Heaps e Árvores Binárias

Eduardo Camponogara

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina

6 de abril de 2009

Sumário

Introdução

Heap

Heap Sort é algoritmo de ordenação

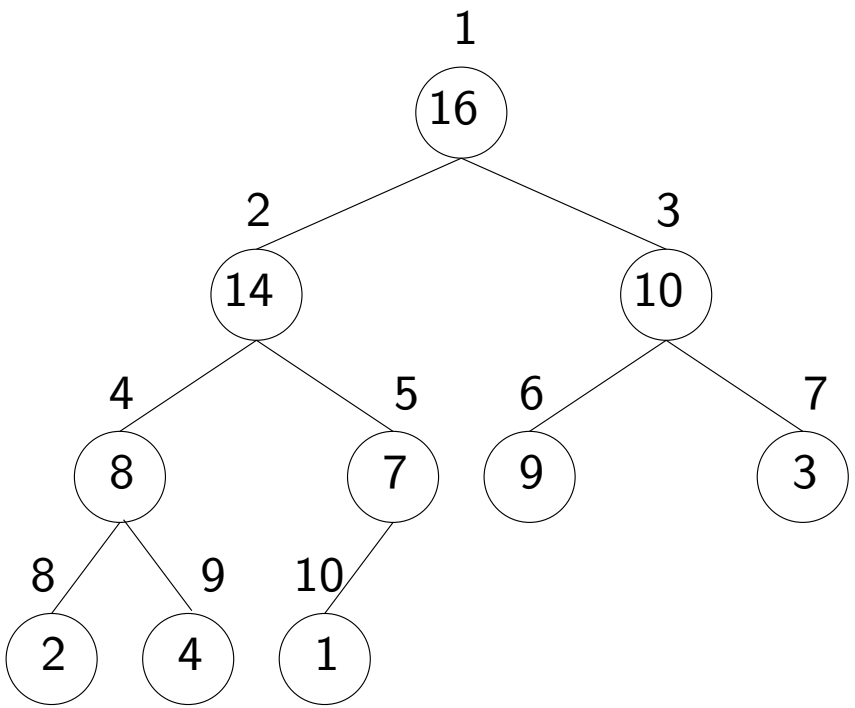
- Como o *Merge Sort*, *Heap Sort* ordena n números em tempo $\Theta(n \lg n)$
- Ordena no local, necessitando apenas um número constante de variáveis de armazenamento.

A estrutura de dados *heap*, além de permitir a implementação do algoritmo *Heap Sort*, também pode ser utilizada como uma fila de prioridades eficiente.

Sumário

Heaps

Um *heap* (binário) é uma estrutura de dados que pode ser vista como uma árvore binária completa.



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heaps

Heaps

- Cada nó da árvore corresponde a um elemento do vetor que amazena o valor do nó.
- A árvore é completa em todos os níveis com excessão do nível mais baixo, o qual é completado da esquerda para a direita.

Heaps

Representação de Um *Heap*

Um vetor A que armazena um *heap* é um objeto com dois atributos:

- $length[A]$: número de elementos do vetor
- $heap\_size[A]$: número de elementos do *heap* armazenados em A.

Heaps

Representação de Um *Heap*

- A raiz da árvore é $A[1]$.
- Dado o índice i de um vértice, podemos calcular os seguintes índices.

Parent(i)
    return $\lfloor i/2 \rfloor$

Left_Child(i)
    return $2i$

Right_Child(i)
    return $2i + 1$

A Propriedade *Heap*

Propriedade *Heap*

Para que um vetor A seja *heap*, cada nó i, com excessão da raiz, deve satisfazer a seguinte propriedade:

$$A[i] \leq A[\text{Parent}(i)]$$

- O valor de cada nó é no máximo o valor do seu pai
- O maior elemento do *heap* está na raiz
- As subárvores a partir de qualquer nó possuem valores menores ou iguais ao valor do nó

A Propriedade *Heap*

Altura da Árvore

- **altura de um nó** da árvore é o número de arestas no caminho mais longo do nó até uma folha da árvore.
- **altura da árvore** é a altura do nó raiz.
- Uma vez que um *heap* induz uma árvore binária completa, sua altura é $\Theta(\lg n)$.

A Propriedade *Heap*

Mantendo a Propriedade *Heap*

- O procedimento Heapify é fundamental na manipulação de heaps.
- A entrada é o vetor A e um índice do vetor.
- Quando é chamado, assume-se que as árvores com raiz em Left_Child(i) e Right_Child(i) são *heaps*, todavia o elemento $A[i]$ pode ser menor do que seus filhos, desta forma violando a propriedade *heap*
- O procedimento *Heapify* recupera a propriedade heap da árvore com raiz em i.

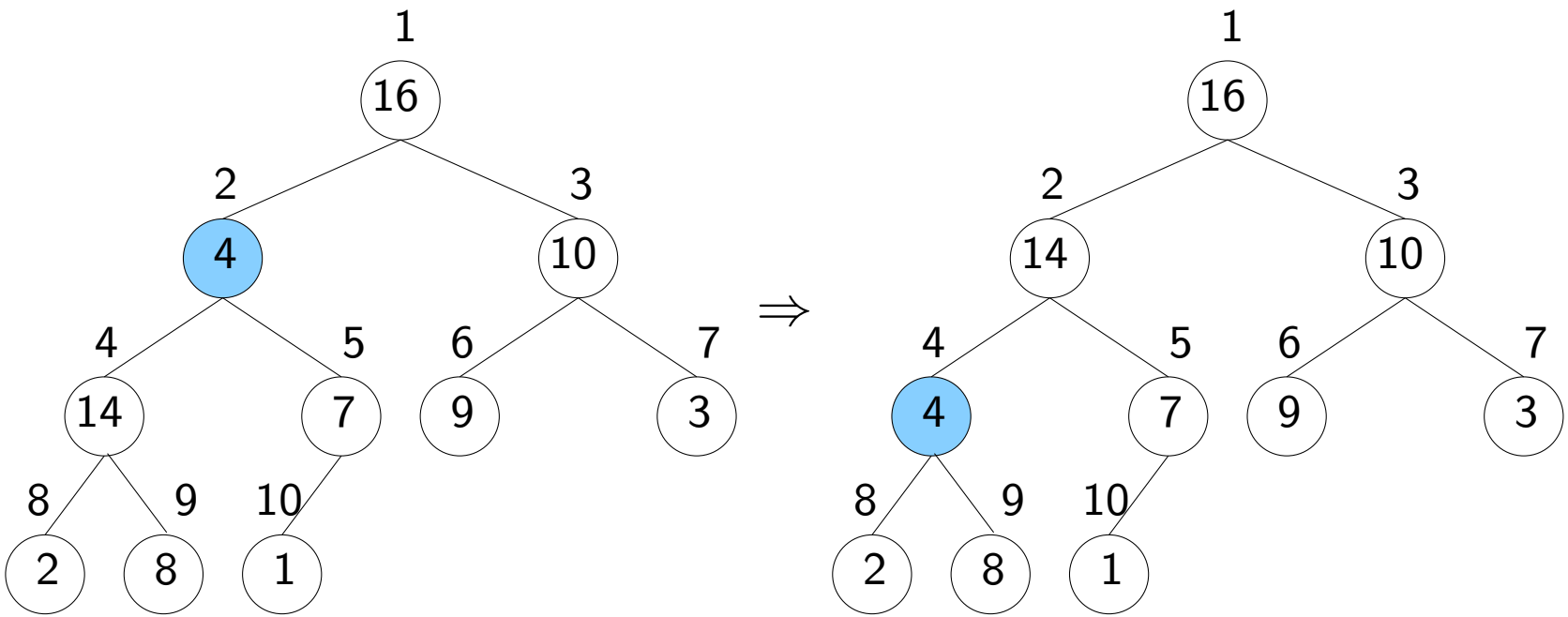
A Propriedade *Heap*

Heapify

Heapify(A, i)
    $l \leftarrow$ Left_Child(i)
    $r \leftarrow$ Right_Child(i)
    if $l \leq heap\_size[A]$ & $A[l] > A[i]$
        $largest \leftarrow l$
    else
        $largest \leftarrow i$
    if $r \leq heap\_size[A]$ & $A[r] > A[largest]$
        $largest \leftarrow r$
    if $largest \neq i$
        exchange $A[i] \leftrightarrow A[largest]$
        Heapify(A, $largest$)

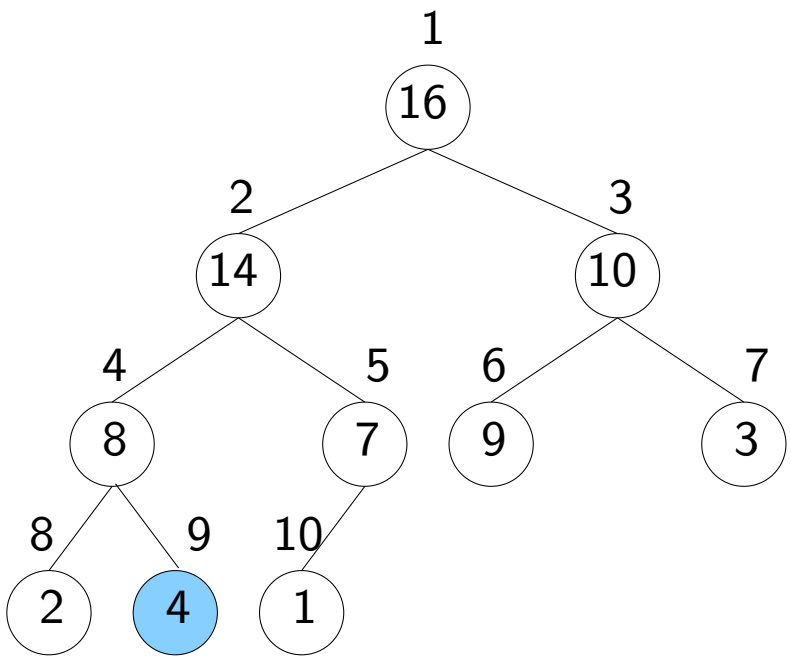
A propriedade heap

Exemplo



A propriedade heap

Exemplo



O tempo de execução da operação *Heapify* é precisamente a altura da árvore enraizada no nó i. Portanto $O(h) = O(\lg n)$

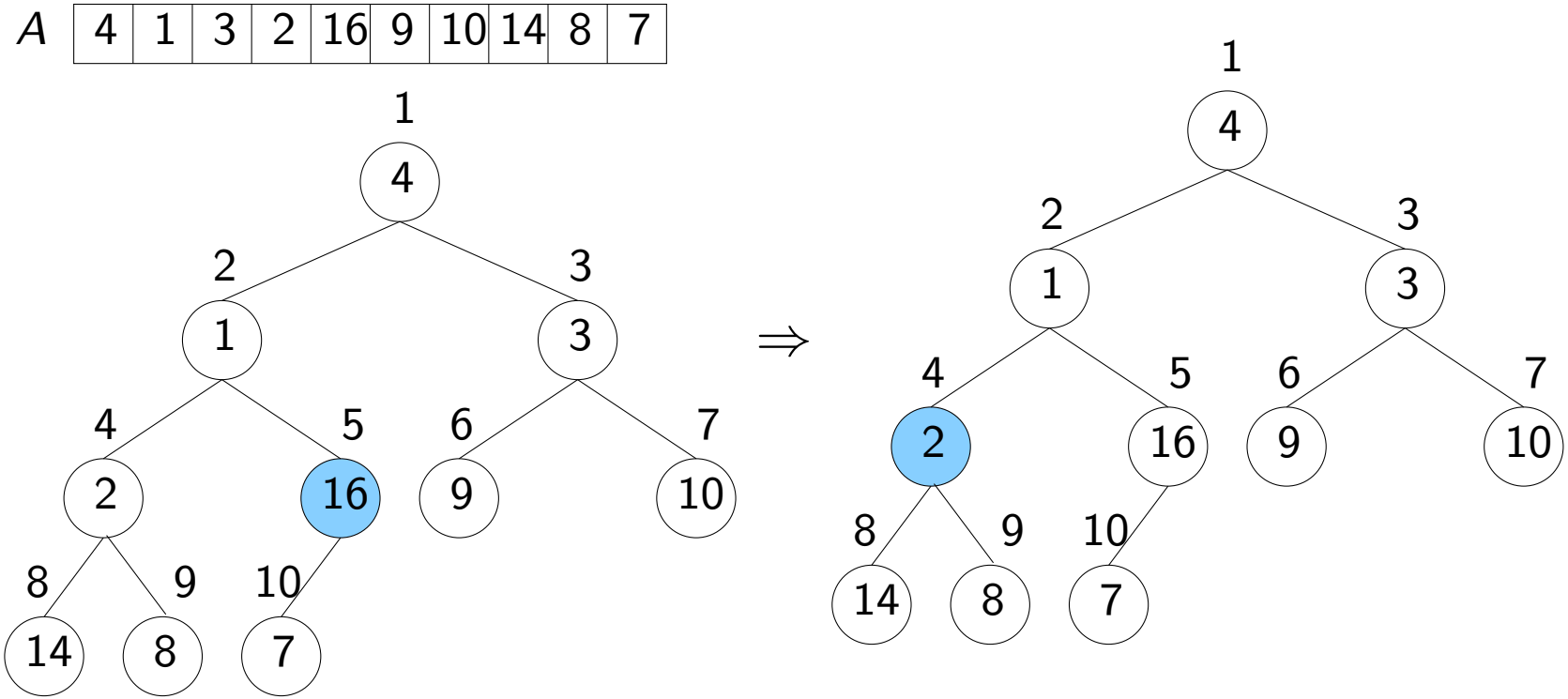
Construindo Um *Heap*

- Podemos utilizar o procedimento Heapify para converter um vetor $A[1, \ldots, n]$, com $n = length[A]$ em um *heap*.
- Como os elementos do subvetor $A[(\lfloor n/2 \rfloor + 1), \ldots, n]$ são folhas da árvore, podemos iniciar o procedimento no nó $\lfloor n/2 \rfloor$
- Cada Heapify executa em $O(\lg n)$, portanto, o Build_Heap executa em tempo $O(n \lg n)$.

Build_Heap(A)
    $heap\_size[A] \leftarrow length[A]$
    for $j \leftarrow \lfloor heap\_size[A]/2 \rfloor$ downto 1
        $Heapify(A, j)$

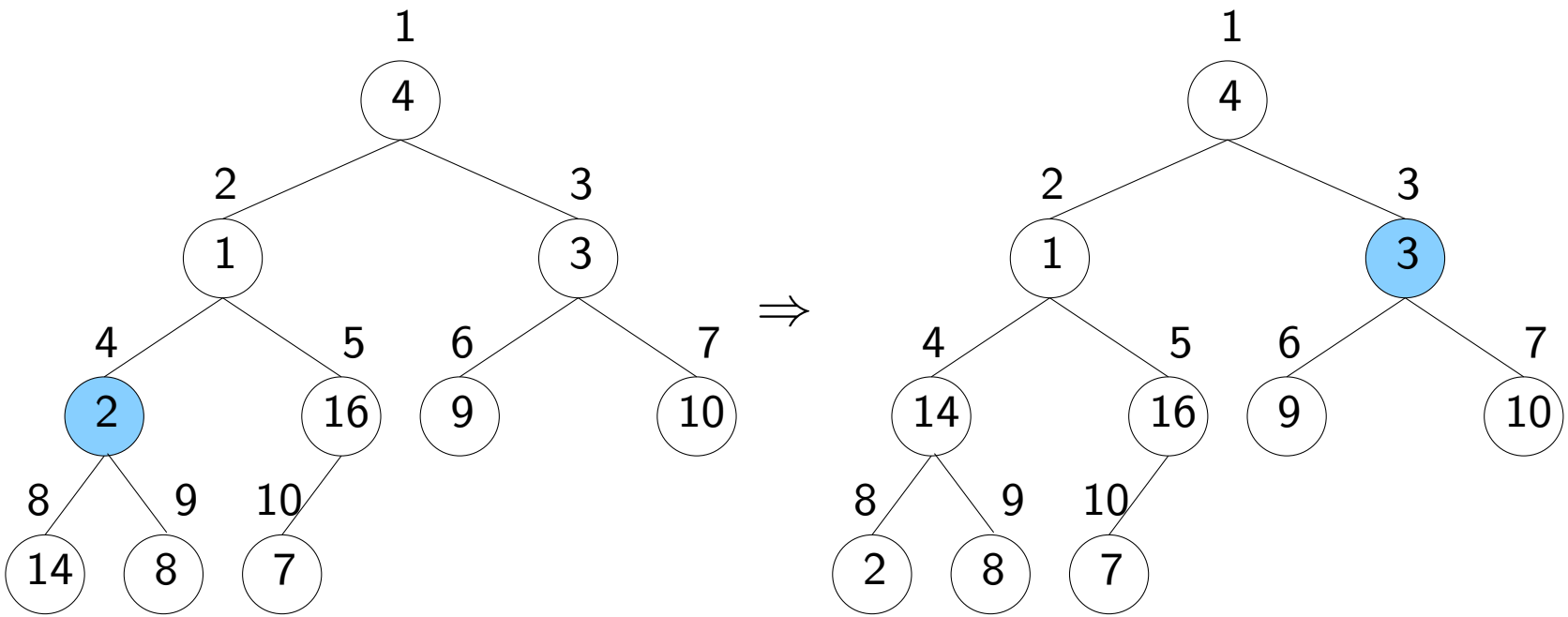
Construindo Um *Heap*

Exemplo



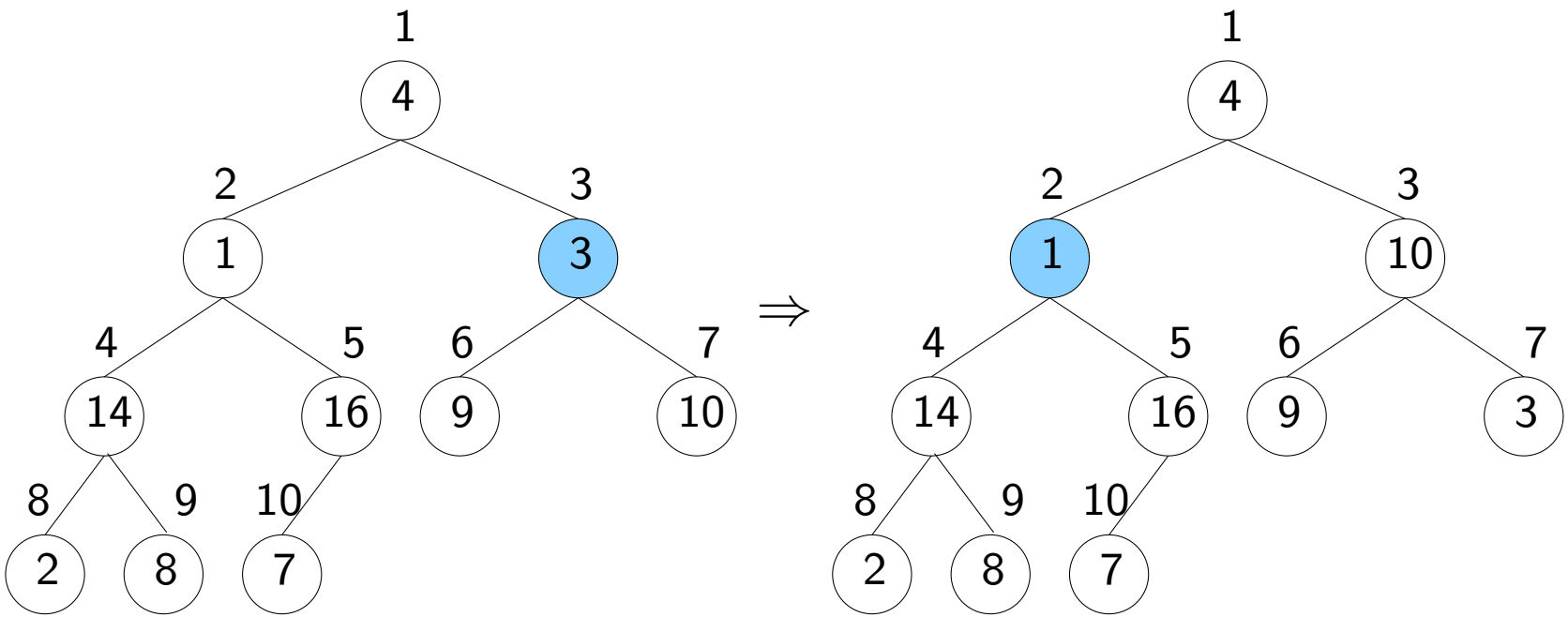
Construindo Um *Heap*

Exemplo



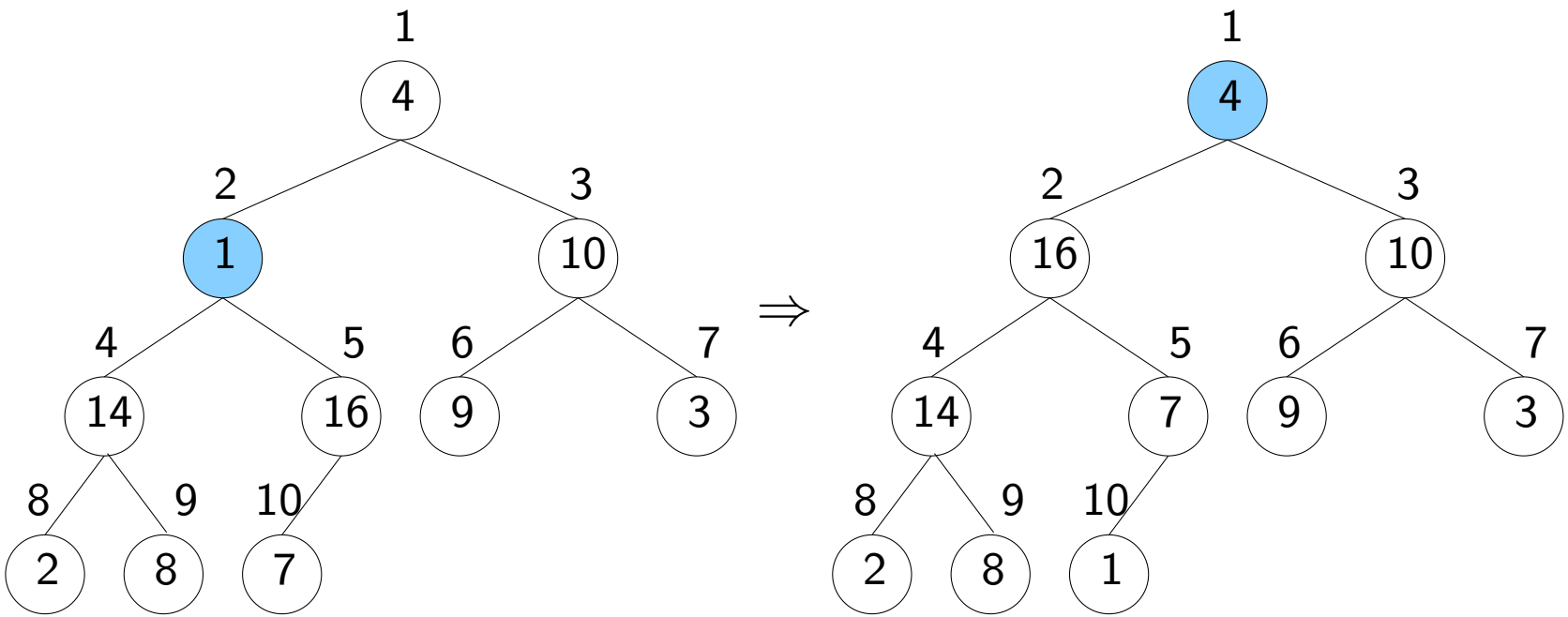
Construindo Um *Heap*

Exemplo



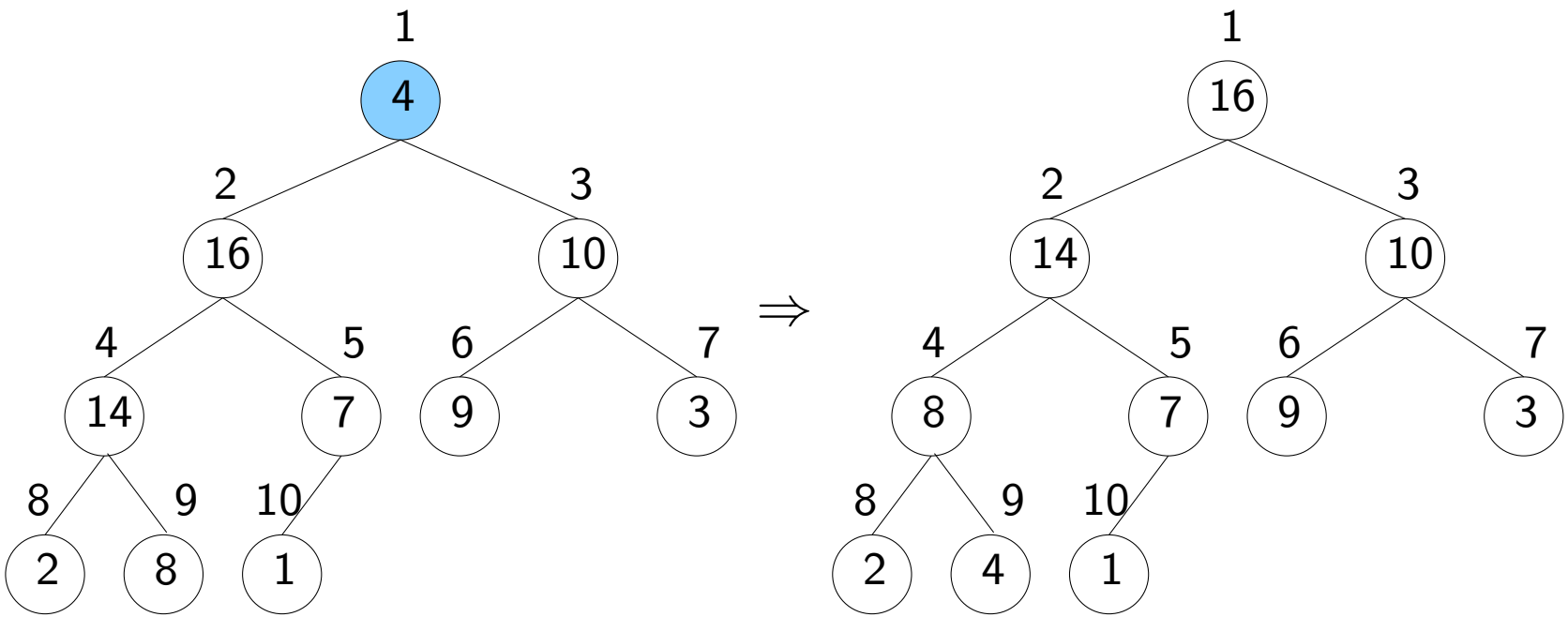
Construindo Um *Heap*

Exemplo



Construindo Um *Heap*

Exemplo



Algoritmo de Ordenação

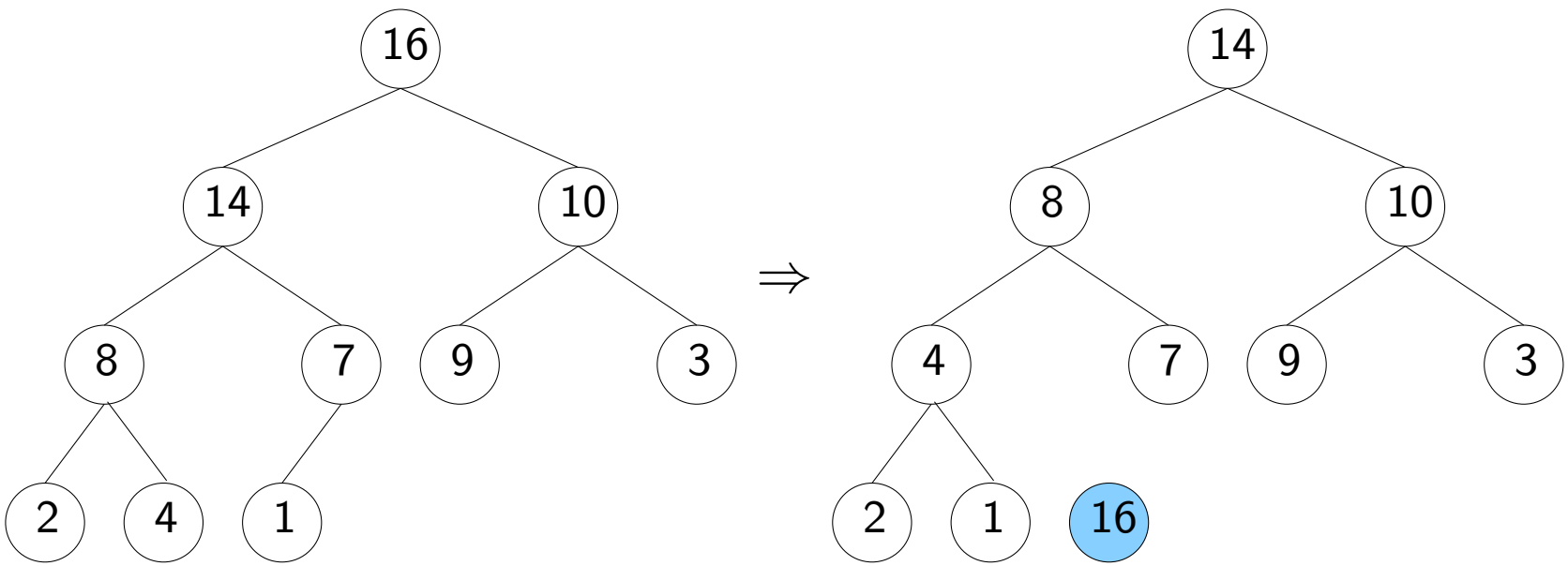
O algoritmo Heapsort

Heapsort(A)
    Build_Heap(A)
    for $i \leftarrow$ *length*$[A]$ downto 2
        exchange $A[1] \leftrightarrow A[i]$
        heap_size$[A] \leftarrow$ *heap_size*$[A] - 1$
        Heapify(A, 1)

- O algoritmo Heapsort executa em tempo $O(n \lg n)$

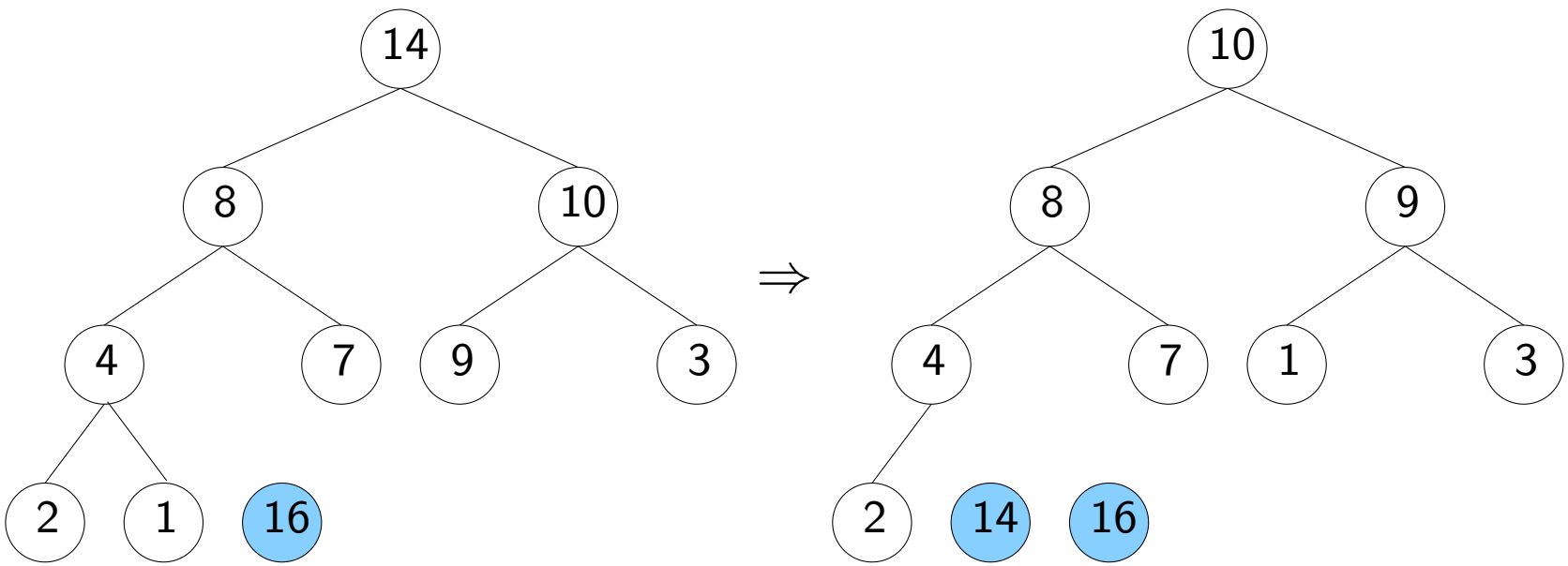
Heapsort

Exemplo



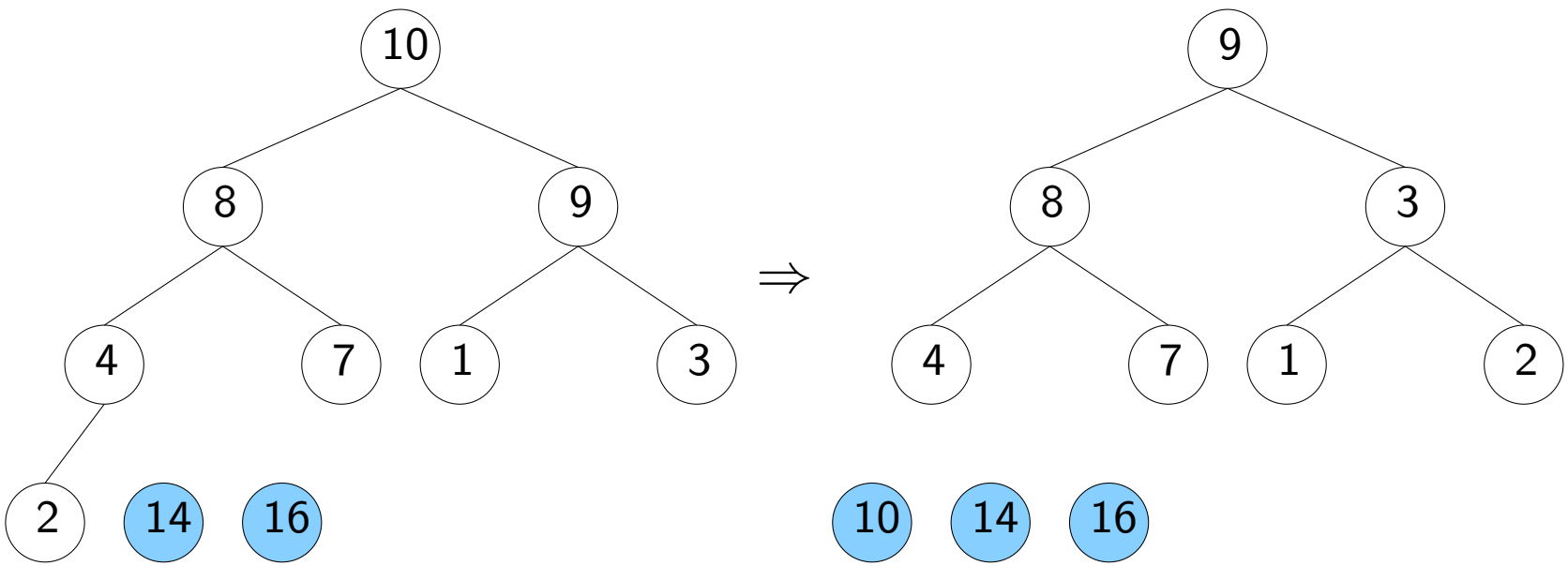
Heapsort

Exemplo



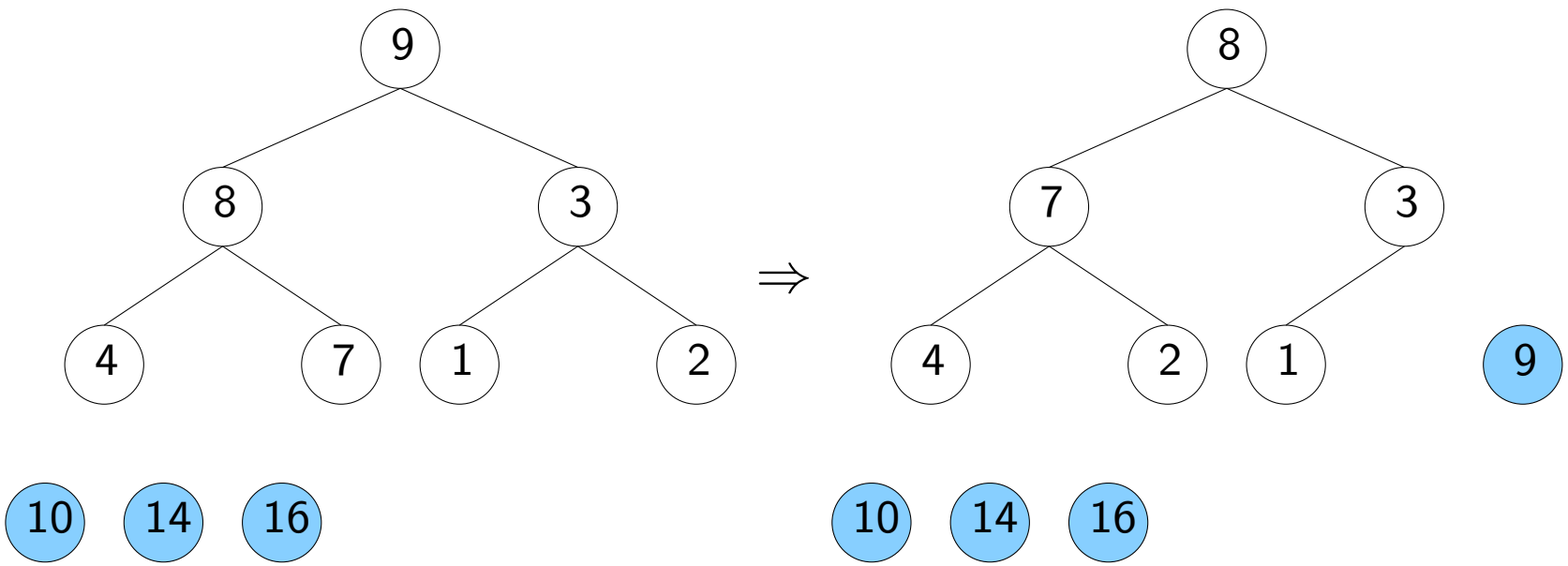
Heapsort

Exemplo



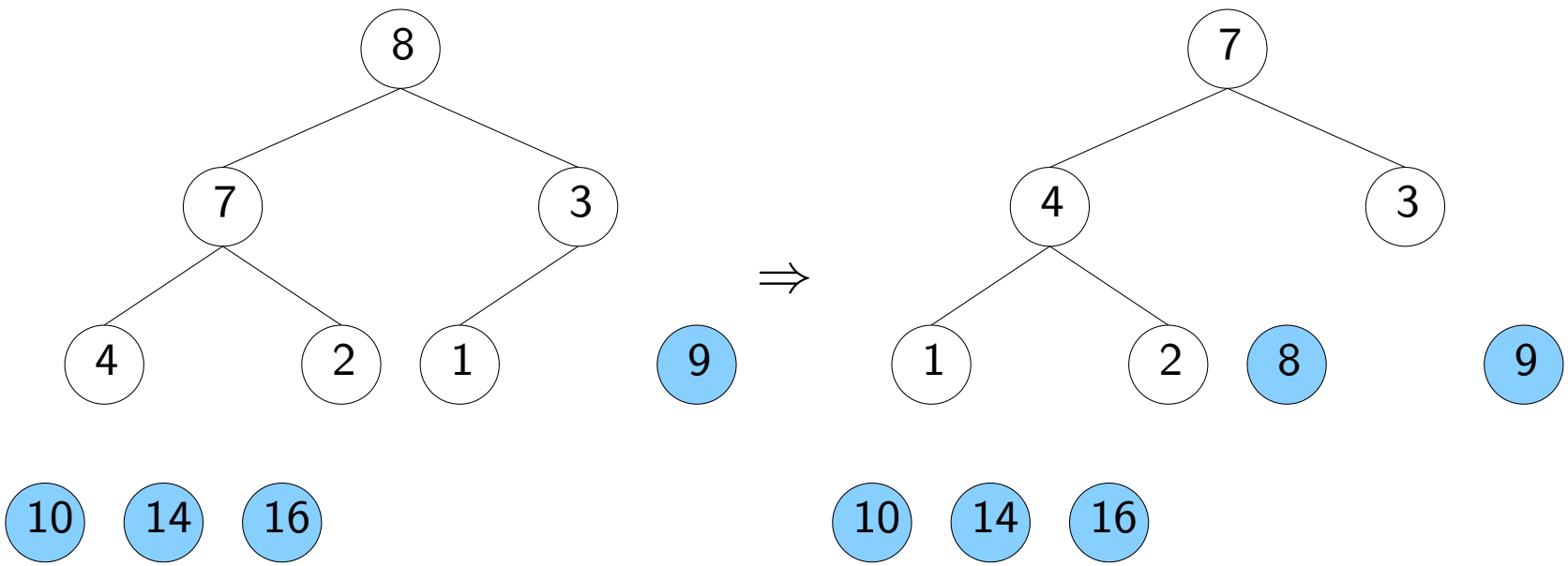
Heapsort

Exemplo



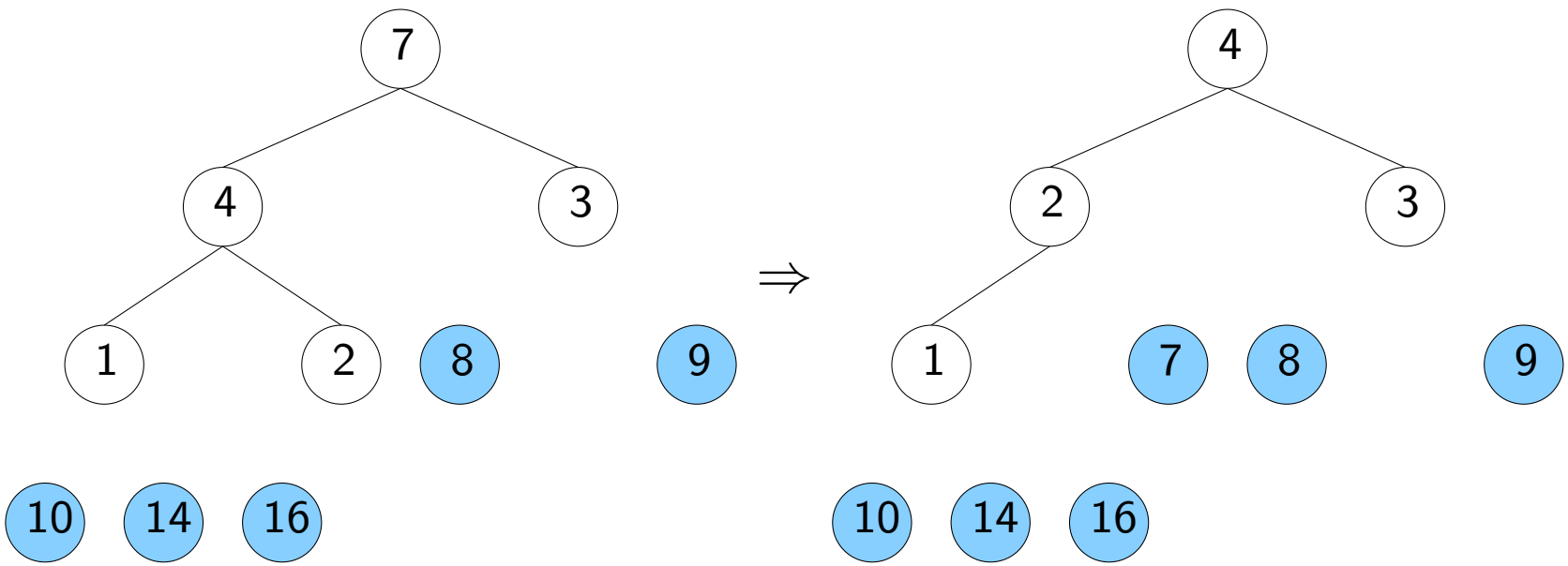
Heapsort

Exemplo



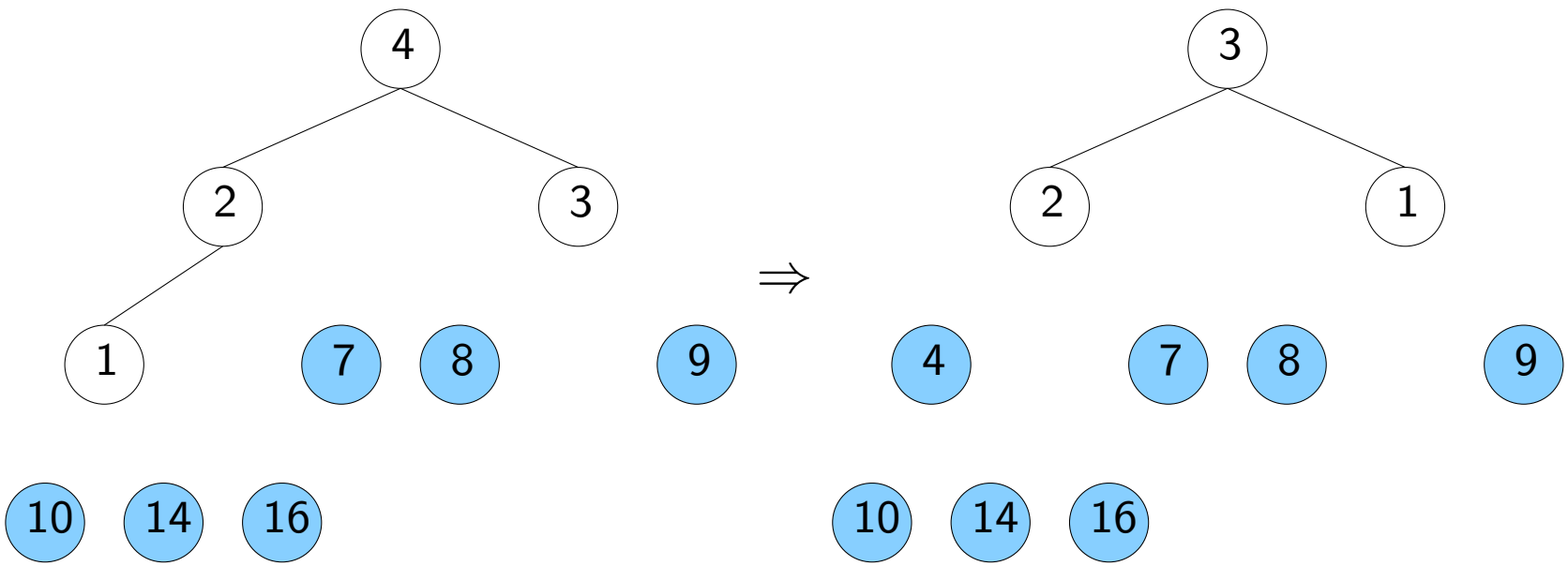
Heapsort

Exemplo



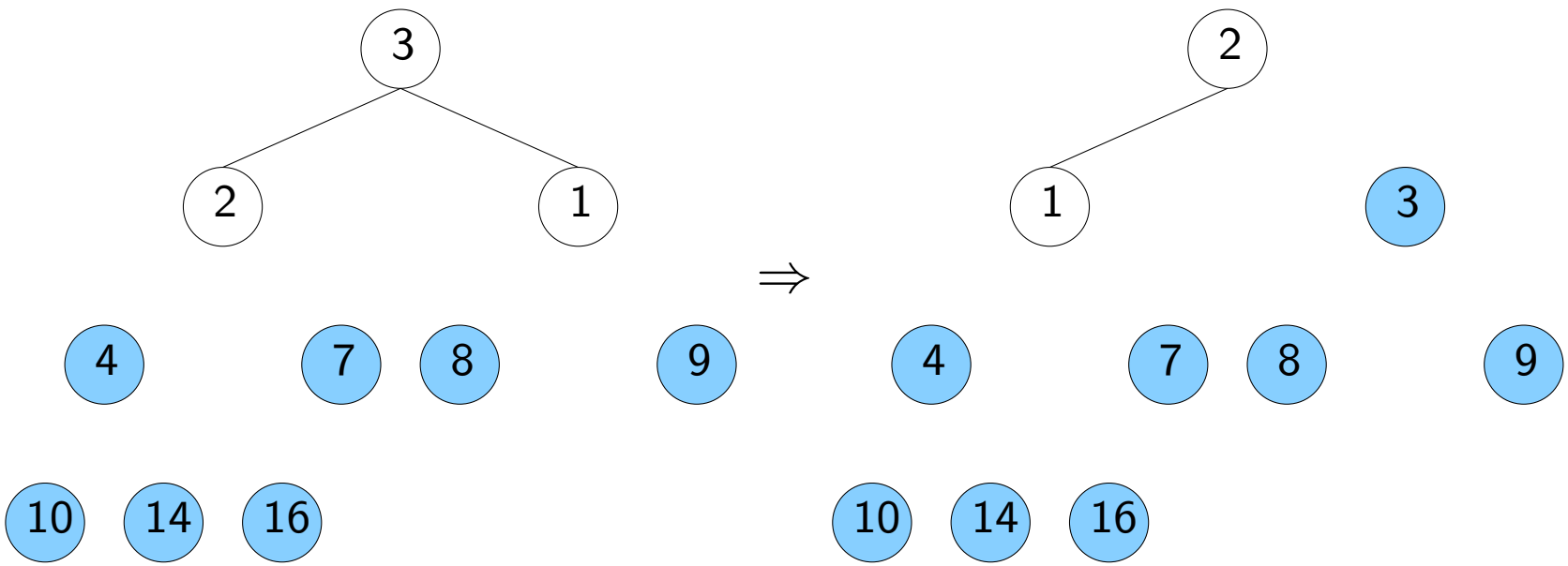
Heapsort

Exemplo



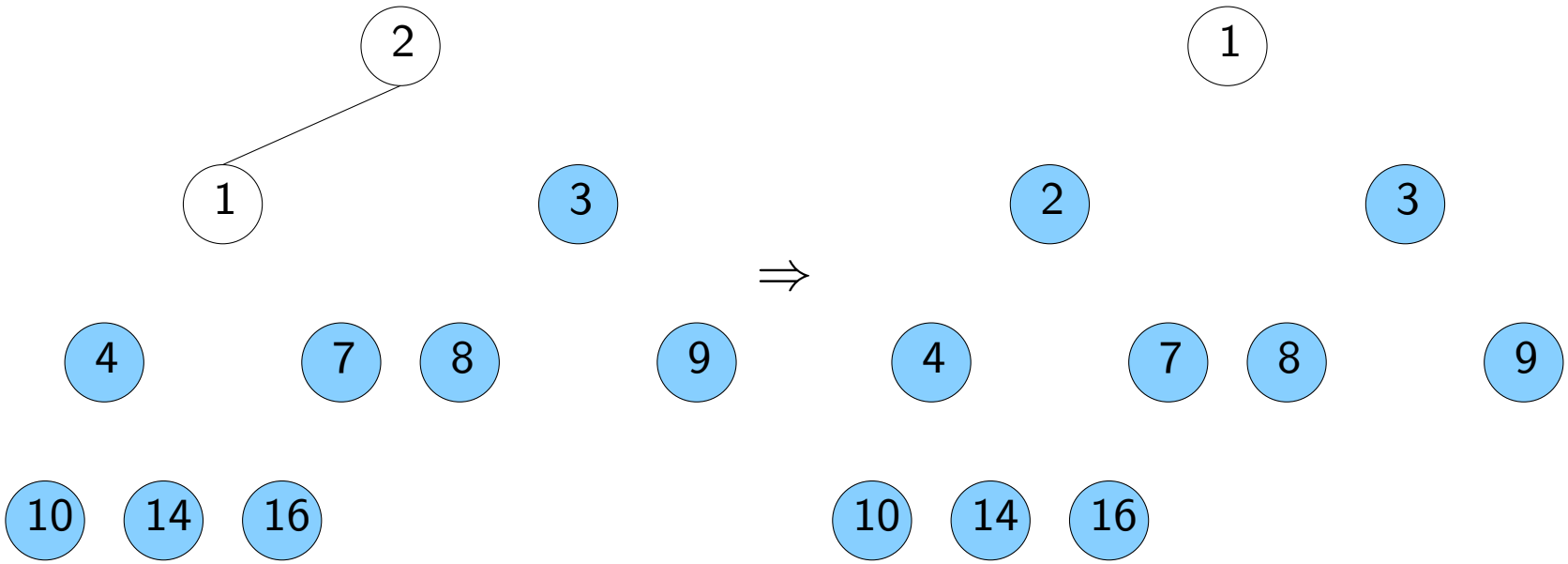
Heapsort

Exemplo



Heapsort

Exemplo



Heap como uma fila de prioridades

Fila de prioridades

- Uma **fila de prioridades** é uma estrutura de dados para manter um conjunto dinâmico S de elementos, cada um com um valor associado, chamado de chave.
- As operações são:
  - $Insert(S, x)$: insere um elemento em S
  - $Maximum(S)$: retorna o elemento de maior chave de S
  - $Extract\_Max(S)$: extrai o elemento de maior chave de S
- Uma aplicação é o escalonamento de tarefas em um computador de memória compartilhada.

Heap como uma fila de prioridades

Extrair o máximo

Heap_Extract_Max(A)
    if $heap\_size[A] < 1$
        error("underflow")
    $max \leftarrow A[1]$
    $A[1] \leftarrow A[heap\_size[A]]$
    $heap\_size[A] \leftarrow heap\_size[A] - 1$
    Heapify(A, 1)
    return max

- O tempo de execução é $O(\lg n)$

Heap como uma fila de prioridades

Inserir novo elemento

1) Expandimos o tamanho do heap introduzindo o novo elemento
2) Navegamos através da árvore até que o elemento atinja a posição correta para manter a propriedade heap.

Heap como uma fila de prioridades

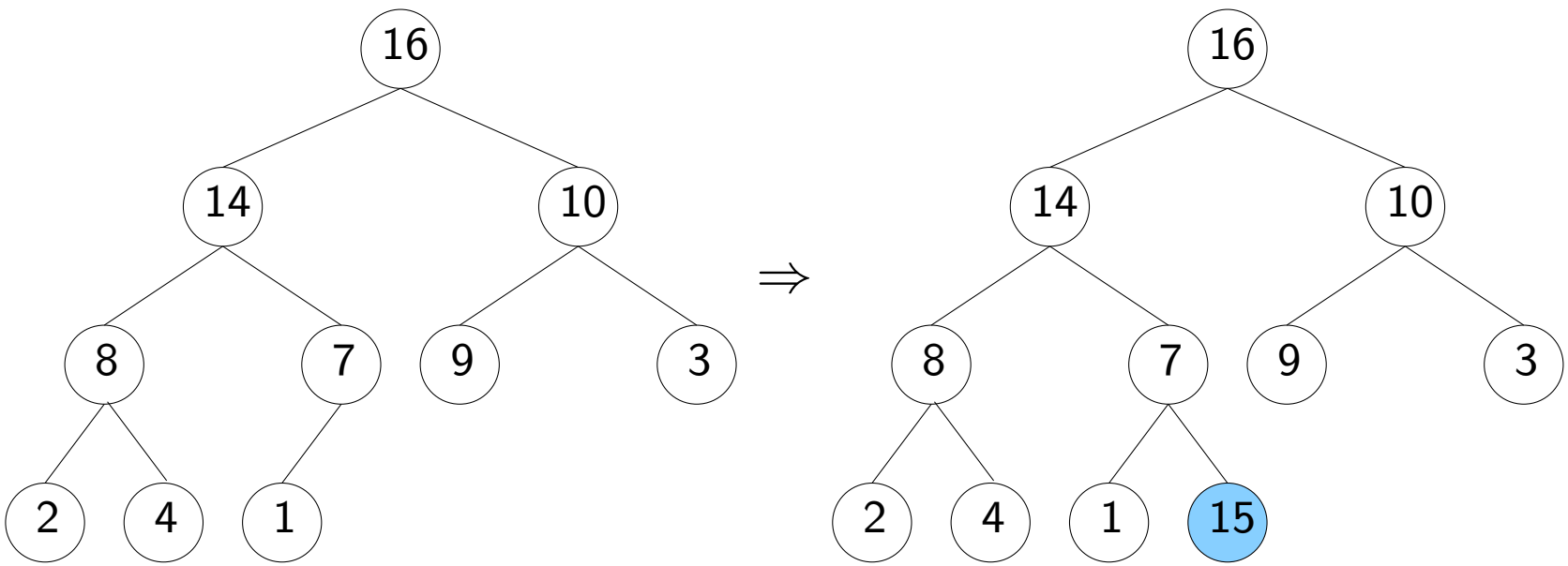
Inserir novo elemento

Heap_Insert(A, key)
    $heap\_size[A] \leftarrow heap\_size[A] + 1$
    $i \leftarrow heap\_size[A]$
    while $i > 1$ & $A[\text{Parent}(i)] < key$
        $A[i] \leftarrow A[\text{Parent}(i)]$
        $i \leftarrow \text{Parent}(i)$
    $A[i] \leftarrow key$

- O procedimento executa em tempo $O(\lg n)$

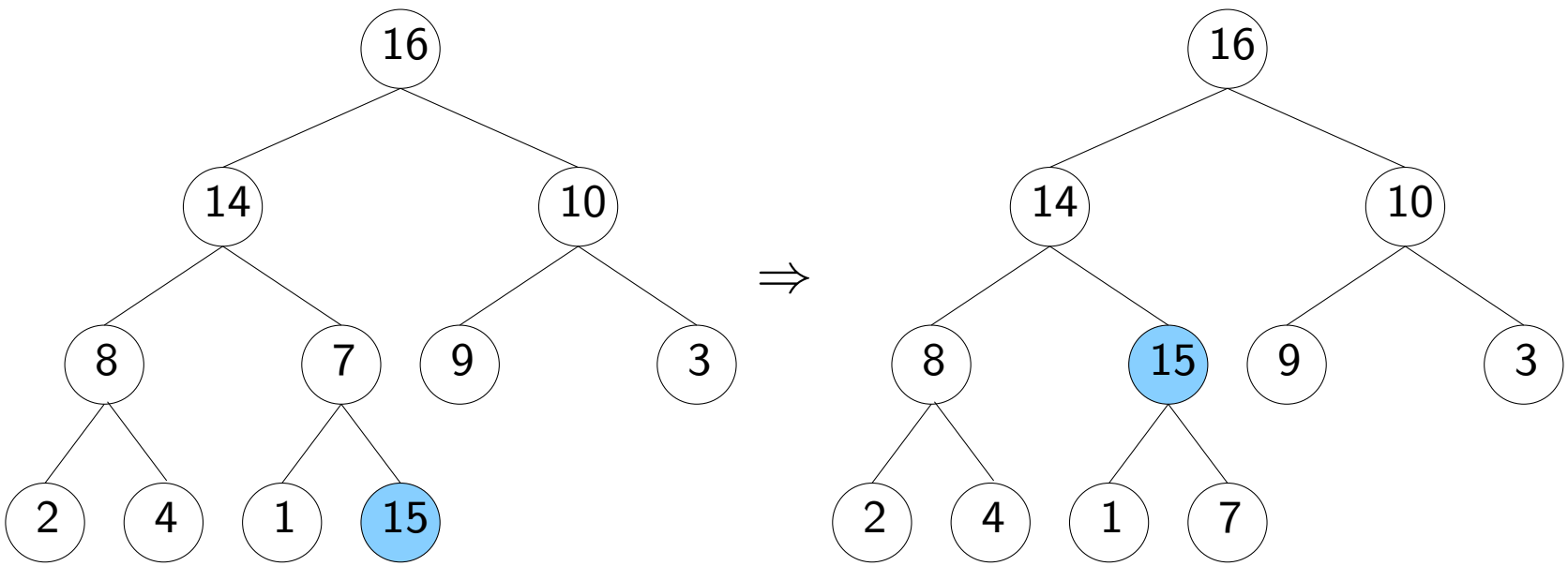
Heap como uma fila de prioridades

Exemplo



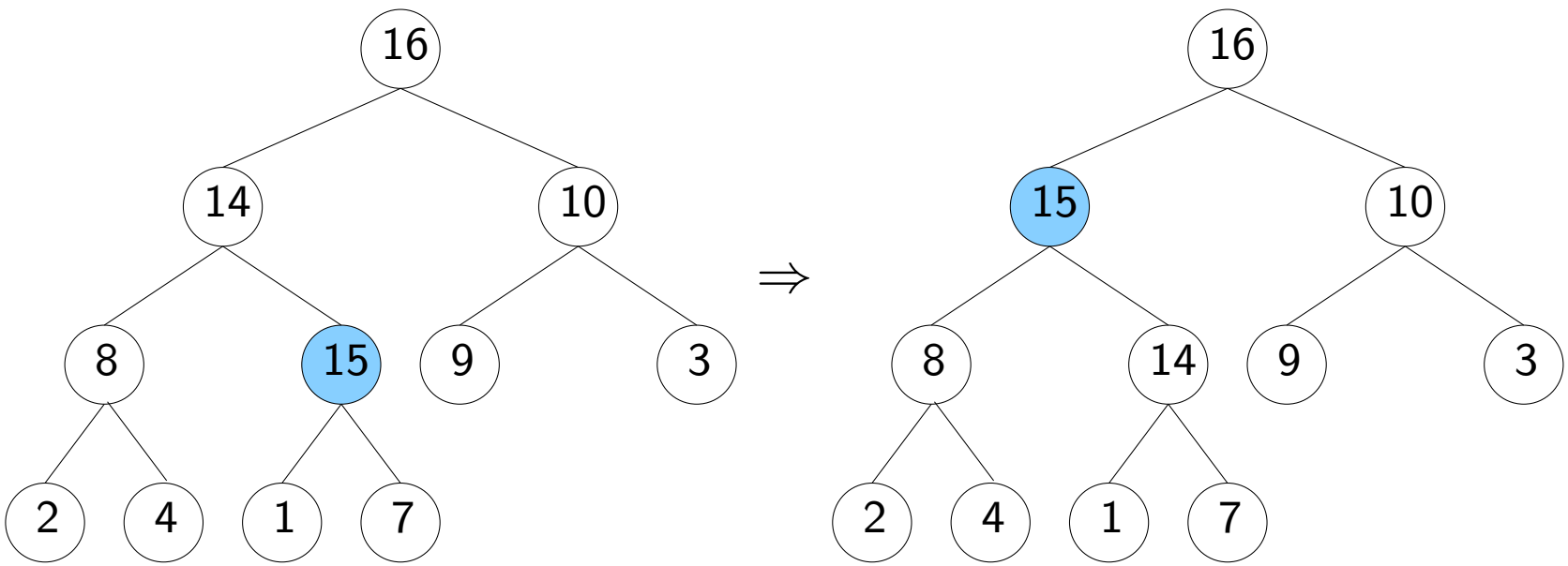
Heap como uma fila de prioridades

Exemplo



Heap como uma fila de prioridades

Exemplo



Sumário

Árvores Binárias de Pesquisa

Introdução

- Árvores binárias de pesquisa são estruturas de dados que suportam muitas das operações executadas sobre conjuntos dinâmicos
- Operações: *Search*, *Minimum*, *Predecessor*, *Insert* e *Delete*
- As Operações levam tempo proporcional à altura da árvore.
- Para uma árvore binária completa de n nós, tais operações levam $\Theta(\lg n)$.
- Todavia, se a árvore é uma lista de nós, as operações podem levar $\Theta(n)$ no pior caso.

Árvores Binárias de Pesquisa

Introdução

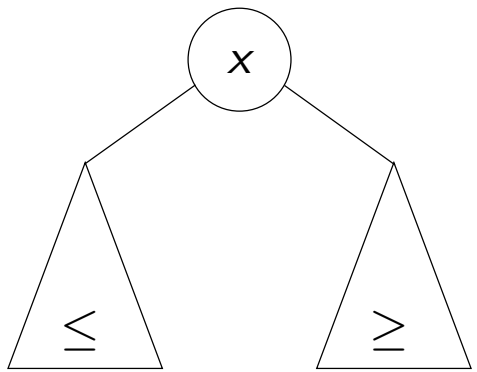
- A altura de árvores binárias construídas randomicamente é $O(\lg n)$, o que garante que as operações levem $\Theta(\lg n)$.
- Na prática, não podemos garantir que as árvores são construídas randomicamente.
- Entretanto, existem implementações de árvores binárias balanceadas, que garantem uma altura $O(\lg n)$ para a árvore, tais como *Red-Black Trees*, *AVL Trees* e *B-Trees*.

Árvores Binárias de Pesquisa

O que é árvore binária de pesquisa?

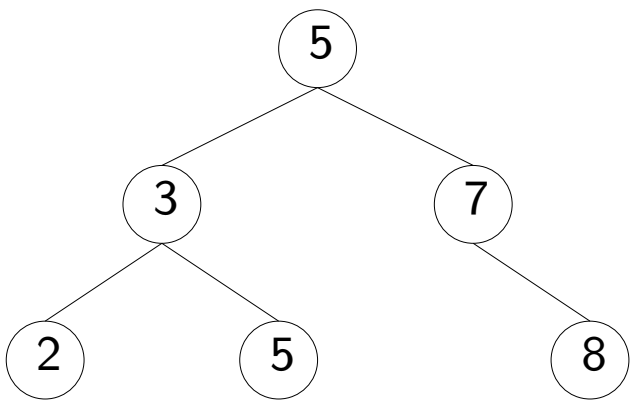
Toda árvore binária que satisfaz a **propriedade árvore-binária de pesquisa**:

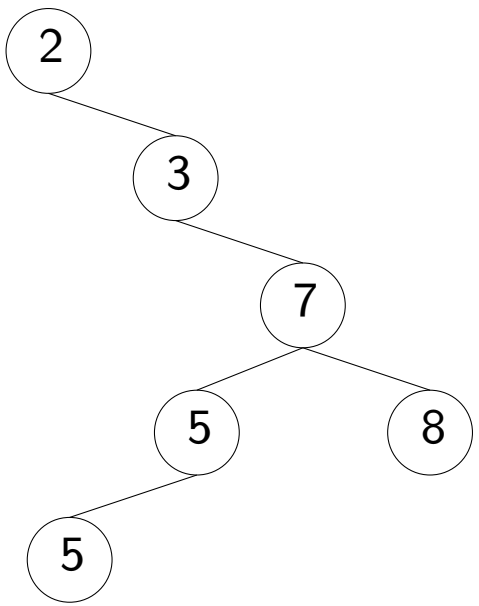
- Seja x um vértice qualquer da árvore binária de pesquisa
- Se y é um nó da subárvore da esquerda, então $key[y] \leq key[x]$
- Se y é um nó da subárvore da direita, então $key[y] \geq key[x]$



Árvores Binárias de Pesquisa

Exemplos





Árvores Binárias de Pesquisa

Inorder tree walk

A **propriedade árvore-binária de pesquisa** nos permite imprimir todas as chaves da árvore binária em ordem com um simples algoritmo recursivo.

```
Inorder_Walk(T, x)
        if x ≠ NIL
                Inorder_Walk( T, left[x] )
                print(key[x])
                Indorder_Walk( T, right[x] )
```

A chamada Indorder_Walk(T, $root[T]$) é executada em tempo $\Theta(n)$, retornando para os exemplos anteriores 2, 3, 5, 5, 7, 8.

Árvores Binárias de Pesquisa

Busca em uma árvore

As operações:

- $Search(T, k)$
- $Minimum(T)$
- $Maximum(T)$
- $Predecessor(T, x)$ e
- $Sucessor(T, x)$

são executadas em tempo $O(h)$, onde h é a altura da árvore T.

Busca em uma árvore

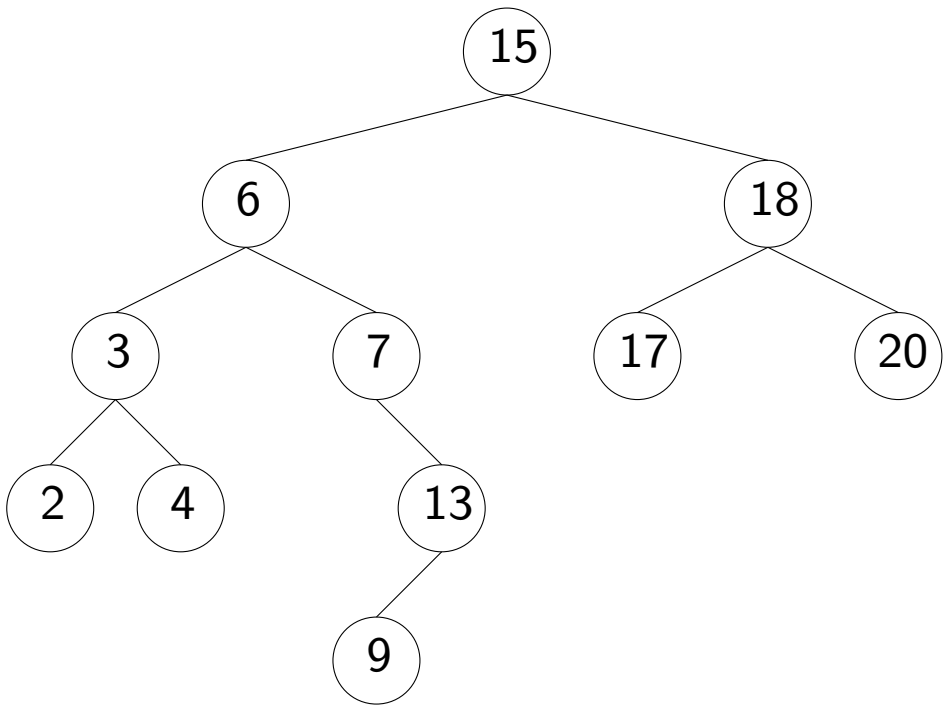
Search

Dados um ponteiro para a raiz T e uma chave k, o procedimento Search(T,k) retorna um ponteiro para um objeto com a chave k se um existe, ou NIL caso contrário.

```
Search(T, k)
        if x = NIL or key[x] = k
                return x
        if k < key[x]
                Search(left[x], k)
        else
                Search(right[x], k)
```

Busca em uma árvore

Exemplo



A busca pela chave 13, segue o caminho $15 \longrightarrow 6 \longrightarrow 7 \longrightarrow 13$.

Mínimo e Máximo

O elemento cuja chave é mínima pode ser encontrado ao ao seguirmos a cadeia "left_child"a partir da raiz até que NIL seja encontrado. Para o máximo, o precedimento é semelhante.

Mínimo

Minimum(x)
    while $left[x] \neq$ NIL
        $x \leftarrow left[x]$
    return x

Máximo

Maximum(x)
    while $right[x] \neq$ NIL
        $x \leftarrow right[x]$
    return x

Sucessor e Predecessor

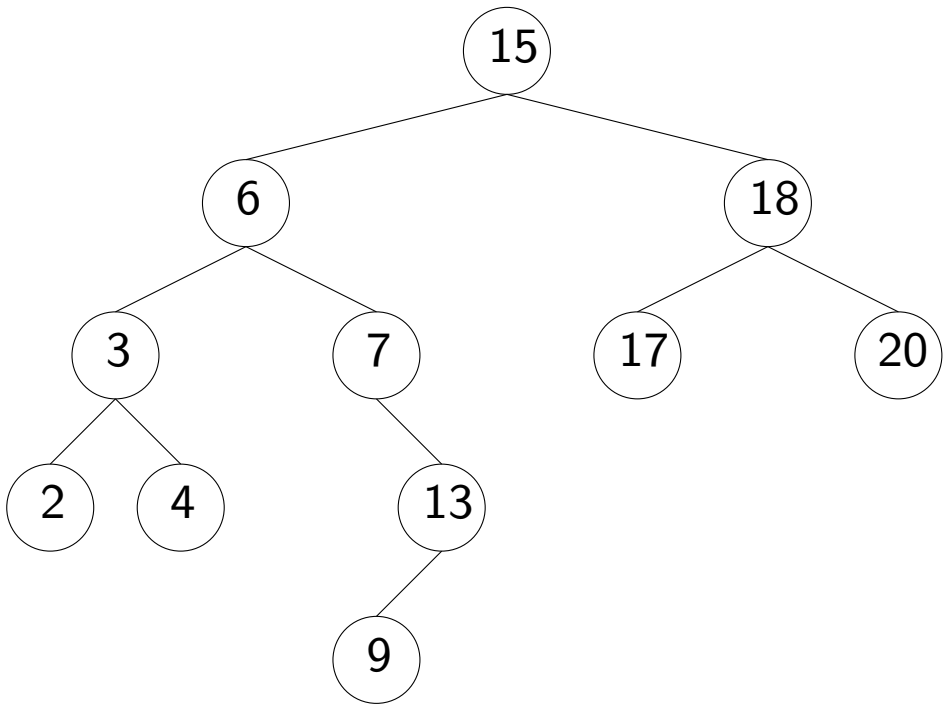
- O sucessor de um nó x é o nó y com a menor chave $key[y]$ que seja maior do que $key[x]$.
- A estrutura da árvore binária nos permite encontrar o sucessor de um nó x, sem fazer nenhuma comparação de chaves.
- Se a sub-árvore à direita de x é não-vazia, então o sucessor de x é o elemento mínimo desta sub-árvore.
- Caso contrário, o sucessor de x é o ancestral mais próximo de x cujo filho da esquerda é também um ancestral de x.

Sucessor

```
Successor(x)
        if right[x] ≠ NIL
                return Minimum(right[x])
        y ← p[x]
        while y ≠ NIL & x = right[y]
                x ← y
                y ← p[y]
        return y
```

Sucessor

Exemplo



- O sucessor de 13 é 15

Inserção de Objetos

Insert

```
Insert(T, z)
    y ← NIL
    x ← root[T]
    while x ≠ NIL
        y ← x
        if key[z] < key[x]
            x ← left[x]
        else
            x ← right[x]
        endif
    endwile
```

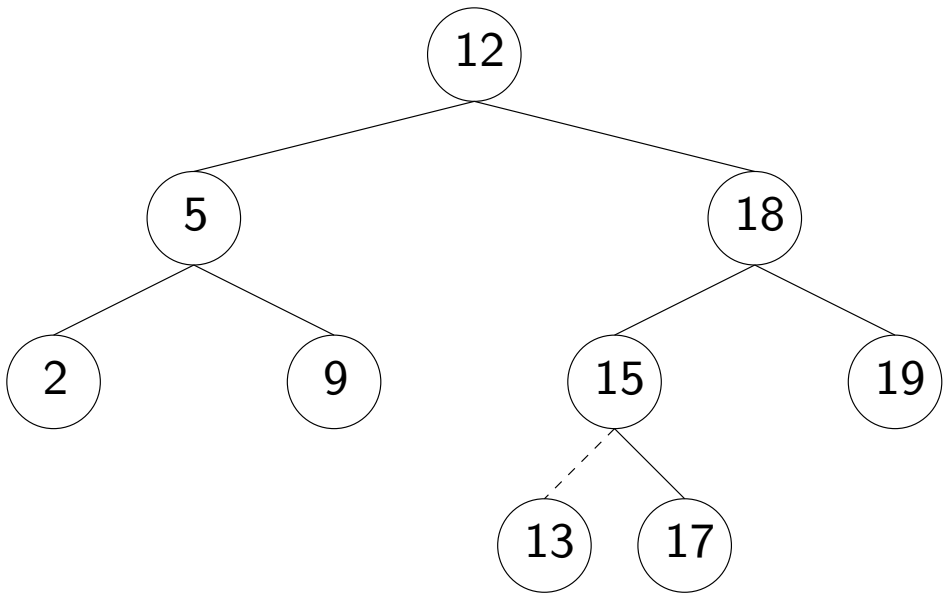
Insert (Continuation)

```
p[z] ← y
if y = NIL
    root[T] ← z
else
    if key[z] < key[y]
        left[y] ← z
    else
        right[y] ← z
    endif
endif
```

Inserção de Objetos

Exemplo

- Inserção de 13



Deleção

O procedimento de deleção de um nó z consiste em 3 casos.

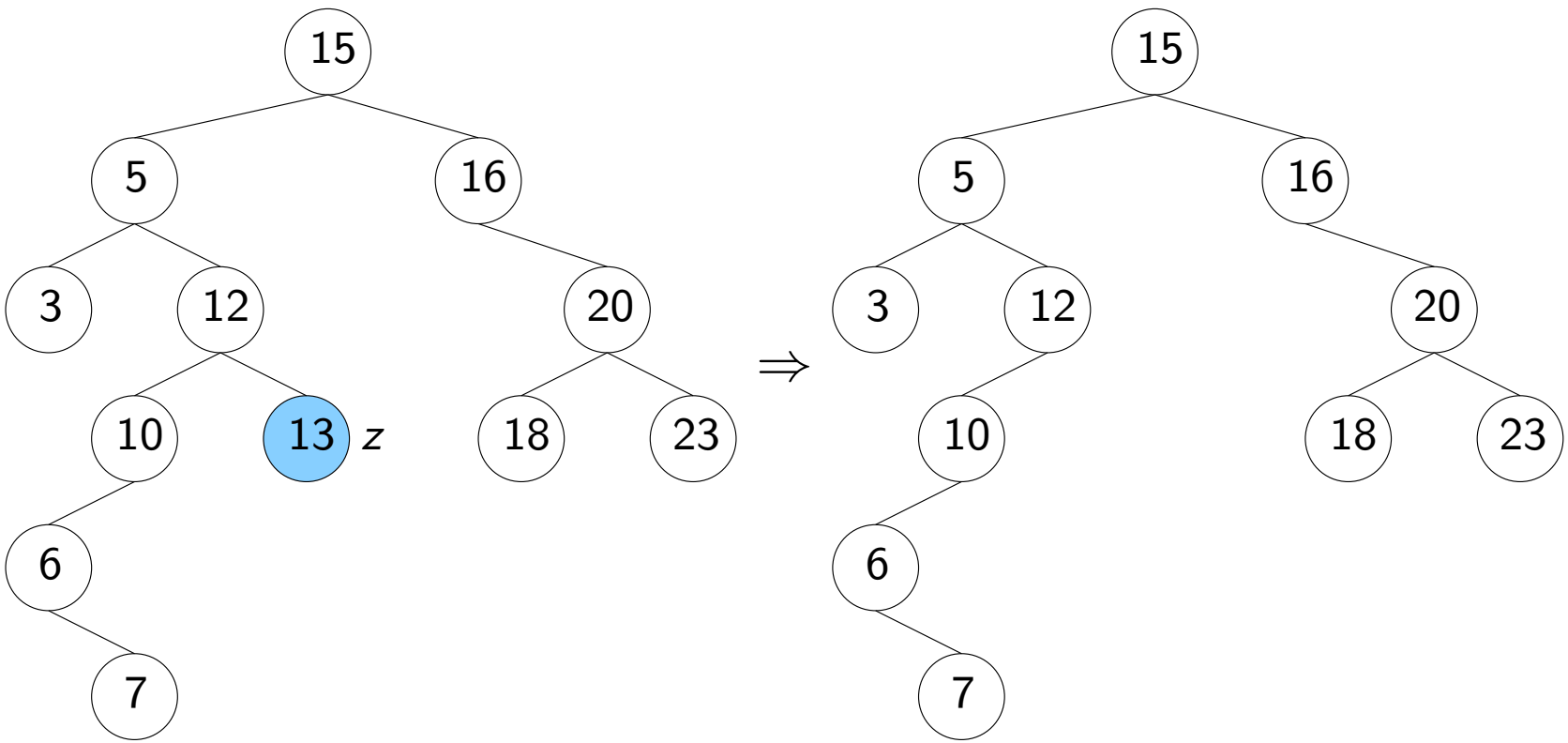
a) Se z não tem filhos, modificar o pai de z, $p[z]$, para que este aponte para NIL.

b) Se z possui apenas um filho, fazemos o pai de z apontar para o filho de z.

c) Se z possui dois filhos, colocamos no lugar de z o seu sucessor, o que com certeza não possui filho à esquerda.

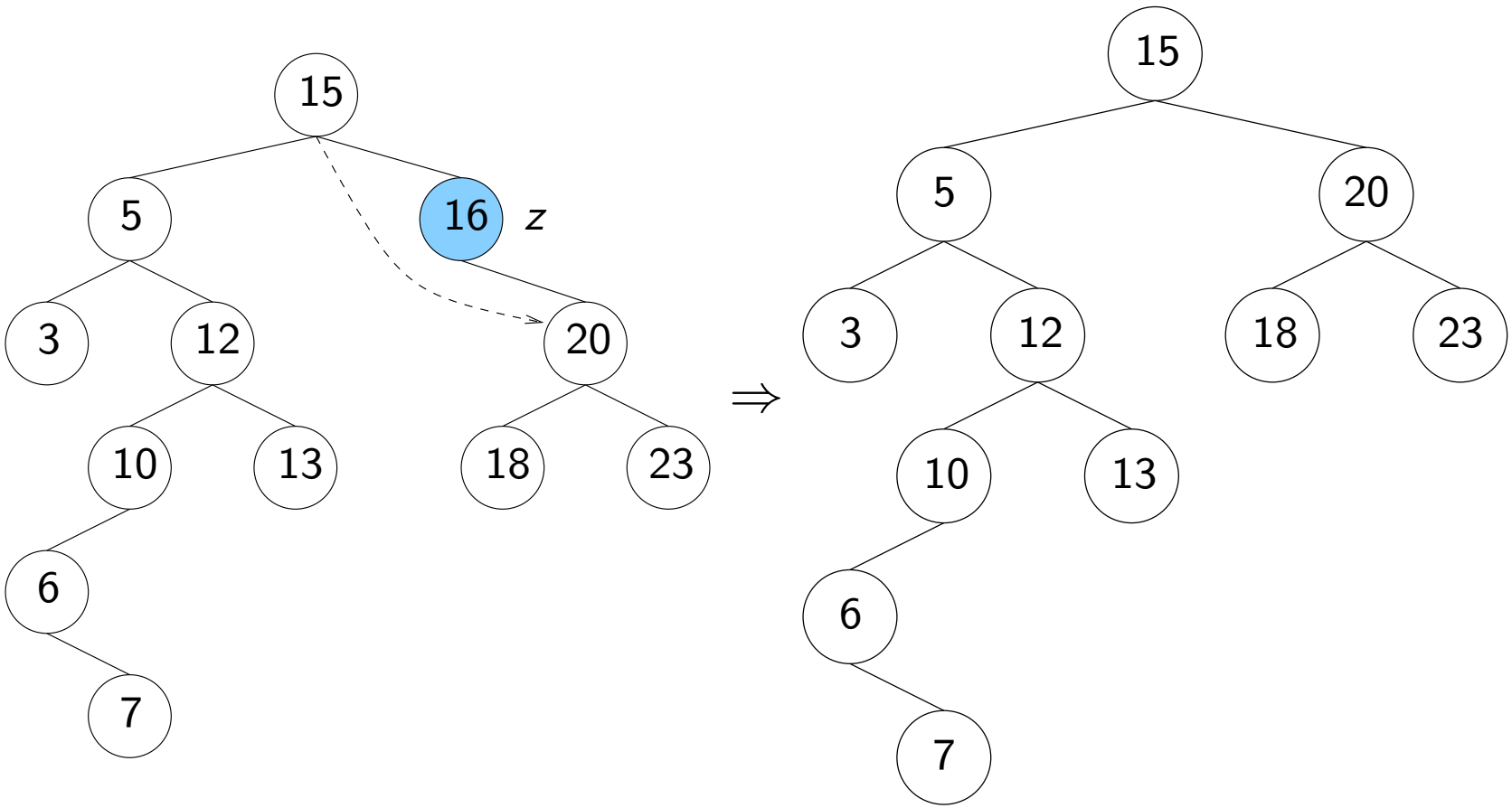
Deleção

Caso a: z não possui filhos



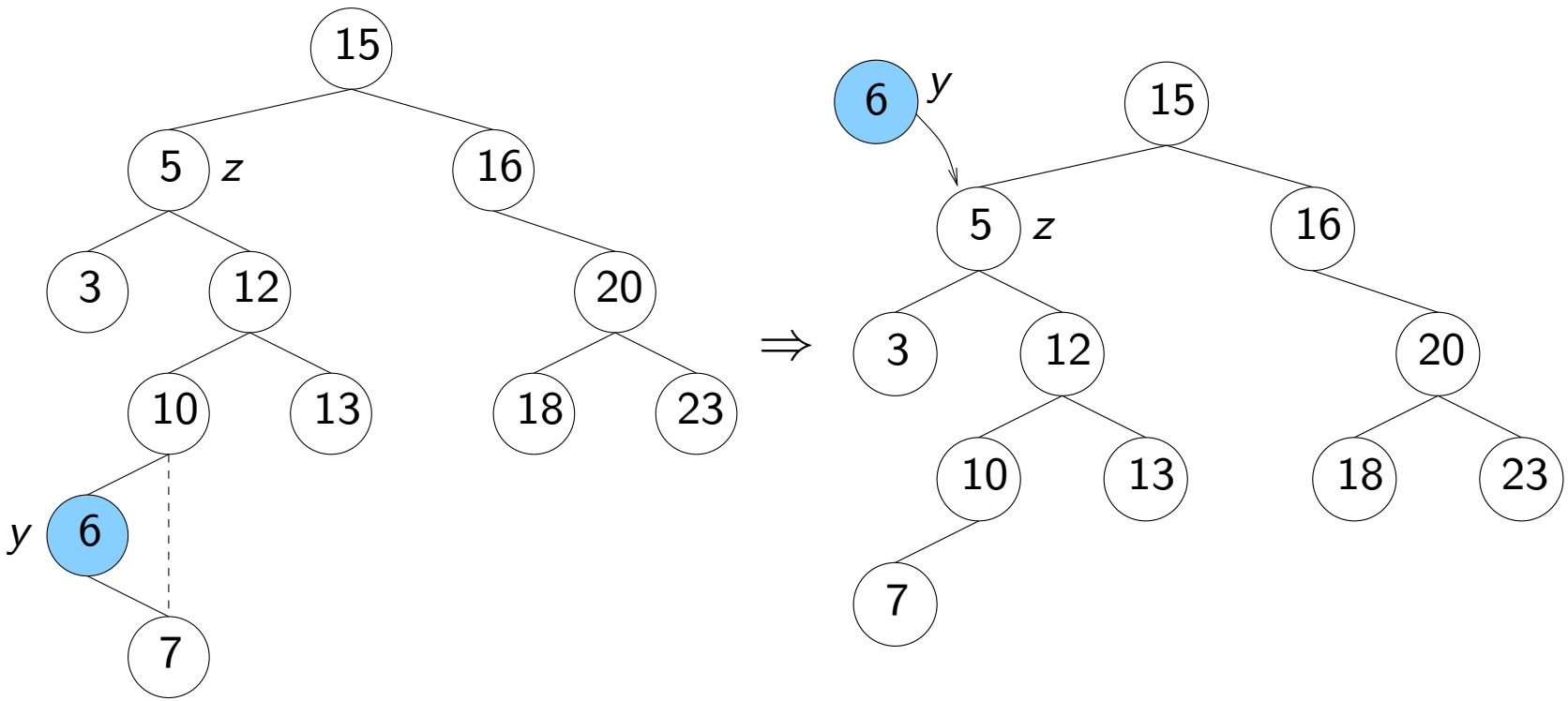
Deleção

Caso b: z possui um filho



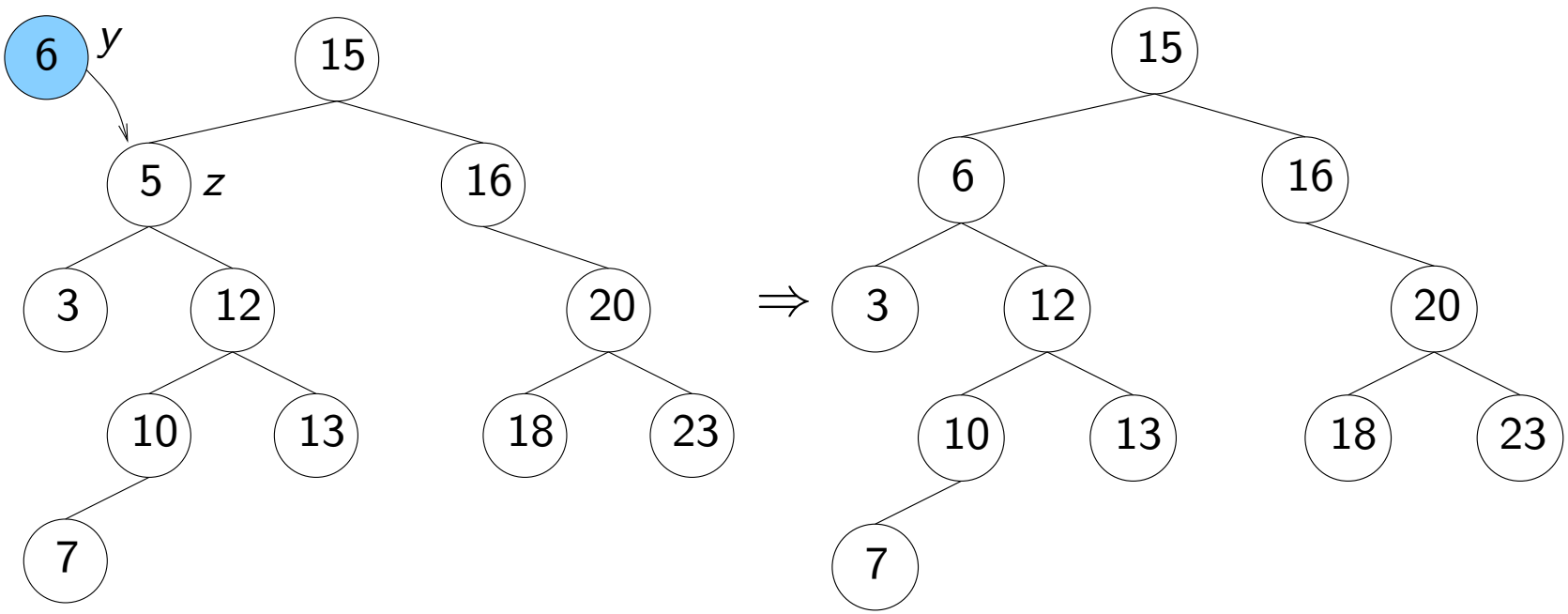
Deleção

Caso c: z possui dois filhos



Deleção

Caso c: z possui dois filhos



Heaps e Árvores Binárias

- Fim!
- Obrigado pela presença