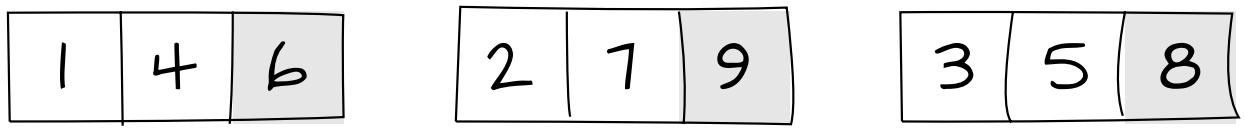
- Intercalando k vetores de tamanho n (*k-way merge*)

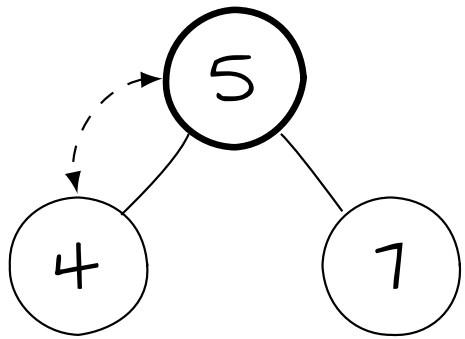


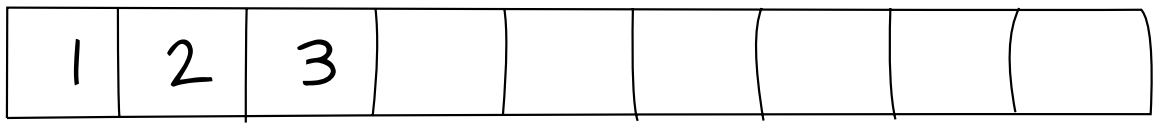
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



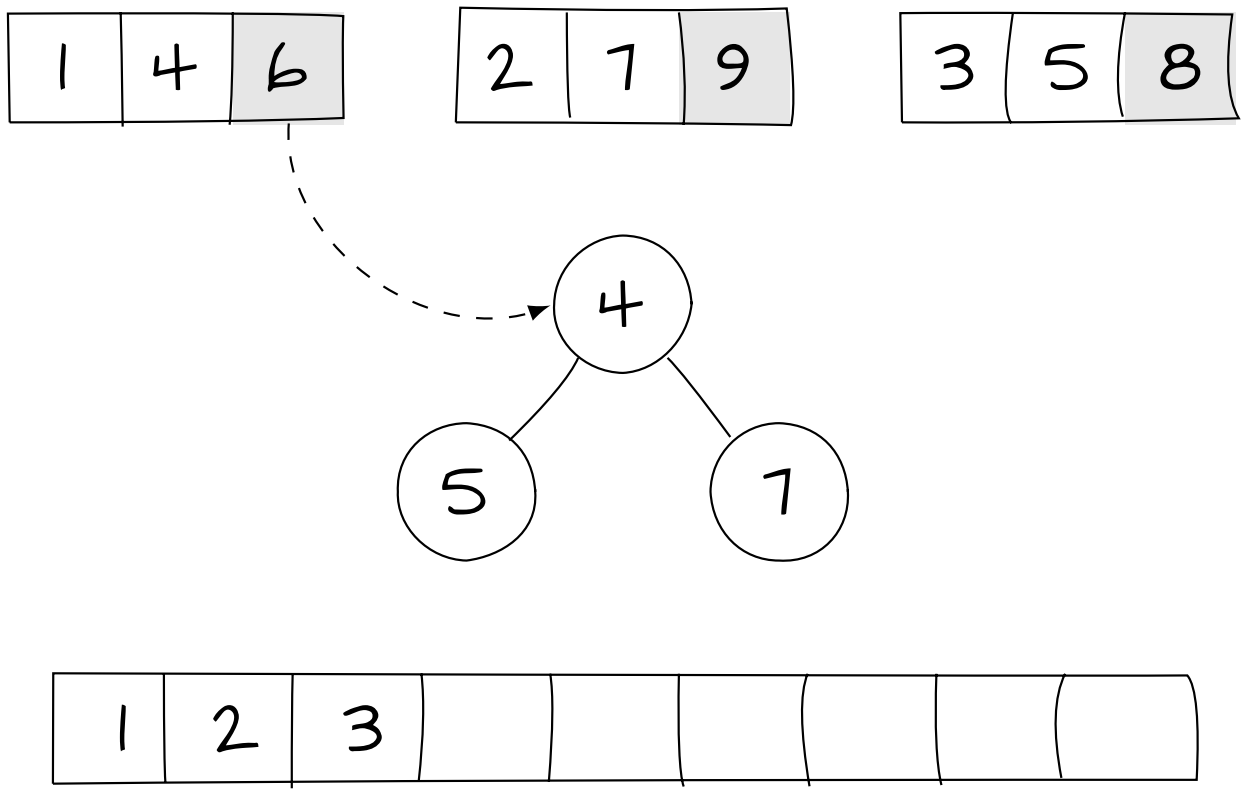




São realizadas kn operações com custo $O(\log k)$

Heapsort

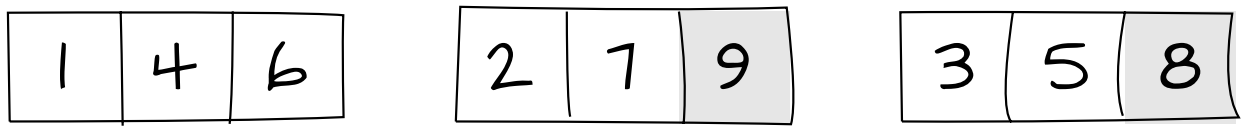
- Intercalando k vetores de tamanho n (*k-way merge*)

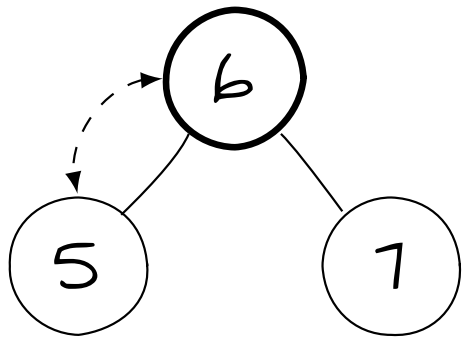


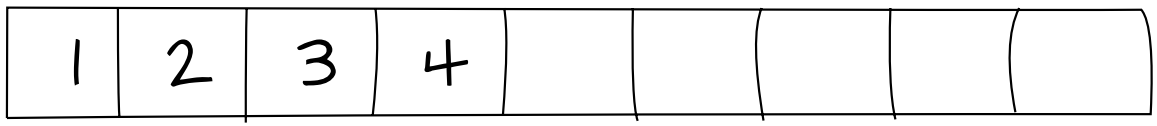
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



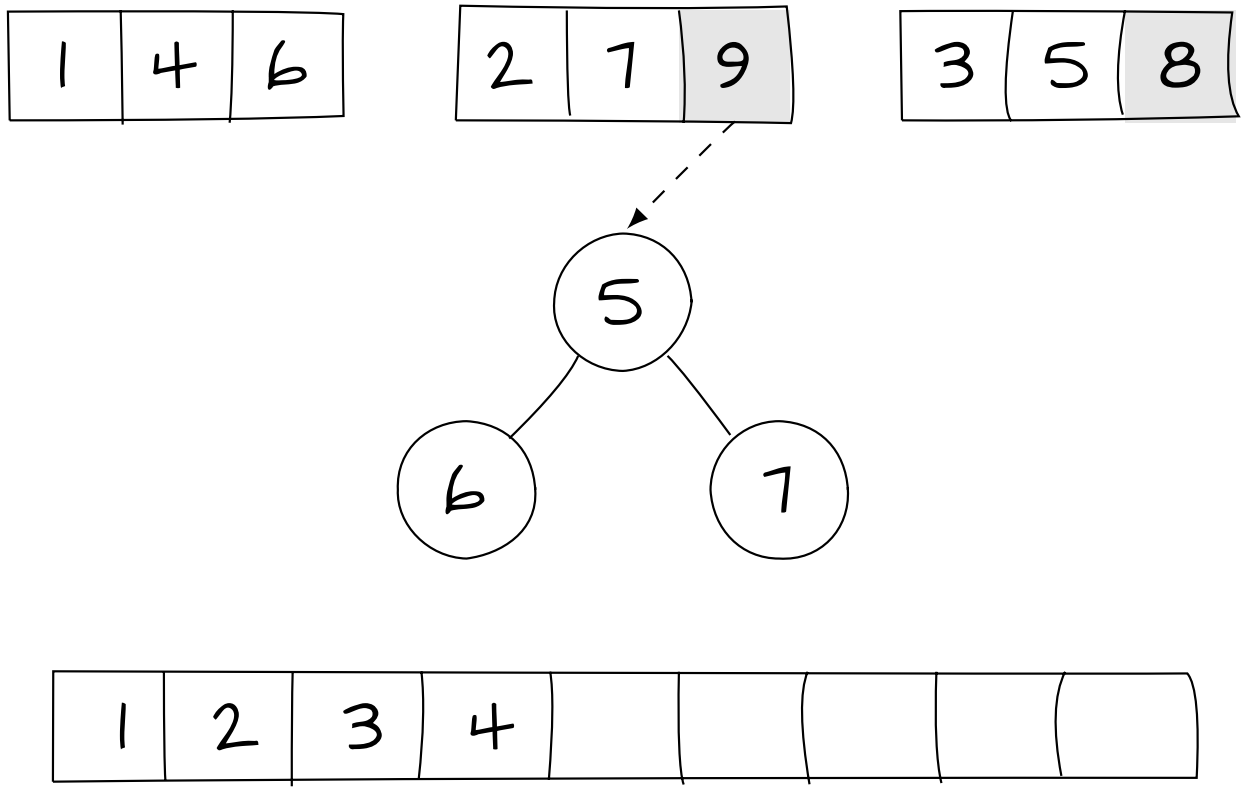




São realizadas kn operações com custo $O(\log k)$

Heapsort

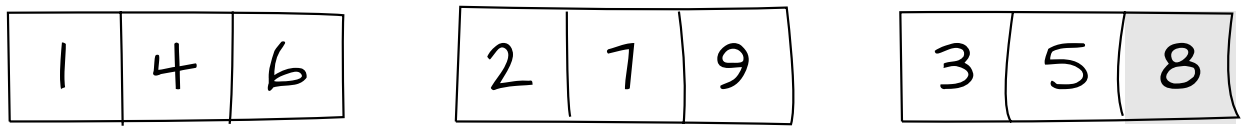
- Intercalando k vetores de tamanho n (*k-way merge*)

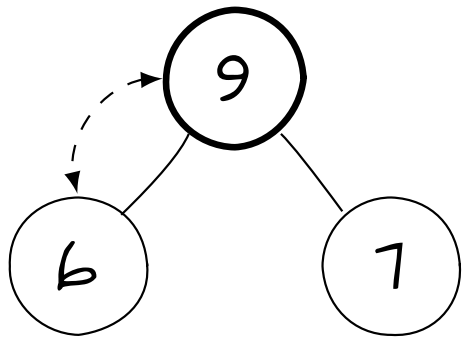


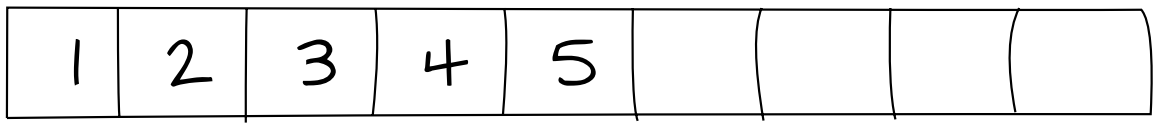
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



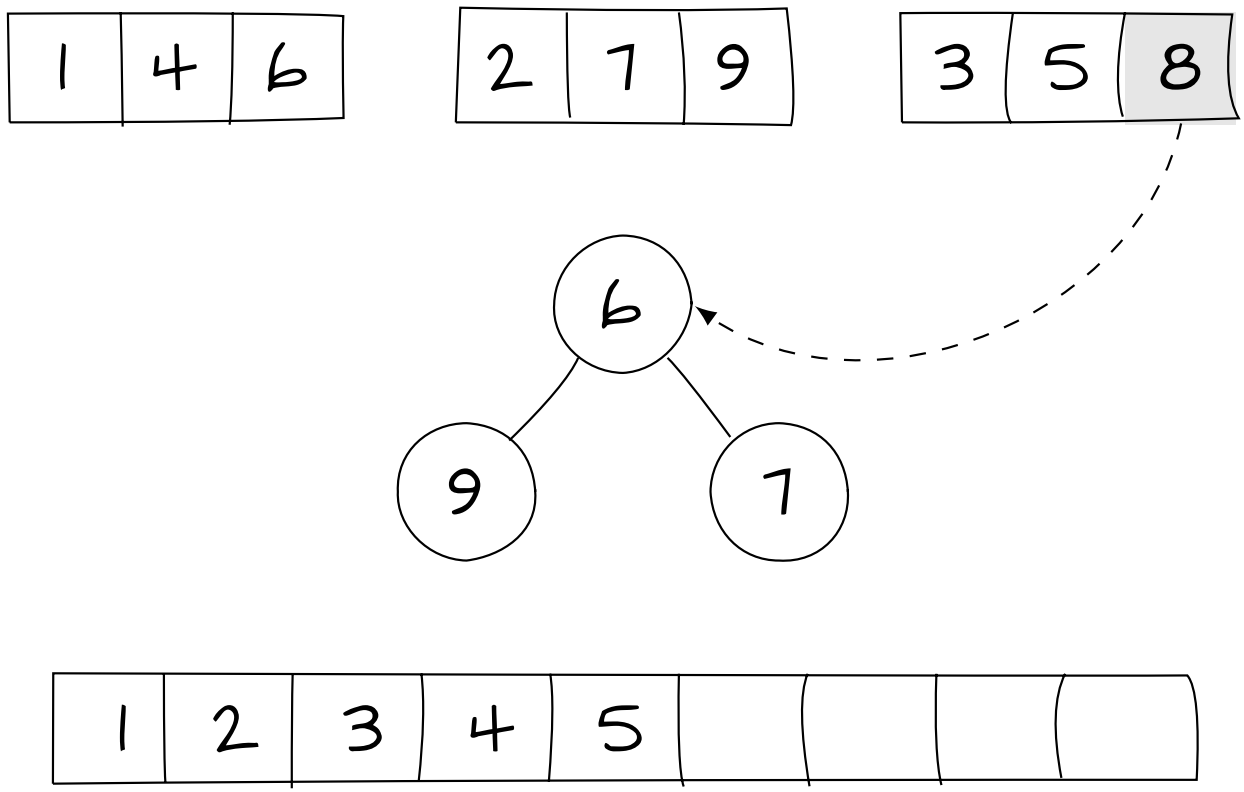




São realizadas kn operações com custo $O(\log k)$

Heapsort

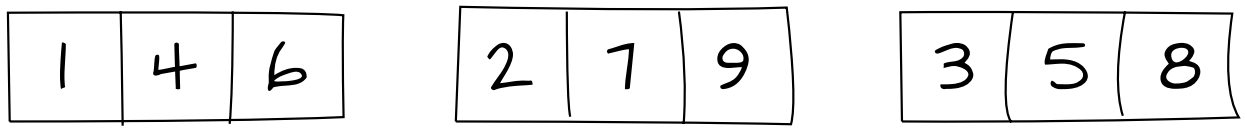
- Intercalando k vetores de tamanho n (k-*way merge*)

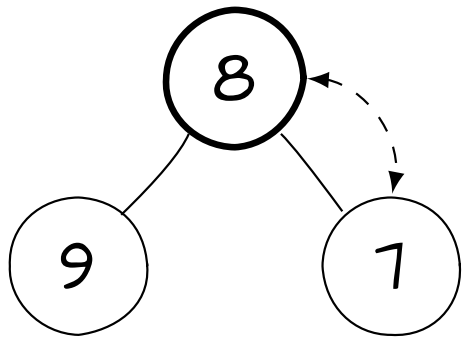


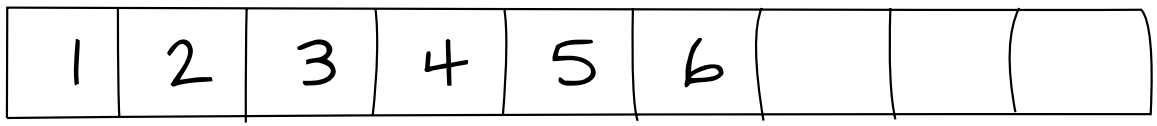
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



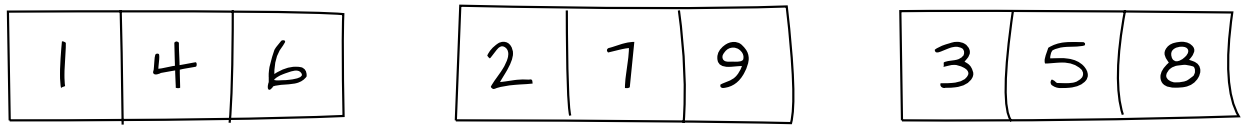


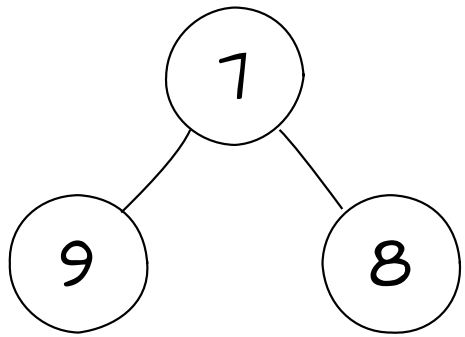


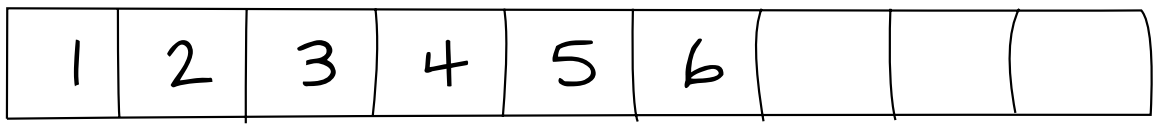
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



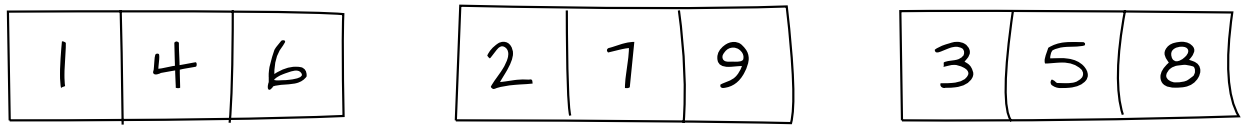


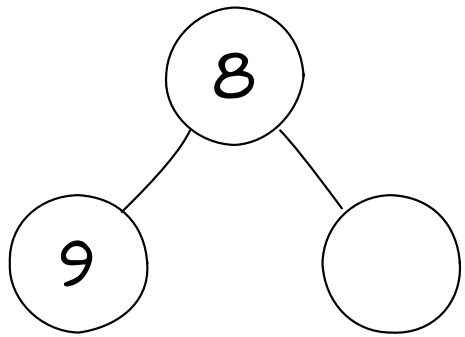


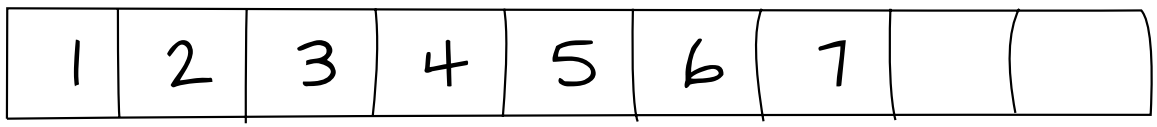
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



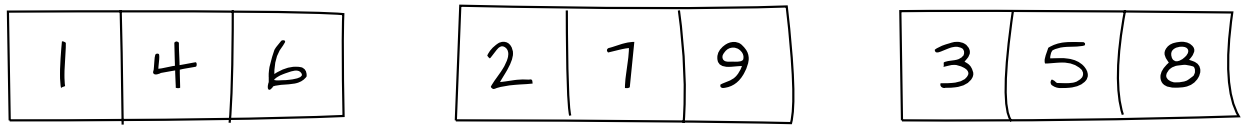


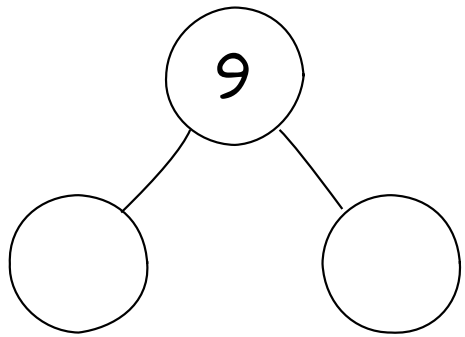


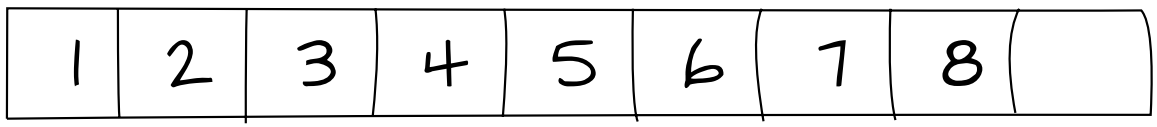
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)



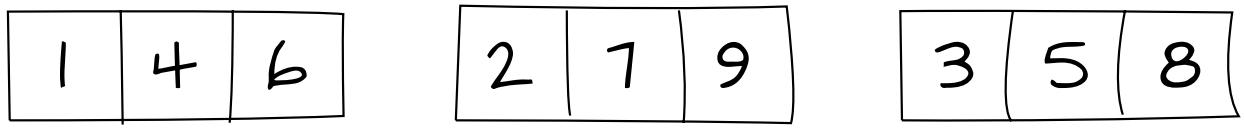


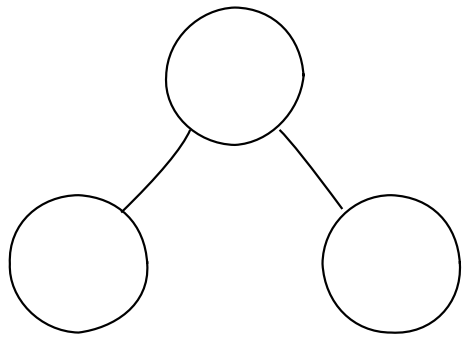


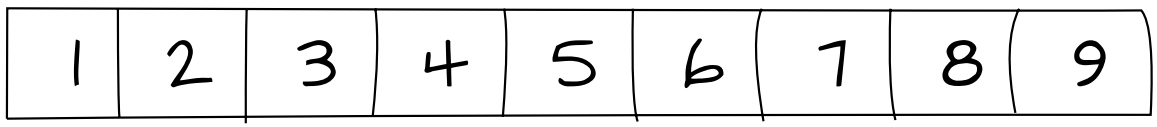
São realizadas kn operações com custo $O(\log k)$

Heapsort

- Intercalando k vetores de tamanho n (*k-way merge*)







Espaço $\Theta(n)$ e tempo $O(kn \log k)$

Exemplo

- Considerando o algoritmo de ordenação Heapsort, ordene o vetor 23, 32, 54, 92, 74, 23, 1, 43, 63 e 12
  - Utilize o critério decrescente de ordenação, com *heap* máximo e mínimo
  - Execute passo a passo cada etapa do algoritmo

Exercício

- A empresa de telecomunicações Poxim Tech está construindo um sistema de comunicação, baseado no protocolo de datagrama do usuário (UDP) para transferência de pacotes em redes TCP/IP
  - Os dados são organizados em sequências de bytes de tamanho variável, mas limitados até o tamanho máximo de 512 bytes
  - Devido às características de roteamento de redes TCP/IP, os pacotes podem chegar ao seu destino desordenados, sendo necessária a ordenação dos pacotes para receber os dados corretamente
  - Para permitir o acesso rápido dos dados, é possível processar as informações recebidas desde que estejam parcialmente ordenadas, com os pacotes iniciais, sendo este processamento disparado por uma determinada quantidade de pacotes recebidas

Exercício

- Formato de arquivo de entrada
  - [#*n total de pacotes*] [*Quantidade de pacotes*]
  - [*Número do pacote*] [#m_1 *Tamanho do pacote*] [*B*1] $\cdots$ [B_{m_1}]
  - $\ldots$
  - [*Número do pacote*] [#m_n *Tamanho do pacote*] [*B*1] $\cdots$ [B_{m_n}]

```
1  6␣2
2  0␣3␣01␣02␣03
3  1␣2␣04␣05
4  2␣4␣06␣07␣08␣09
5  4␣2␣0F␣10
6  3␣5␣0A␣0B␣0C␣0D␣0E
7  5␣6␣11␣12␣13␣14␣15␣16
```

Exercício

- Formato de arquivo de saída
  - Quando uma quantidade determinada de pacotes é recebida, é feita a ordenação parcial dos pacotes para verificar se é possível exibir a parte inicial completa dos dados que já foram recebidos

```
0: 01 02 03 04 05
1: 06 07 08 09
2: 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
```