

# CES-11



## Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches  
Juliana de Melo Bezerra

# CES-11

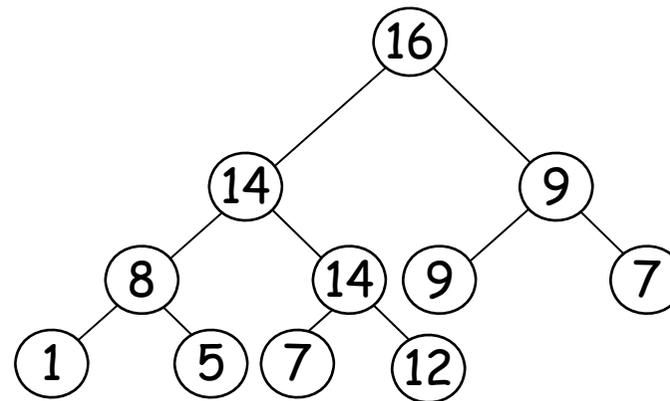


- Algoritmos de Ordenação
  - *Heap*
  - Filas de prioridade
  - *HeapSort*

# A estrutura *heap*

- *Heap* é uma árvore binária com duas propriedades:
  - 1) Balanceamento: é uma árvore completa, com a eventual exceção do último nível, onde as folhas estão sempre nas posições mais à esquerda.
  - 2) Estrutural: o valor armazenado em cada nó não é menor que os de seus filhos.

Exemplo:

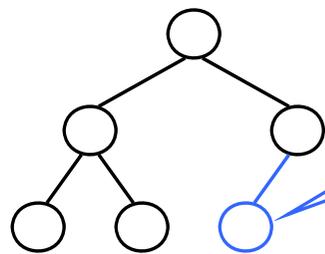


- Observação: há também o caso análogo, em que o valor de cada nó não é maior que os de seus filhos.

# Criação de um *heap*

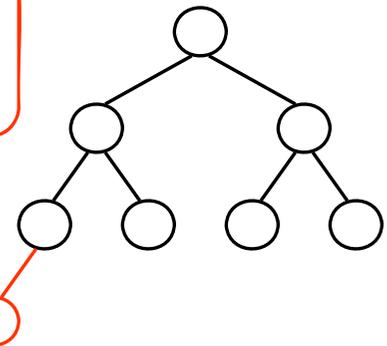
- Uma árvore com um único nó já é automaticamente um *heap*.
- Procedimento para adição de novos nós a um *heap*:
  - Inicialmente, para se garantir a propriedade de balanceamento, o novo nó deverá ser folha no último nível, na primeira posição disponível mais à esquerda.
  - Se este nível estiver cheio, começa-se um novo.

## ■ Exemplos:



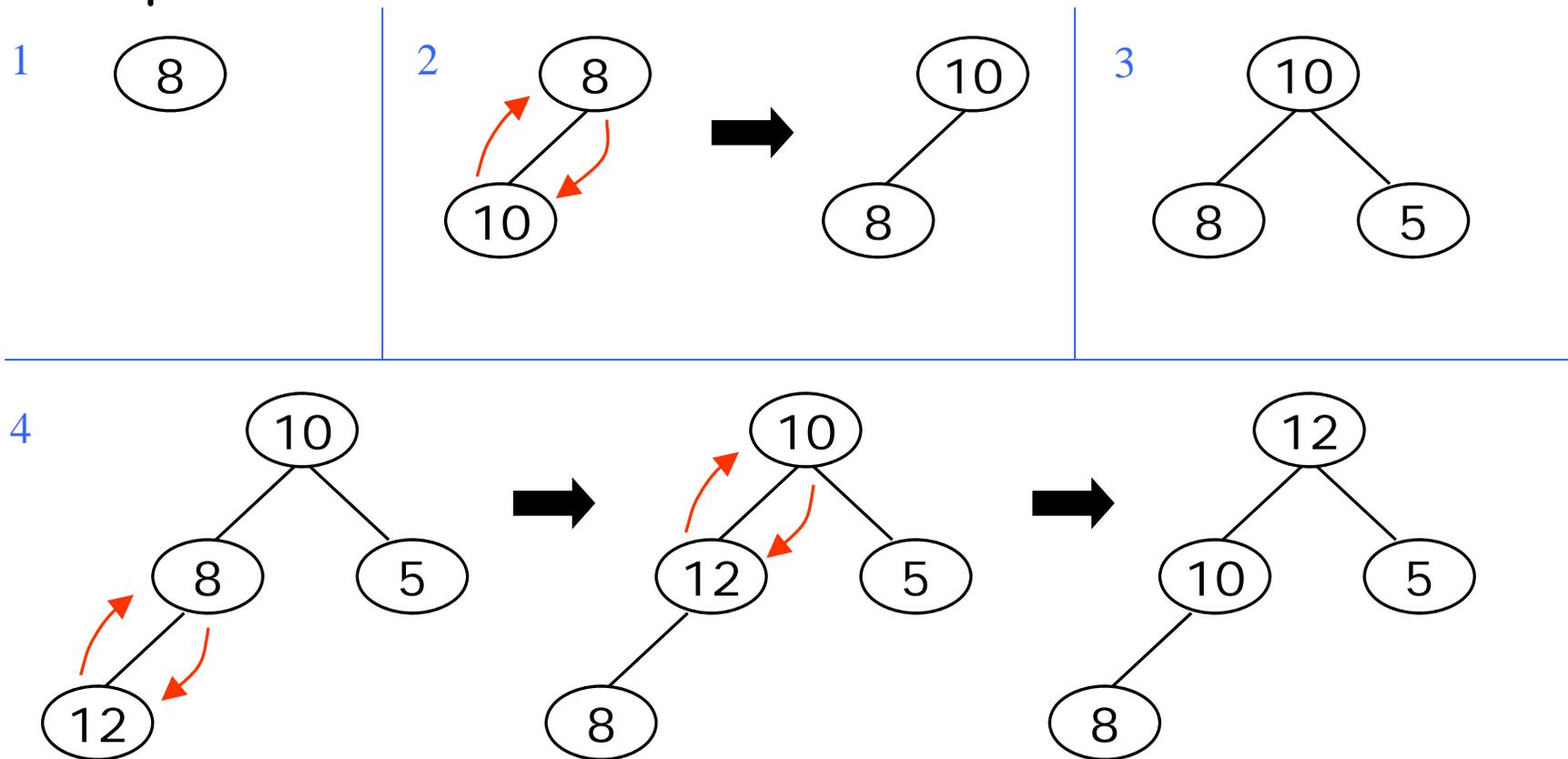
Aqui o nó pode ser adicionado

Aqui o nó pode ser adicionado



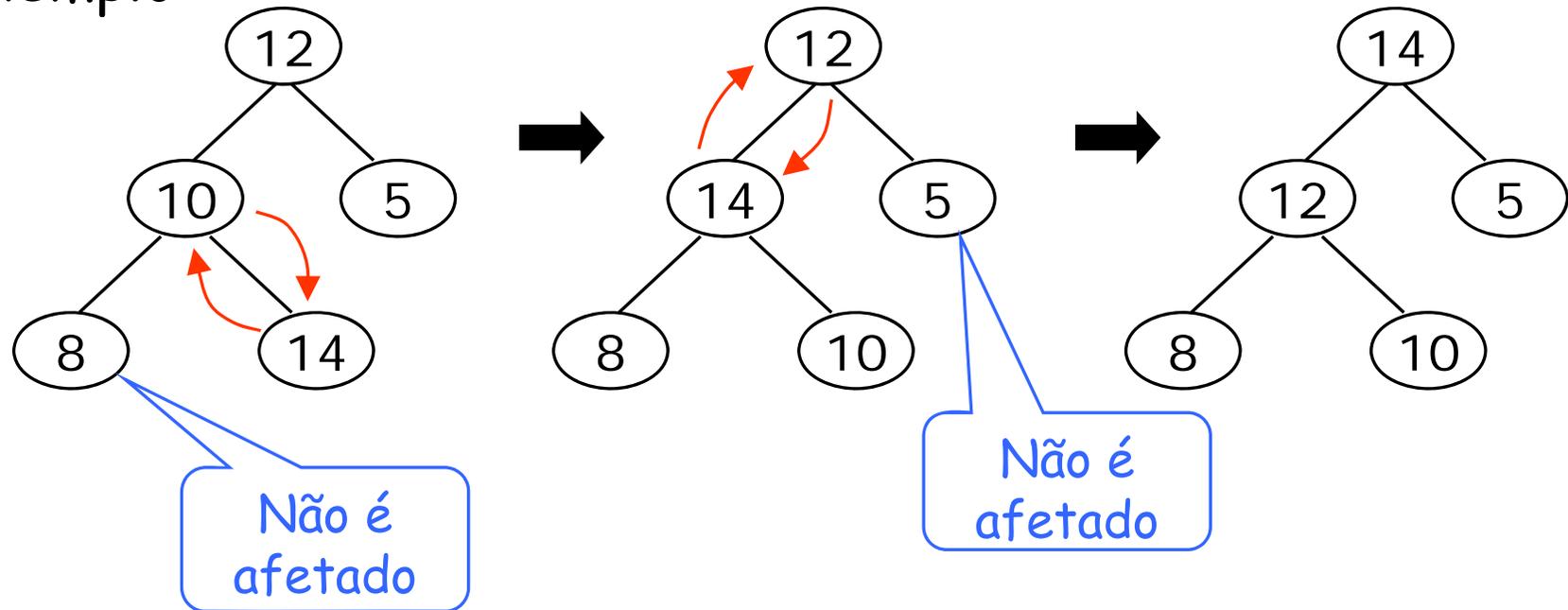
# Criação de um *heap*

- Durante o acréscimo de um novo nó, se seu pai perder a propriedade estrutural, basta trocar de posição com ele. Esse procedimento pode eventualmente chegar à raiz.
- Exemplo:



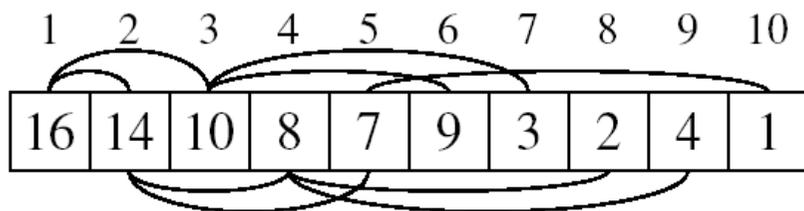
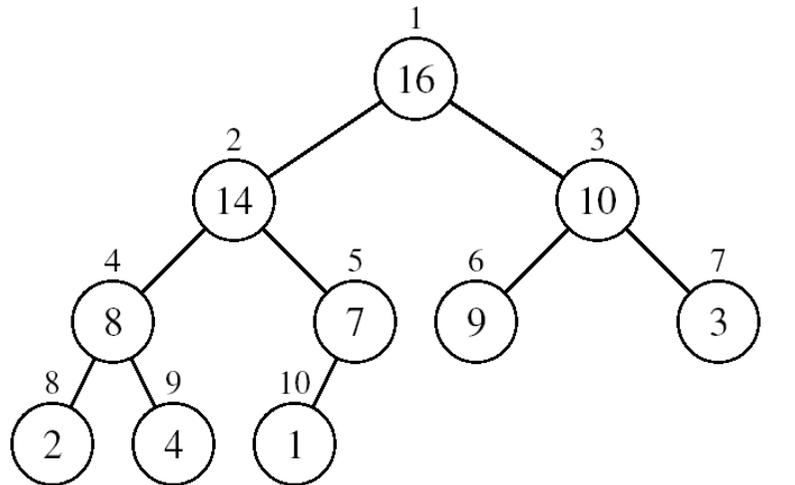
# Criação de um *heap*

- Essas trocas de valores não afetam a propriedade estrutural dos nós irmãos.
- Exemplo:



- Portanto, cada inserção de um novo nó pode ser feito em tempo proporcional à altura atual do *heap*.

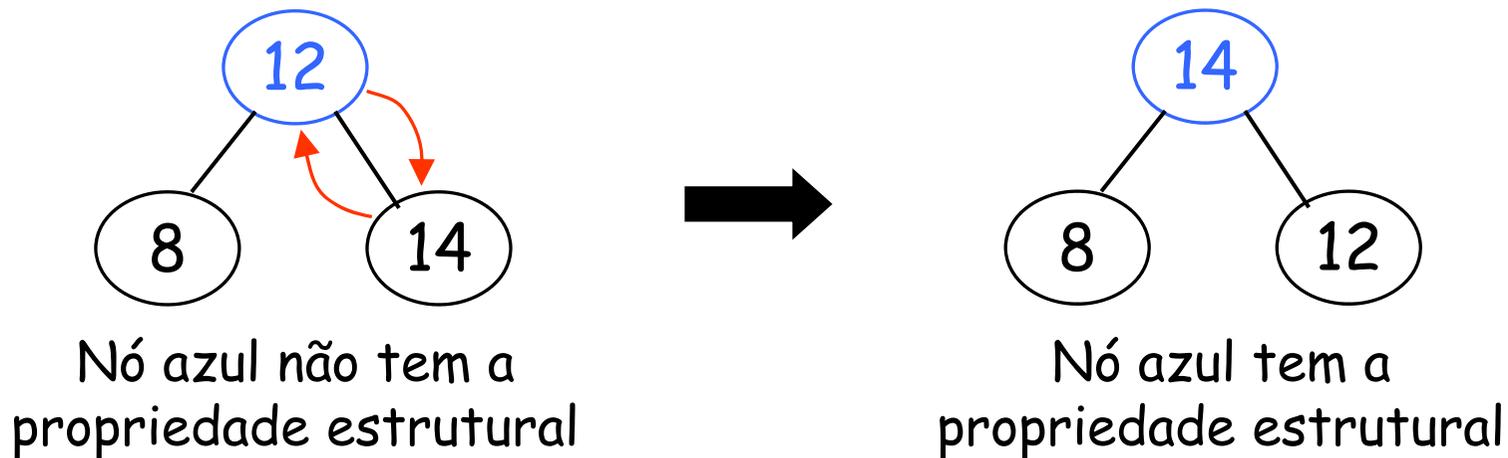
# Representação de *heap* com vetor



- Armazenamento de um *heap* com  $n$  elementos em um vetor  $v$ :
  - A raiz está em  $v[1]$
  - O filho esquerdo de  $v[i]$  é  $v[2i]$
  - O filho direito de  $v[i]$  é  $v[2i+1]$
- O pai de  $v[i]$  será  $v[\lfloor i/2 \rfloor]$ .
- Os elementos do subvetor  $v[(\lfloor n/2 \rfloor + 1) .. n]$  são as folhas.

# Algoritmo *Sift*

- Dado um *heap*, suponhamos que ocorra uma alteração no valor presente na sua raiz.
- Caso ela perca a propriedade estrutural, poderá recuperá-la trocando de valor com o seu filho maior.
- Isso pode ser feito através do algoritmo *Sift*:



- Como o filho trocado também pode perder a propriedade estrutural, será preciso chamar *Sift* para ele.

# Algoritmo *Sift*

Reorganiza "para baixo" o *heap* alterado na posição  $i$ :

```
sift(i, n){
    esq = 2i;
    dir = 2i+1;
    maior = i;
    if (esq <= n && v[esq] > v[i])
        maior = esq;
    if (dir <= n && v[dir] > v[maior])
        maior = dir;
    if (maior != i) {
        aux = v[i];
        v[i] = v[maior];
        v[maior] = aux;
        Sift(maior, n);
    }
}
```

Tempo:  $O(\log n)$

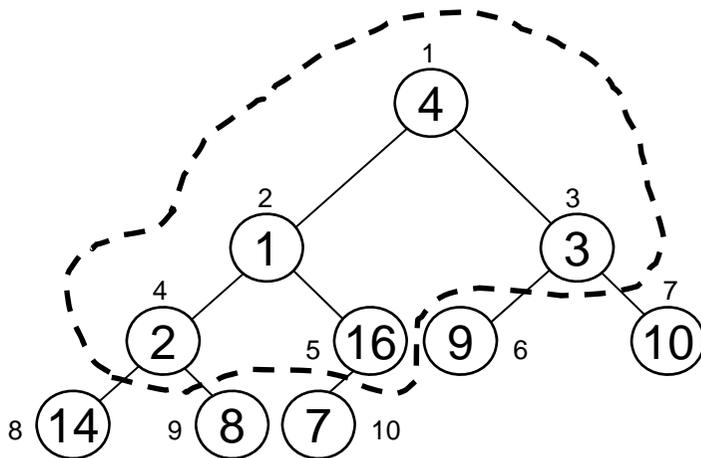
Exercício: reescrever *Sift* em formato não recursivo.

# Transformação de um vetor em um *heap*

- O algoritmo *Build* transforma o vetor  $v[1..n]$ , já inicializado, em um *heap* de tamanho  $n$ .
- Como as posições entre  $\lfloor n/2 \rfloor + 1$  e  $n$  serão as folhas do *heap*, basta aplicar *Sift* entre as posições 1 e  $\lfloor n/2 \rfloor$ .
- Exemplo:

v

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

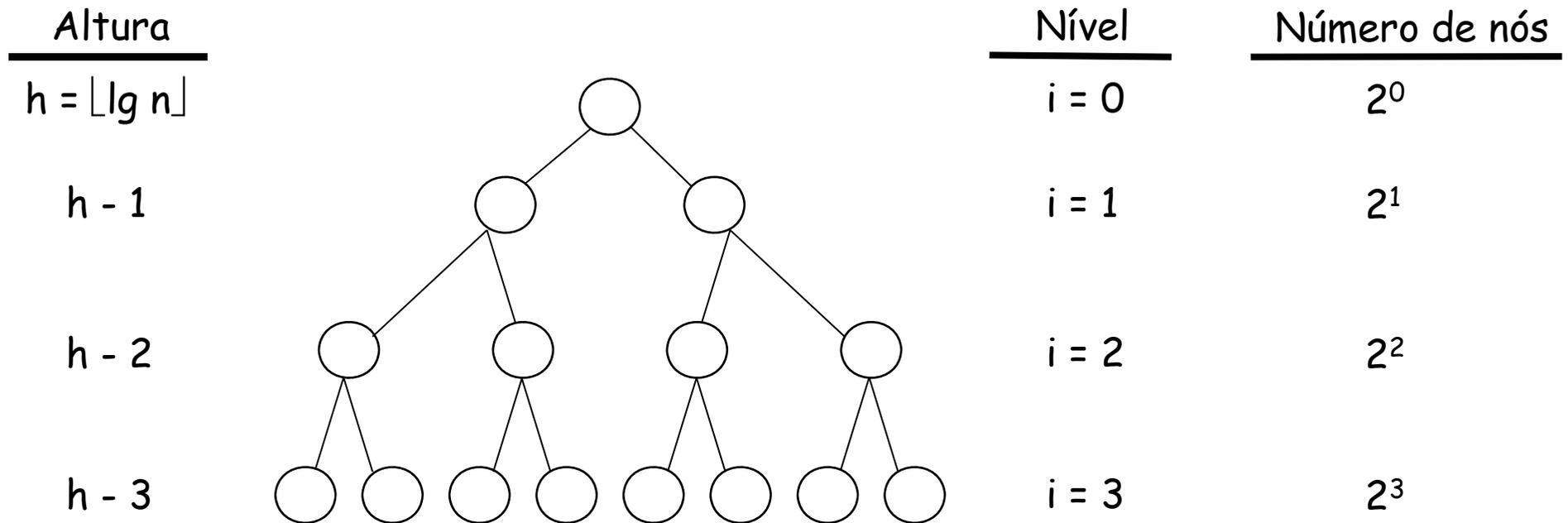


```
Build(v) {  
    for (i =  $\lfloor n/2 \rfloor$ ; i > 0; i--)  
        Sift(i, n);  
}
```

Complexidade de tempo:  $O(n \cdot \log n)$  ?

# Complexidade de tempo de *Build*

- O tempo gasto por  $Sift(i, n)$  é proporcional à altura do nó  $i$ .
- O pior caso é com a árvore completa:



$$T(n) = 2^0 h + 2^1 (h-1) + \dots + 2^h (h-h)$$

$$T(n) = \sum_{i=0}^h 2^i (h-i)$$

# Complexidade de tempo de *Build*

Tempo de pior caso, correspondente a uma árvore completa com  $n$  nós e altura  $h$ :

$$T(n) = \sum_{i=0}^h 2^i (h - i)$$

Multiplicando o numerador e o denominador por  $2^h$ :

$$= \sum_{i=0}^h \frac{h - i}{2^{h-i}} 2^h$$

Troca de variáveis ( $k = h - i$ ):

$$= 2^h \sum_{k=0}^h \frac{k}{2^k}$$

Sabemos que  $h = \lg n$  e que essa somatória é menor que a correspondente somatória até  $\infty$ :

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

Sabemos também que essa somatória é menor que 2:

$$= O(n)$$

# CES-11



- Algoritmos de Ordenação
  - *Heap*
  - Filas de prioridade
  - *HeapSort*

# Filas de prioridade

- Fila de prioridade é uma estrutura de dados com as seguintes operações:
  - *Max* (ou *Min*): retorna o valor máximo (ou mínimo) presente na fila
  - *ExtractMax* (ou *ExtractMin*): extrai e retorna o valor máximo (ou mínimo) presente na fila
  - *Modify(k, x)* : atribui o valor  $x$  à posição  $k$  da fila
  - *Insert(x)* : insere o valor  $x$  na fila
- *Heap* é uma boa implementação para filas de prioridade.
- Supomos que o *heap* utilize um vetor  $v$ , e que a variável `size` armazene o seu tamanho corrente.

# Operação *Max*

- Passos:

- Basta retornar o valor armazenado na primeira posição do *heap*.

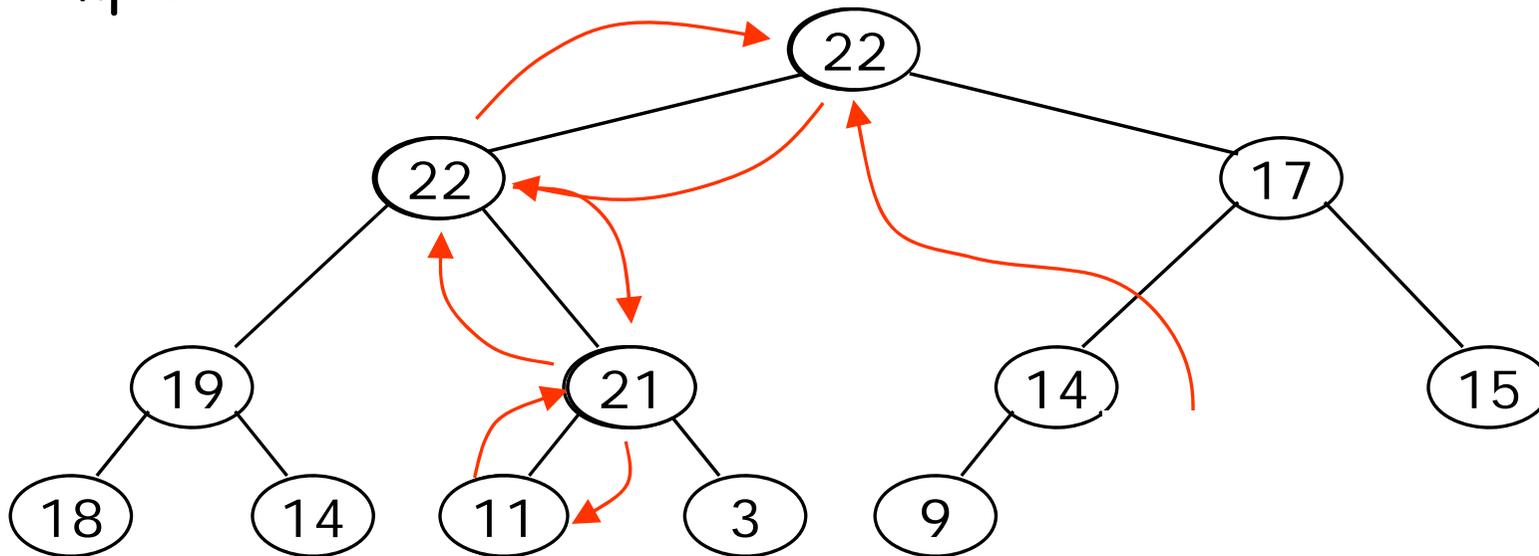
```
Max(){  
    return v[1];  
}
```

Tempo: *constante*

Um *heap* com operação *Min* é análogo.

# Remoção da raiz

- Remover a raiz equivale a extrair o máximo valor presente no *heap*.
- Em seguida, colocaremos em seu lugar a última folha e recuperaremos a propriedade estrutural com a aplicação de vários *Sifts*.
- Exemplo:



# Operação *ExtractMax*

## ■ Passos:

- 1) Substituir a raiz pelo último elemento do *heap*.
- 2) Decrementar o seu tamanho atual.
- 3) Chamar *Sift* desde a raiz.

```
ExtractMax(){  
    if (size < 1)  
        Erro("heap underflow");  
    else {  
        max = v[1];  
        v[1] = v[size--];  
        Sift(1, size);  
        return max;  
    }  
}
```

Tempo: *logarítmico*

# Operação *Modify(k, x)*

## ■ Passos:

- 1) Modificar a posição correspondente.
- 2) Se a propriedade estrutural for perdida, ir trocando de valor para cima ou para baixo da árvore, até "consertar" o *heap*.

```
Modify(k, x){
  if (k > size || k < 1)
    Erro("Index error");
  else {
    v[k] = x;
    while (k > 1 && v[⌊k/2⌋] < v[k]) { //conserta para cima
      aux = v[k];
      v[k] = v[⌊k/2⌋];
      v[⌊k/2⌋] = aux;
      k = ⌊k/2⌋;
    }
    Sift(k, size); // ou conserta para baixo
  }
}
```

Tempo: *logarítmico*

# Operação *Insert(x)*

- Passos:

- 1) Aumentar uma posição no final do *heap*.
- 2) Chamar *Modify* nessa posição, colocando o valor que se deseja inserir.

```
Insert(x){  
    Modify(++size, x);  
}
```

Tempo: *logarítmico*

# Sumário

---

- A tabela abaixo indica as complexidades de tempo das operações de uma fila de prioridades implementada com um *heap* :

<u>Operação</u>	<u>Tempo</u>
<i>Build</i>	Linear
<i>Max</i> (ou <i>Min</i> )	Constante
<i>ExtractMax</i> (ou <i>ExtractMin</i> )	Logarítmico
<i>Modify</i>	Logarítmico
<i>Insert</i>	Logarítmico

# Exercícios



- Implemente uma fila de prioridades utilizando lista ligada e calcule a complexidade de tempo de cada uma das suas operações.
- Compare esses resultados com a implementação que utiliza *heap*.

# CES-11



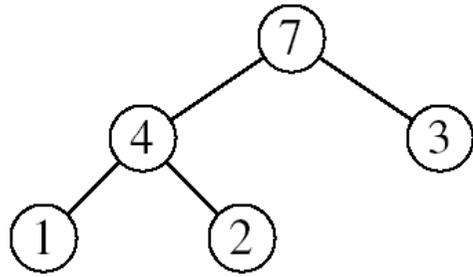
- Algoritmos de Ordenação
  - *Heap*
  - Filas de prioridade
  - *HeapSort*

# *HeapSort* (Williams, 1964)

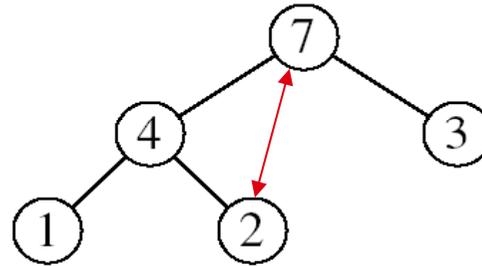
- A estrutura de *heap* permite a elaboração de um eficiente algoritmo de ordenação.
- Ideia:
  - 1) Transformar o vetor inicial em um *heap*.
  - 2) Laço de repetição:
    - a) Trocar a raiz (elemento de valor máximo) com o último elemento do *heap*.
    - b) Desconsiderar esse último elemento.
    - c) Chamar *Sift* para os demais elementos do vetor.

# Exemplo

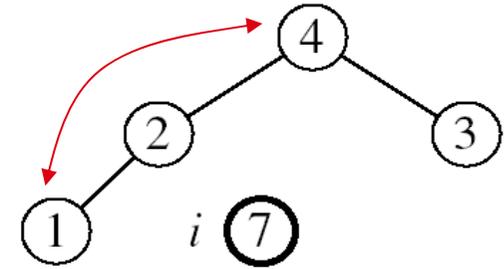
Heap já construído:



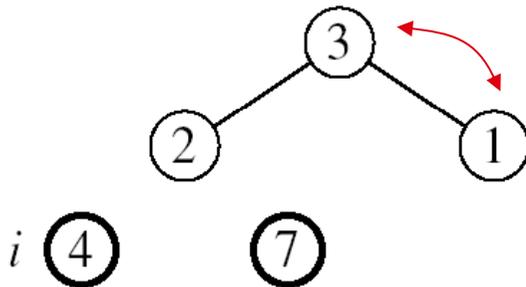
7	4	3	1	2
---	---	---	---	---



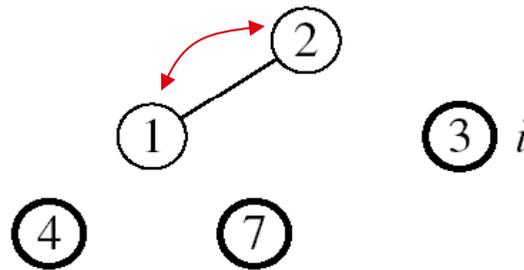
Sift(1, 4)



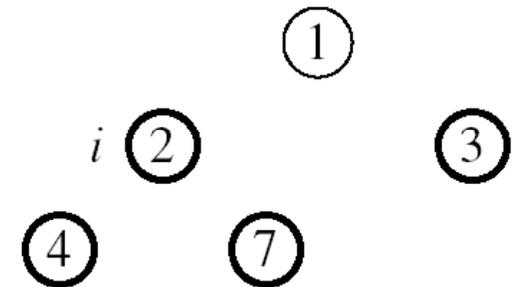
Sift(1, 3)



Sift(1, 2)



Sift(1, 1)



1	2	3	4	7
---	---	---	---	---

# Algoritmo *HeapSort*

```
HeapSort(v) {  
    Build(v);  
    for (i=n; i>1; i--) {  
        aux = v[i];  
        v[i] = v[1];  
        v[1] = aux;  
        Sift(1, i-1);  
    }  
}
```

- Complexidade de tempo:
  - *Build*:  $O(n)$
  - $n-1$  *Sifts*:  $O(n \cdot \log n)$
- Total:  $O(n \cdot \log n)$