Estrutura de Dados – 1º semestre de 2021

Professor Mestre Fabio Pereira da Silva

Fila em Algoritmos de Ordenação

- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
- Basta utilizar repetidamente a operação de inserção para construir a fila de prioridades.
- Em seguida, utilizar repetidamente a operação de retirada para receber os itens na ordem reversa.
- O uso de listas lineares não ordenadas corresponde ao método da seleção.
- O uso de listas lineares ordenadas corresponde ao método da inserção.
- O uso de heaps corresponde ao método Heapsort.

Fila em Algoritmos de Ordenação

- Estrutura de dados composta de chaves, que suporta duas operações básicas: inserção de um novo item e remoção.
- A chave de cada item reflete a prioridade em que se deve tratar aquele item
- Aplicações: Sistemas operacionais, paginação de memória, ordenação, simulação de eventos

Definição de Árvores

- Árvores são um conjunto finito de elementos.
  - Um elemento é chamado de **raíz**
- Os outros são divididos em subconjuntos disjuntos onde cada um define uma árvore.
  - Cada elemento é um **Nó ou Vértice da árvore**
  - Arcos ou arestas conectam os vértices

Definição de Árvores

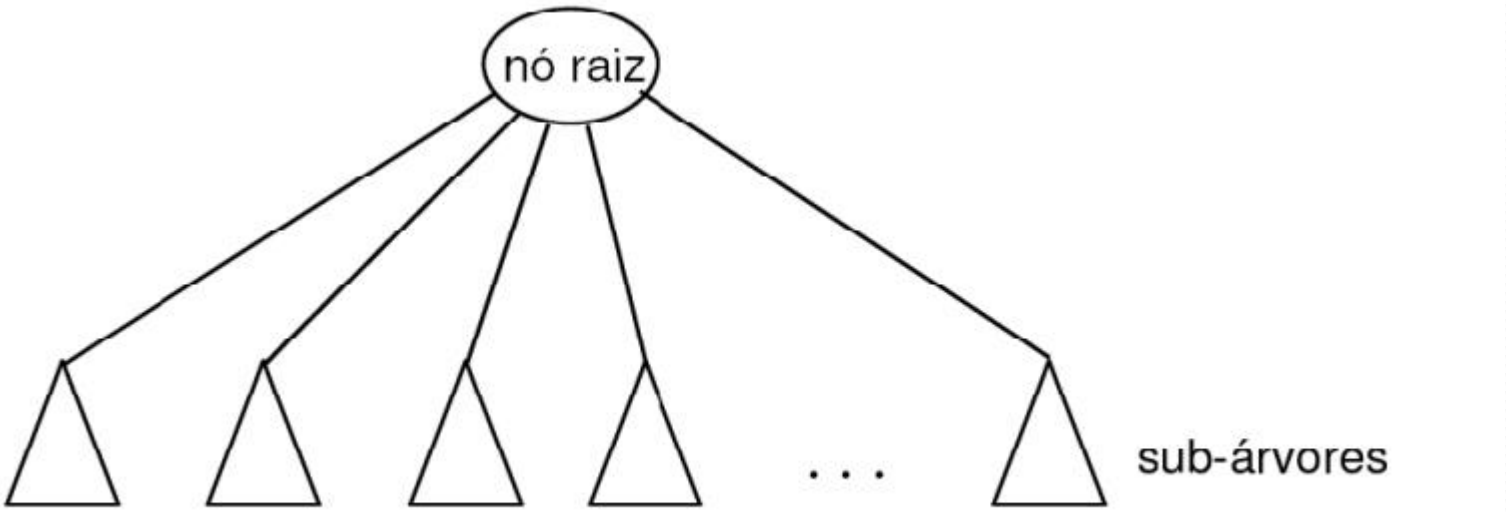
- Uma coleção não vazia de vértices e ramos que satisfazem a certos requisitos.
- Vértice (Ou Nó)
  - É um objeto simples que pode ter um nome e mais alguma outra informação associada.
- Arco ou aresta (direcionado ou não)
  - É a conexão entre dois Nós

Definição de Árvores

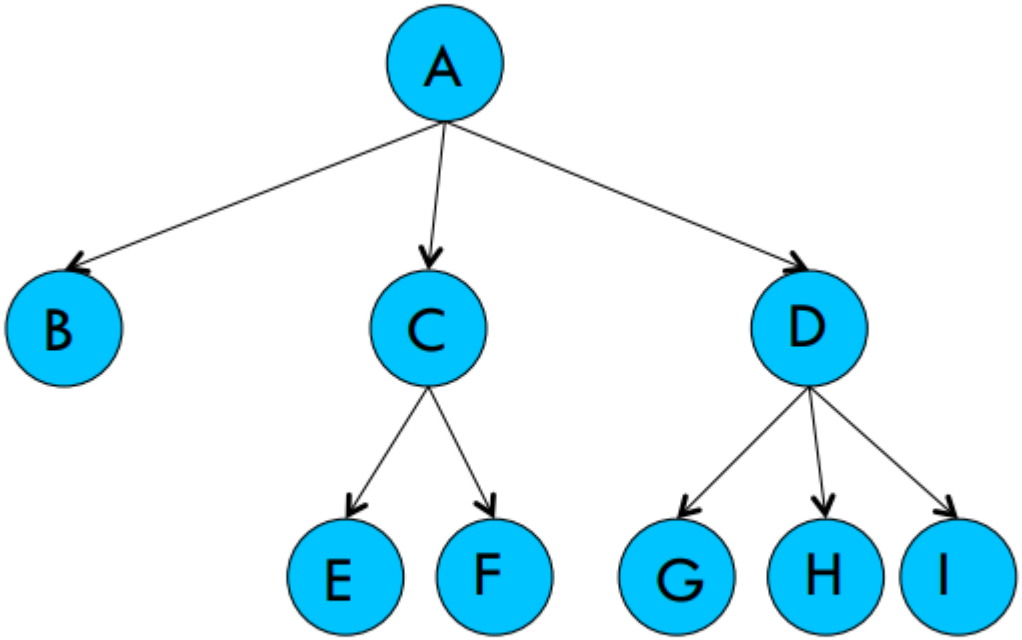
- Nós filhos, pais, tios, irmãos e avô
- Grau de saída (número de filhos de um nó)
- Nó folha (grau de saída nulo) e nó interior (grau de saída diferente de nulo)
- Grau de uma árvore (máximo grau de saída)
- Floresta (conjunto de árvores)

Definição de Árvores

- Um conjunto de nós tal que:
  - Existe um nó r, denominado raiz, com zero ou mais sub-árvores, cujas raízes estão ligadas a r
  - Os nós raízes destas sub-árvores são os filhos de r
  - Os nós internos da árvore são os nós com filhos
  - As folhas ou nós externos da árvore são os nós sem filhos



Representação de árvore



Heap Sort

- Heap – Dicionário Merriam-Webster:
  - Coleção de coisas jogadas uma em cima da outra – monte.
  - Grande número ou grande quantidade – lote.
- Em computação, dois sentidos :
  - Espaço de memória variável onde são criados objetos;
  - Estrutura de dados para armazenar dados segundo uma regra particular próximo do sentido original, traduzido como monte.
- O algoritmo de ordenação HeapSort utiliza a estrutura de dados heap para ordenar um vetor.

Heap Sort

- Algoritmo criado por John Williams (1964)
- Complexidade O(NlogN) no pior e médio caso
- Mesmo tendo a mesma complexidade no caso médio que o QuickSort, o HeapSort acaba sendo mais lento que algumas boas implementações do QuickSort
- Porém, além de ser mais rápido no pior caso que o QuickSort, necessita de menos memória para executar
- QuickSort necessita de um vetor O(logN) para guardar as estruturas enquanto o HeapSort não necessita de um vetor auxiliar

Heap Sort

- Utiliza a abordagem proposta pelo SelectionSort
- O SelectionSort pesquisa entre os n elementos o que precede todos os outros n-1 elementos
- Para ordenar em ordem ascendente, o heapsort põe o maior elemento no final do array e o segundo maior antes dele, etc.
- O heapsort começa do final do array pesquisando os maiores elementos, enquanto o selection sort começa do início do array pesquisando os menores.

Heap Sort

- Heapsort é um método de ordenação cujo princípio de funcionamento é o mesmo utilizado para a ordenação por seleção.
- Selecione o maior (ou menor) item do vetor e a seguir troque-o com o item que está na Heapsort a seguir troque-o com o item que está na última (ou primeira) posição do vetor; repita estas duas operações com os n - 1 itens restantes; depois com os n - 2 itens; e assim sucessivamente.

Heap Sort

- O projeto por indução que leva ao HeapSort é essencialmente o mesmo do Selection Sort: selecionamos e posicionamos o maior (ou menor) elemento do conjunto e então aplicamos a hipótese de indução para ordenar os elementos restantes.
- A diferença importante é que no HeapSort utilizamos a estrutura de dados heap para selecionar o maior (ou menor) elemento eficientemente.
- **Um heap é um vetor que simula uma árvore binária completa, a menos, talvez, do último nível, com estrutura de heap.**

Heap Sort

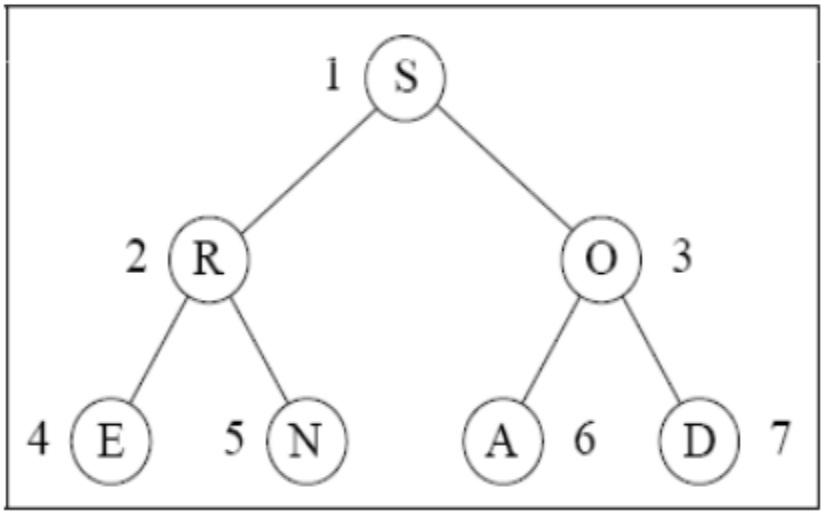
- Cada nó da árvore corresponde a um elemento do vetor que armazena o valor no nó.
- A árvore está completamente preenchida em todos os níveis, exceto possivelmente no nível mais baixo, que é preenchido da esquerda para a direita até certo ponto.
- Um vetor V representa uma estrutura heap através de dois parâmetros:
  - comprimento de V (V.length): tamanho total do vetor;
  - comprimento do heap (heapComp): comprimento da parte do vetor que contém elementos da estrutura heap.

Heap Sort

- Transformação do vetor em um heap binário máximo (Construção do Heap)
- Ordenação – a cada iteração seleciona-se o maior elemento (na raiz do heap) e o adiciona no início de um segmento ordenado
- Após cada seleção de elemento, o heap deve ser reorganizado para continuar sendo um heap binário máximo

Heap Sort

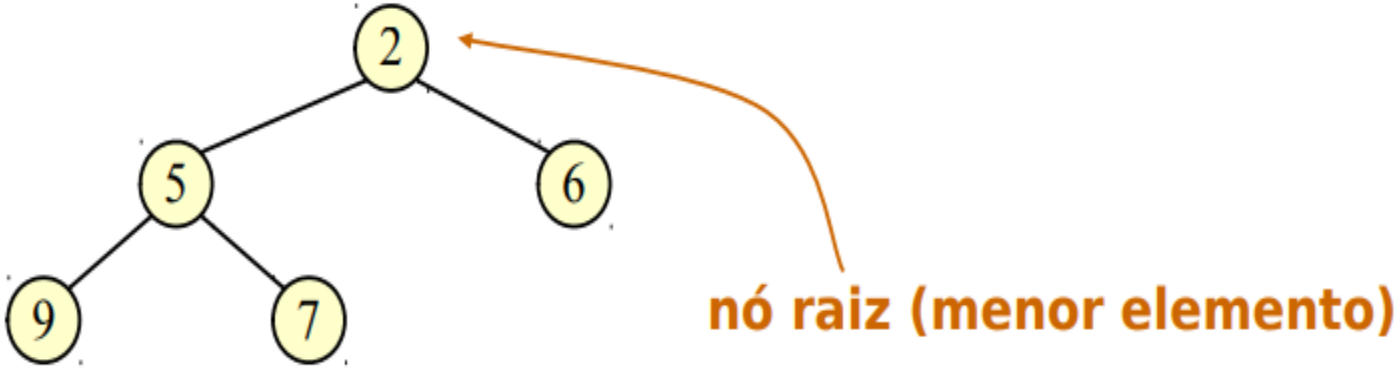
- Essa definição pode ser facilmente visualizada em uma árvore binária completa:



Árvores vão ser vistas em detalhes nos algoritmos de pesquisa

Heap Sort

- É uma árvore binária em que um nó filho é sempre maior ou igual a um nó pai. • Ou seja: chave(v) >= chave(pai(v))



Estrutura do Heap - Heap Máximo

- $A[pai(i)] \geq A[i]$.
- Isto é, o valor de um nó é no máximo o valor de seu pai.
- O maior elemento do heap está na raiz.
- As subárvores de um nó possuem valores menores ou iguais ao do nó.

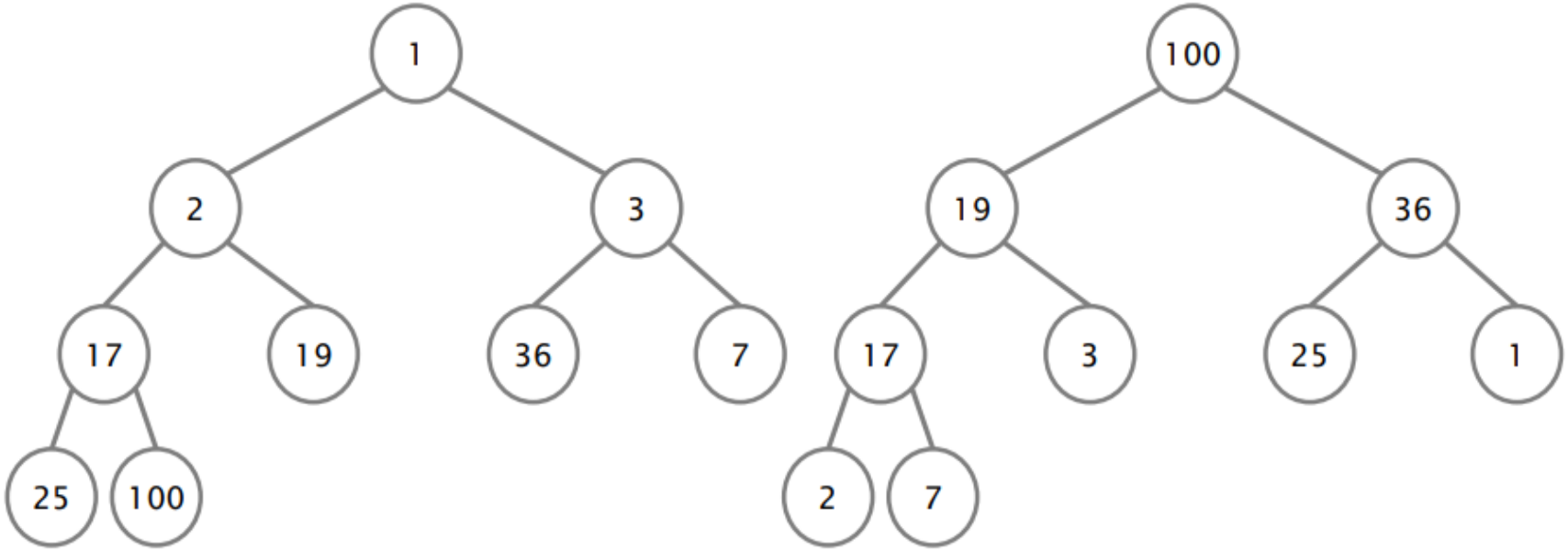
Estrutura do Heap - Heap Mínimo

- A[pai(i)] ≤ A[i].
- Isto é, o valor de um nó é maior ou igual o valor de seu pai.
- O menor elemento do heap está na raiz
- As subárvores de um nó possuem valores maiores ou iguais ao do nó.

Árvore binária **completa**

Min heap: Cada nó é **menor** que seus filhos

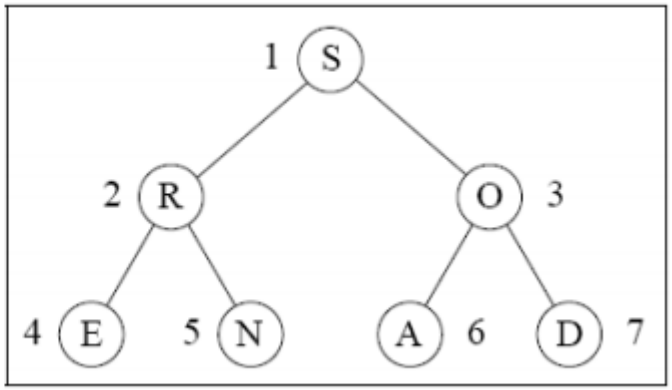
Max heap: Cada nó é **maior** que seus filhos



Estrutura do Heap - Heap Mínimo

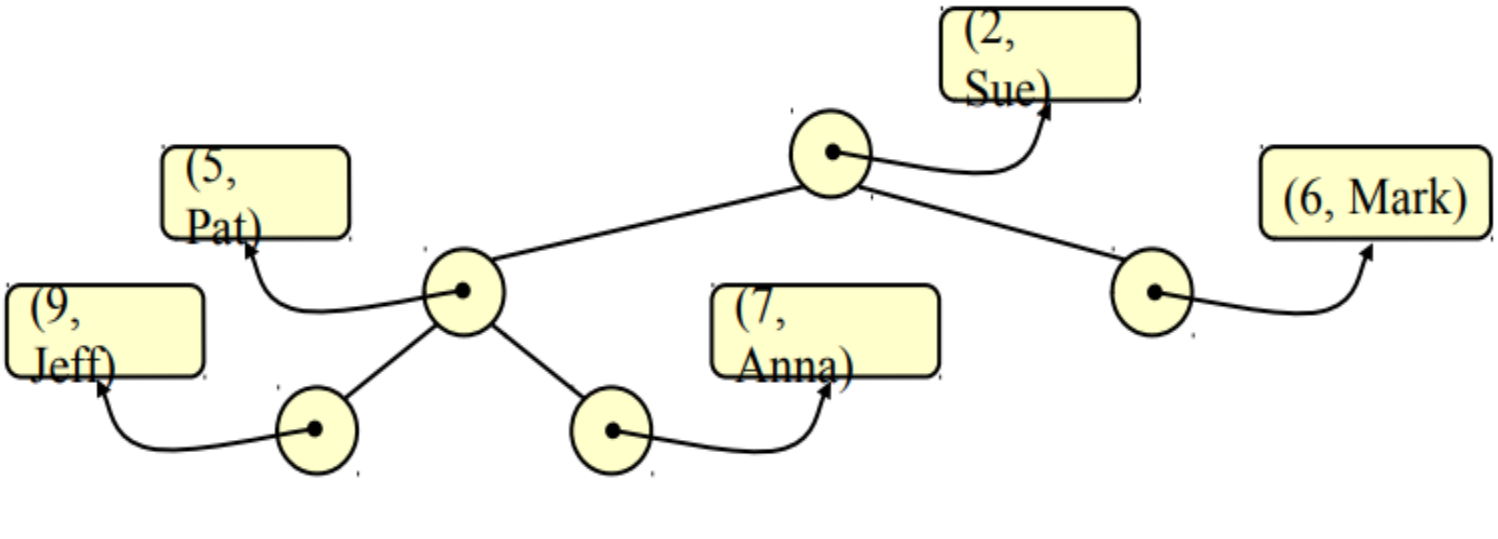
- Correspondência entre representação em árvore e representação em vetor
- Nós são numerados de 1 a n
- O primeiro é chamado raiz
- O nó k/2 é o pai do nó k, $1 < k \leq n$
- Os nós 2k e 2k+1 são filhos da esquerda e direita do nó k, para $1 \leq k \leq n/2$

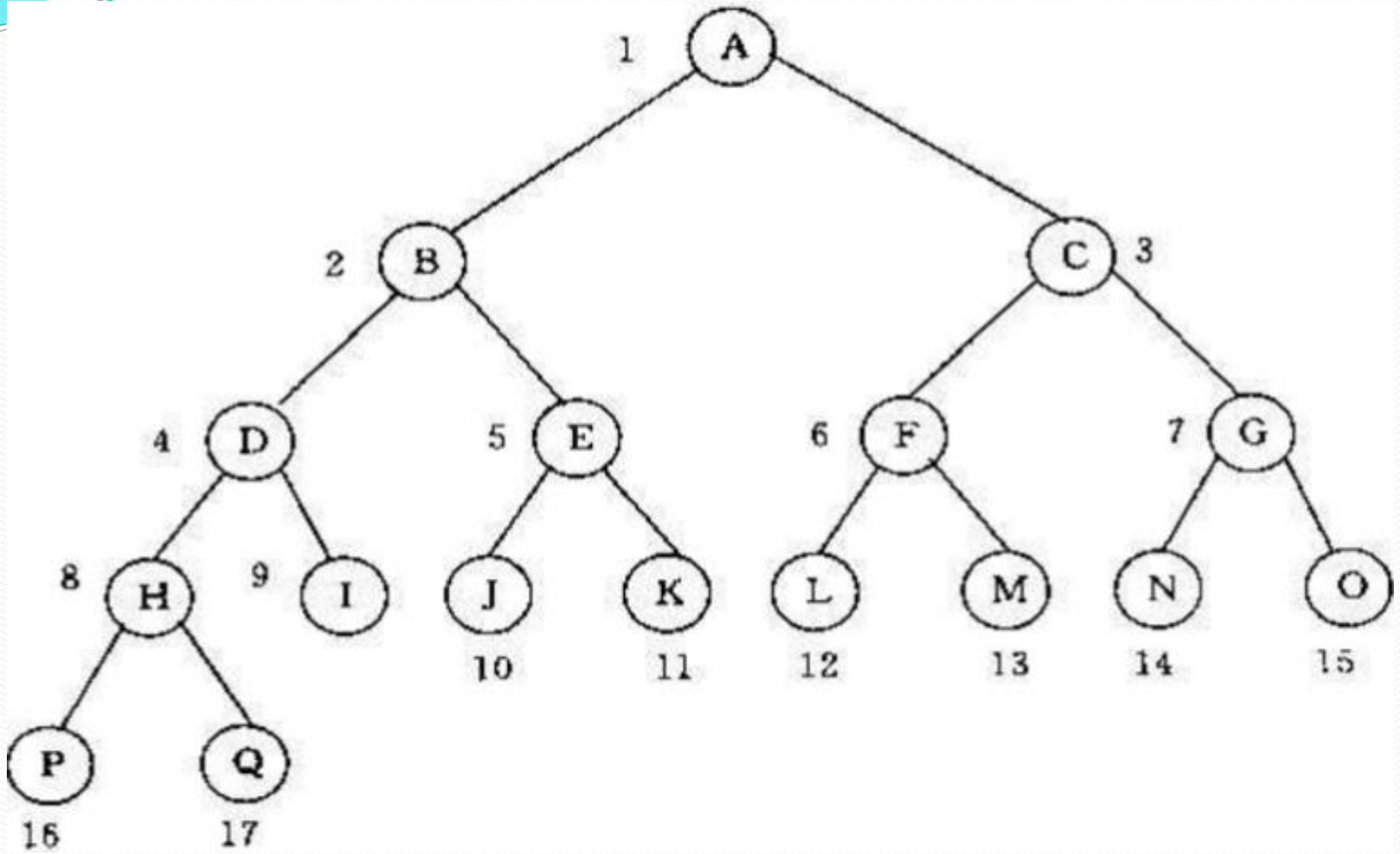
1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*



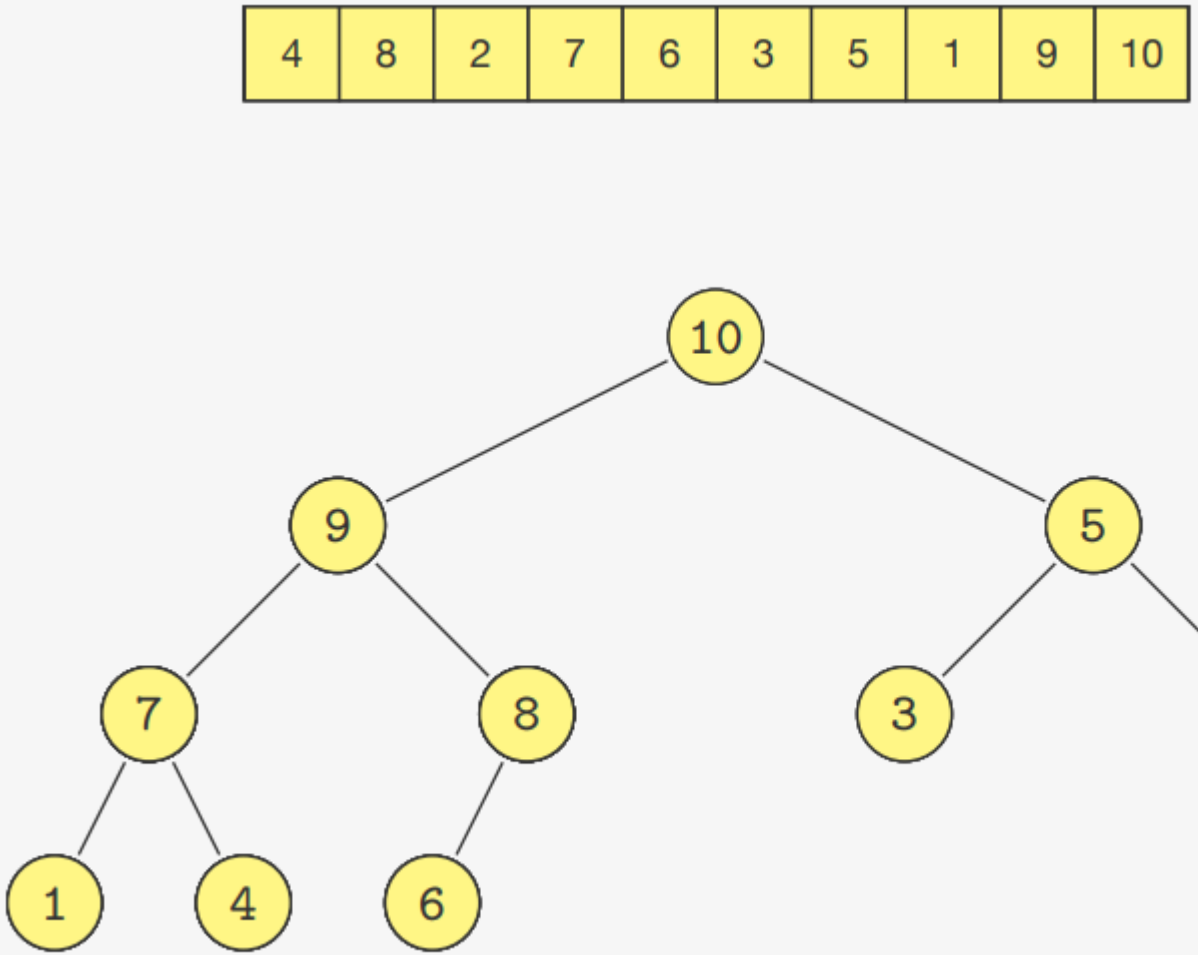
Estrutura do Heap

- Cada nó possui uma tupla (chave, elemento)
- Assim, cada nó do heap armazena todo o item sendo.




1
A
2
B
C
3
4
D
5
E
6
F
7
G
8
H
9
I
J
K
L
M
N
O
10
11
12
13
14
15
P
Q
16
17

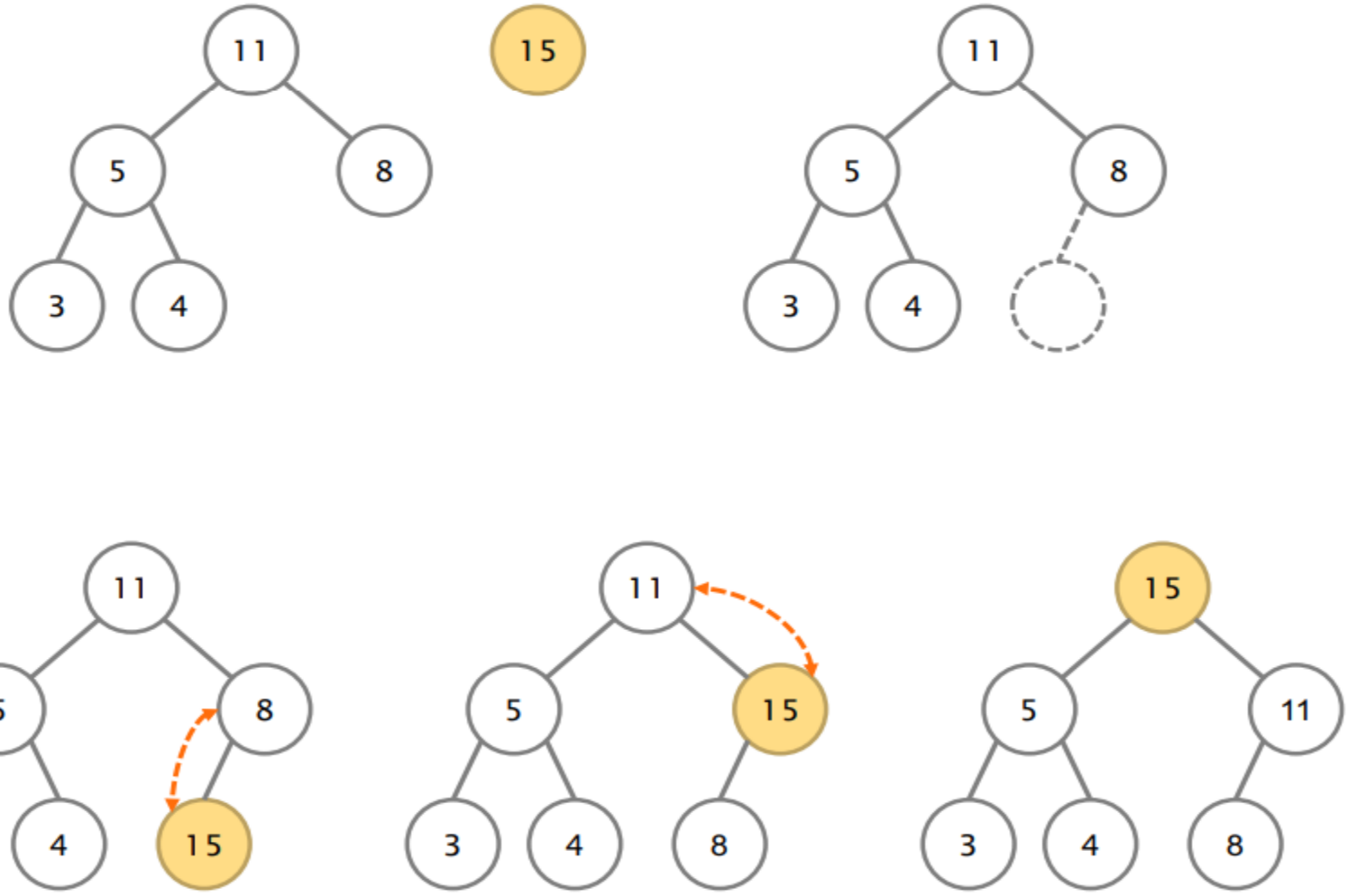
Transformando um vetor em um Heap



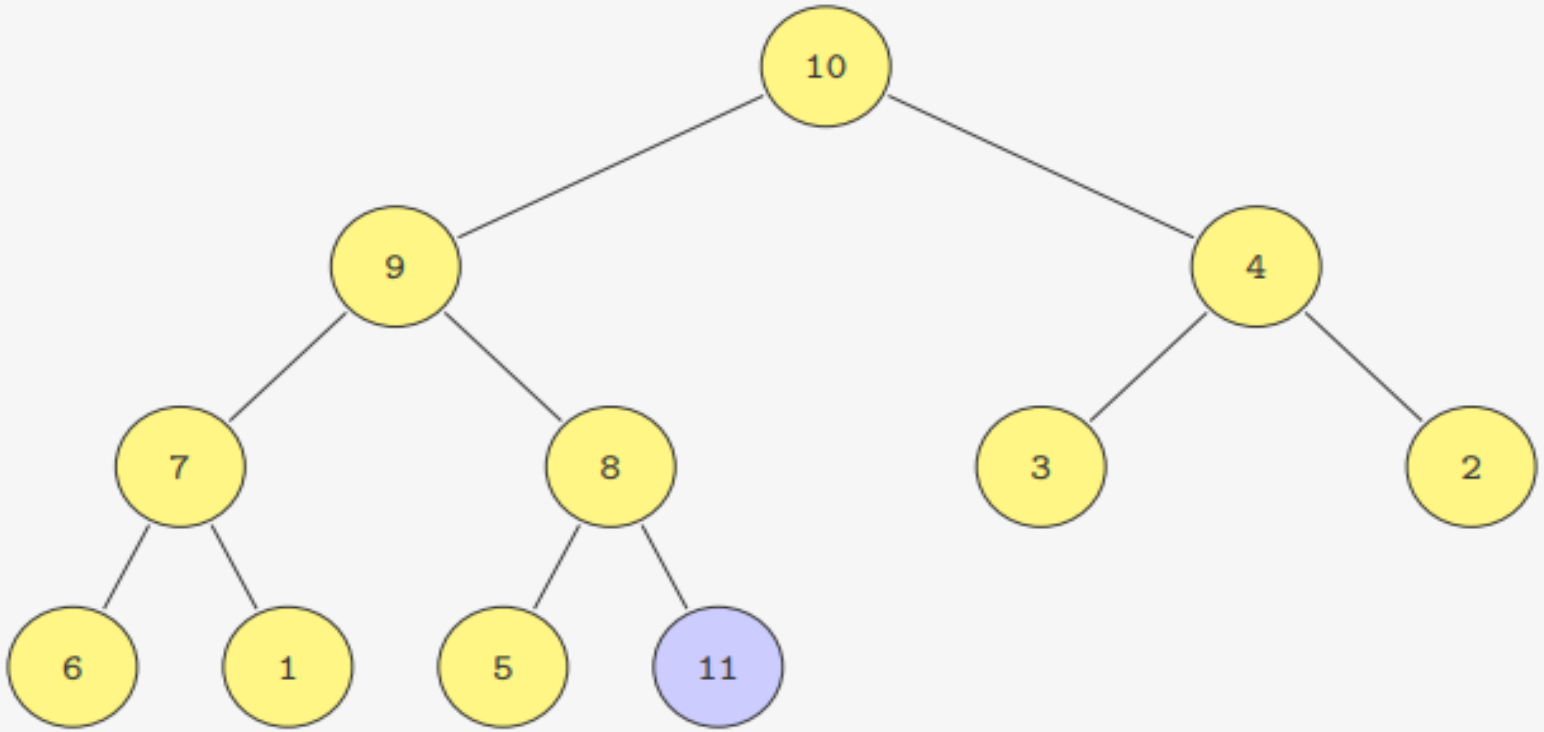
Inserir um elemento

- Insira o elemento no final do heap
- Compare o elemento a ser inserido com o Nó pai
  - Se estiver em ordem, a inserção foi realizada corretamente
  - Se não estiver em ordem, troque com o elemento pai e repita o passo 2 até terminar ou chegar a raiz.

Procedimento para Inserção no Heap Maximo



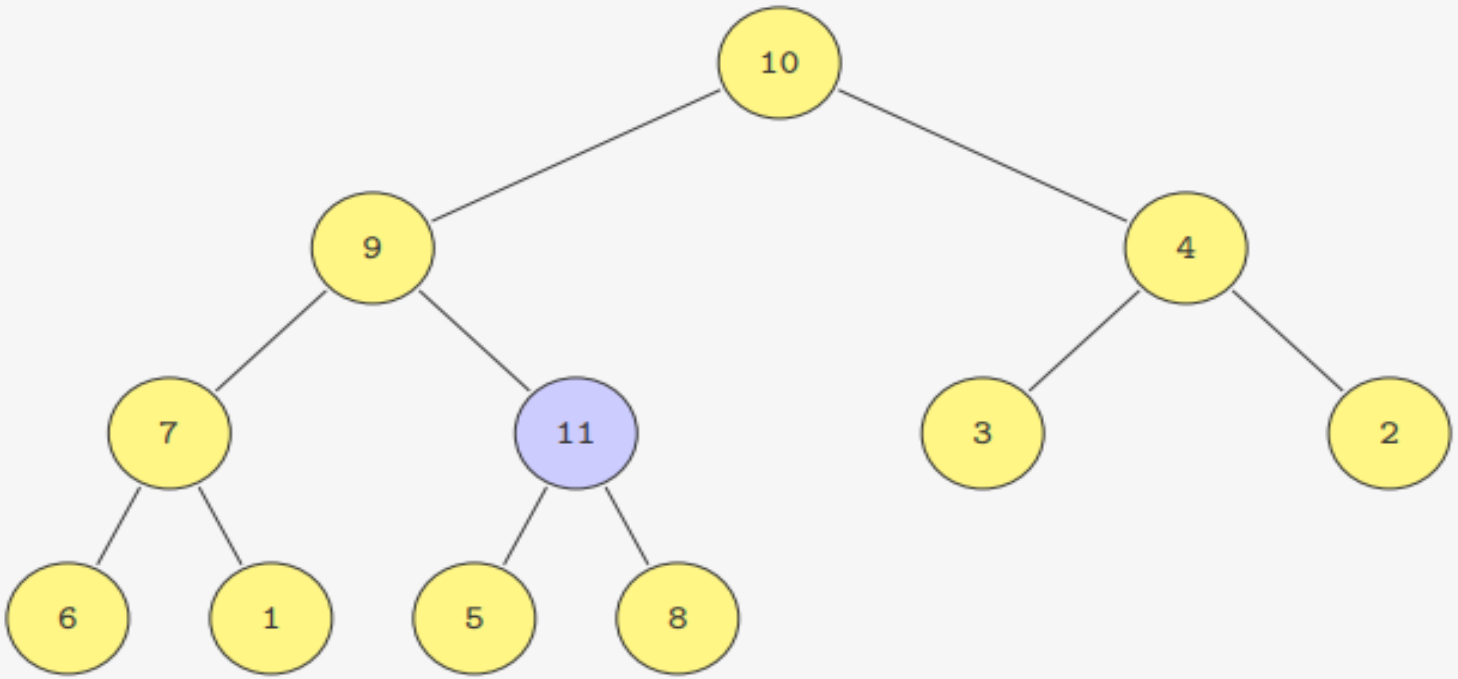
Exemplo de Inserção no Heap Máximo



Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

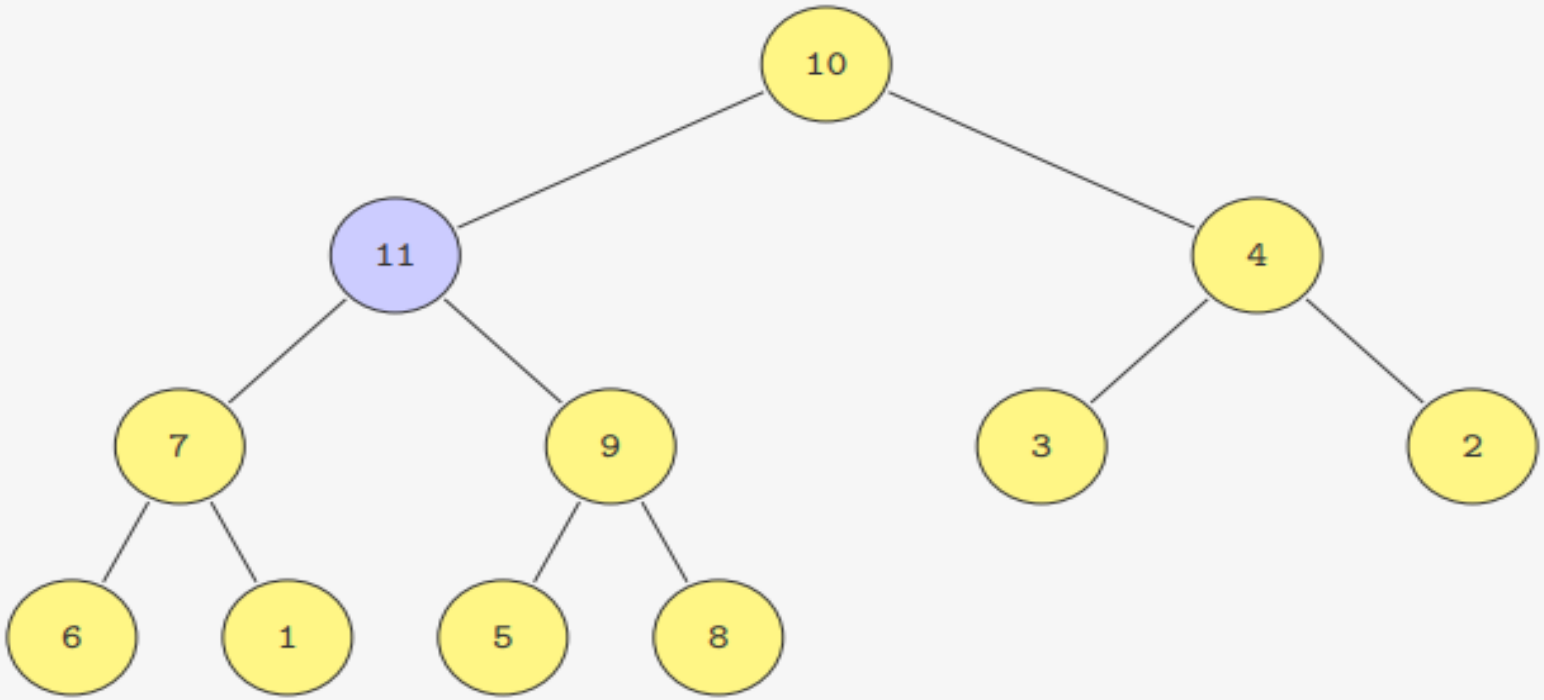
Exemplo de Inserção no Heap Máximo



Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

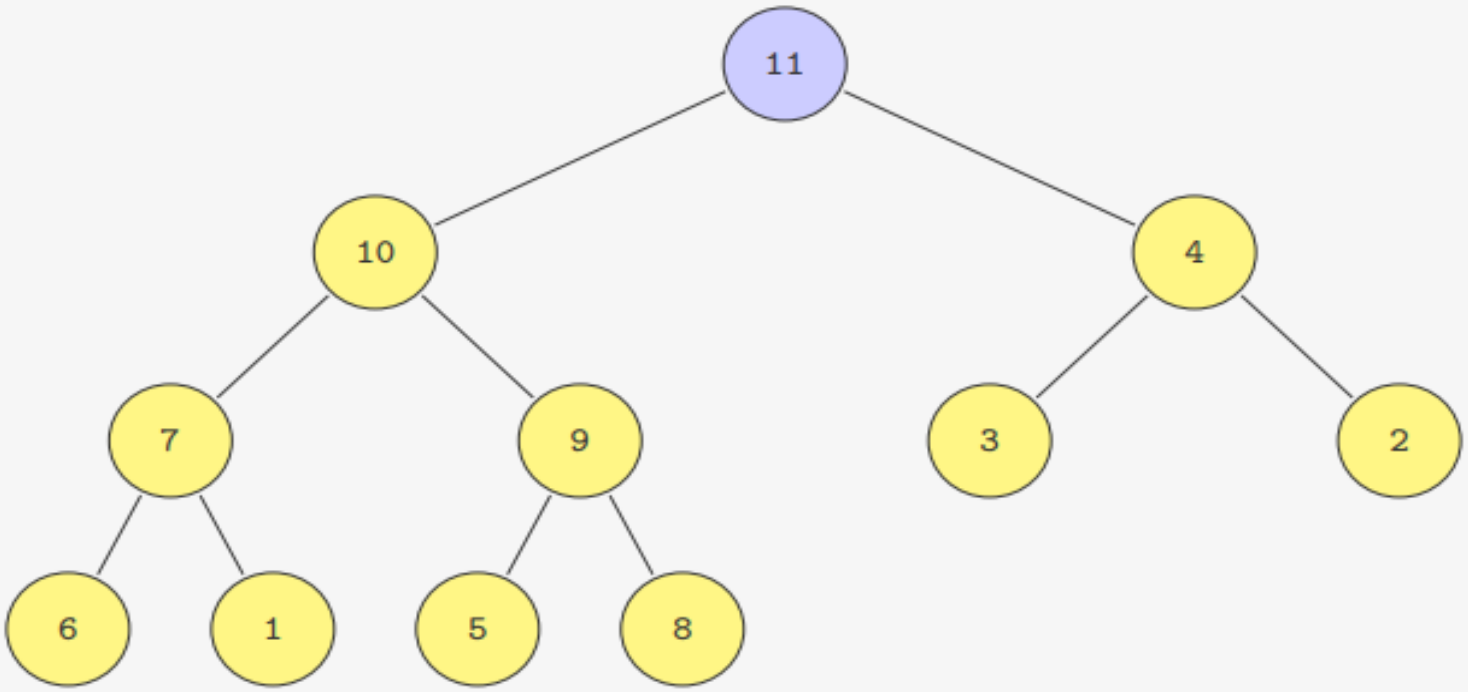
Exemplo de Inserção no Heap Máximo



Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

Exemplo de Inserção no Heap Máximo



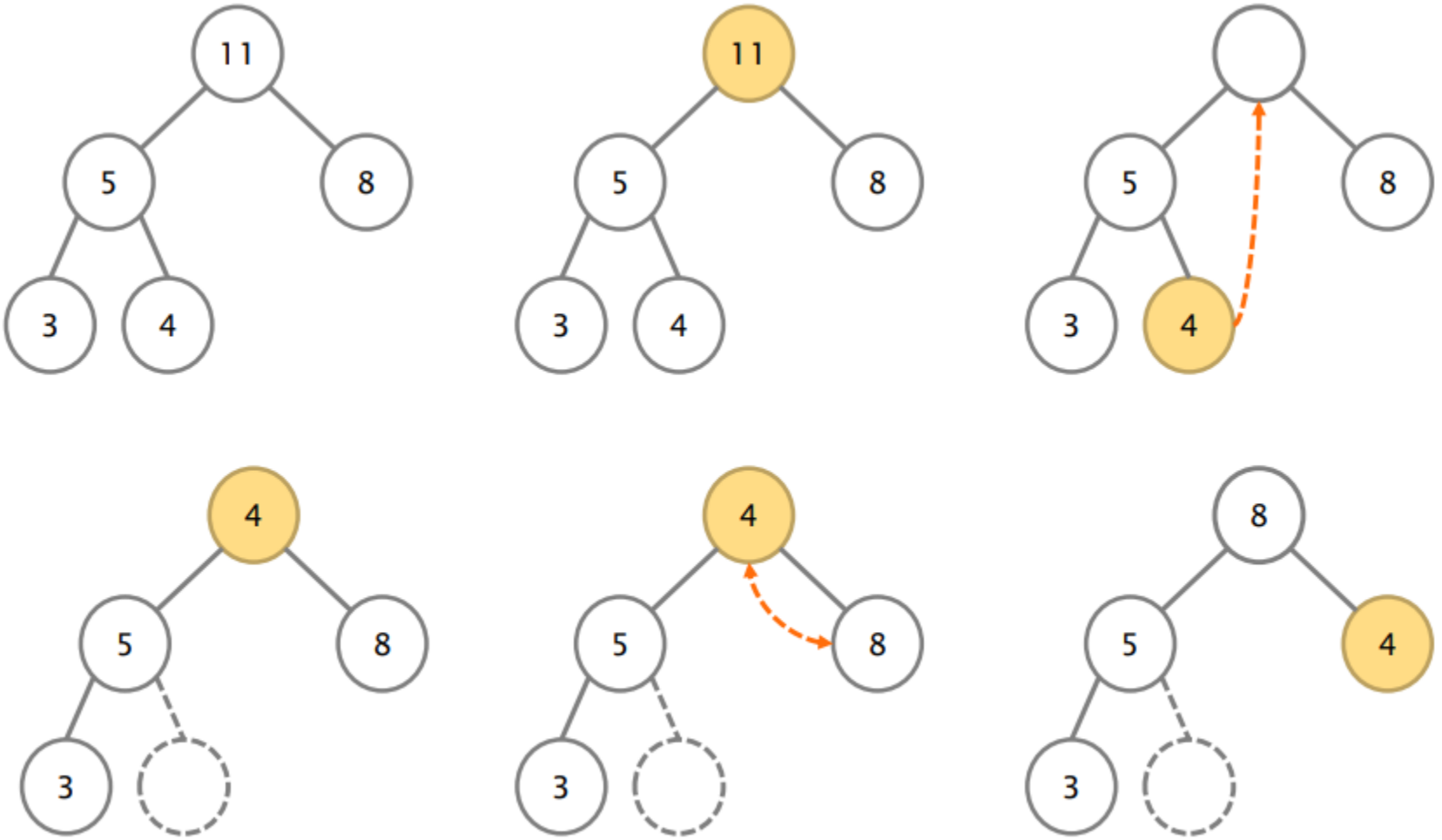
Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

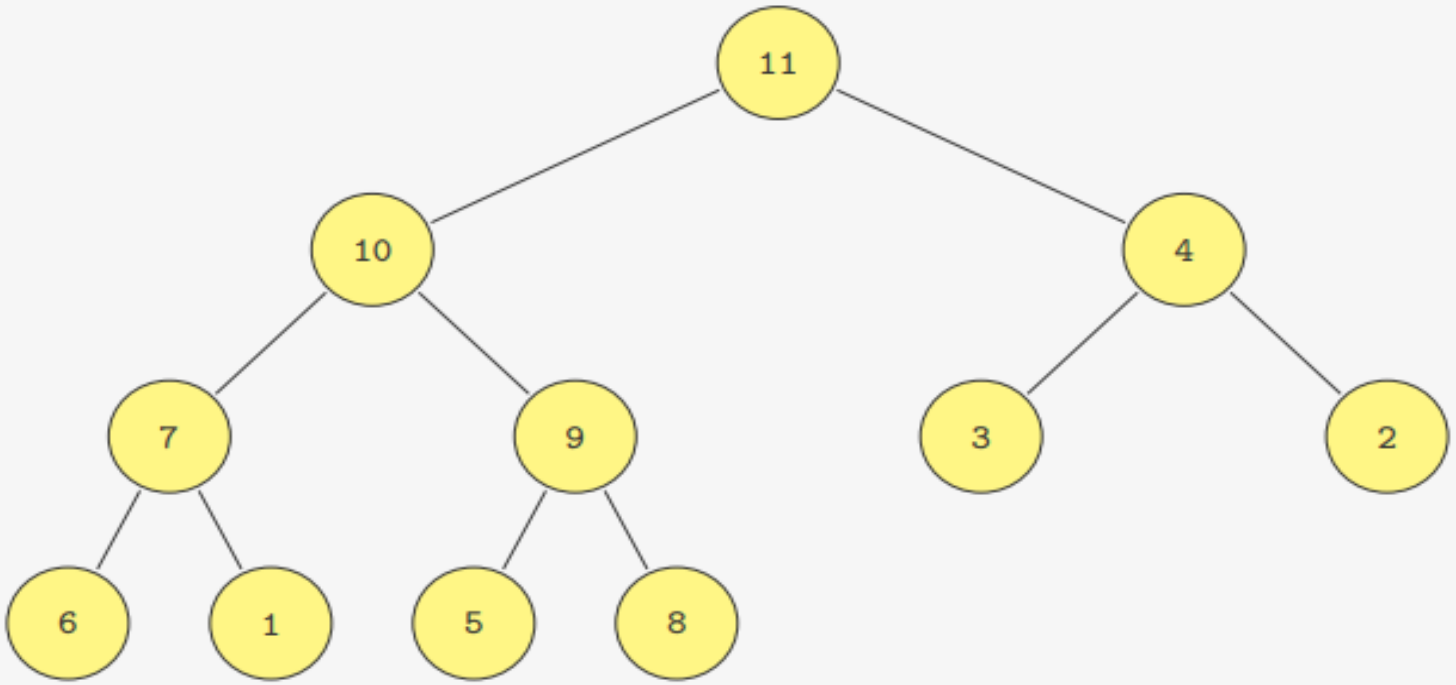
Remover um elemento

- Coloque na raiz o último elemento
- Compare ele com os seus filhos
  - Se estiver em ordem, a remoção foi concluída
  - Se não estiver em ordem, troque com o maior filho e repita o passo 2 até terminar ou chegar em um Nó folha

Procedimento para Remoção no Heap Máximo

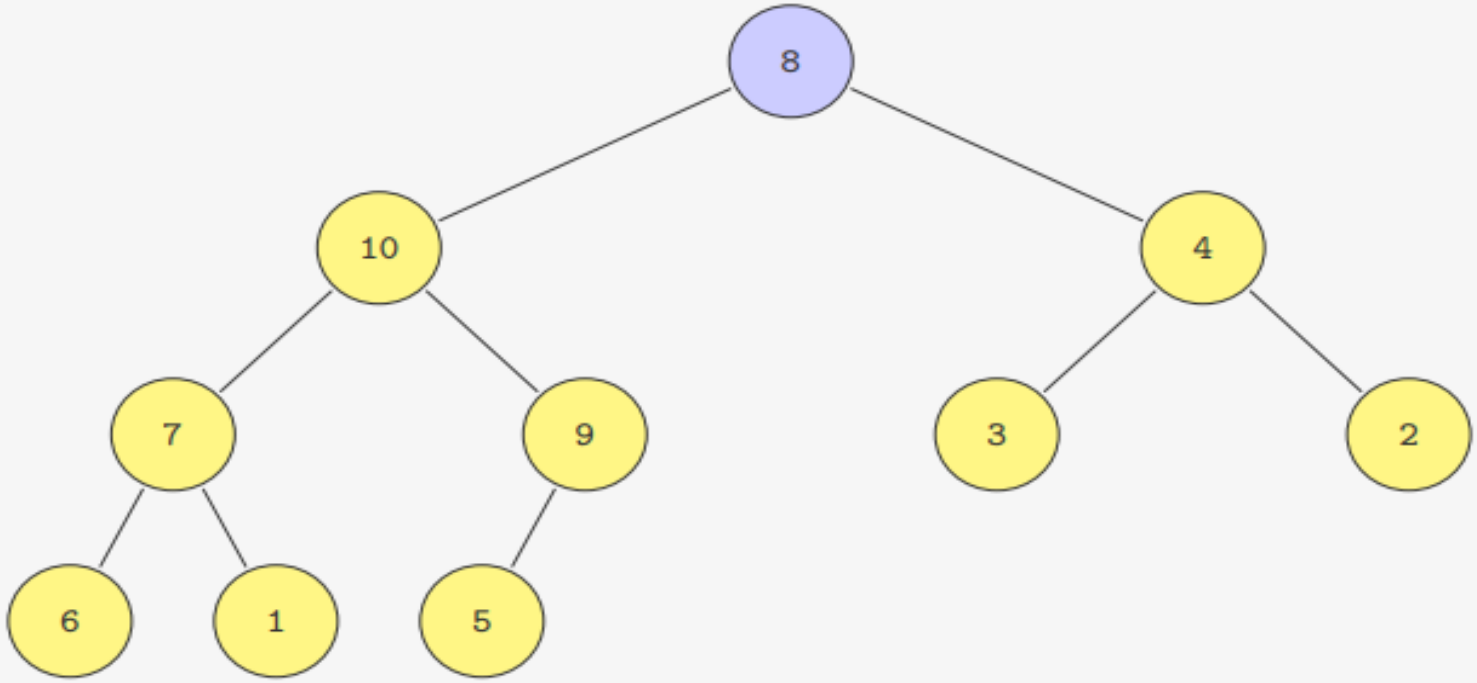


Exemplo de Remoção no Heap Máximo



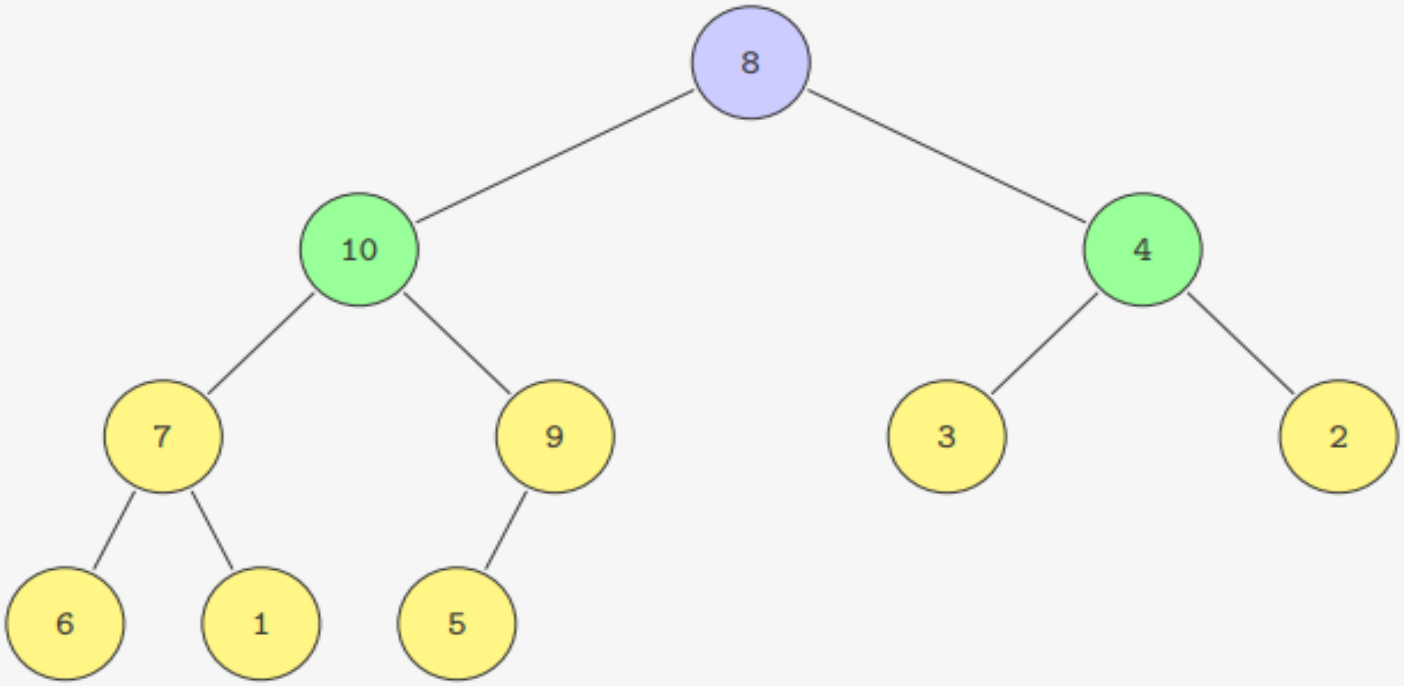
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Exemplo de Remoção no Heap Máximo



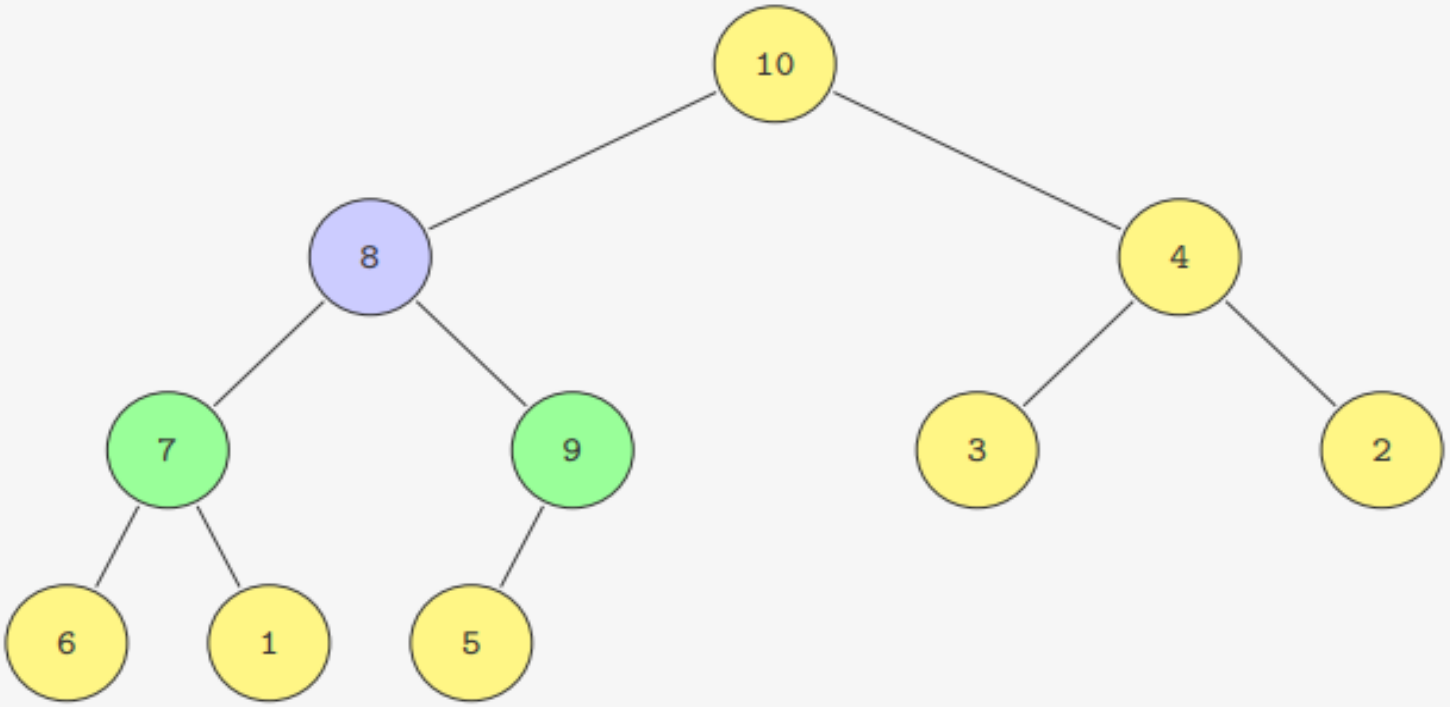
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Exemplo de Remoção no Heap Máximo



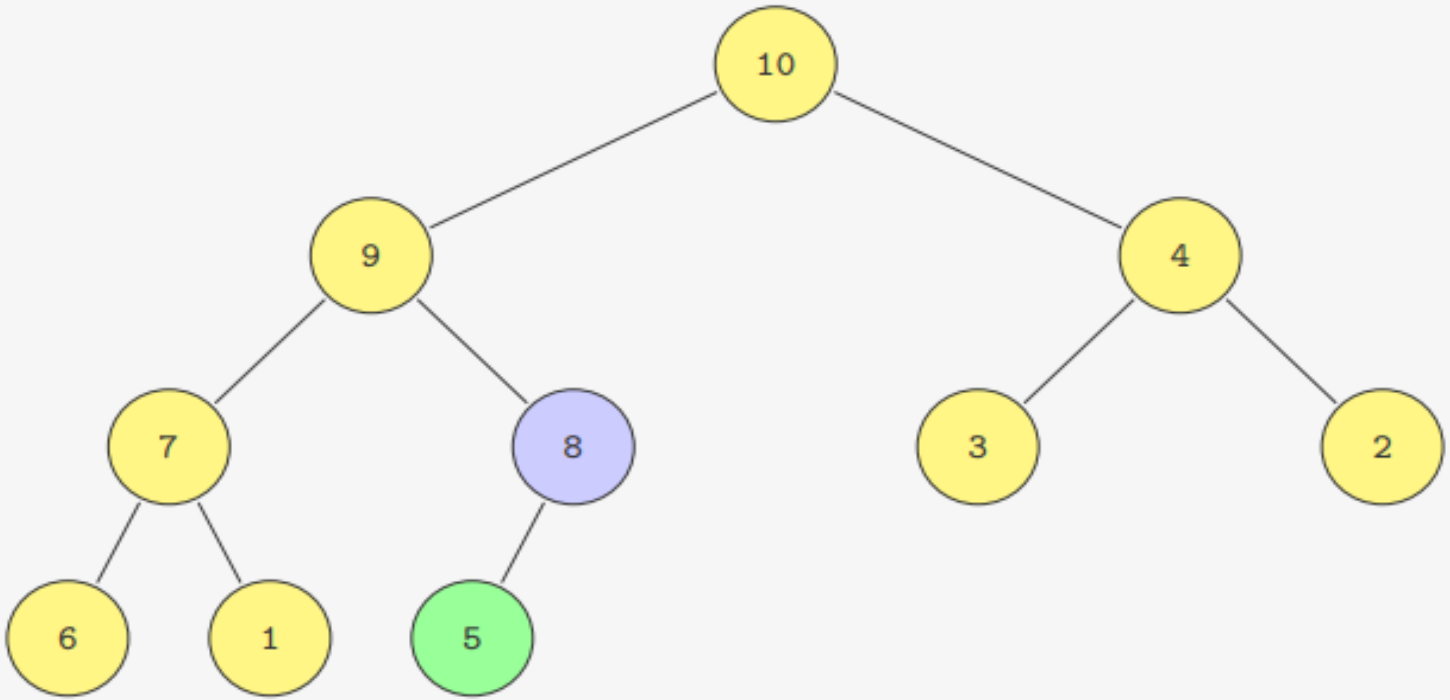
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Exemplo de Remoção no Heap Máximo



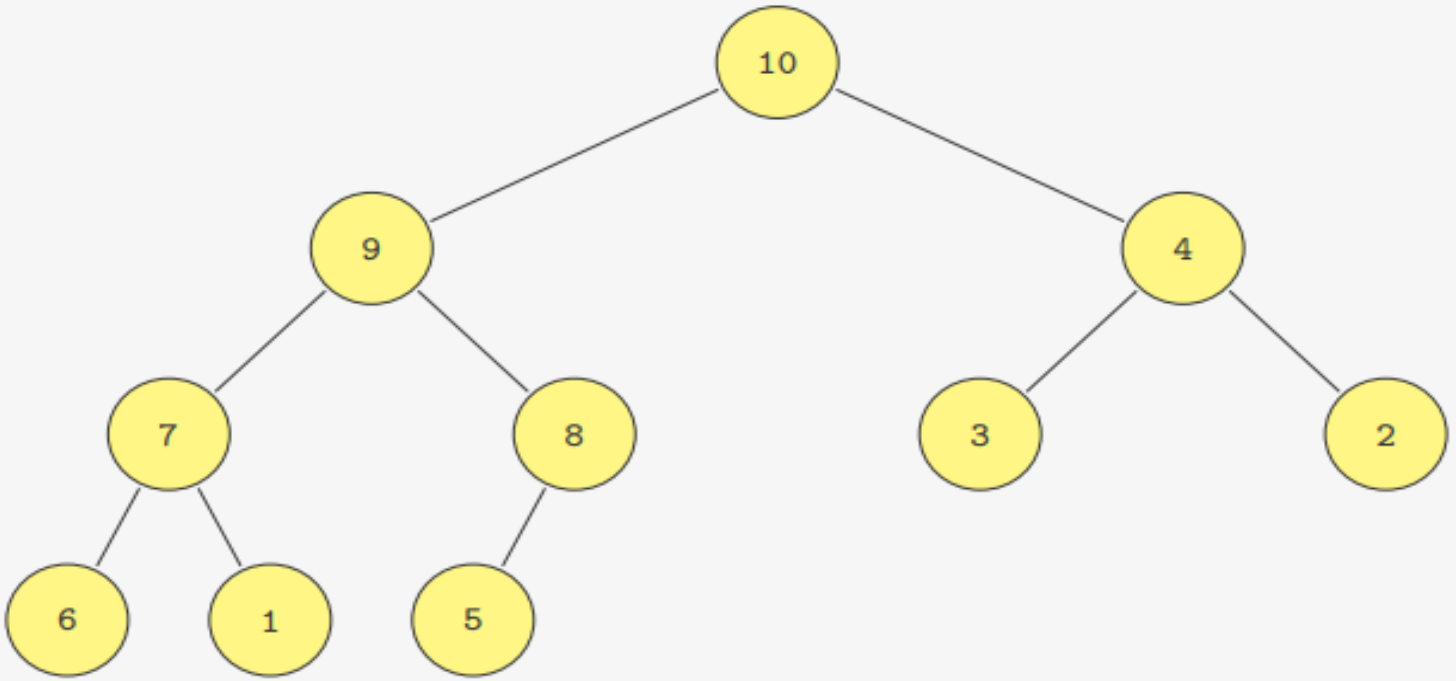
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Exemplo de Remoção no Heap Máximo



- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

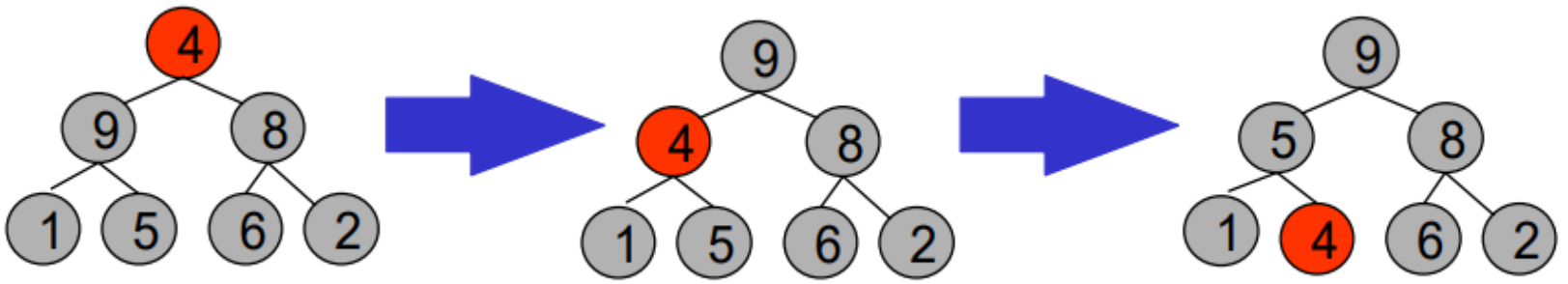
Exemplo de Remoção no Heap Máximo



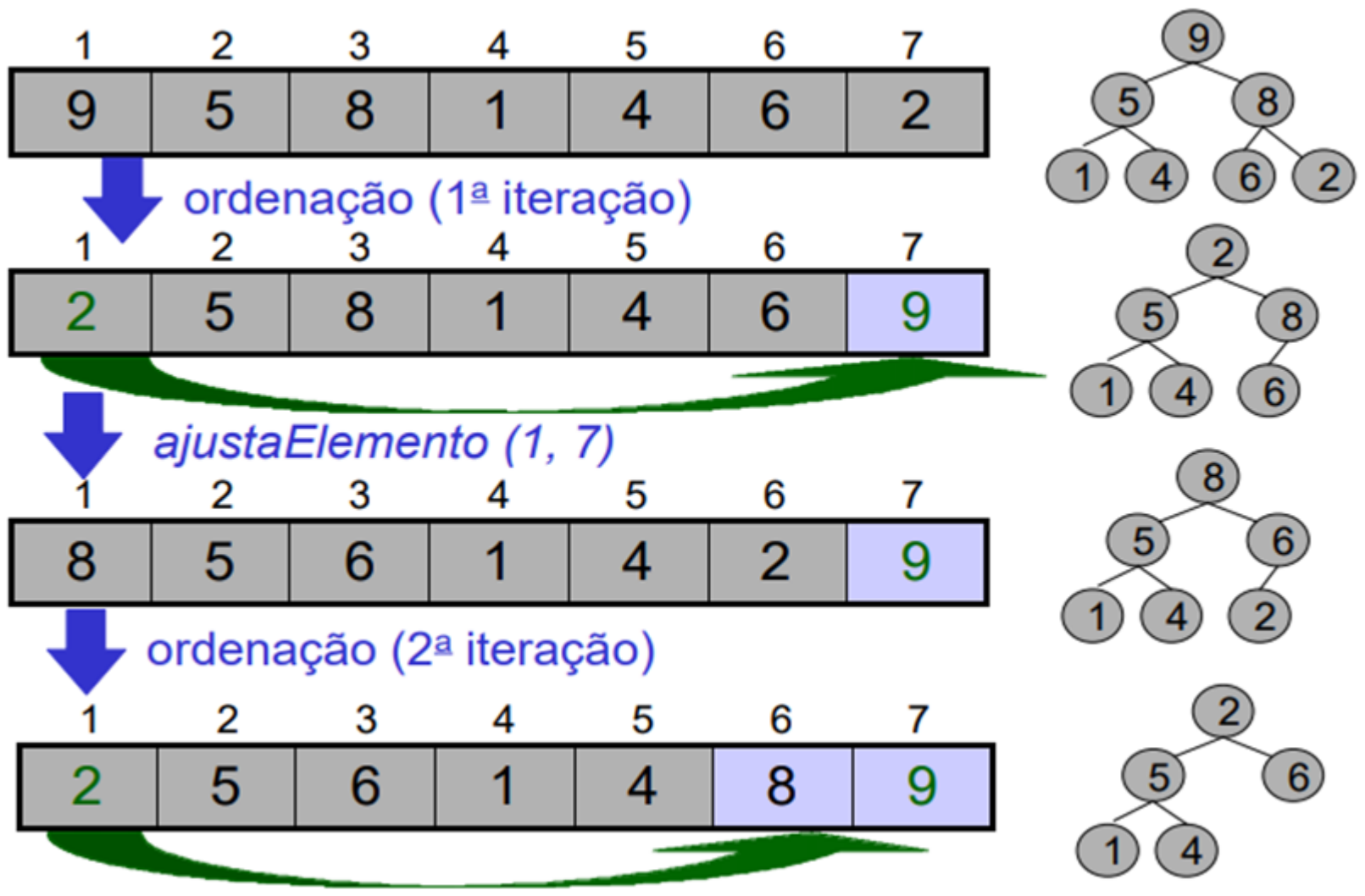
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Etapas da ordenação

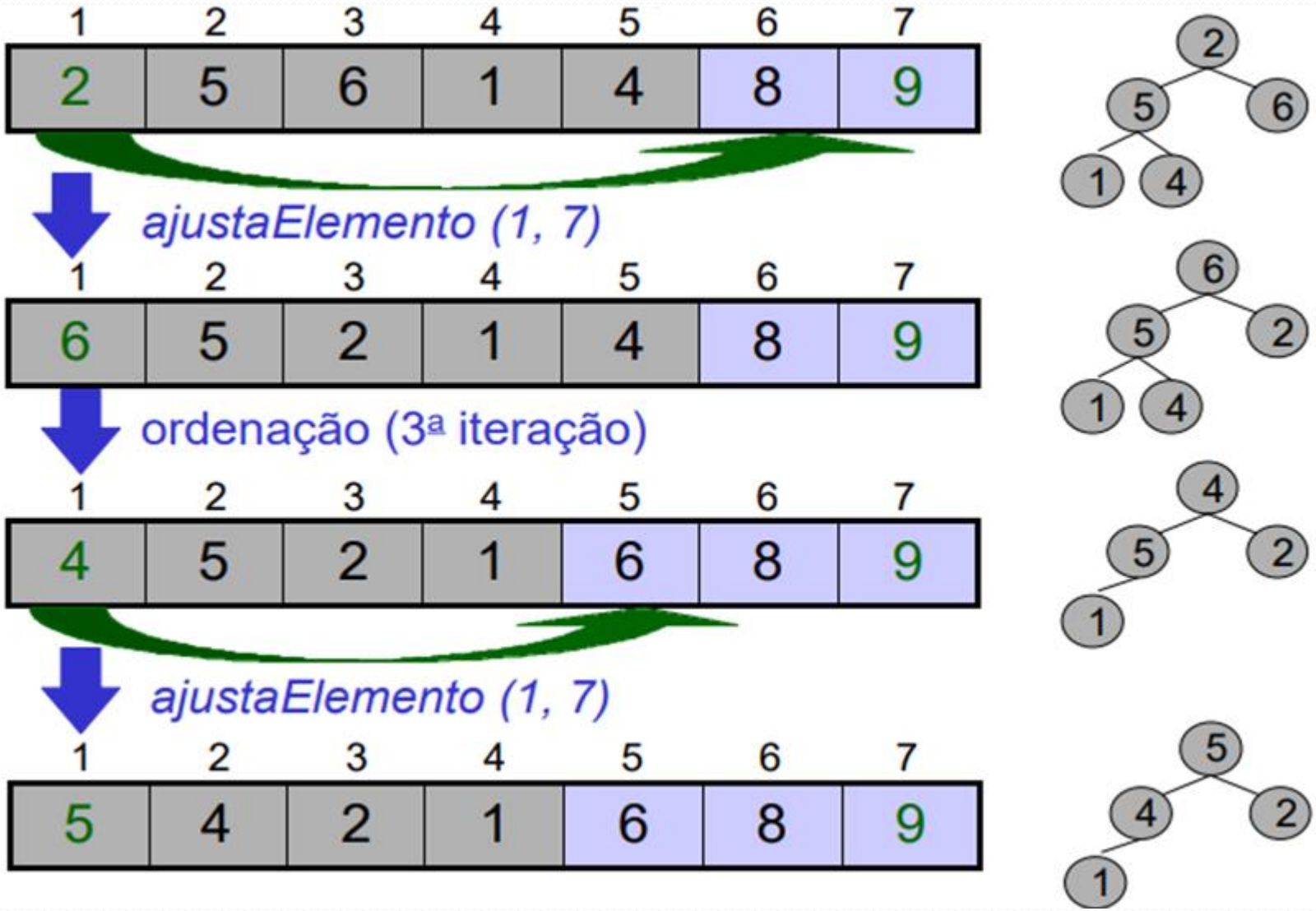
- Método auxiliar responsável pelo ajuste de um elemento no heap
  - método ajustaElemento(posição, vetor.lenght)
- Realiza trocas no heap para posicionar corretamente um elemento • Exemplo: ajustaElemento(1, 7)



Etapas da ordenação



Etapas da ordenação



```
void constroiHeapMax(int [] V) {
   int compHeap = V.length;
   for(int i = (V.length)/2 - 1; i >= 0 ; i--)
      refazHeapMax(V, i, compHeap);
}
```

```
void refazHeapMax(int V[], int i, int compHeap) {
  int esq, dir, maior, temp;
  esq = esquerda(i); dir = direita(i);
  if(esq < compHeap && V[esq] > V[i]) {
    maior = esq;
  } else {
    maior = i;
  }
  if(dir < compHeap && V[dir] > V[maior]) {
    maior = dir;
  }
  if(maior != i) {
    // trocar V[i] <==> V[maior]
    temp = V[i];
    V[i] = V[maior];
    V[maior] = temp;
    // Ajusta a posicao de maior, se incorreta.
    refazHeapMax(V, maior, compHeap);
  }
}
```

```
void heapSort(int V[]) {
  int i, compHeap, temp;
  // Constrói o heap máximo do arranjo todo
  compHeap = V.length;
  constroiHeapMax(V);

  for(i = V.length-1; i > 0; --i) {
    // Troca V[0] <==> V[i]
    temp = V[0];
    V[0] = V[i];
    V[i] = temp;
    // Diminui o heap, pois V[i] está posicionado
    compHeap--;
    refazHeapMax(V, 0, compHeap);
  }
}
```

Desempenho

- QuickSort é ruim no pior caso (ocorre raramente)
- Na prática, QuickSort tem um desempenho melhor que o HeapSort, considerando que a sua média de iterações é menor (proporcional a log n)

	QuickSort	*HeapSort*
Pior caso	$O(n^2)$	$O(n \log n)$
Caso médio	$O(n \log n)$	$O(n \log n)$
Melhor caso	$O(n \log n)$	$O(n \log n)$

Desempenho

- É o mais interessante para arquivos com menos do que 20 elementos.
- O método é estável.
- Possui comportamento melhor do que o método da bolha (Bubblesort) que também é estável.
- Sua implementação é tão simples quanto as implementações do Bubblesort e Seleção.
- Para arquivos já ordenados, o método é O(n).
- O custo é linear para adicionar alguns elementos a um arquivo já ordenado.

Desempenho

- Vantagens
  - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- Desvantagens
  - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.
  - O Heapsort não é estável.
- Recomendado
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.

Contatos

- Email: fabio.silva321@fatec.sp.gov.br
- Linkedin: https://br.linkedin.com/in/b41a5269
- Facebook: https://www.facebook.com/fabio.silva.56211