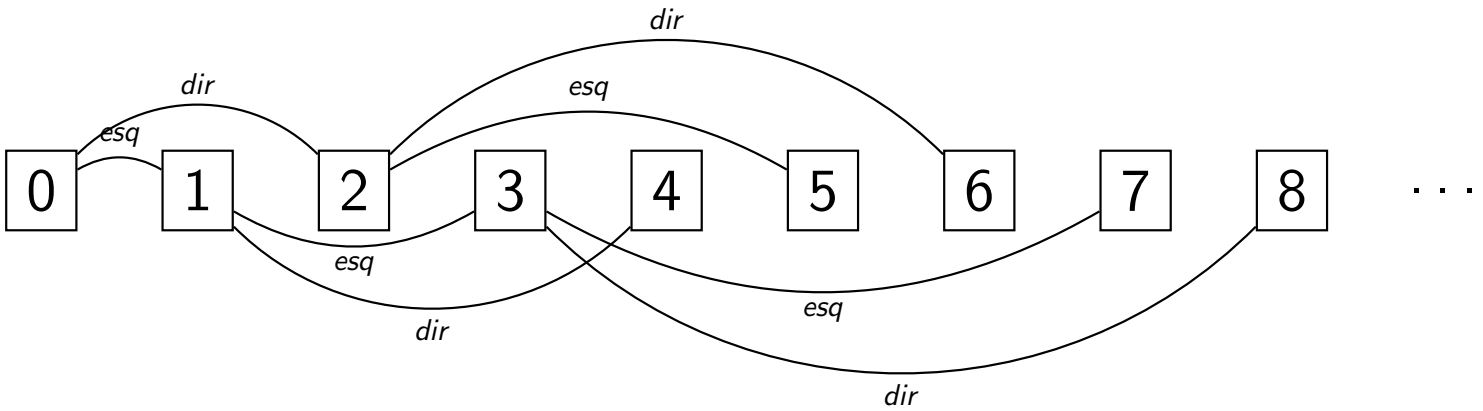
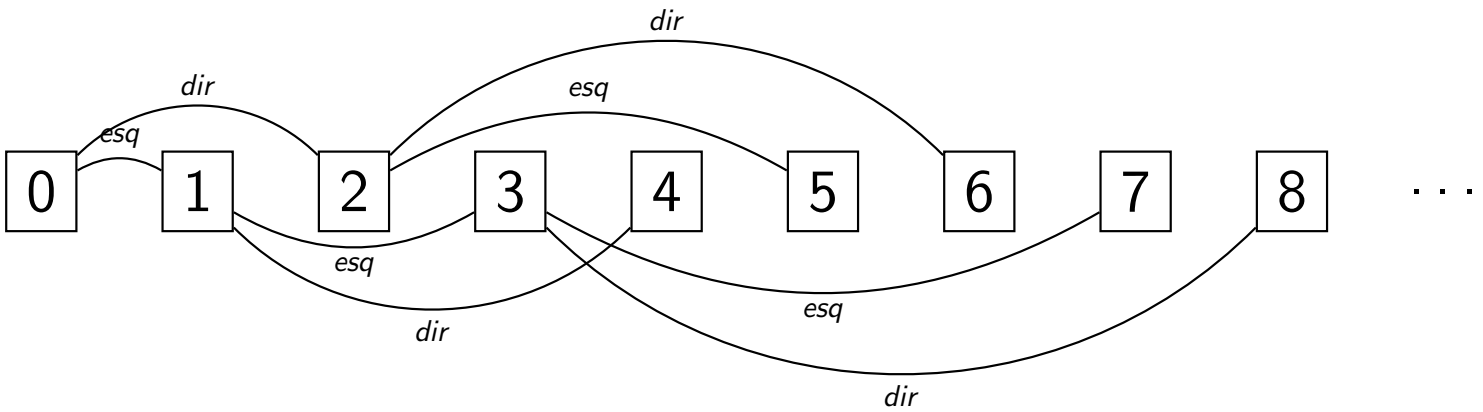
Estruturas de Dados - Heap

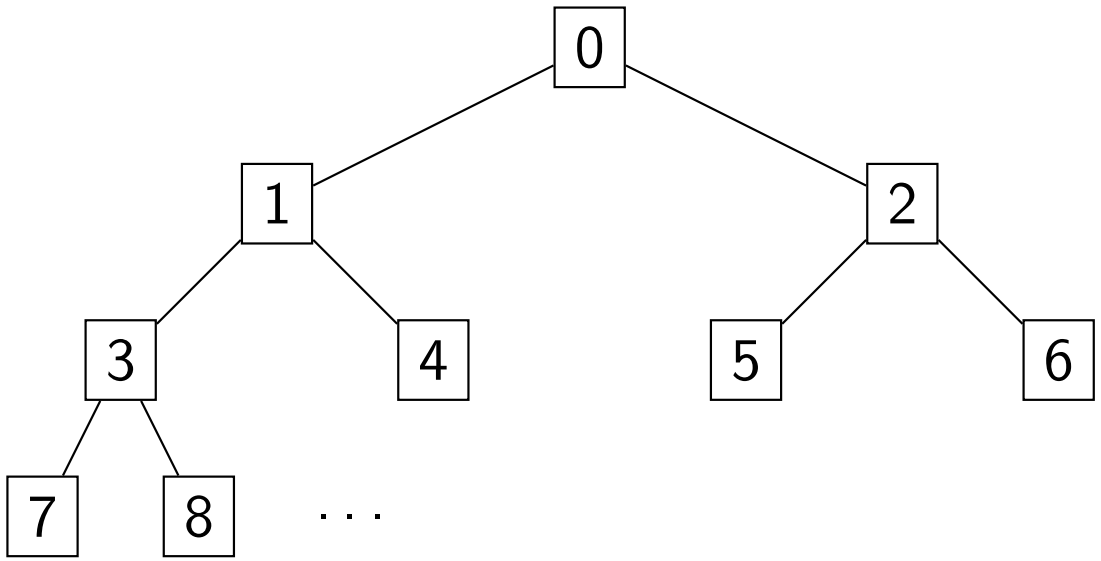
Heap

- Heap: ideia de árvore aplicada a um vetor
- Especificamente, trabalharemos a ideia de árvore binária
  - cada nó tem apenas dois filhos
  - apesar disso, também podemos montar heaps com mais!
- Novamente, trabalharemos com um vetor de inteiros
- Características de uma árvore binária:
  - cada nó tem um pai
  - cada nó tem dois filhos, *esq* e *dir*



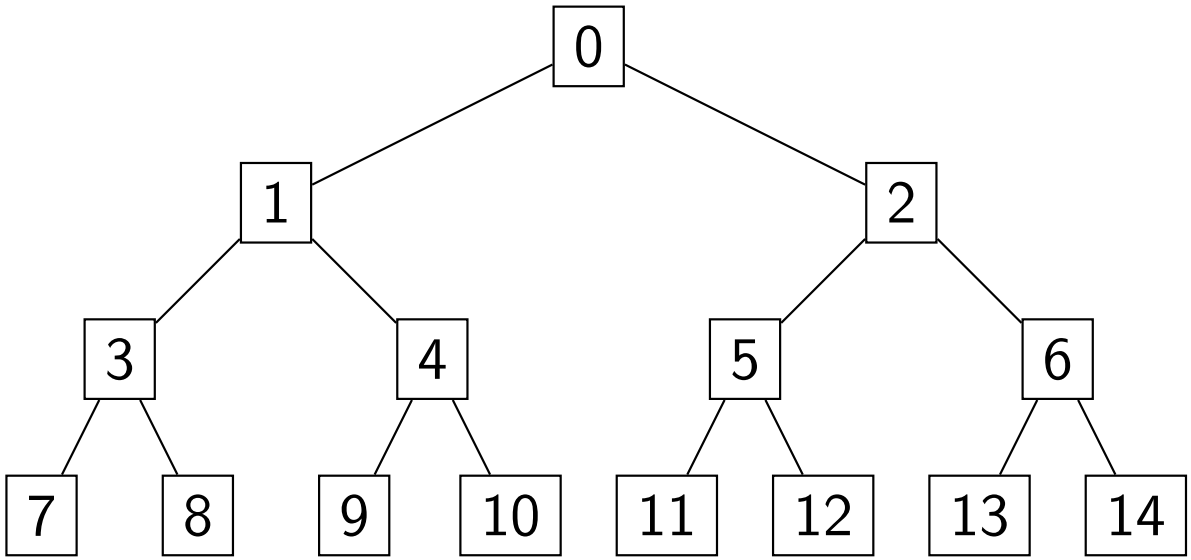
Heap





Heap

- Cada nó tem um pai e dois filhos
- Portanto, dado um nó, devemos saber quem é seu pai e quem são os seus filhos
  - obs: a raiz não tem pai, e há nós sem filhos, ou apenas um filho
  - seu pai e seus filhos, caso existam, estão armazenados em posições do vetor
- Como fazer isso em um vetor?



Heap

- Maneira eficiente: checar pai e filhos a partir do índice no vetor
- Note que para um nó na posição (índice) i, o pai e os filhos são sempre os mesmos:
  - seu pai está na posição $\left\lfloor \frac{i-1}{2} \right\rfloor$
  - seu filho esquerdo está na posição $2i+1$
  - seu filho direito está na posição $2i+2$
- Criaremos funções $\textsc{Pai}(i)$, $\textsc{Esq}(i)$ e $\textsc{Dir}(i)$ a partir disto!
- Obs: vamos simplificar estas ideias, ignorando os casos onde os nós não tem pai ou não tem um dos filhos

Heap: pai, filho esquerdo e filho direito

Algoritmo: Pai(i)

Entrada: índice i

Saída: índice do pai do nó i

1 **retorne** $\left\lfloor \frac{i-1}{2} \right\rfloor$

Algoritmo: Esq(i)

Entrada: índice i

Saída: índice do filho esquerdo do nó i

1 **retorne** $2i + 1$

Algoritmo: Dir(i)

Entrada: índice i

Saída: índice do filho direito do nó i

1 **retorne** $2i + 2$

Heap

- Dois tipos de heap
  - Heap de máximo (ou Max Heap):
    - a raiz é o maior elemento do vetor
    - cada nó é maior que os seus filhos
  - Heap de mínimo (ou Min Heap):
    - a raiz é o menor elemento do vetor
    - cada nó é menor que os seus filhos
- Trabalharemos com métodos voltados ao Max Heap
- Estrutura de um heap:
  - H: heap (vetor) com n posições
  - $H.max$: quantidade máxima de elementos no heap
  - $H.pos$: próxima posição disponível no vetor
- Obs: as folhas estarão nos índices de $\left\lfloor \frac{H.pos}{2} \right\rfloor$ a $H.pos - 1$

Heap: criar

Algoritmo: CriaHeap(n)

Entrada: tamanho n do heap

Saída: heap H

1 criar novo heap H com um vetor de n posições

2 $H.max = n$

3 $H.pos = 0$

4 **retorne** H

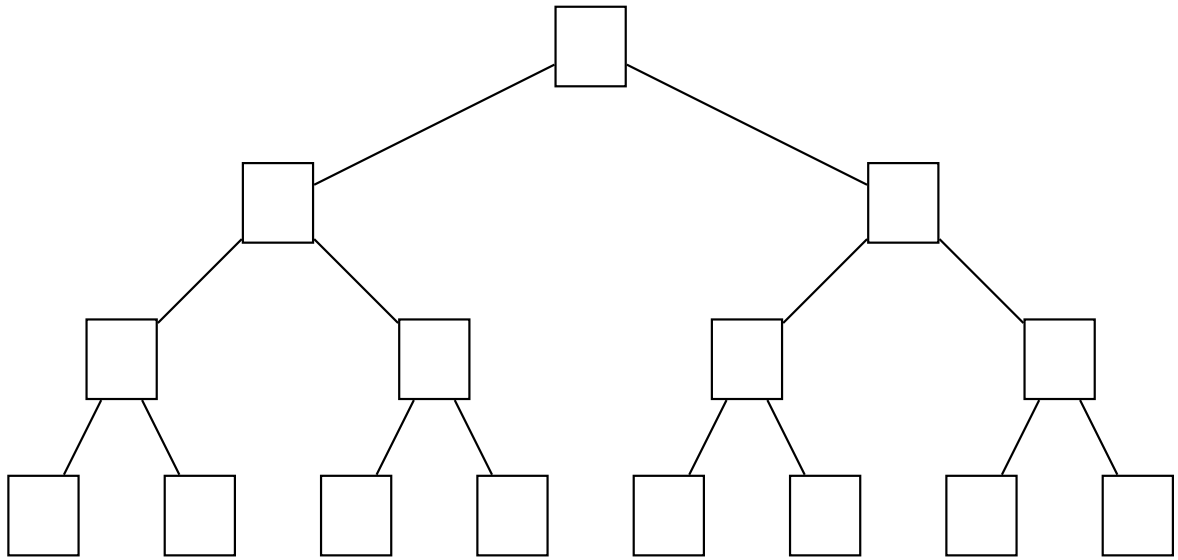
Complexidade: $O(1)$

Heap

- Ao inserirmos um elemento em um heap, devemos inserir na posição $H.pos$
  - $H.pos$ representa ao mesmo tempo a quantidade de elementos no heap, e a próxima posição livre para inserirmos no heap
    - mesma ideia de cardinalidade!
- Porém, ao inserirmos, devemos verificar se estamos quebrando a propriedade de heap
  - sempre inserimos na posição $H.pos$
  - é possível que um elemento inserido em $H[H.pos]$ seja maior que o seu pai
- Portanto, a cada inserção devemos fazer uma verificação
  - se a propriedade de heap continua válida, ok
  - senão, devemos trocar elementos de posição para satisfaze-la

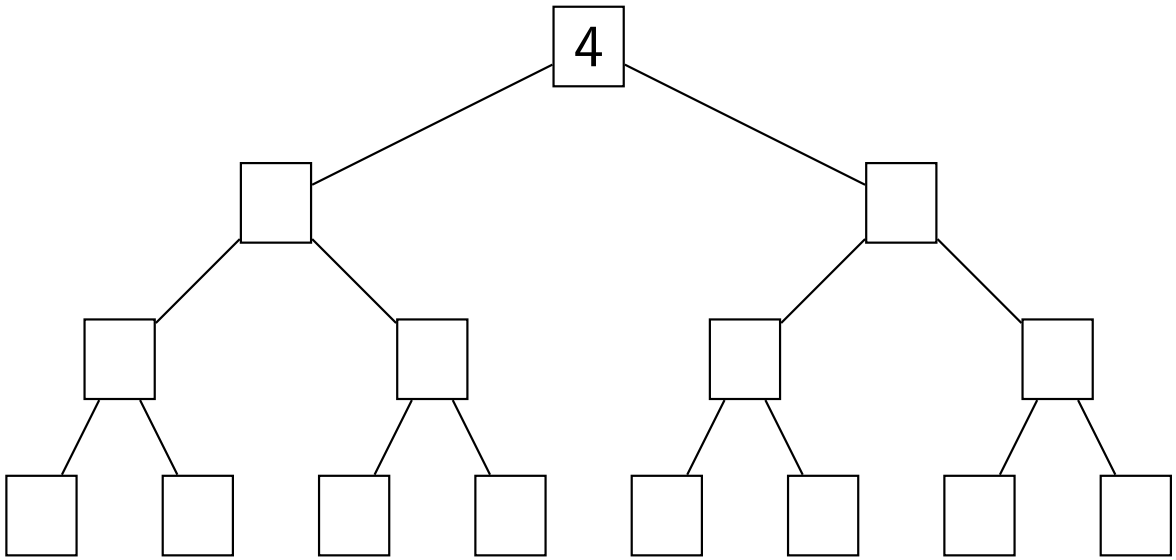
Heap: exemplo

Exemplo: inserir valores 4, 2, 3 e 6, nesta ordem, num heap inicialmente vazio



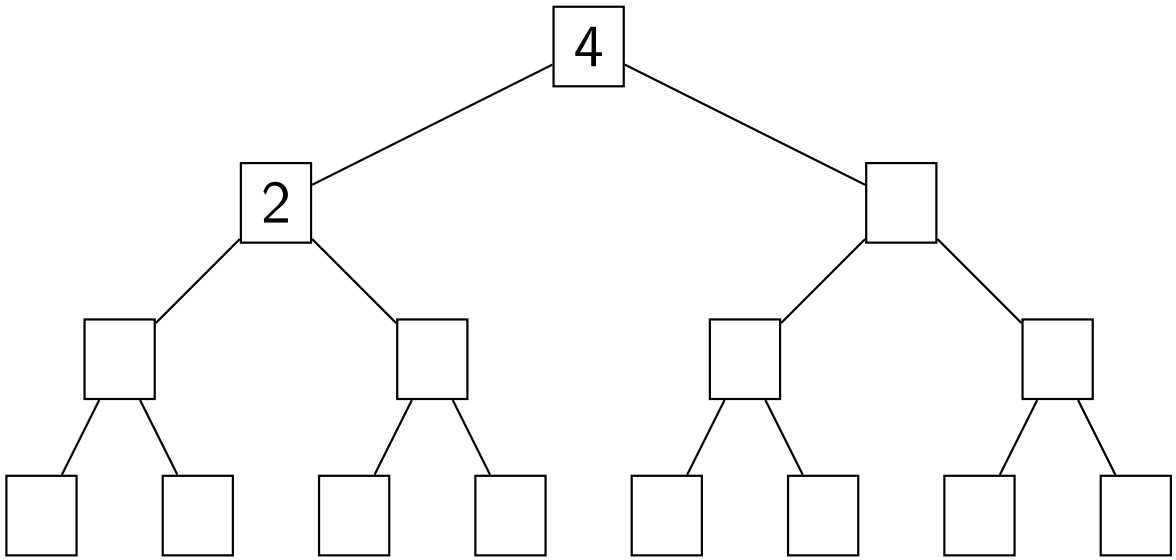
Heap: exemplo

Inserir 4



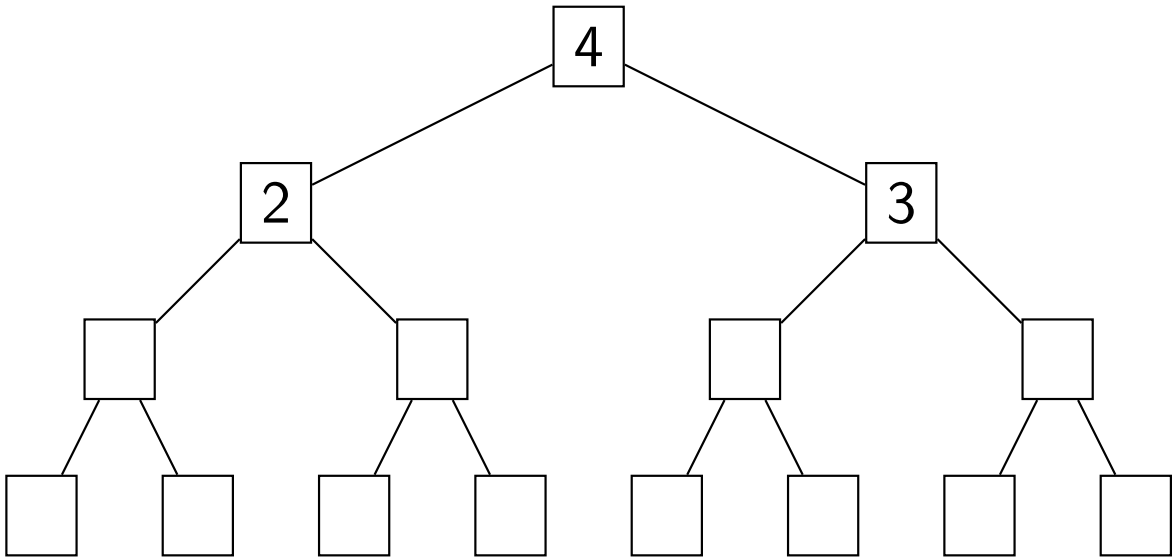
Heap: exemplo

Inserir 2



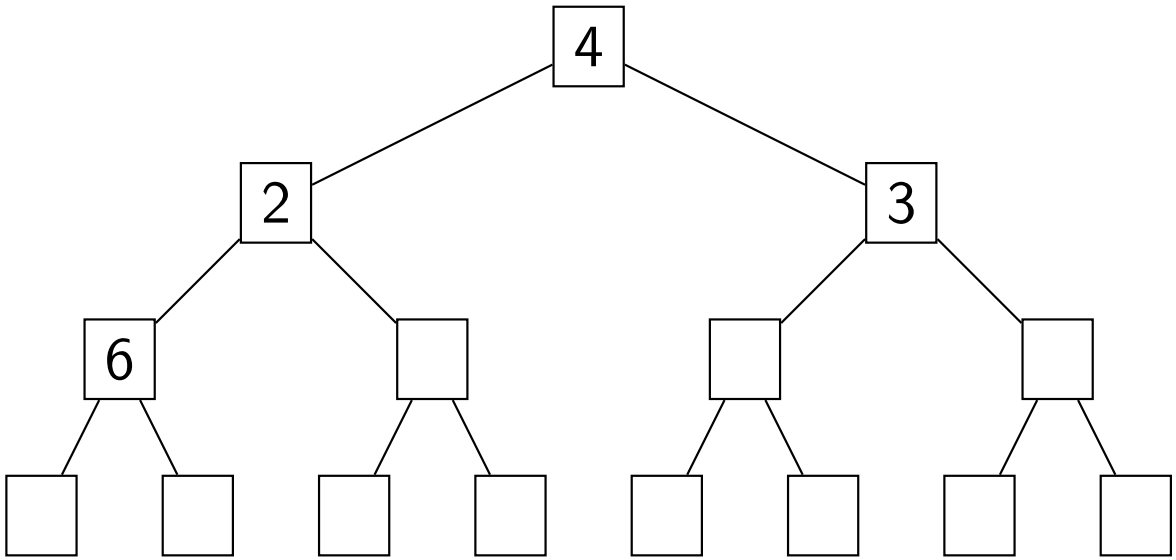
Heap: exemplo

Inserir 3

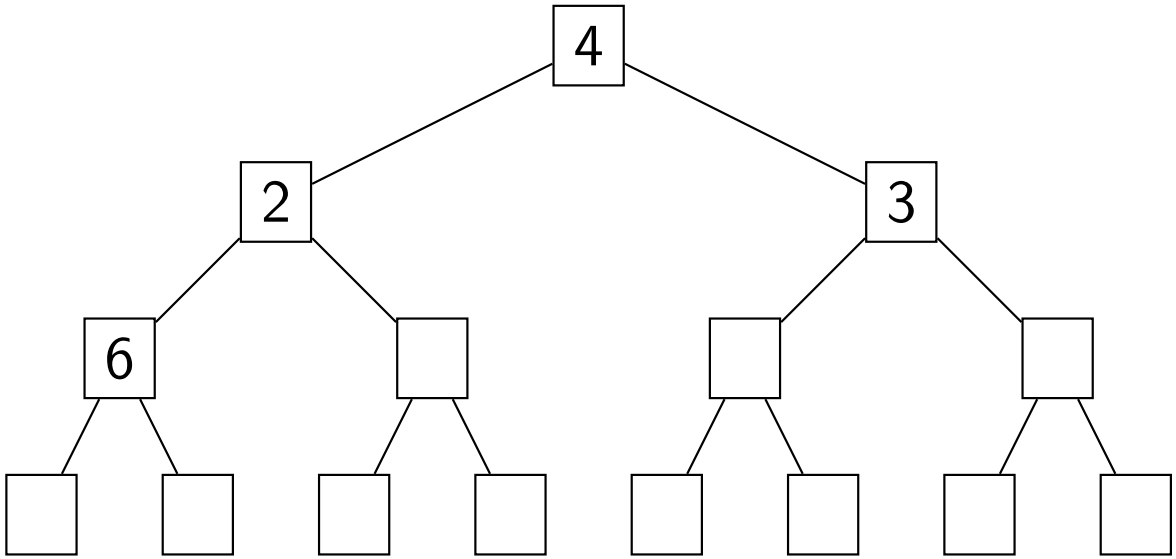


Heap: exemplo

Inserir 6



Heap: exemplo

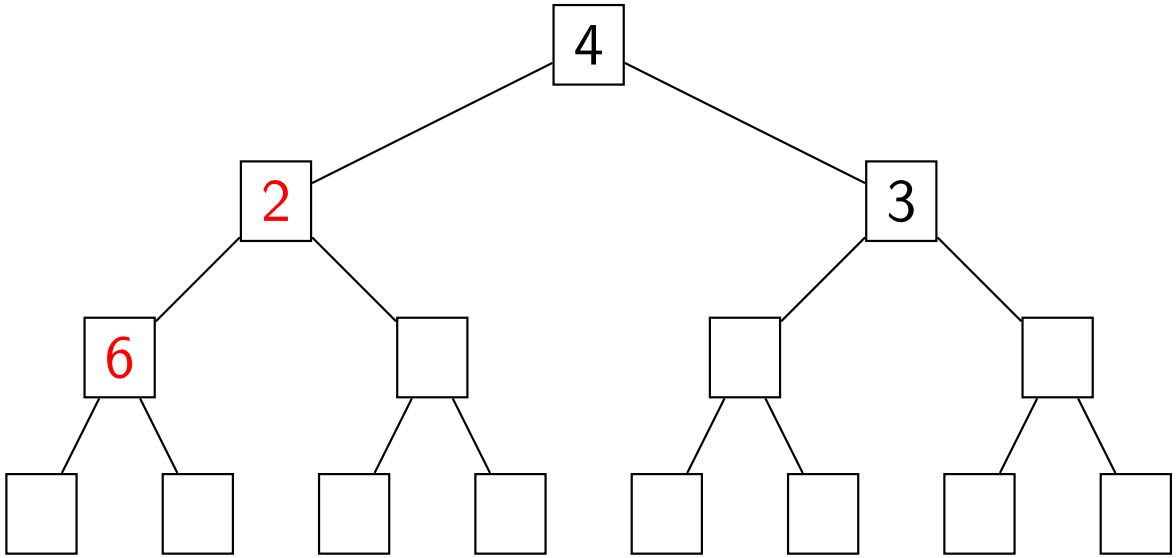


Problema: quebramos a propriedade de heap
Como resolver este problema?

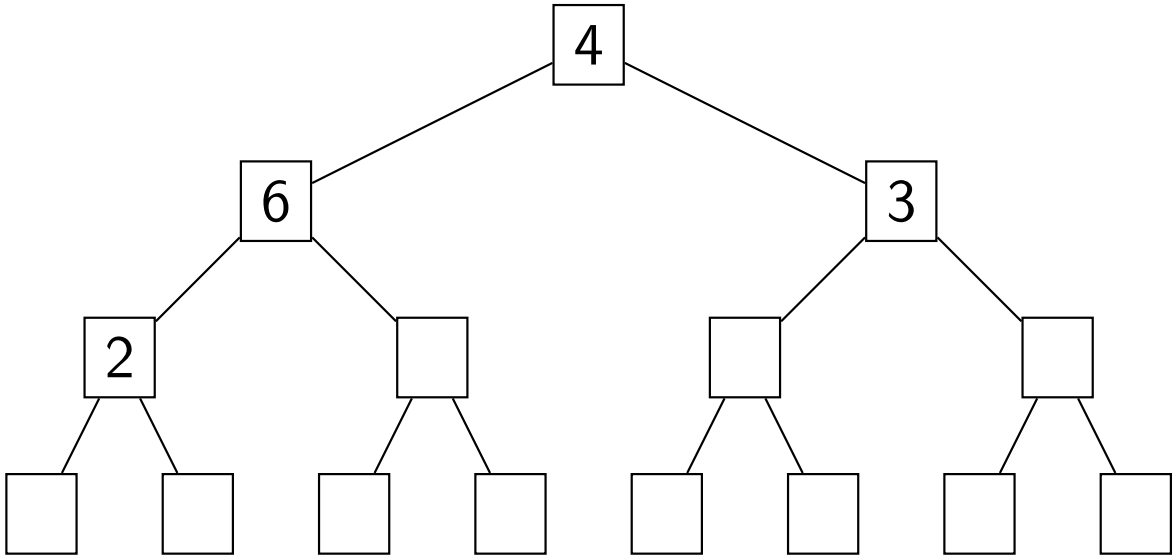
Heap

- A cada inserção, devemos analisar se a propriedade de heap foi quebrada
  - obs: a propriedade a ser analisada é a de Max Heap
  - para todo nó, seu pai é maior do que ele
- Ou seja, a cada novo elemento inserido, devemos analisar se o elemento inserido é maior do que o seu pai
  - se ele não for maior, ok
  - se ele for maior, devemos trocar os dois de posição
- Obs: uma única troca pode não ser o bastante
  - devemos continuar trocando enquanto o pai for menor que o filho, ou até chegarmos na posição zero

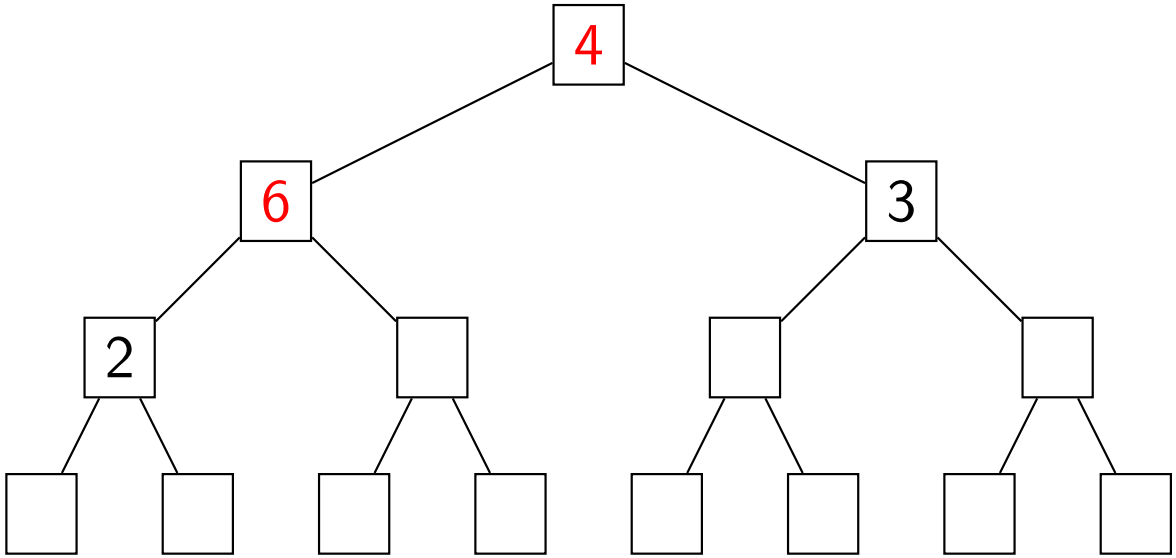
Heap: exemplo



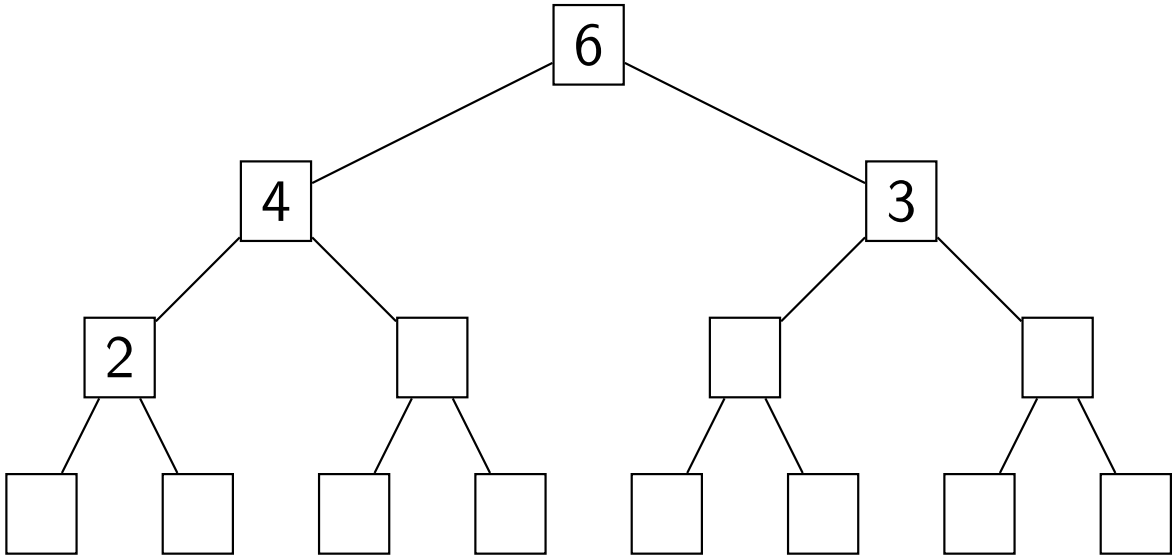
Heap: exemplo



Heap: exemplo

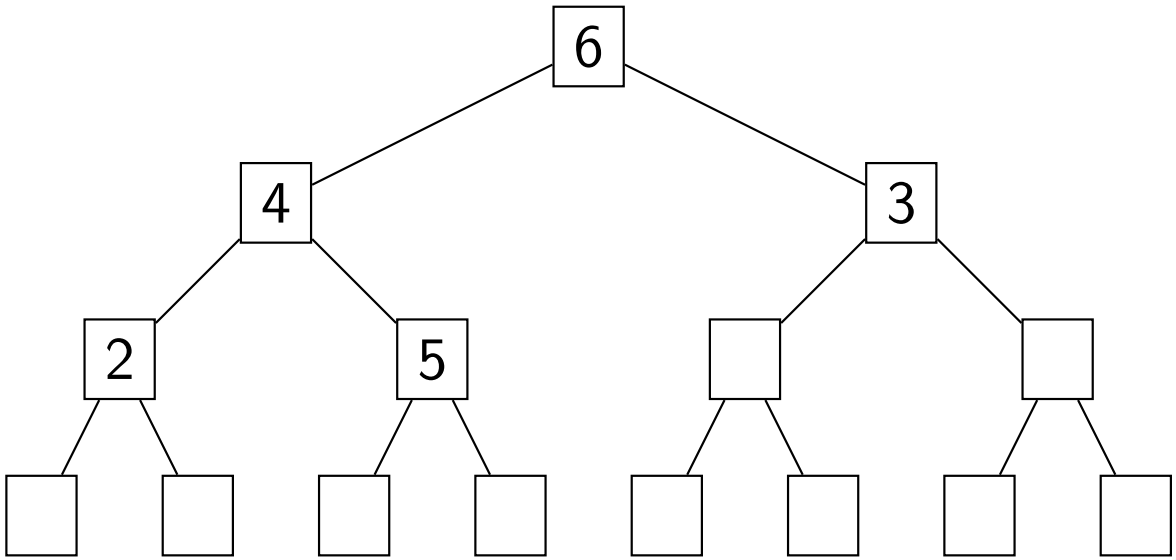


Heap: exemplo

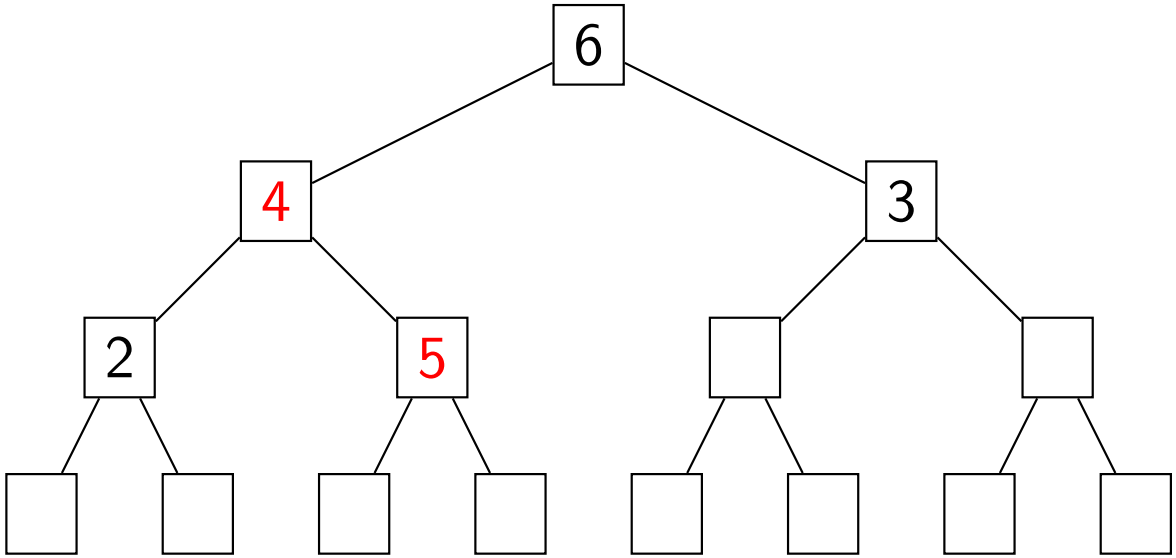


Heap: exemplo

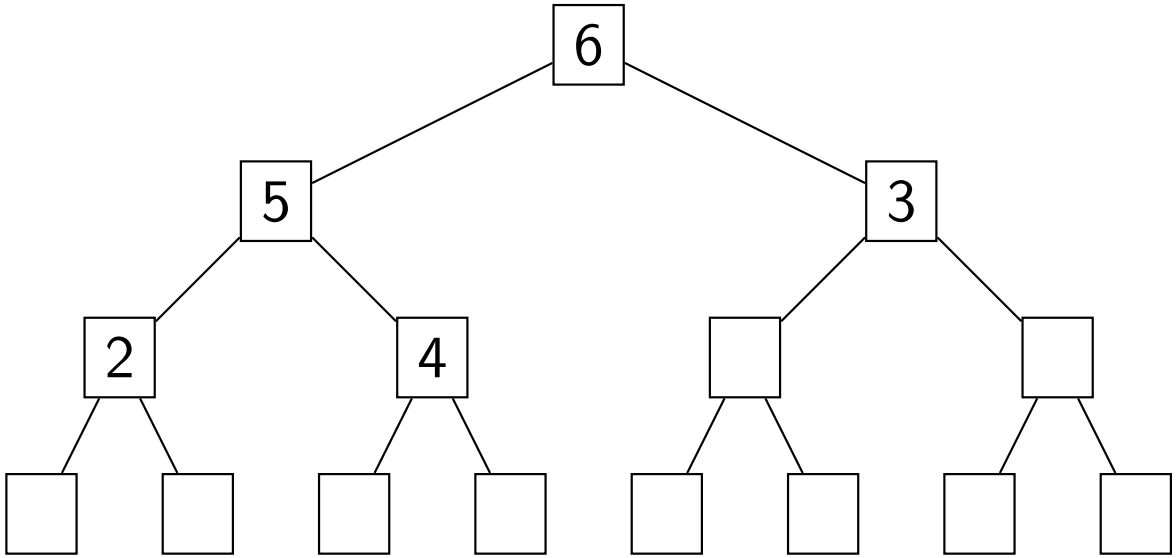
Inserir 5



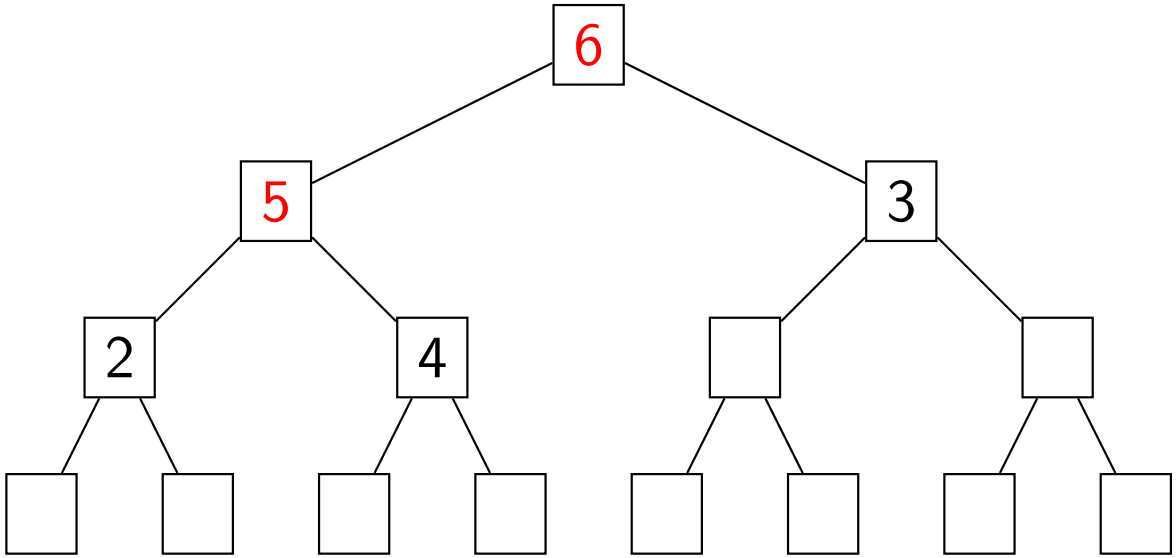
Heap: exemplo



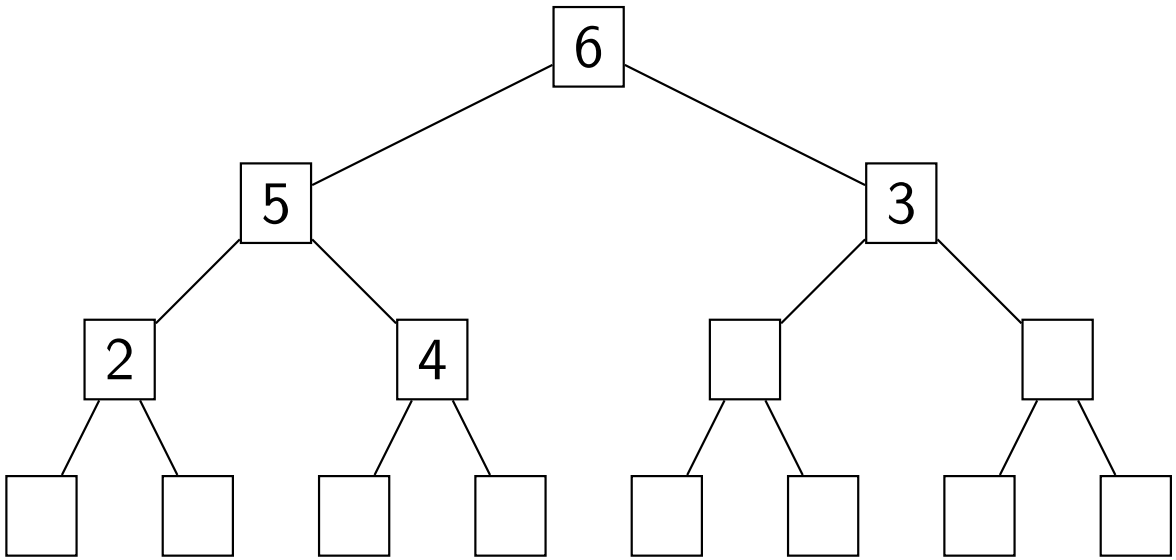
Heap: exemplo



Heap: exemplo

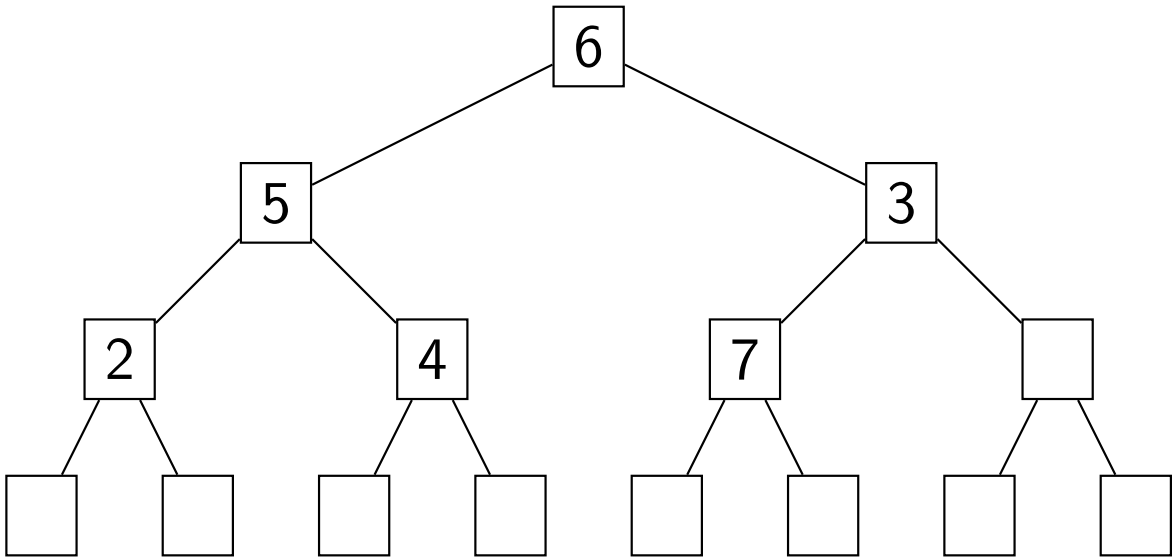


Heap: exemplo

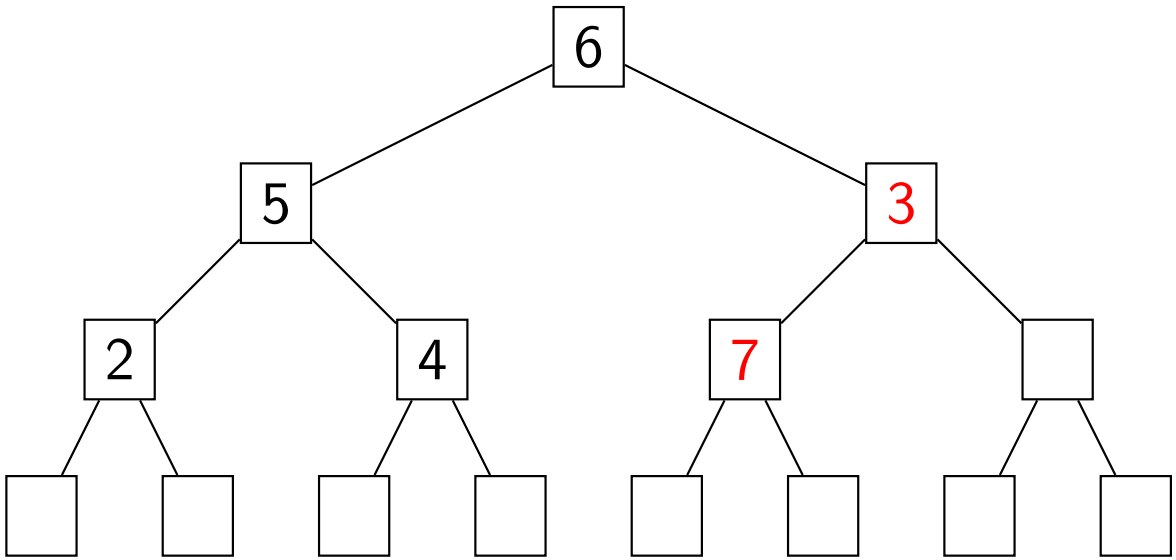


Heap: exemplo

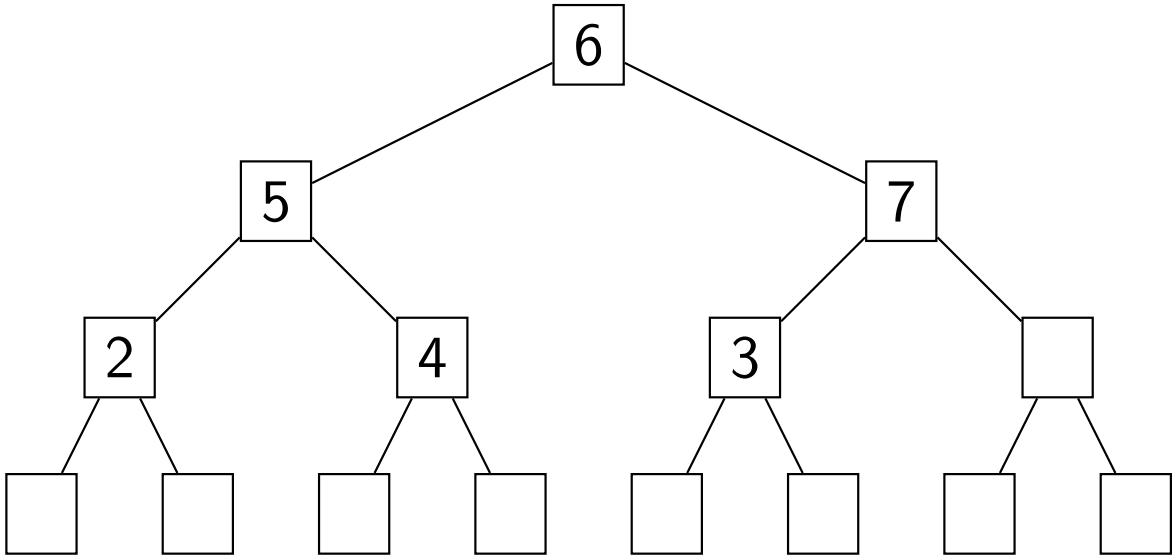
Inserir 7



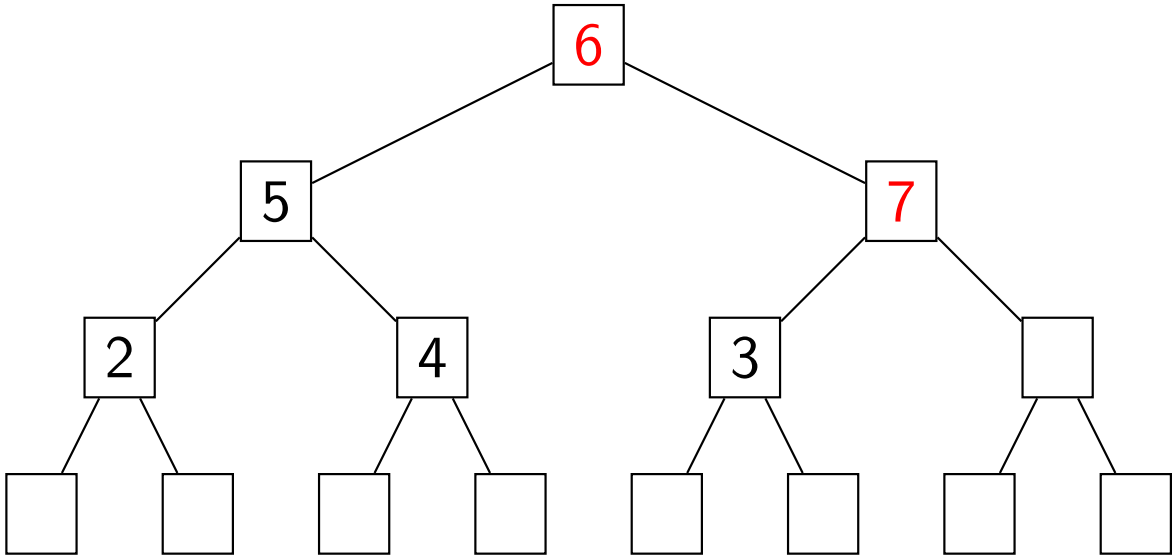
Heap: exemplo



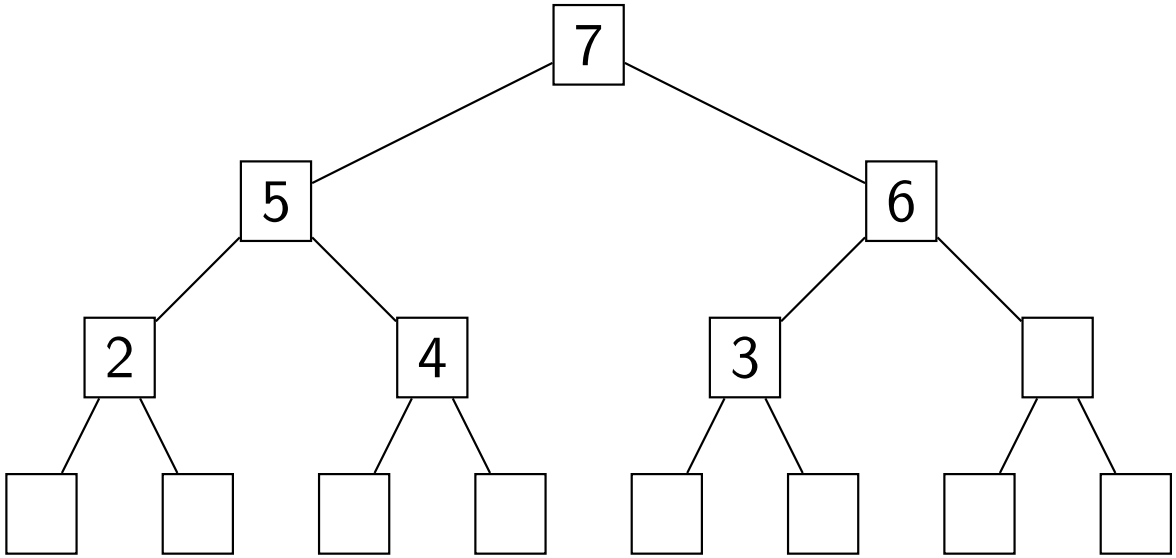
Heap: exemplo



Heap: exemplo



Heap: exemplo



Heap: inserir

Algoritmo: InserirHeap(H, x)

Entrada: heap H, valor x

Resultado: x inserido em H mantendo a propriedade de heap

```
1 se H.pos == H.max então
2 |   retorne "Erro: heap cheio!"
3 i = H.pos
4 H[H.pos] = x;
5 H.pos = H.pos + 1
6 enquanto i > 0 e H[i] > H[Pai(i)] faça
7 |   trocar H[i] com H[Pai(i)]
8 |   i = Pai(i)
```

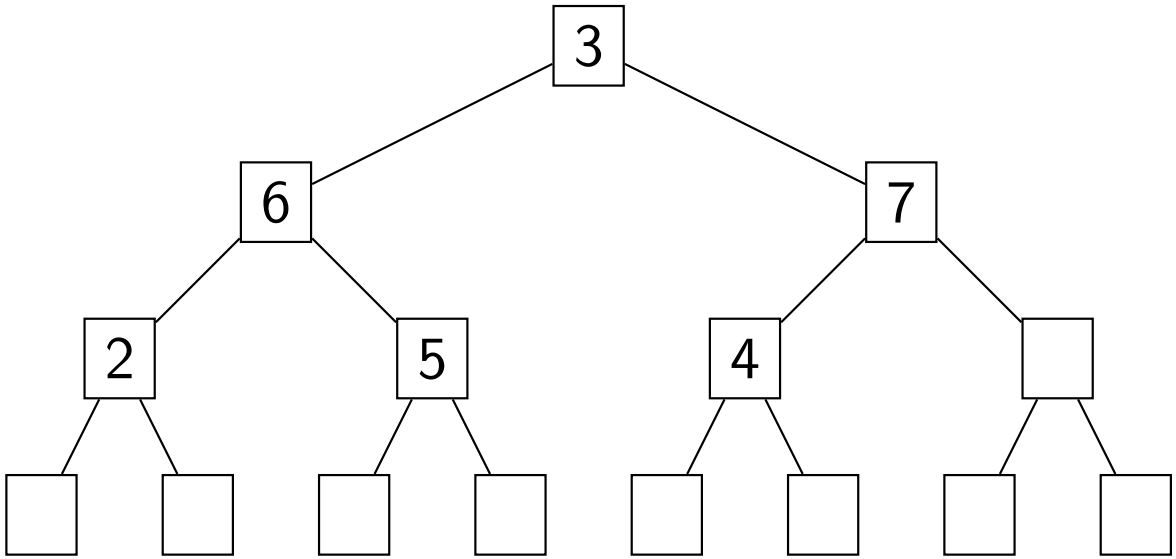
Complexidade: $O(\log n)$

Heap

- E se tivermos um vetor (já preenchido) e quisermos transformá-lo em um heap?
- Uma opção é criar um heap e inserir cada um dos elementos do vetor
  - porém, não é eficiente quanto à memória
- Podemos aproveitar o espaço alocado do vetor, e reposicionar os elementos para torná-lo um heap
  - para um dado elemento, ele deve ser maior que os seus filhos
- Objetivo: colocarmos um certo elemento de uma posição i no seu lugar no heap
- Para um certo elemento, verificamos se algum dos seus filhos é maior do que ele
  - se não, então ele está na posição correta
  - se sim, então devemos trocar com o maior dos filhos
    - além disso, devemos repetir recursivamente para a posição do filho que foi trocado, já que pode haver alterações

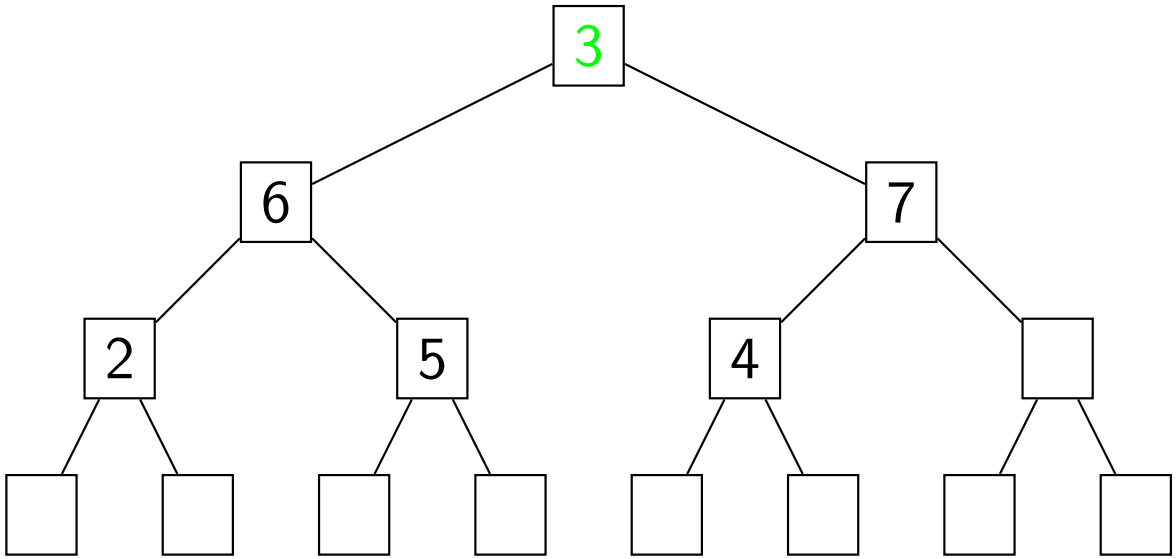
Heap: exemplo

Exemplo: encontrar a posição correta para o valor 3



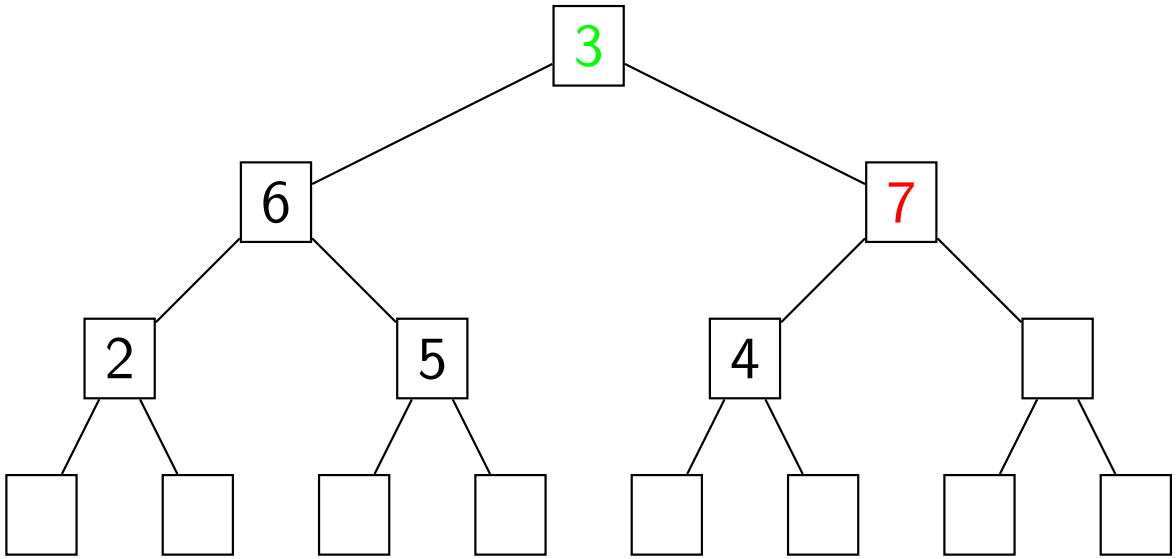
Heap: exemplo

Exemplo: encontrar a posição correta para o valor 3



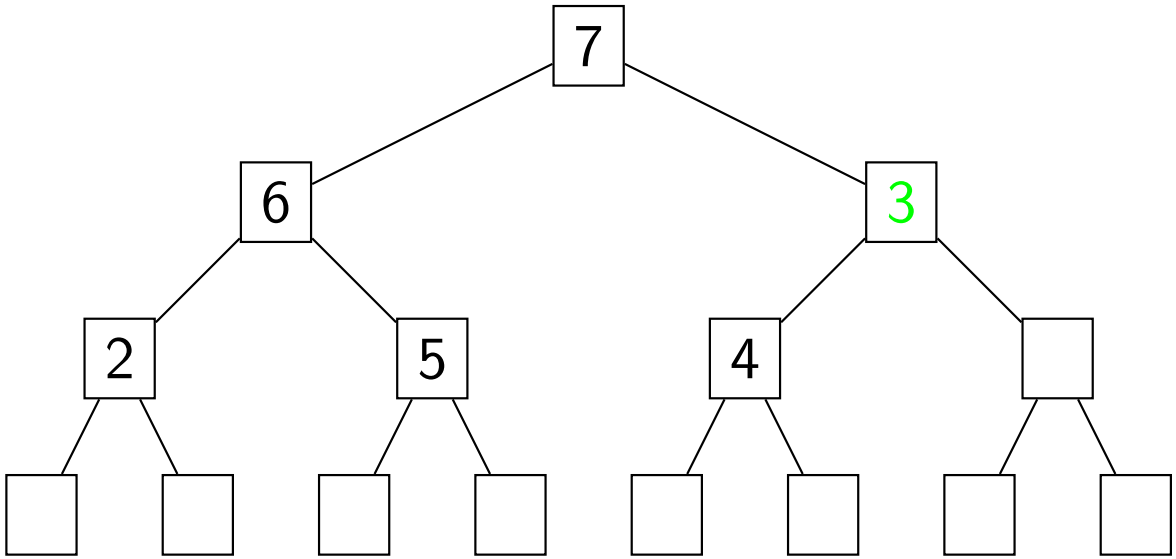
Heap: exemplo

Exemplo: encontrar a posição correta para o valor 3



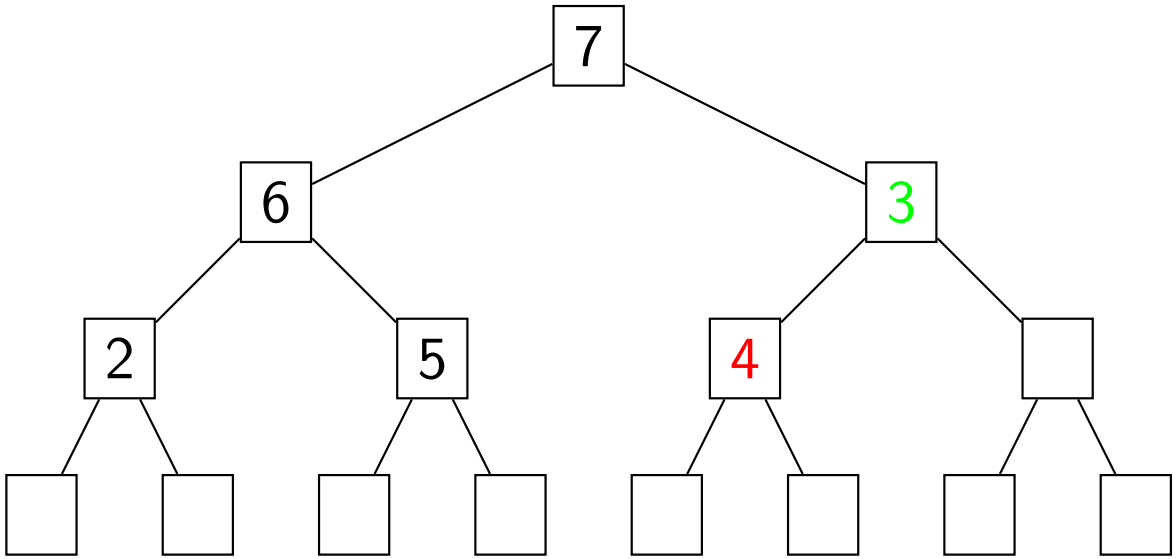
Heap: exemplo

Exemplo: encontrar a posição correta para o valor 3



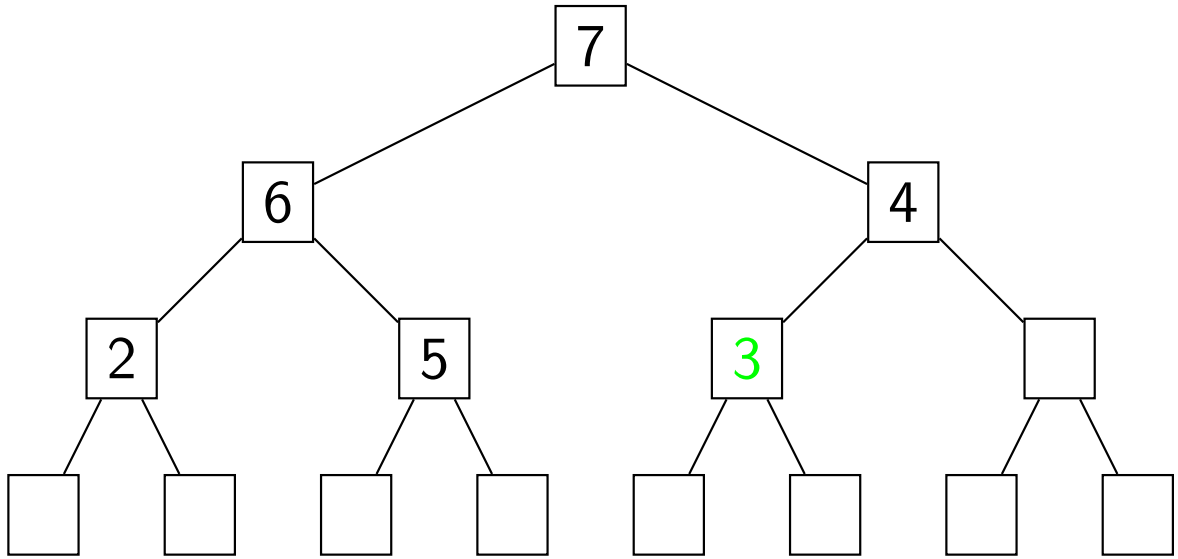
Heap: exemplo

Exemplo: encontrar a posição correta para o valor 3



Heap: exemplo

Exemplo: encontrar a posição correta para o valor 3



Heap: heapify

Algoritmo: Heapify(S, n, i)

Entrada: vetor S, quantidade n de elementos no vetor, índice i

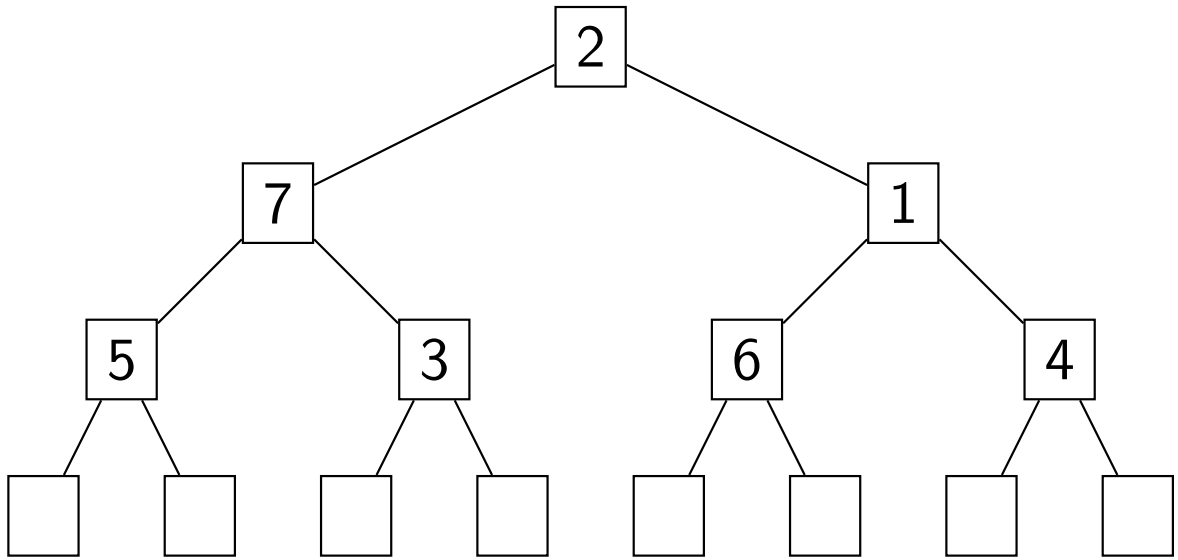
Resultado: "descer" o elemento na posição i até o seu lugar no heap

```
1  e = Esq(i)
2  d = Dir(i)
3  maior = i
4  se e < n e S[e] > S[maior] então
5  |   maior = e
6  se d < n e S[d] > S[maior] então
7  |   maior = d
8  se maior ≠ i então
9  |   trocar S[i] com S[maior]
10 |   Heapify(S, n, maior)
```

Complexidade: $O(\log n)$

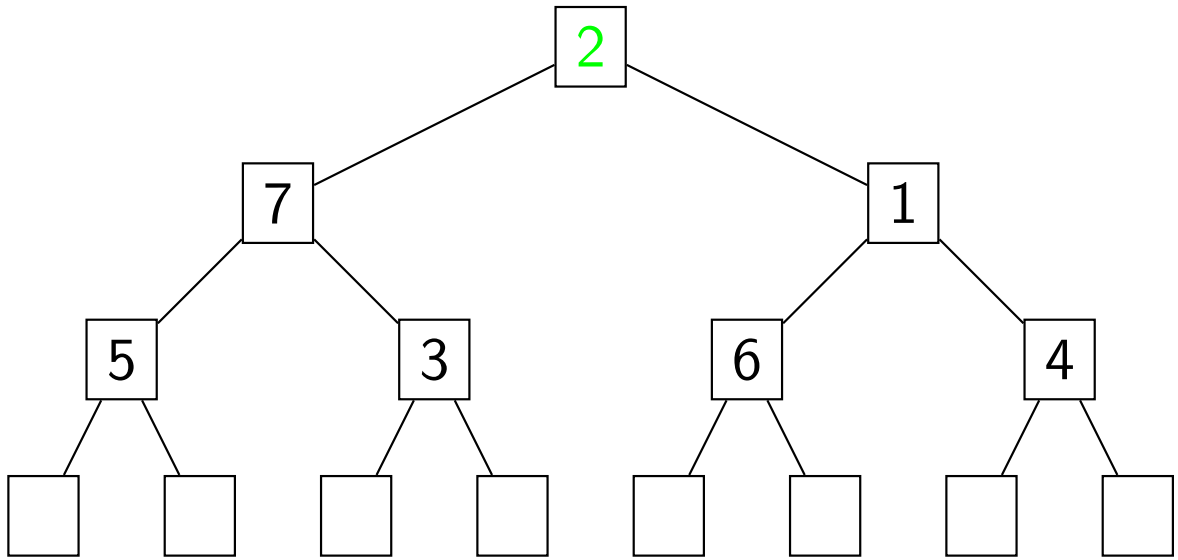
Heap: exemplo

Exemplo 1: encontrar a posição correta para o valor 2 (raiz do heap)



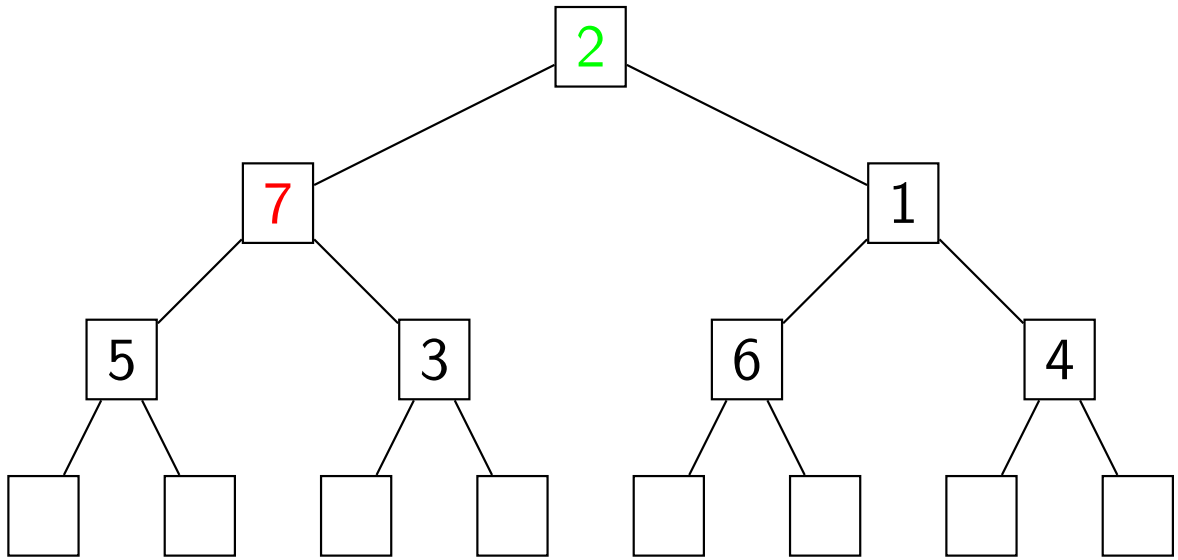
Heap: exemplo

Exemplo 1: encontrar a posição correta para o valor 2 (raiz do heap)



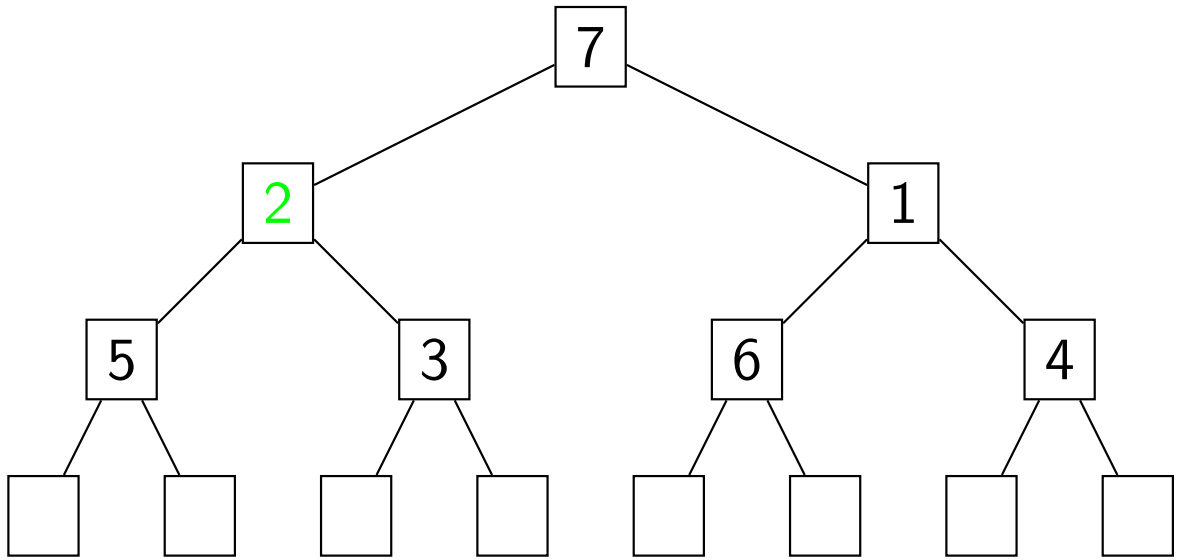
Heap: exemplo

Exemplo 1: encontrar a posição correta para o valor 2 (raiz do heap)



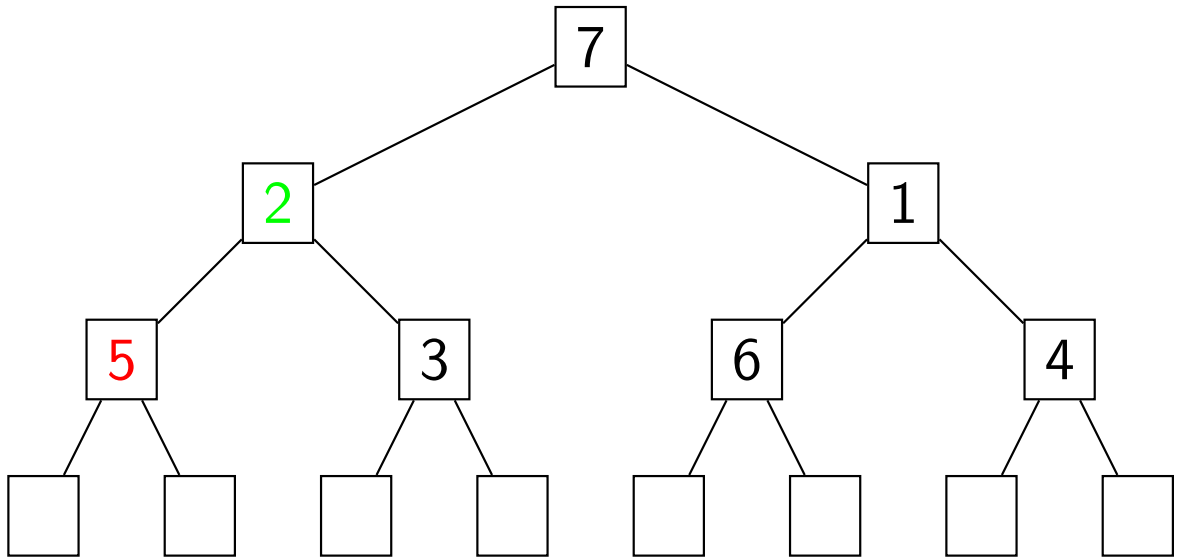
Heap: exemplo

Exemplo 1: encontrar a posição correta para o valor 2 (raiz do heap)



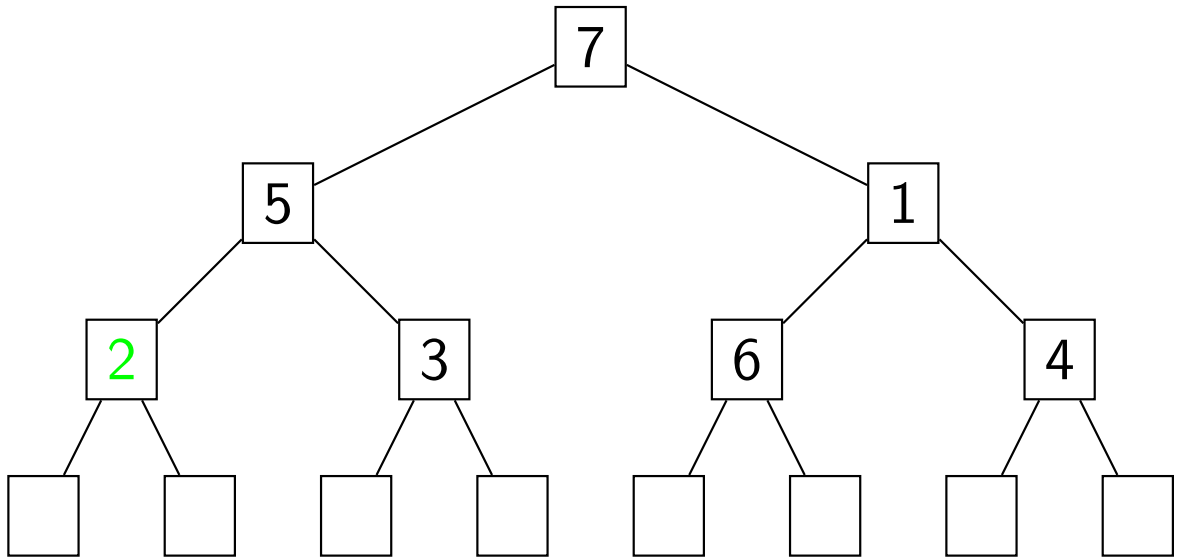
Heap: exemplo

Exemplo 1: encontrar a posição correta para o valor 2 (raiz do heap)

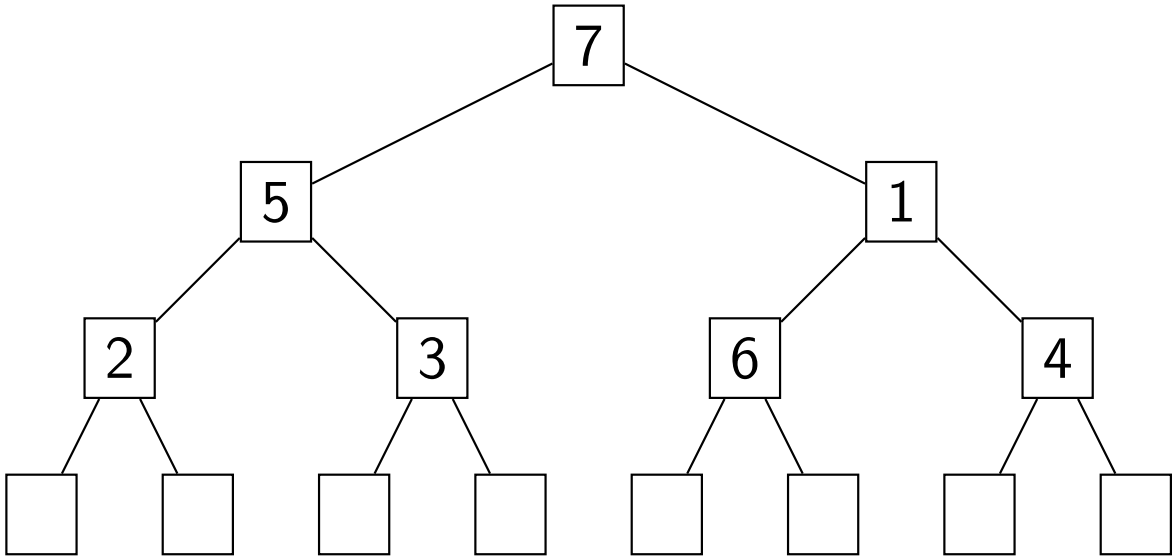


Heap: exemplo

Exemplo 1: encontrar a posição correta para o valor 2 (raiz do heap)



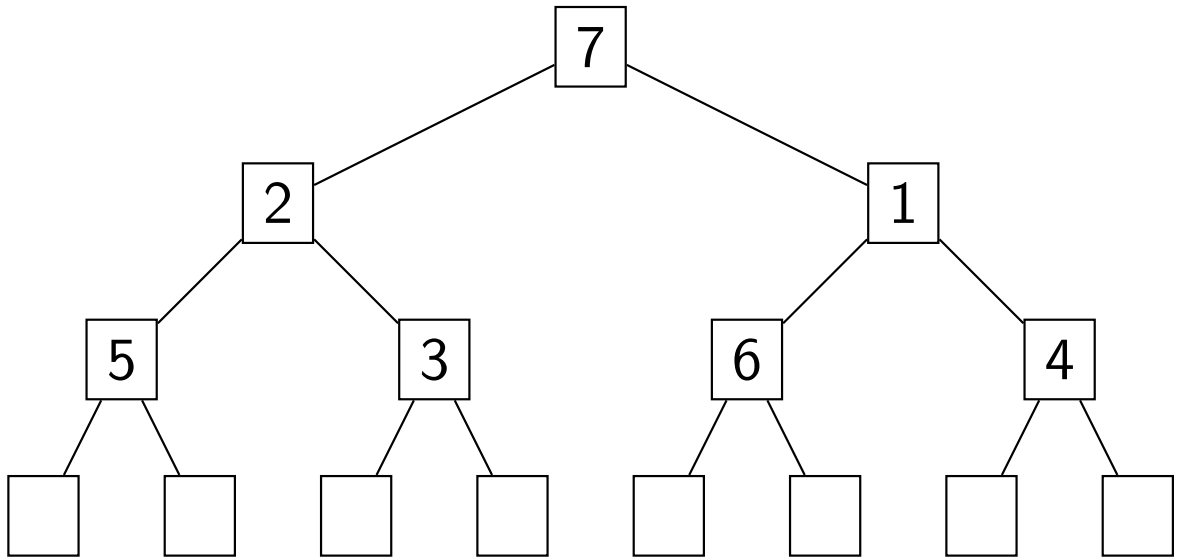
Heap: exemplo



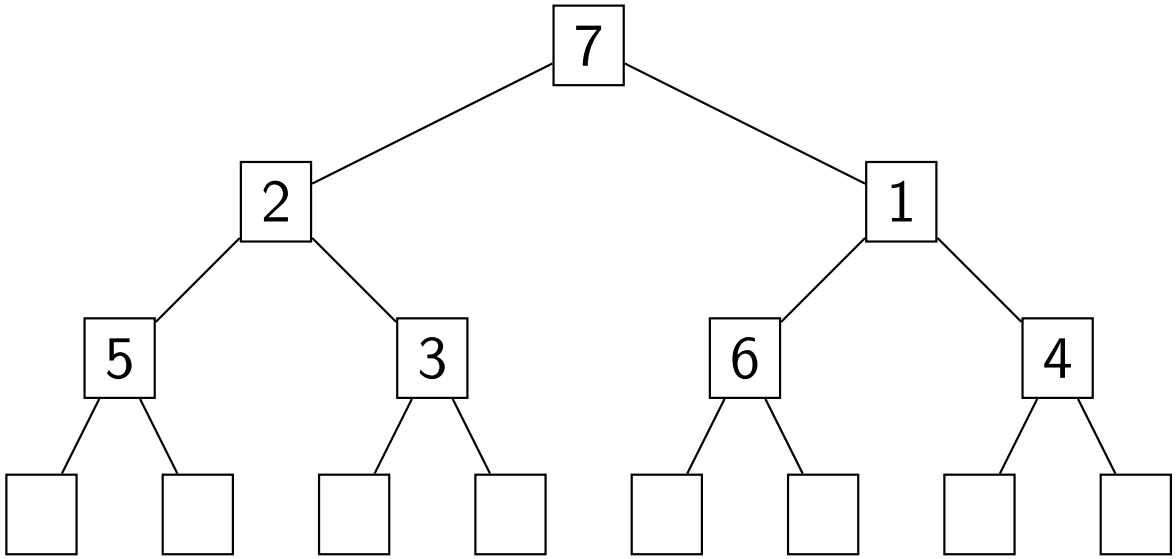
Não é um heap!

Heap: exemplo

Exemplo 2: encontrar a posição correta para o valor 7 (raiz do heap)



Heap: exemplo



Não é um heap!

Heap

- Nota: uma única execução do Heapify sobre a raiz (posição 0) não é o bastante para transformarmos um vetor em um heap
  - há a possibilidade de sequer trocarmos elementos de posição
- Porém, uma aplicação sobre a raiz funciona caso os filhos da raiz sejam heaps!
- Portanto, temos que transformar cada filho possível desse vetor em um heap
- Para isso, podemos usar nossa função Heapify!

Heap

- Obs: podemos ignorar as folhas, já que elas individualmente já são um heap
  - índices das folhas: de $\left\lfloor \frac{H.pos}{2} \right\rfloor$ a $H.pos - 1$
- Portanto, devemos transformar em heap dos índices 0 a $\left\lfloor \frac{H.pos}{2} \right\rfloor - 1$
- Importante: devemos fazer na ordem inversa!
  - começar aplicando Heapify sobre o índice $\left\lfloor \frac{H.pos}{2} \right\rfloor - 1$ e decrescer até chegar a 0
  - ou seja, a raiz é a última a aplicarmos o Heapify
  - justificativa: só depois de aplicarmos Heapify sobre todos os outros, é que temos os dois filhos da raiz como heaps

Heap: construir heap

Algoritmo: BuildHeap(S, n)

Entrada: vetor S, quantidade n de elementos no vetor

Resultado: reposicionar elementos de S formando um heap

1 **para** $i = \left\lfloor \frac{n}{2} \right\rfloor - 1$ **até** 0 **faça**

2 | Heapify(S, n, i)

Complexidade: $O(n \log n)$

Heap

- A partir destes métodos, podemos desenvolver um método de ordenação!
- Considere que temos um vetor arbitrário com n elementos
- Como fazer?
  1. construímos um Max Heap
  2. o maior elemento do heap estará na posição 0
  3. trocamos os valores da posição 0 e $n-1$ (última posição do vetor)
  4. decrementamos o n
     - ou seja, não trabalharemos mais com a posição $n-1$
  5. repetimos os passos 1 a 4 até que só haja um elemento
     - quando $n = 1$ há só um elemento

Heap: Heap Sort

Algoritmo: HeapSort(S, n)

Entrada: vetor S, quantidade n de elementos no vetor

Resultado: ordenar os elementos de S em ordem crescente

```
1 BuildHeap(S, n)
2 para i = n − 1 até 1 faça
3 |   trocar S[0] com S[i]
4 |   BuildHeap(S, i)
```

Heap

- Problema: complexidade
  - uma execução de BuildHeap no início do método
  - $n - 1$ execuções de BuildHeap no loop do método
  - complexidade do BuildHeap: $O(n \log n)$
  - complexidade do método: $O(n^2 \log n)$
- Esperávamos um $O(n \log n)$
- Como fazer melhor?

Heap

- Solução: em vez de usar BuildHeap várias vezes, utilizamos Heapify!
- $\textsc{BuildHeap}(S, n)$:
  - Transformar um vetor em um heap: $O(n \log n)$
- $\textsc{Heapify}(S, n, i)$
  - "Descer" um elemento até o seu lugar no heap: $O(\log n)$
- Por que isto funciona?
  - primeiro, construímos um heap
  - após construirmos, temos certeza de que cada sub-árvore é um heap
  - se trocamos os elementos das posições 0 e $n - 1$, temos a certeza de que a nova raiz (o novo elemento na posição 0) está fora de posição
  - além disso, as duas sub-árvores desta raiz são heaps!
  - portanto, em vez de construirmos um novo heap, basta procurarmos a posição correta para a nova raiz!

Heap

- Solução: em vez de usar BuildHeap várias vezes, utilizamos Heapify!
- A partir disto, podemos desenvolver um método de ordenação de complexidade $O(n \log n)$!
- Considere que temos um vetor arbitrário com n elementos
- Como fazer?
  1. construímos um Max Heap
  2. o maior elemento do heap estará na posição 0
  3. trocamos os valores da posição 0 e $n-1$ (última posição do vetor)
  4. decrescemos o n
     - ou seja, não trabalharemos mais com a posição $n-1$
  5. executamos Heapify sobre a posição 0 para descermos o elemento até o seu lugar no heap
  6. repetimos os passos 2 a 5 até que só haja um elemento
     - quando $n = 1$ há só um elemento

Heap: Heap Sort

Algoritmo: HeapSort(S, n)

Entrada: vetor S, quantidade n de elementos no vetor

Resultado: ordenar os elementos de S em ordem crescente

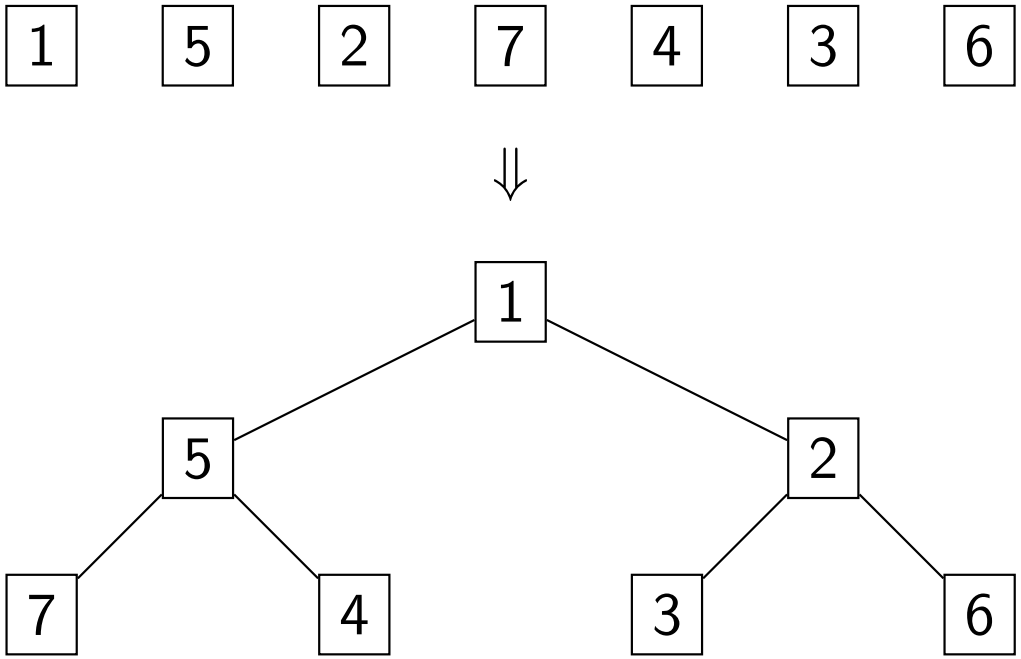
```
1 BuildHeap(S, n)
2 para i = n − 1 até 1 faça
3 |   trocar S[0] com S[i]
4 |   Heapify(S, i, 0)
```

Heap Sort

- Vantagens:
  - complexidade $O(n \log n)$ no pior caso
  - pouco uso de memória auxiliar
    - não usa vetores auxiliares, só uma quantidade constante
- Desvantagens:
  - não é estável
    - há como fazer uma versão estável, mas é complicado
  - implementação um pouco complicada
    - o laço do algoritmo é complexo
  - caso trabalhemos com poucos elementos, este algoritmo é complicado demais para ter um ganho muito pequeno

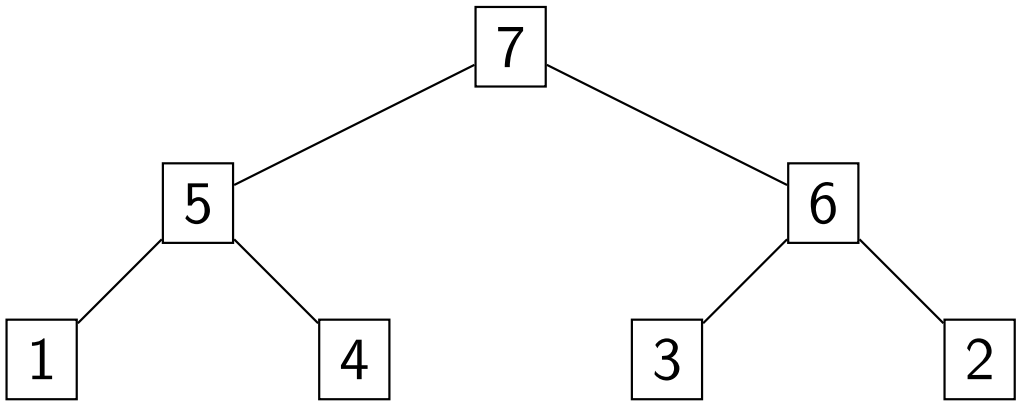
Heap Sort: exemplo

Ordenar o vetor $1, 5, 2, 7, 4, 3, 6$ ($n = 7$ elementos)



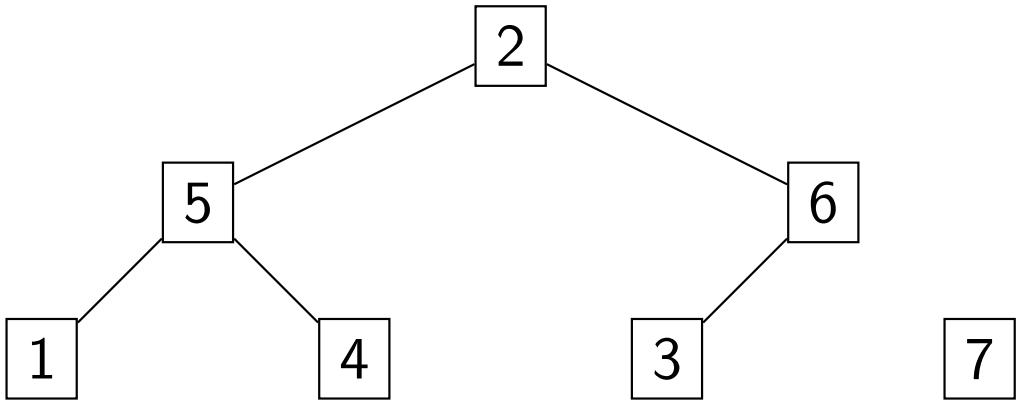
Heap Sort: exemplo

Transformar em um heap (aplicando BuildHeap)
$n = 7$



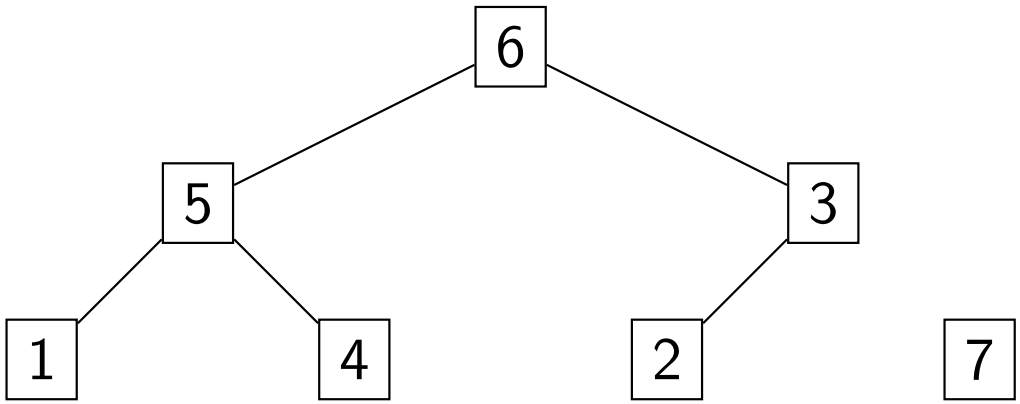
Heap Sort: exemplo

Trocar $S[0]$ com $S[n-1]$ e decrementar n
$n = 7 - 1 = 6$



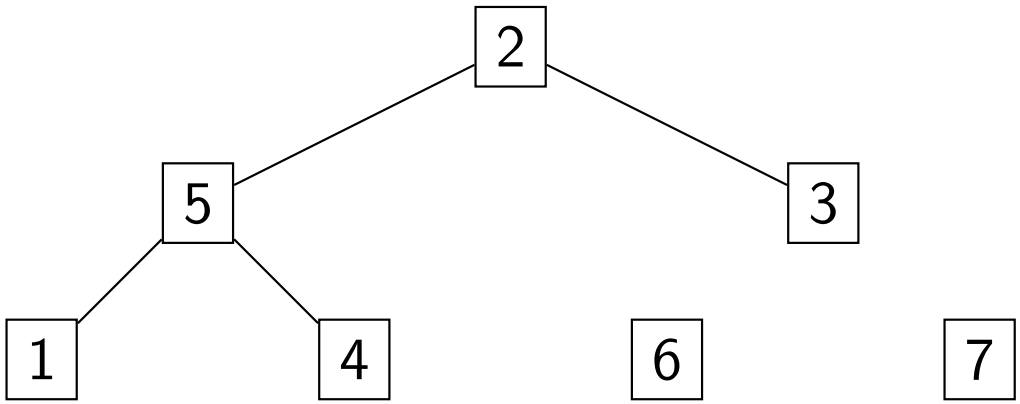
Heap Sort: exemplo

Encontrar a posição correta para a nova raiz (aplicando Heapify)
$n = 6$



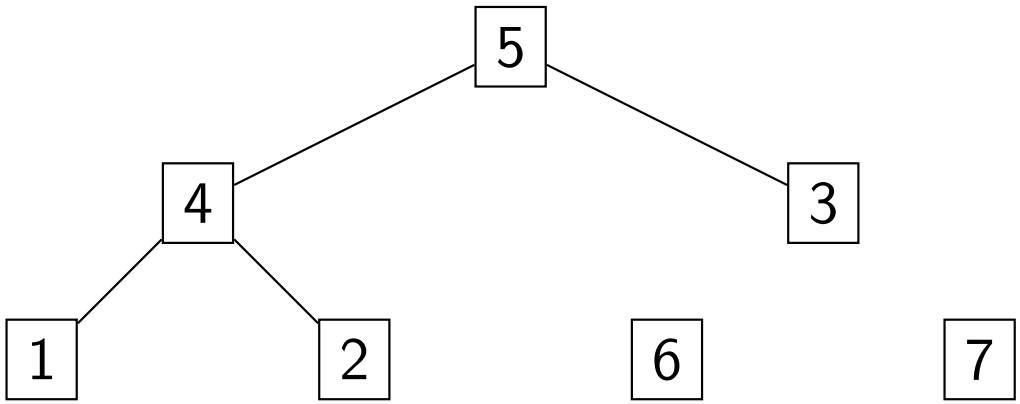
Heap Sort: exemplo

Trocar $S[0]$ com $S[n-1]$ e decrementar n
$n = 5$



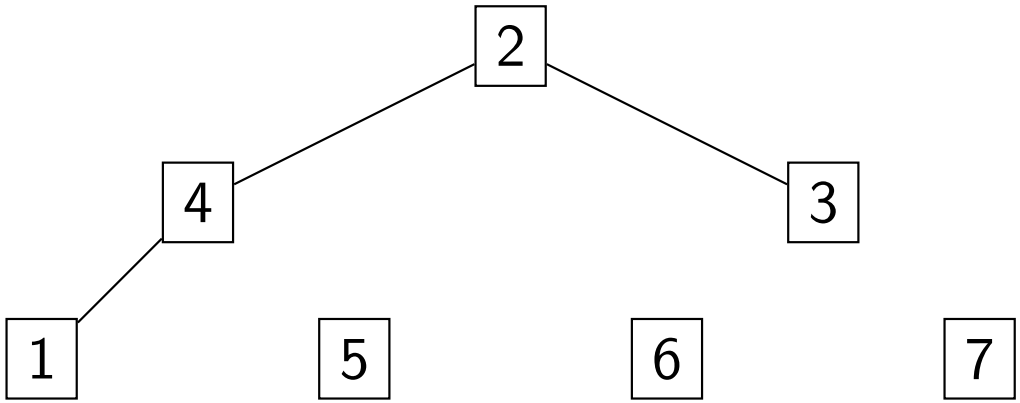
Heap Sort: exemplo

Encontrar a posição correta para a nova raiz (aplicando Heapify)
$n = 5$



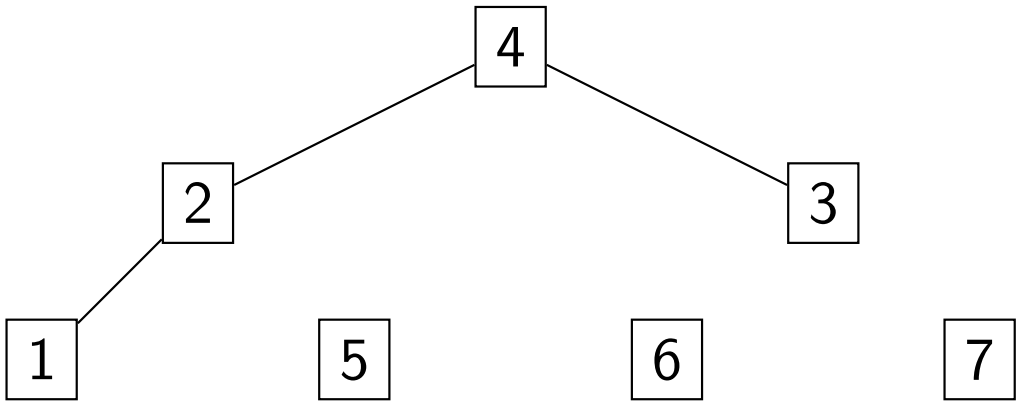
Heap Sort: exemplo

Trocar $S[0]$ com $S[n-1]$ e decrementar n
$n = 5 - 1 = 4$



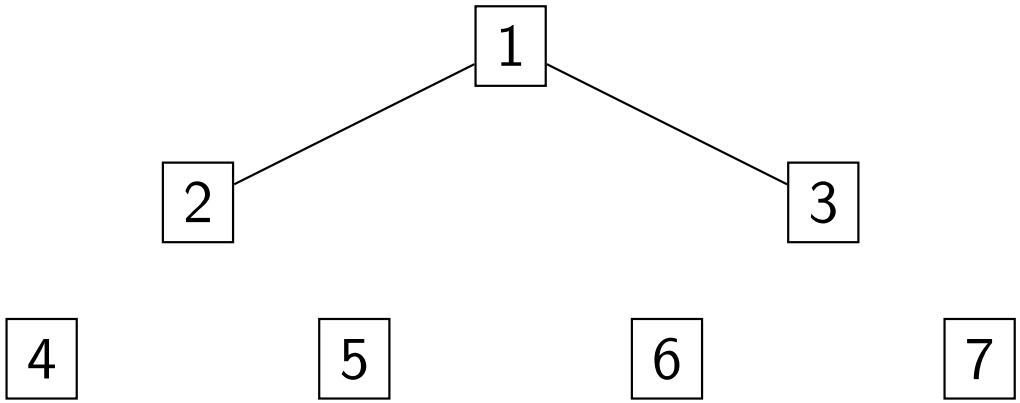
Heap Sort: exemplo

Encontrar a posição correta para a nova raiz (aplicando Heapify)
$n = 4$



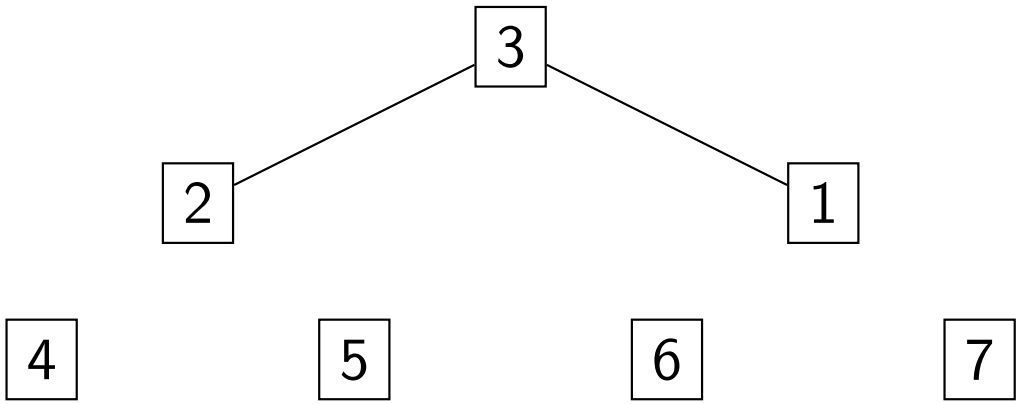
Heap Sort: exemplo

Trocar $S[0]$ com $S[n-1]$ e decrementar n
$n = 4 - 1 = 3$



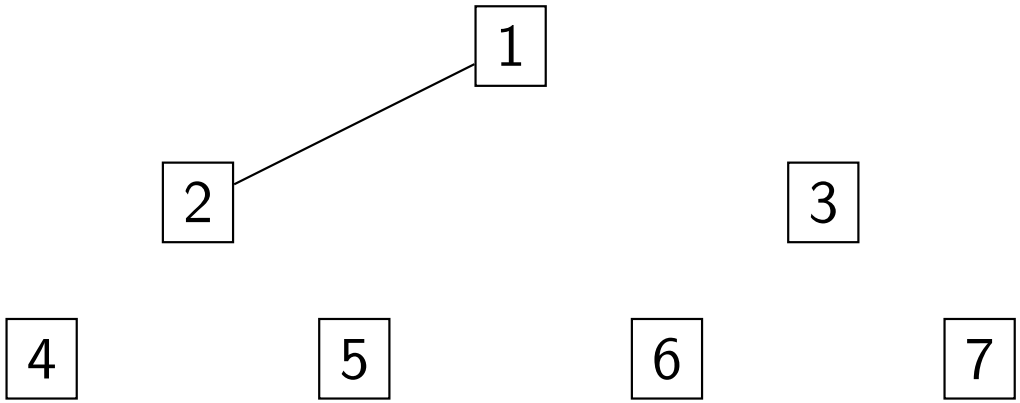
Heap Sort: exemplo

Encontrar a posição correta para a nova raiz (aplicando Heapify)
$n = 3$



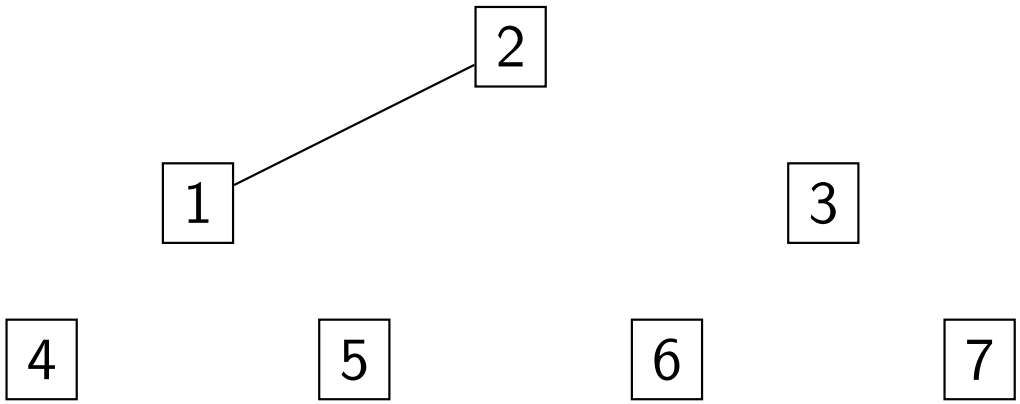
Heap Sort: exemplo

Trocar $S[0]$ com $S[n-1]$ e decrementar n
$n = 3 - 1 = 2$



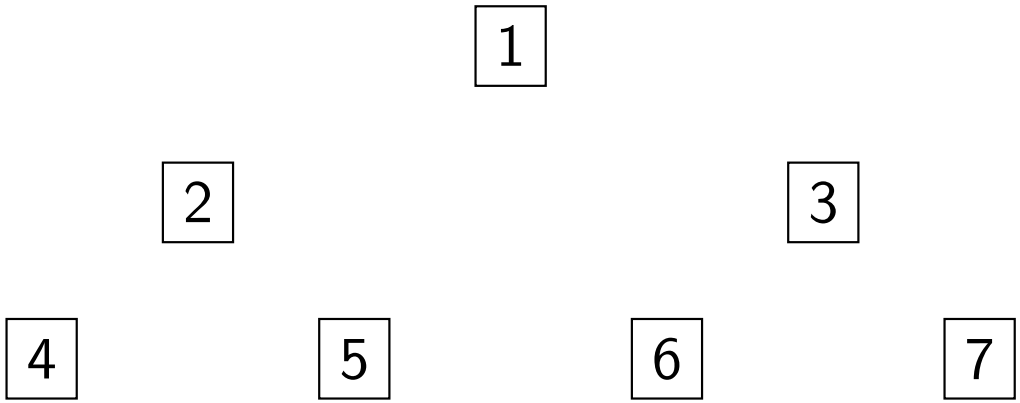
Heap Sort: exemplo

Encontrar a posição correta para a nova raiz (aplicando Heapify)
$n = 2$



Heap Sort: exemplo

Trocar $S[0]$ com $S[n-1]$ e decrementar n
$n = 2 - 1 = 1$



Heap Sort: exemplo

Condição de parada atingida: $n = 1$
Vetor ordenado: