5. Algoritmos de Ordenação

Prof. Renato Tinós

Local: Depto. de Computação e Matemática
(FFCLRP/USP)

Principais Tópicos

5.1. Ordenação por Inserção
5.2. Ordenação por Seleção
5.3. Método da Bolha
5.4. Ordenação por Fusão
5.5. Heapsort
    5.5.1. Estrutura Heap
    5.5.2. Ordenação por Heapsort
5.6. Quicksort
5.7. Considerações sobre o Problema de Ordenação
5.8. Ordenação em Tempo Linear

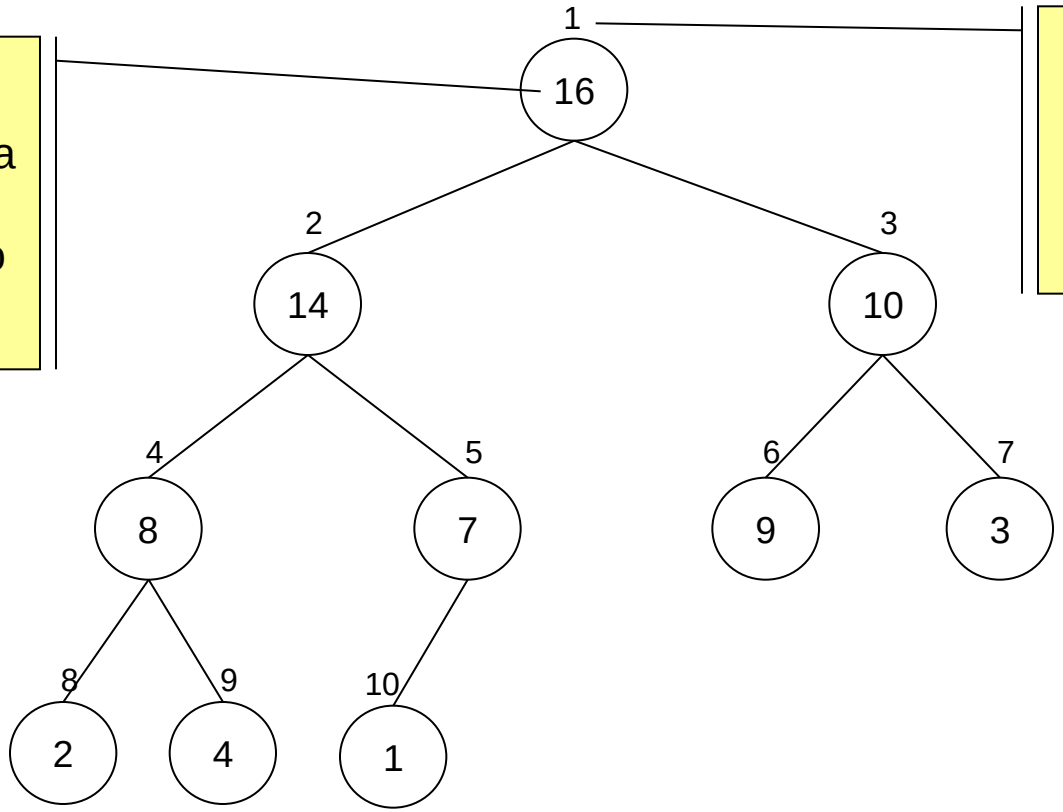
5.5.1. Estrutura Heap

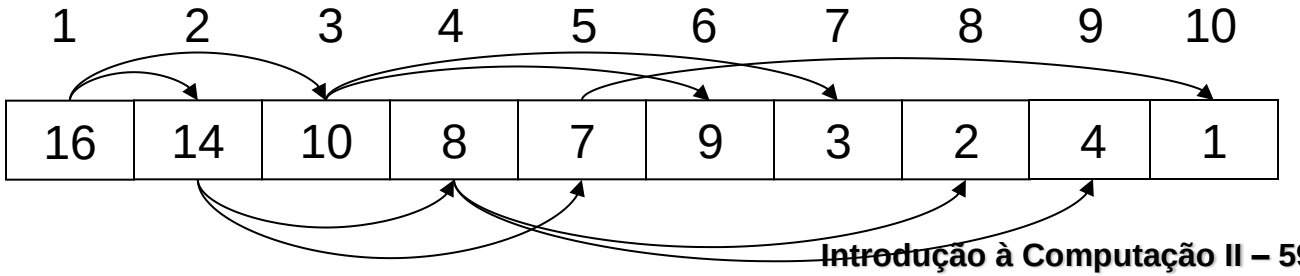
- A estrutura de dados **heap** (binário) é um objeto arranjo (vetor) que representa uma árvore binária
  - Cada nó da árvore corresponde a um elemento do vetor
  - A árvore é completamente preenchida em todos os níveis, exceto o nível mais baixo, que pode ou não ser totalmente preenchido

5.5.1. Estrutura Heap

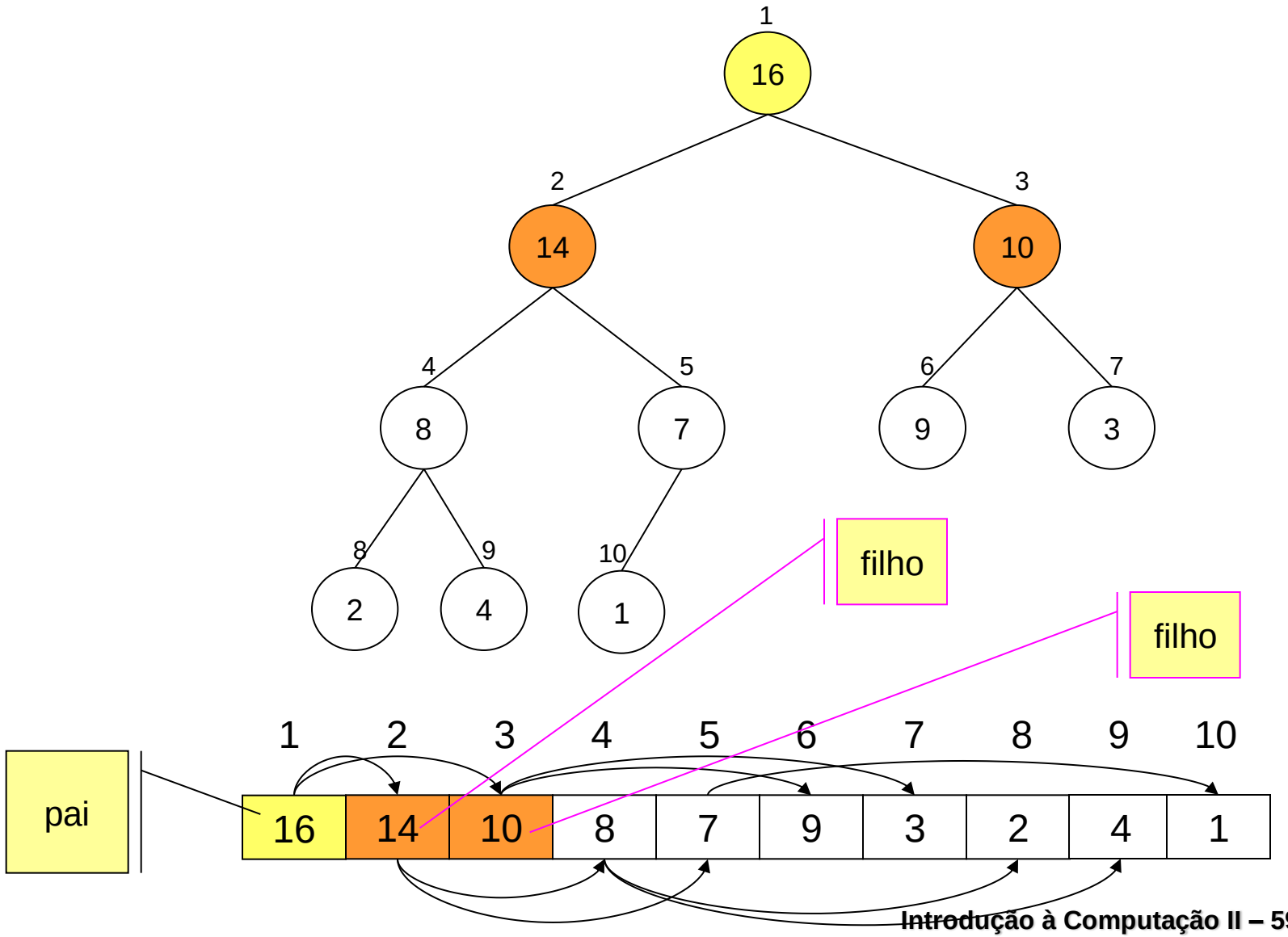
O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó

O número acima de um nó é o índice correspondente no vetor





5.5.1. Estrutura Heap



5.5.1. Estrutura Heap

- Um vetor **a** que representa um heap é um objeto com dois atributos:
  - *N = comprimento*(**a**), que é o número de elementos do vetor e
  - R = *tamanho-do-heap*(**a**), que é o número de elementos no heap armazenado no vetor **a**
- Assim, embora *a[1], ...a[N]* possa conter valores válidos, nenhum elemento além de *a[R],* na qual *R*<=*N*, é um elemento do heap
- A raiz da árvore é *a[1]* e, dado o índice *i* de um nó, os índices
  - de seu pai: PARENT(*i*) = $\lfloor i/2 \rfloor$
  - do filho da esquerda: LEFT(*i*) = $2 \times i$
  - do filho da direita: RIGHT(*i*) = $2 \times i + 1$

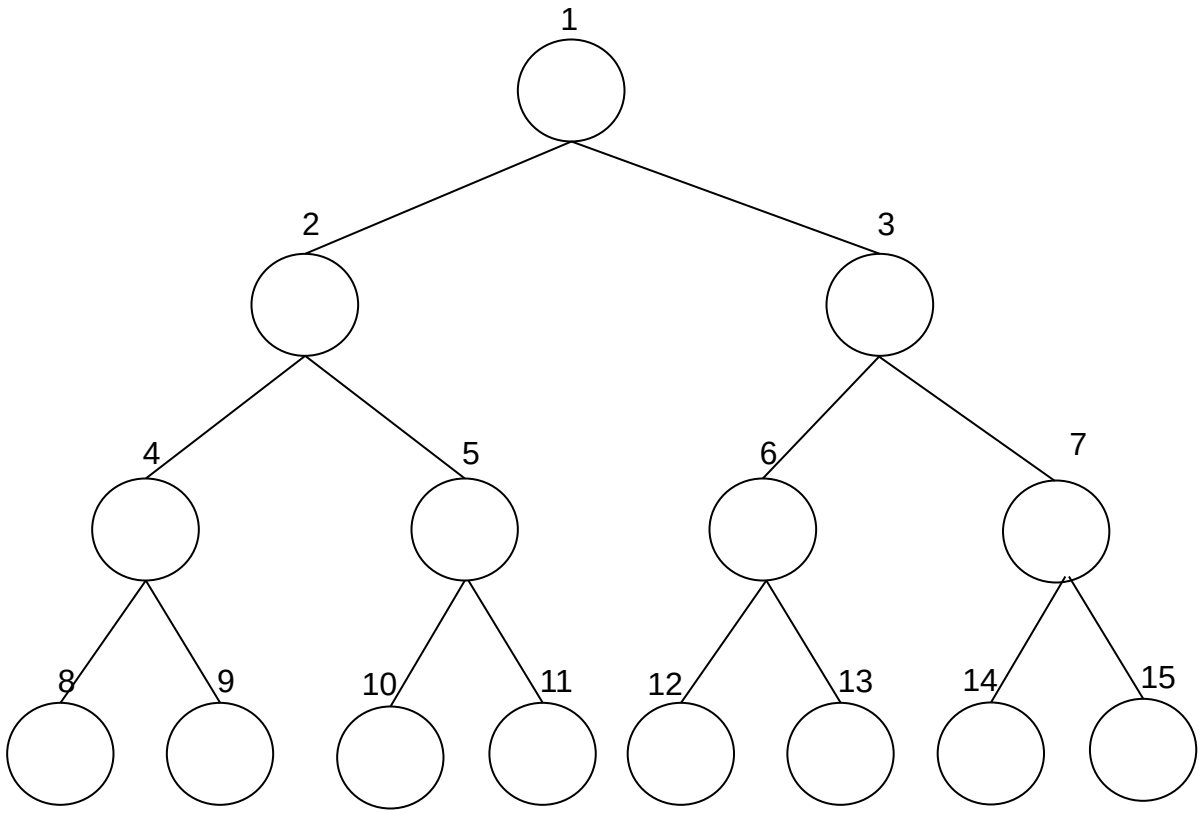
5.5.1. Estrutura Heap

- Existem dois tipos de heaps binários: **heaps máximos** e **heaps mínimos**
  - Em ambos os tipos, os valores nos nós satisfazem a uma **propriedade de heap**, cujos detalhes específicos dependem do tipo de heap
- Em um **heap máximo**, a propriedade de heap máximo é que para todo nó *i* diferente da raiz:
  - *a[PARENT(i)] >= a[i]*
    - isto é, o valor de um nó é, no máximo, o valor de seu pai.
      - o maior elemento em um heap máximo é armazenado na raiz, e
      - a subárvore que tem raiz em um nó contém valores menores que o próprio nó
- Um **heap mínimo** é organizado de modo oposto; a propriedade de heap mínimo é que para todo nó ***i*** diferente da raiz
  - *a[PARENT(i)] <= a[i]*
    - o menor elemento de um heap mínimo está na raiz
- Para o algoritmo Heapsort, utilizaremos heaps máximos
  - toda referência a um heap deste ponto em diante se refere a um heap máximo

5.5.1. Estrutura Heap

- A **altura** de um nó em um heap é o número de arestas (ou arcos) no caminho descendente simples mais longo desde o nó até uma folha
  - A **altura** do heap é a altura de sua raiz
- Tendo em vista que o heap é completamente preenchido em todos os níveis, com possível exceção do último nível, temos que sua altura é a altura de uma árvore binária:
  - Último nível com 1 nó:
    $$h = \log_2(N)$$
  - Último nível completo:
    $$h = \log_2(N+1) - 1$$
  - Generalizando:
    $$h = \lfloor \log_2 N \rfloor$$
- Assim, as operações básicas sobre heaps são executadas em um tempo máximo proporcional à altura da árvore, e assim tem complexidade $O(\log_2 N)$

5.5.1. Estrutura Heap



Altura: h_i	Número de nós na camada i: N_i
0	1=2^0
1	2=2^1
2	4=2^2
3	8=2^3

total de nós no heap:
N=soma{i=0 até h}(N_i)
N=soma{i=0 até h}(2^i)
N=soma{i=0 até h}(2^i)
N=2^(h+1)-1

2^(h+1)=N+1
h+1=log_2 (N+1)
h=log_2 (N+1) -1

5.5.1. Estrutura Heap

- Os elementos *a[m], ..., a[N]* de um vetor **a** sendo $m = \lfloor N/2 \rfloor + 1$ satisfazem as propriedades de um heap, uma vez que nenhum par de índices (*i,j*) é tal que *j = 2i* ou *j = 2i+1*

- Esses elementos formam a linha inferior da árvore binária a eles associada, entre os quais nenhuma relação de ordem é exigida

- O heap pode agora ser estendido para a esquerda, sendo que um novo elemento é incluído a cada passo e posicionado apropriadamente por meio de uma operação de escorregamento, que nos leva ao procedimento **heapify**

primeira função que vai precisar no heapSort

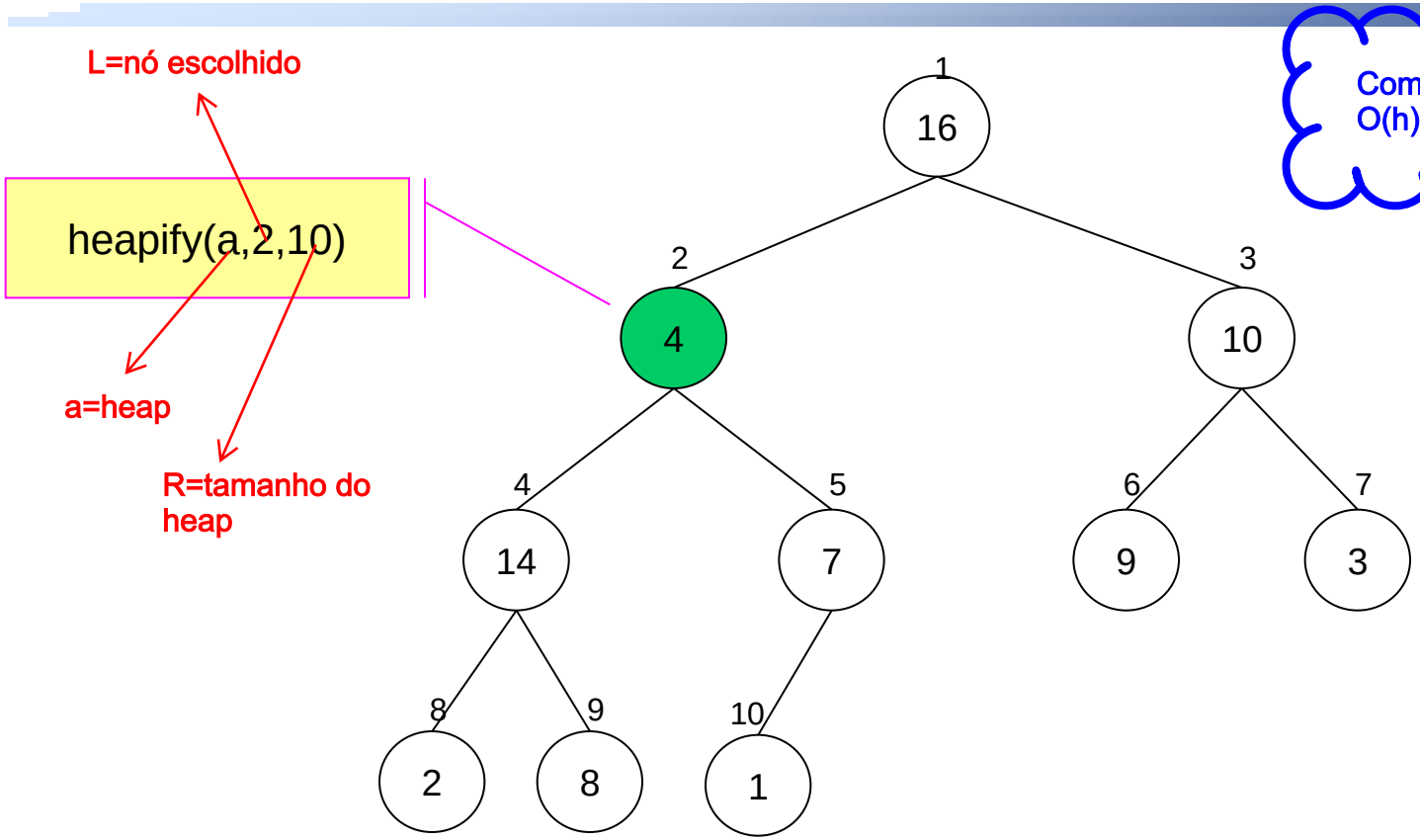
5.5.1. Estrutura Heap

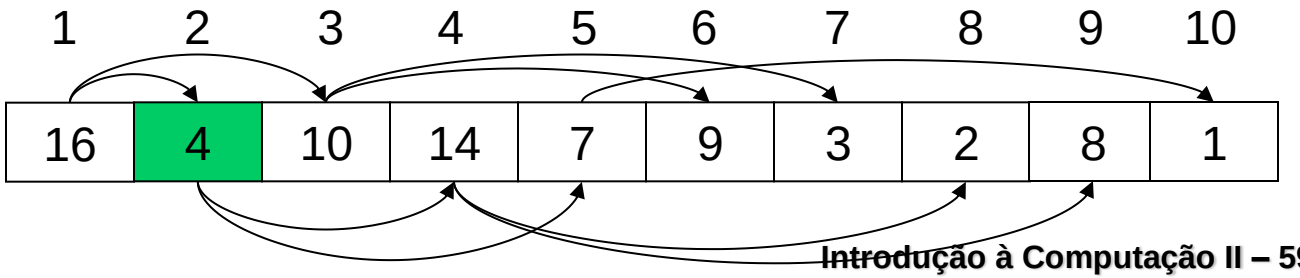
heapify(a[ ],L,R):
faz com que o elemento em a[L], do heap dado por a[1] até a[R], obedeça a propriedade do heap máximo

Algoritmo heapify(*a[ ] , L , R*)

```
i ← L
j ← 2*L
x ← a[ L ]
se ( ( j < R ) e ( a[ j ] < a[ j +1 ] ) )
        j ← j + 1
fim se
enquanto ( ( j ≤ R ) e ( x < a[ j ] ) )
        a[ i ] ← a[ j ]
        i ← j
        j ← 2*j
        se ( ( j < R ) e ( a[ j ] < a[ j+1 ] ) )
                j ← j + 1
        fim se
fim enquanto
a[ i ] ← x
```

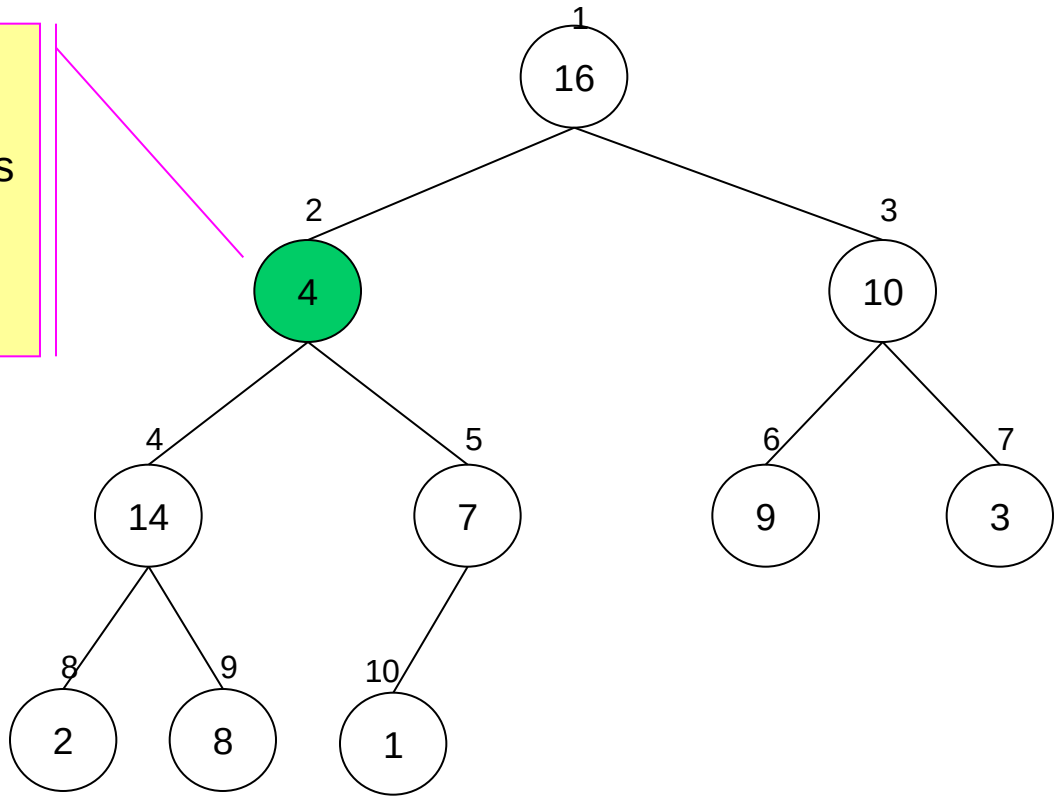
5.5.1. Estrutura Heap

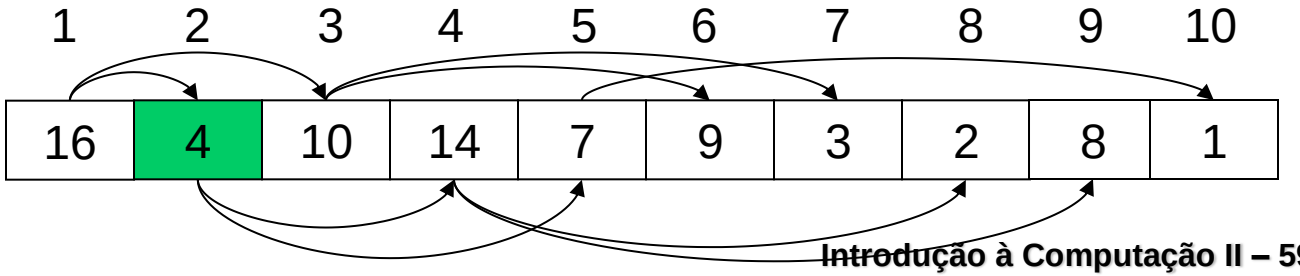




5.5.1. Estrutura Heap

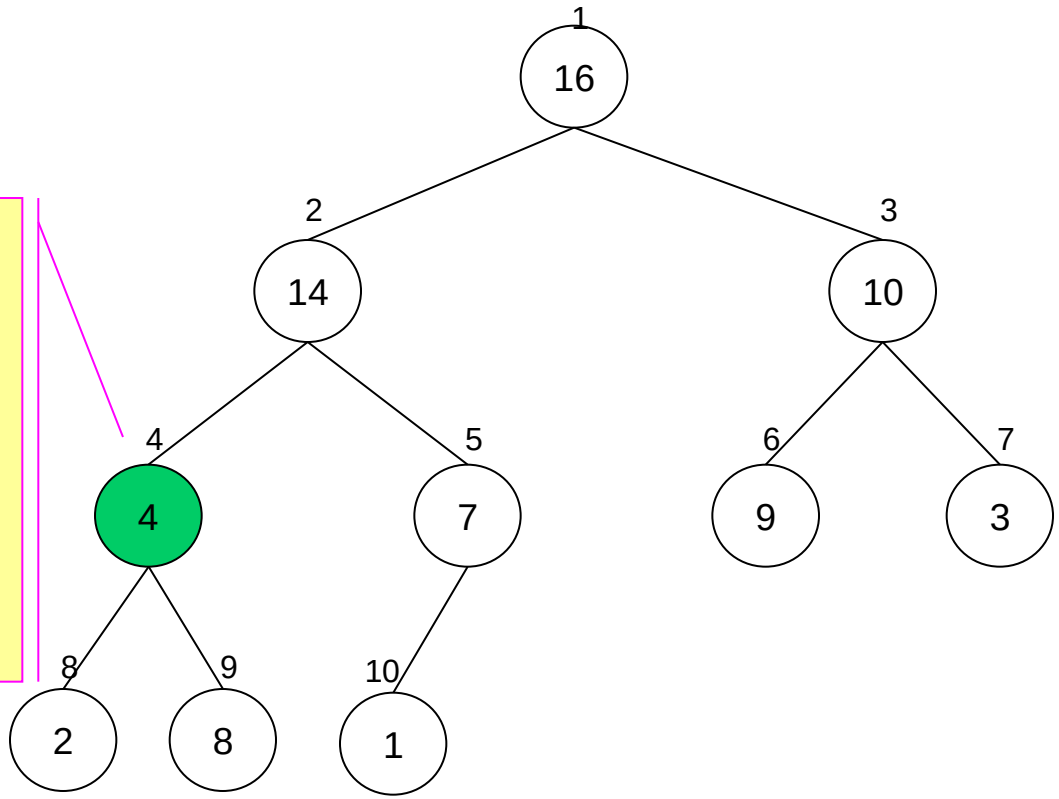
a[2] = 4 viola a propriedade de heap máximo, pois ele não é maior que ambos os filhos

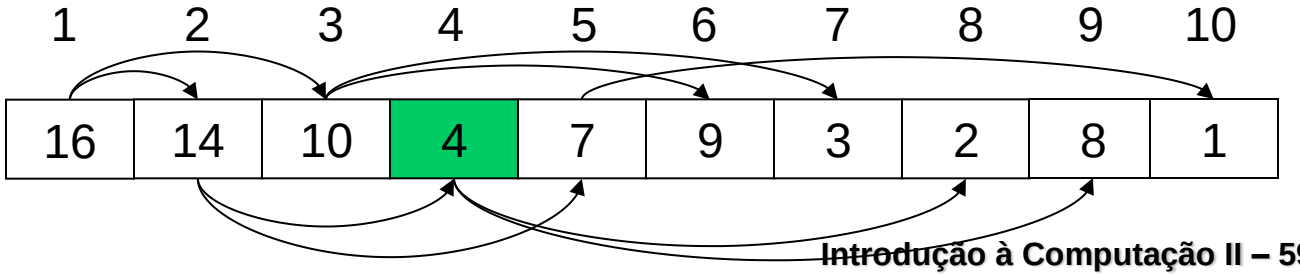




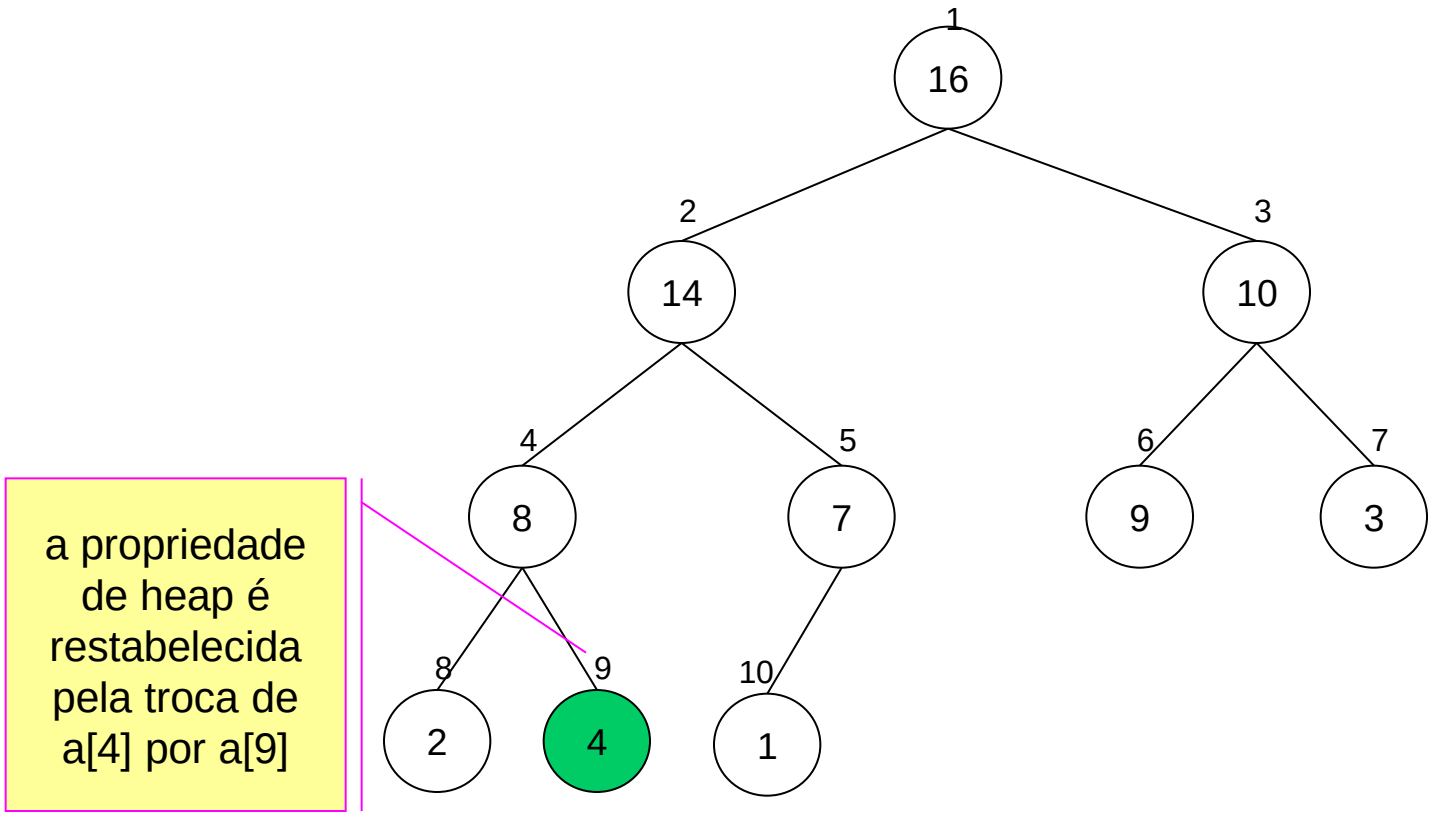
5.5.1. Estrutura Heap

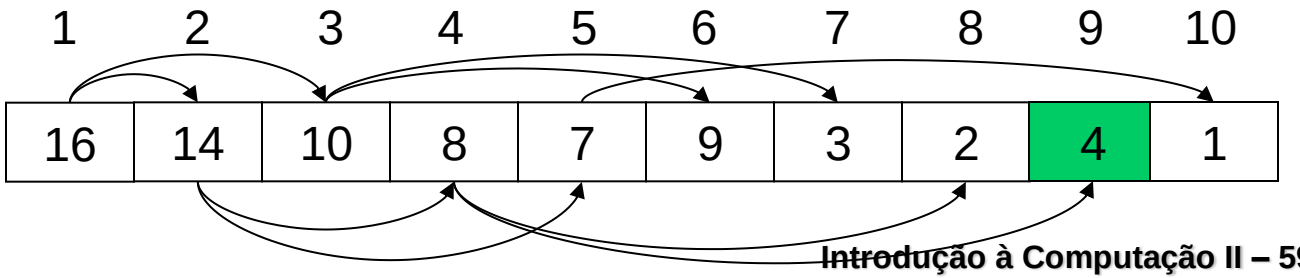
A propriedade de heap máximo é restabelecida para o nó 2 pela troca de a[2] por a[4], o que destrói a propriedade de heap para o nó 4





5.5.1. Estrutura Heap





5.5.1. Estrutura Heap

- Podemos utilizar o procedimento **heapify** de baixo para cima, a fim de converter um vetor **a**=[*a(1)...a(N)*] em um heap
- Os elementos *a(N/2+1),...,a(N)* são todos folhas da árvore, então cada um deles é um heap de 1 elemento com o qual podemos começar
- O procedimento para construção de um heap percorre os nós restantes da árvore e executa **heapify** sobre cada um

```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify(a,L,N)
fim para
```

5.5.1. Estrutura Heap

torna um heap qualquer em um heap máximo

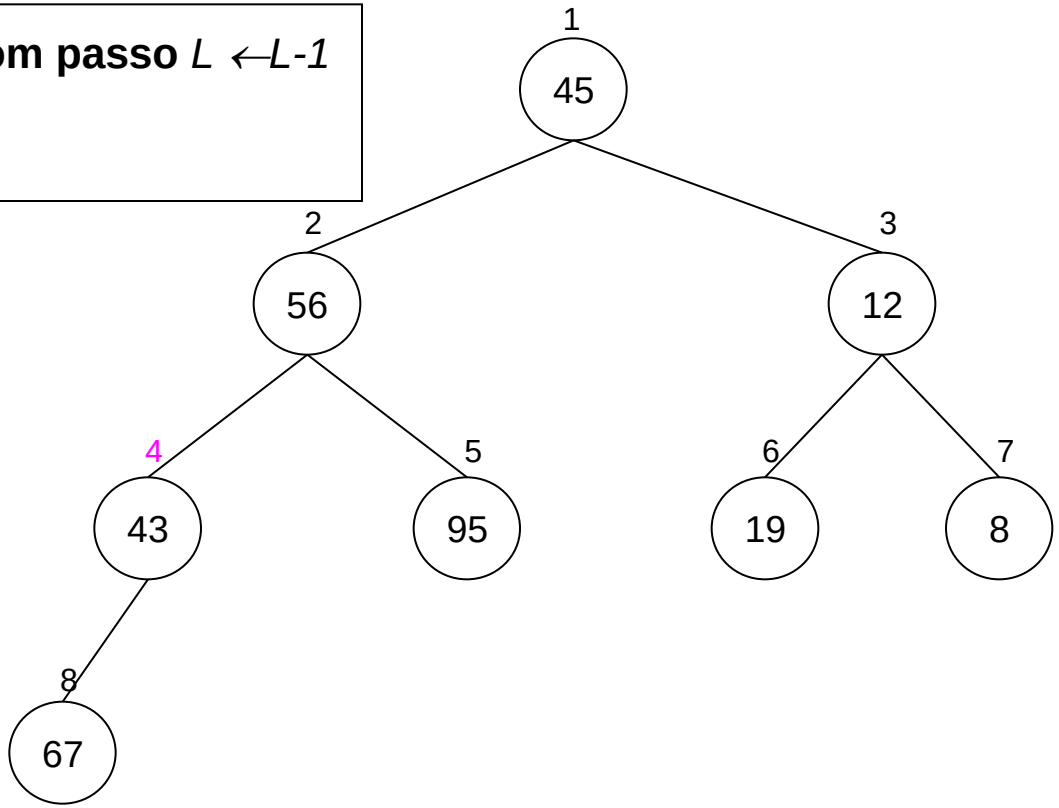
para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

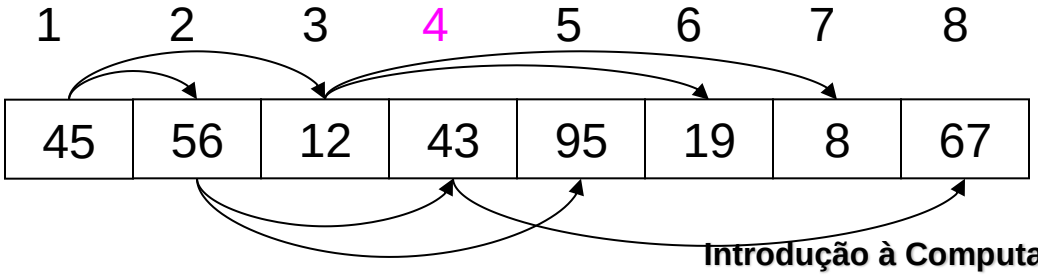
N = 8

L = 4

como heapify é O(log_2 N), então:
f(N)=N/2 * O(log_2 N)

então f(N) é O(N * log_2 N)



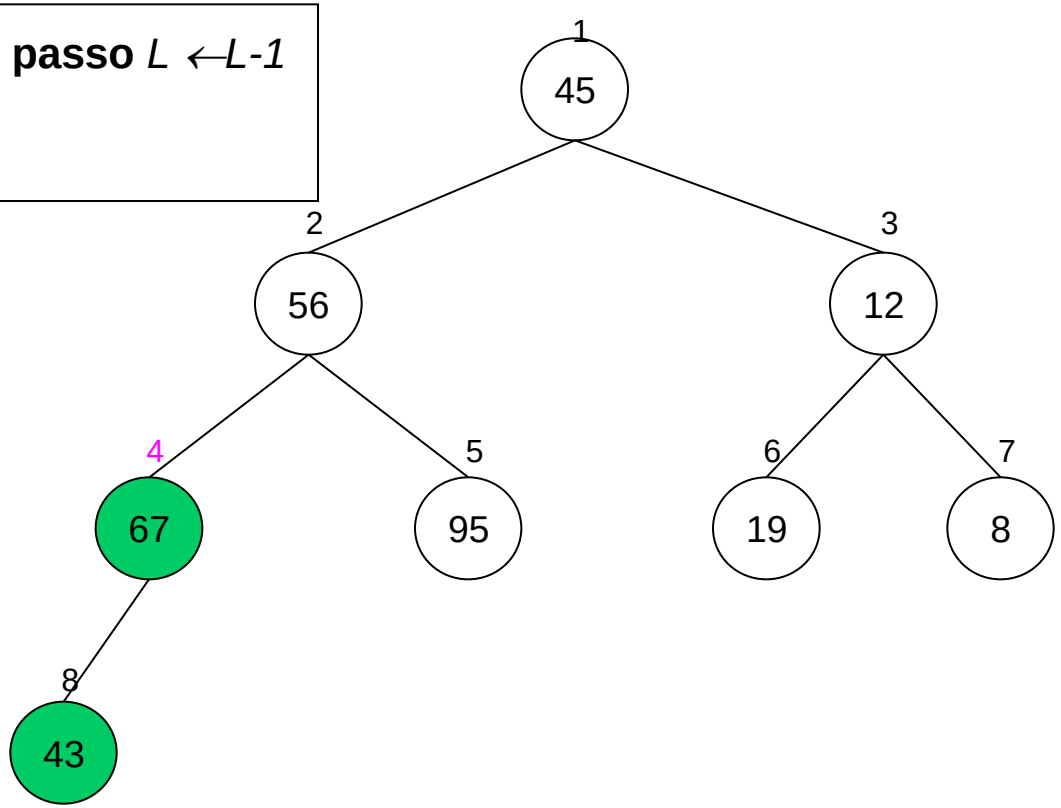


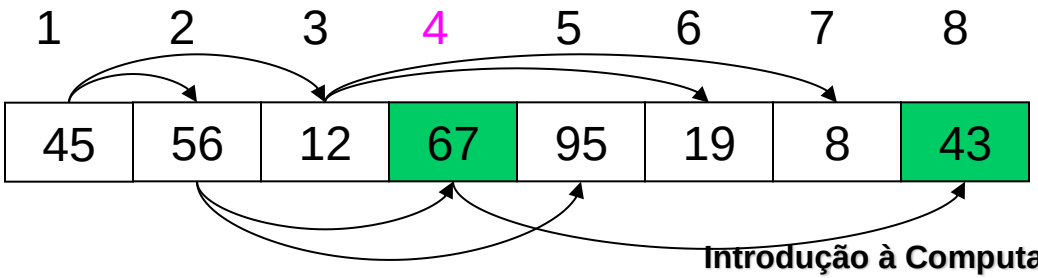
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 4



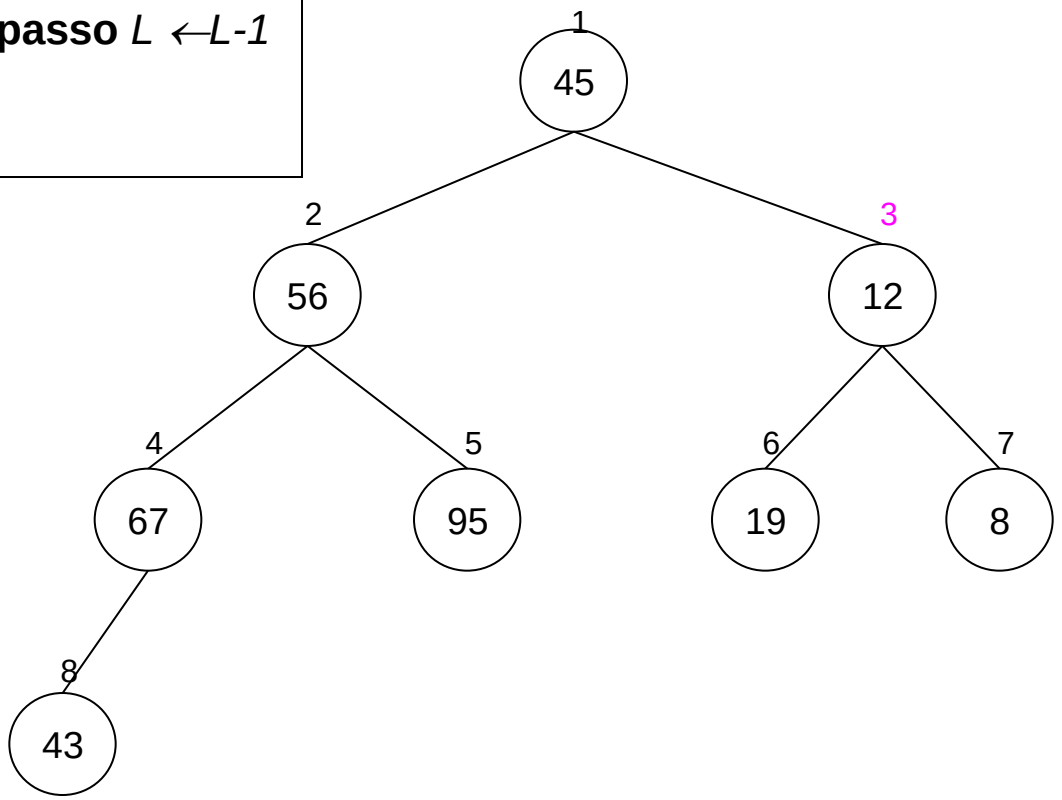


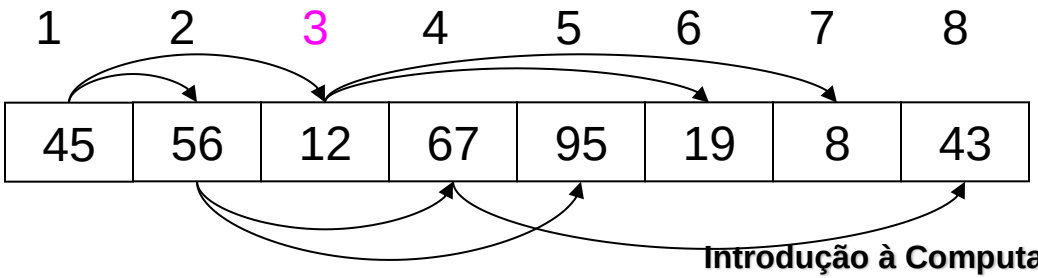
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 3



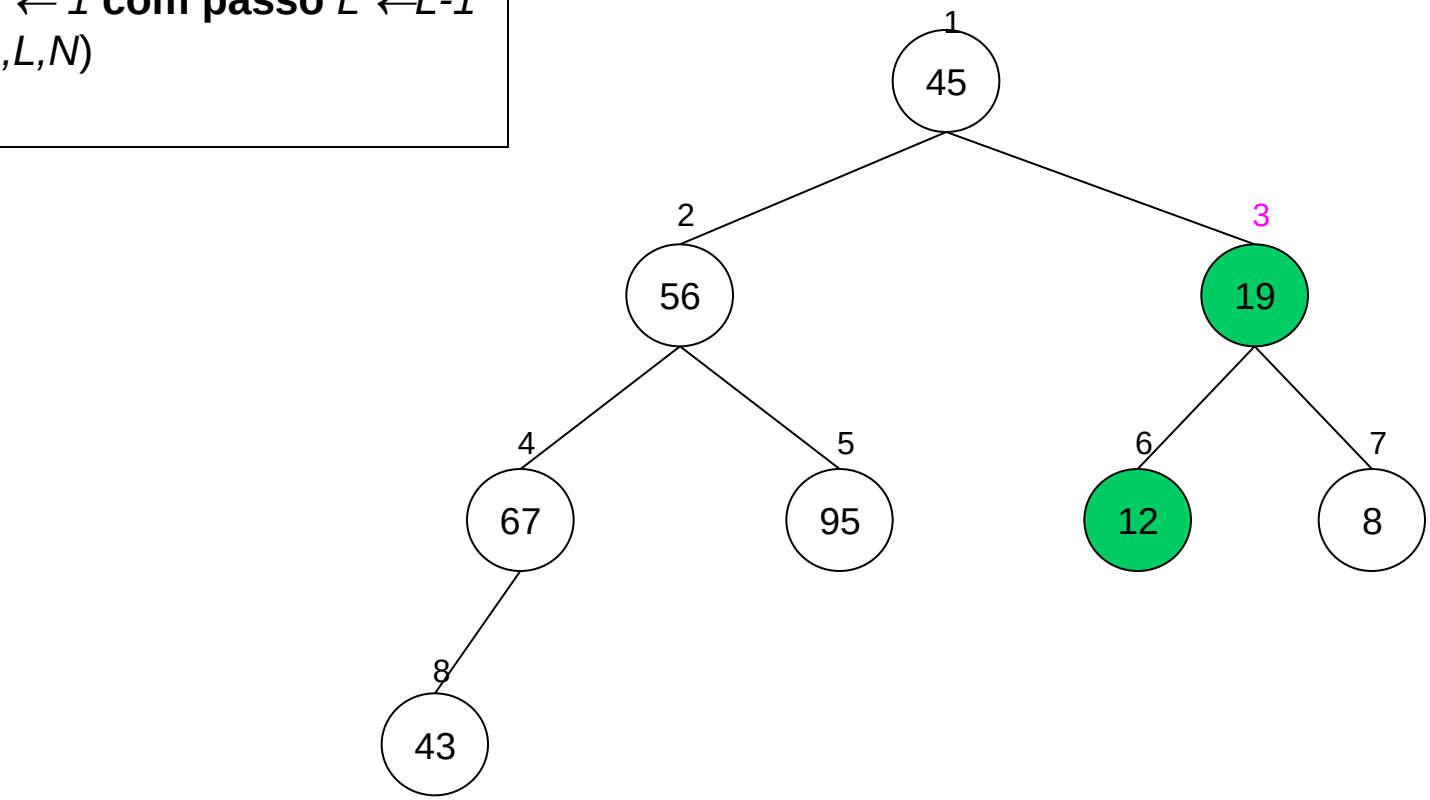


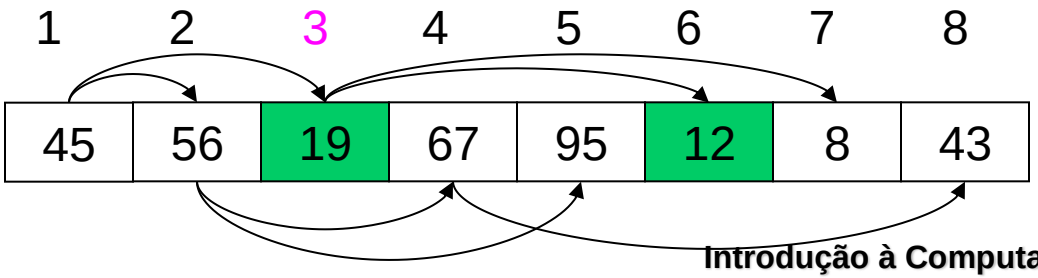
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 3



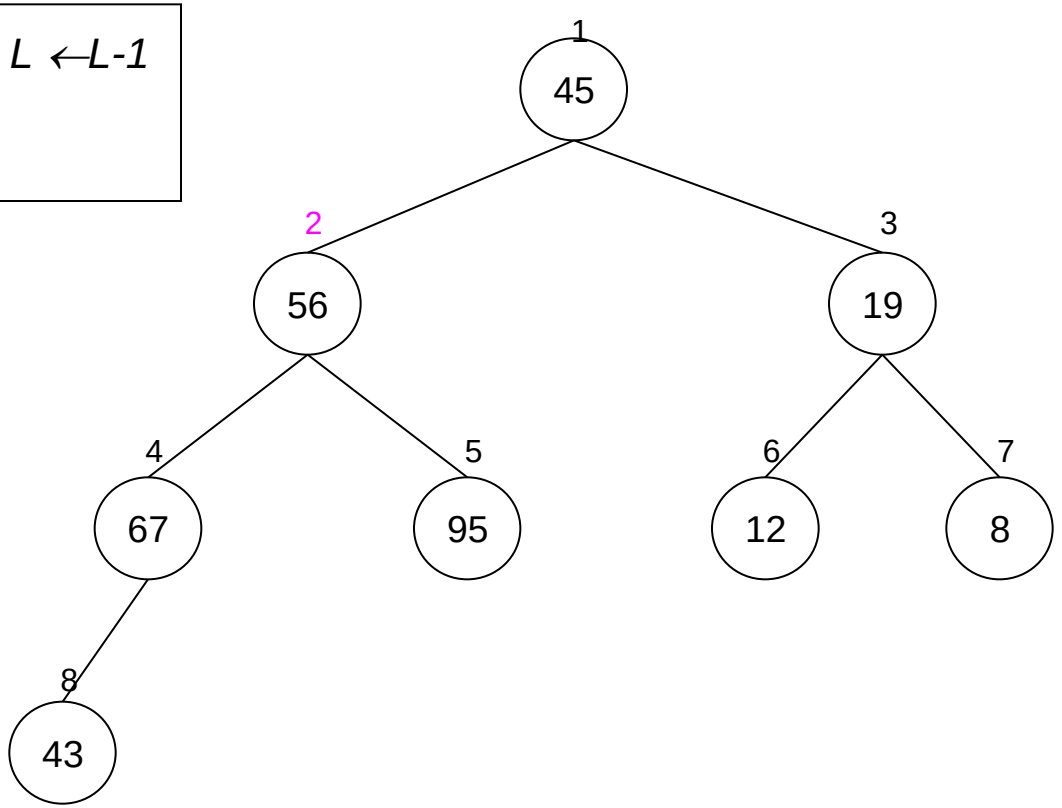


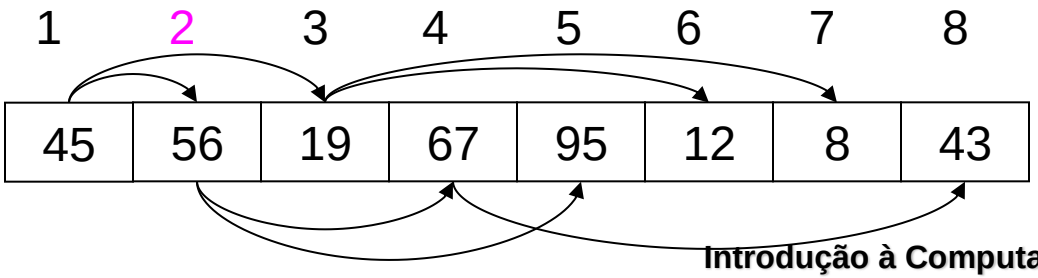
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 2



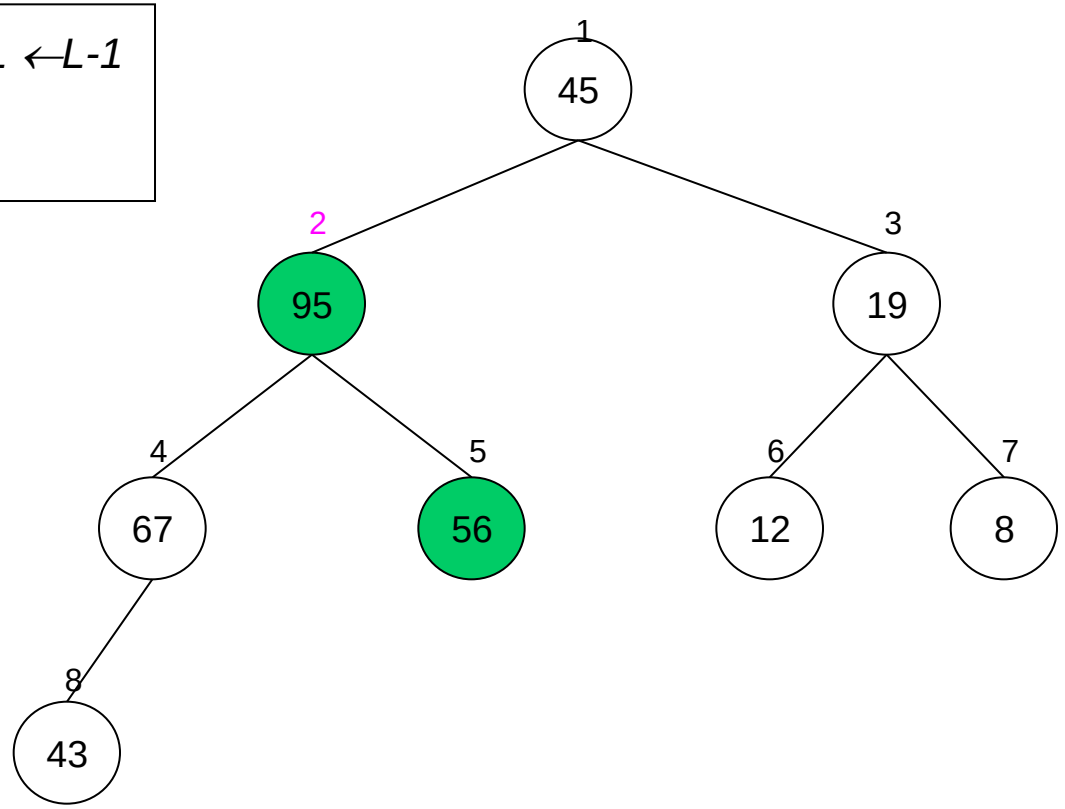


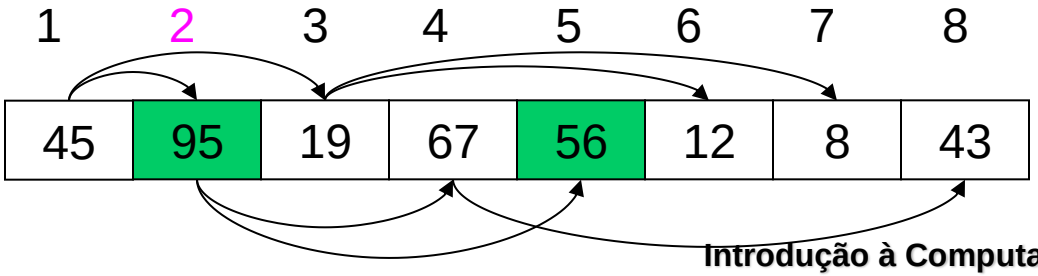
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 2



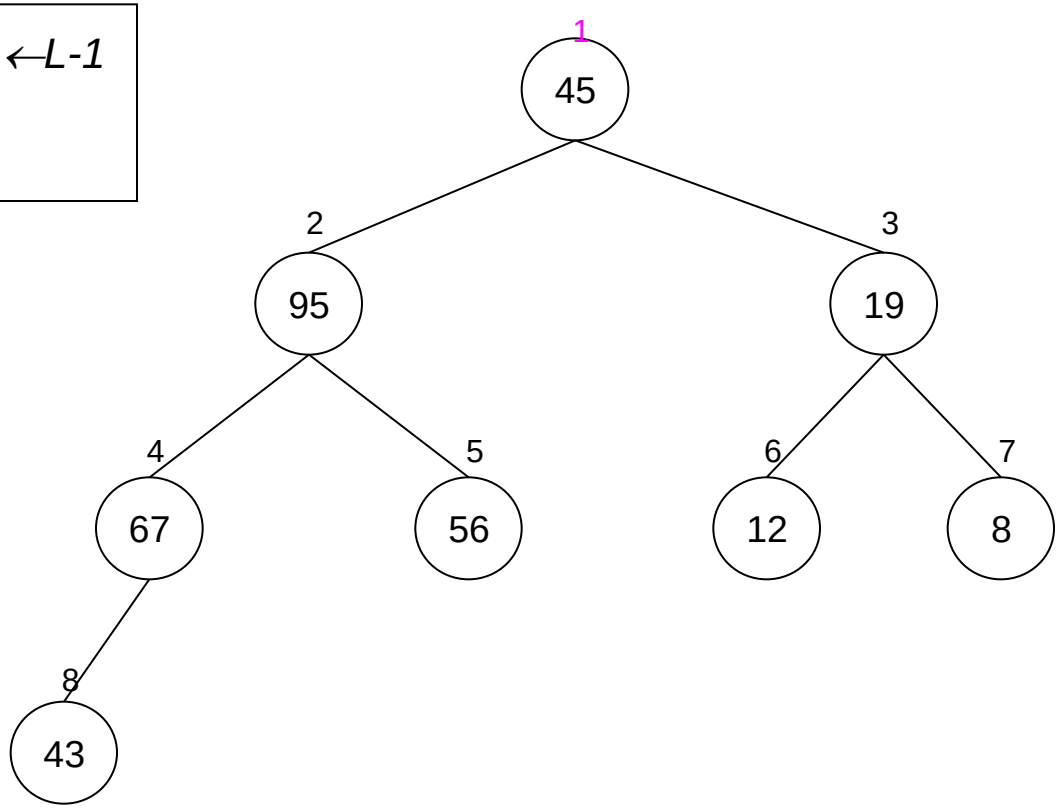


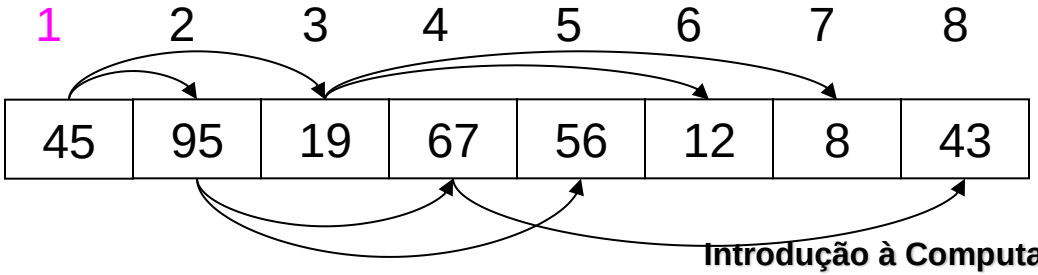
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 1



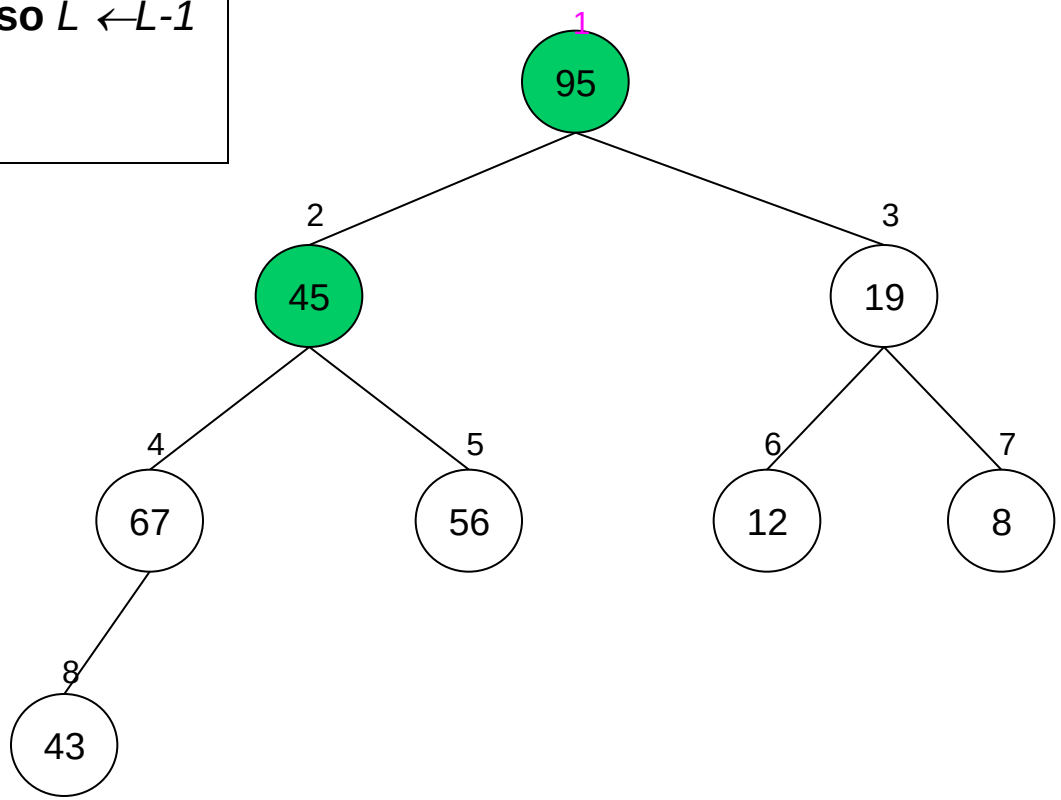


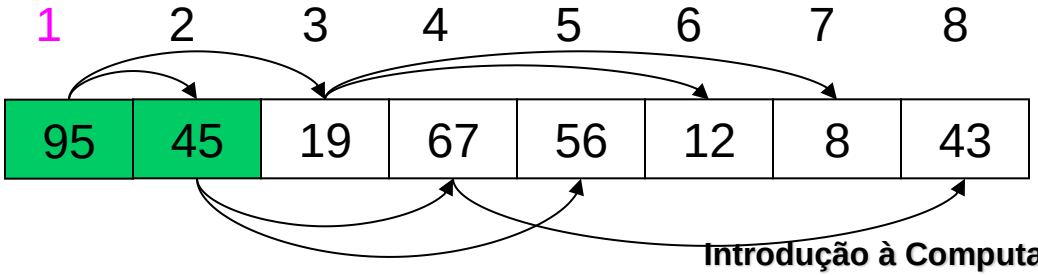
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a,L,N*)
fim para

N = 8

L = 1



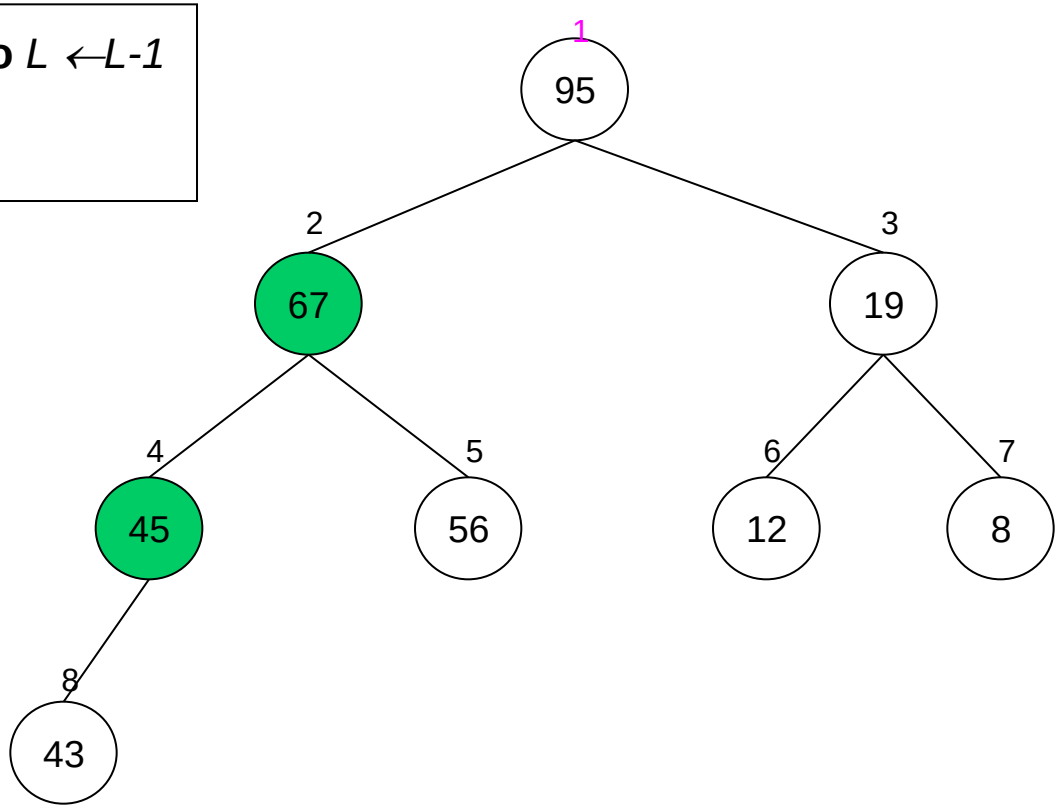


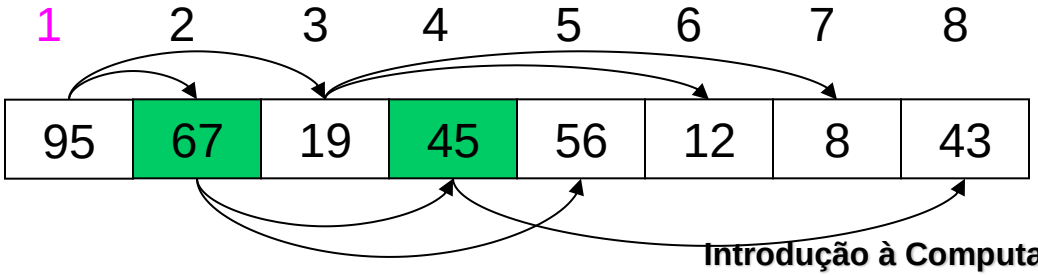
5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8

L = 1

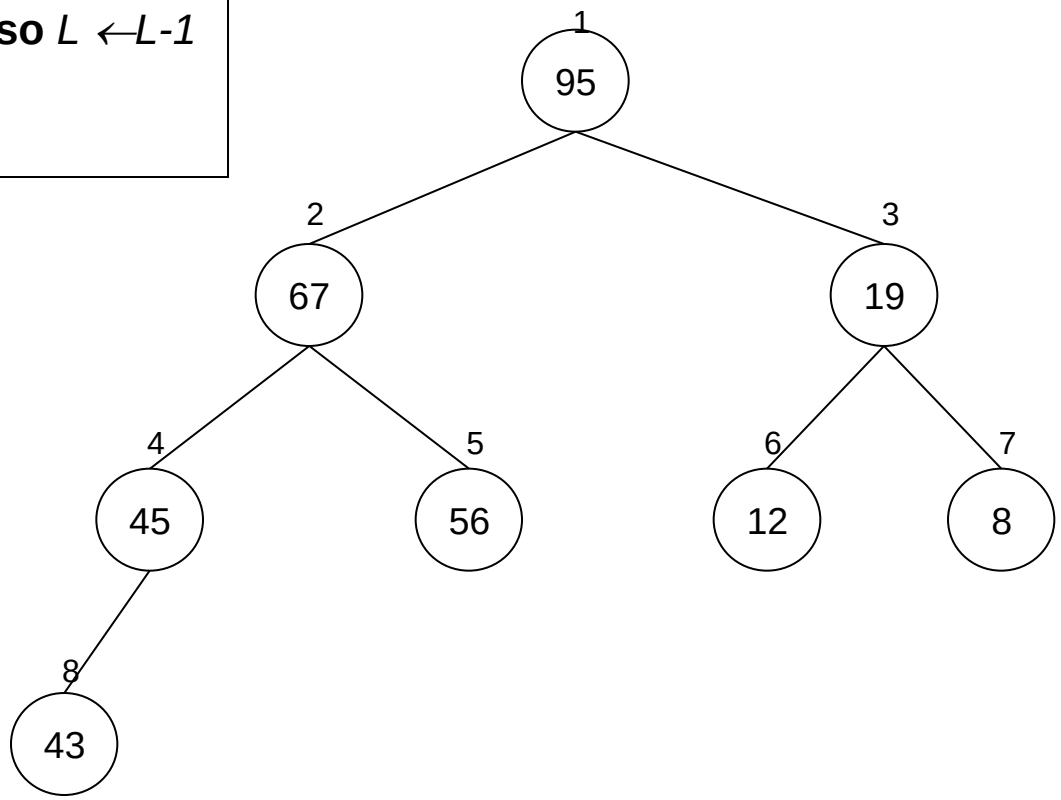


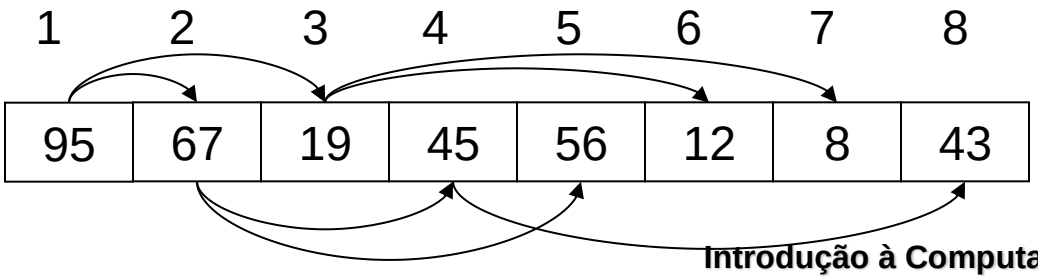


5.5.1. Estrutura Heap

para *L* ← *N/2* **até** *L* ← *1* **com passo** *L* ←*L-1*
heapify(*a*,*L*,*N*)
fim para

N = 8





5.5.2. Ordenação por Heapsort

- Com a finalidade de obter uma ordenação completa dos *N* elementos deve-se seguir *N* passos de escorregamento, em que, após a execução de cada passo, o novo elemento pode ser retirado do topo do heap

- Uma questão que surge é onde armazenar os elementos que emergem do topo e se seria possível uma ordenação diretamente em um único heap

- Tal solução existe: em cada passo, é necessário mover o elemento do topo do heap no nó liberado antes ocupado por *w* e permitir que *w* escorregue para a sua posição adequada

```
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```

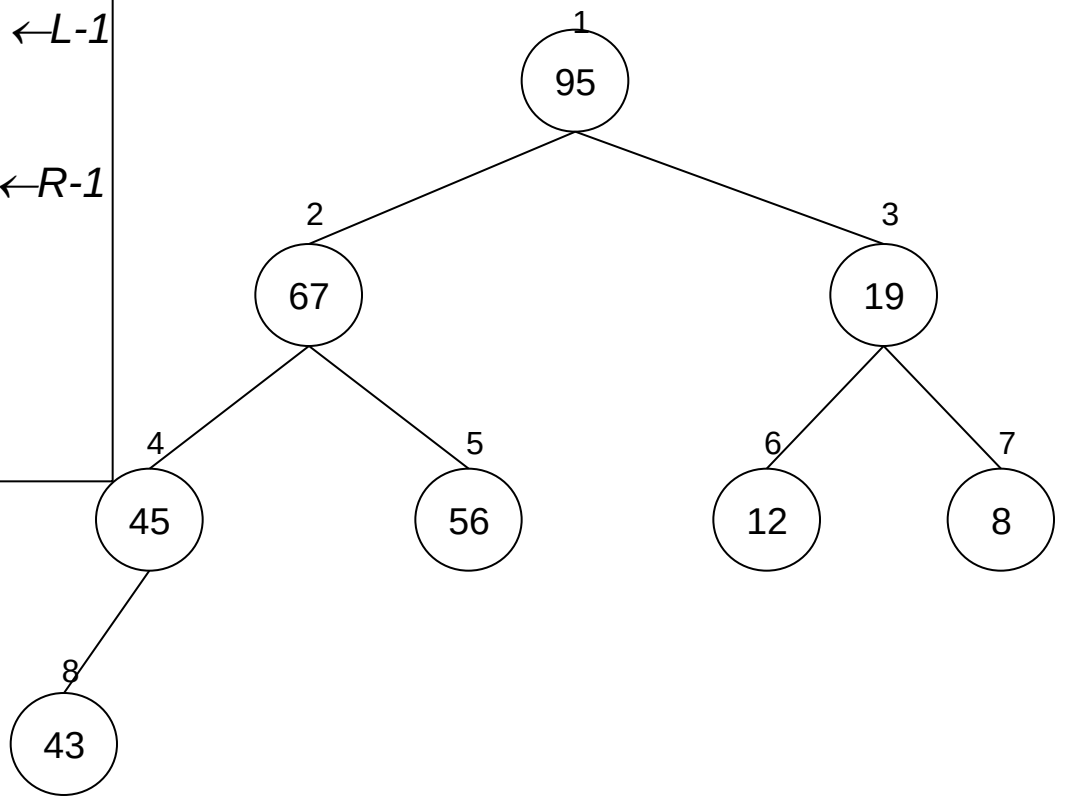
5.5.2. Ordenação por Heapsort

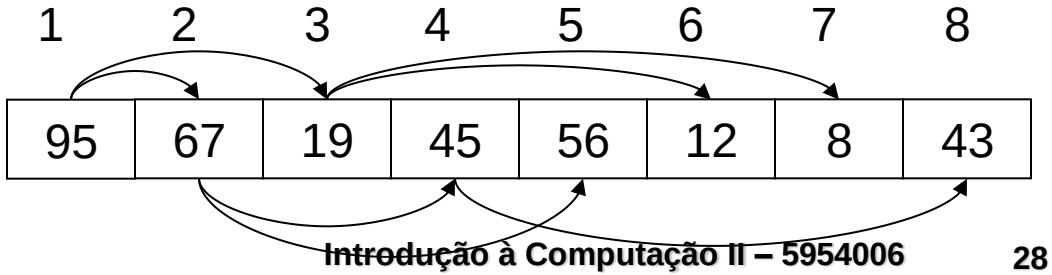
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```

segunda função para implementar heapSort

N = 8

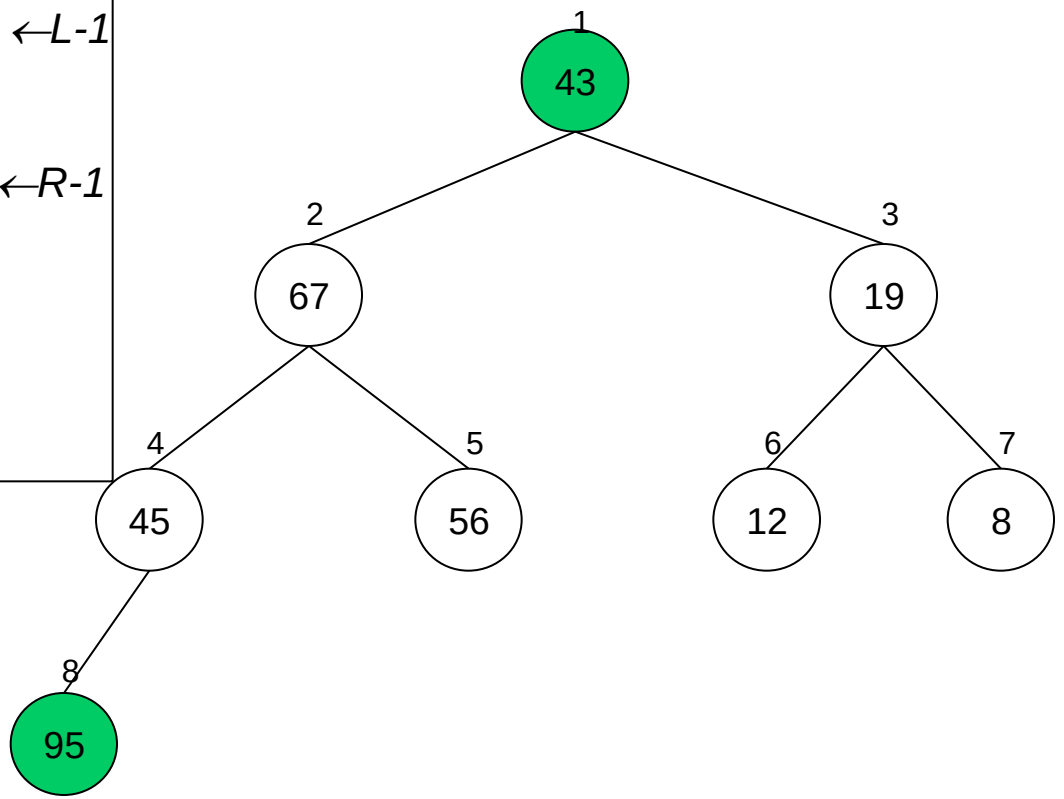
R = 8





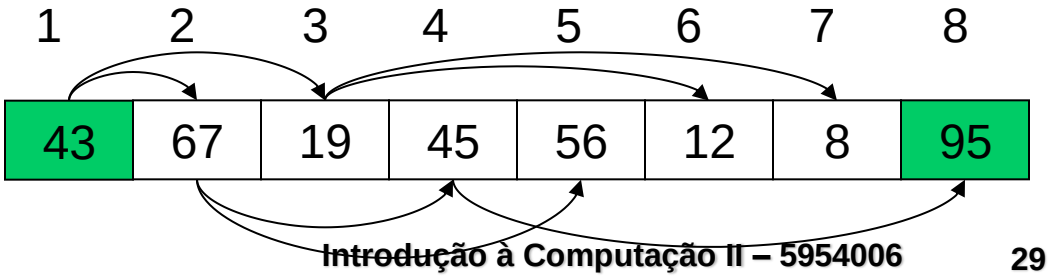
5.5.2. Ordenação por Heapsort

```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



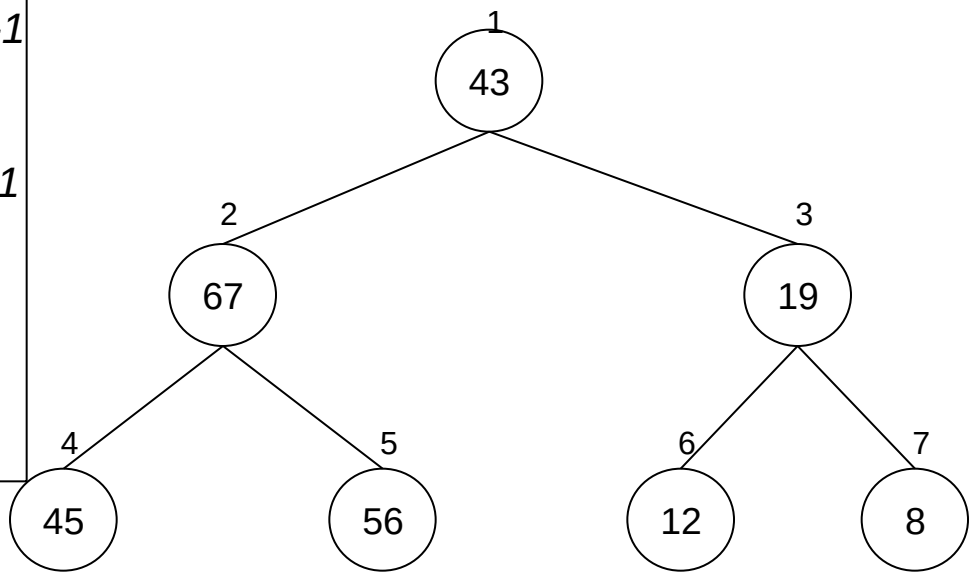
N = 8

R = 8



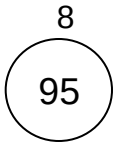
5.5.2. Ordenação por Heapsort

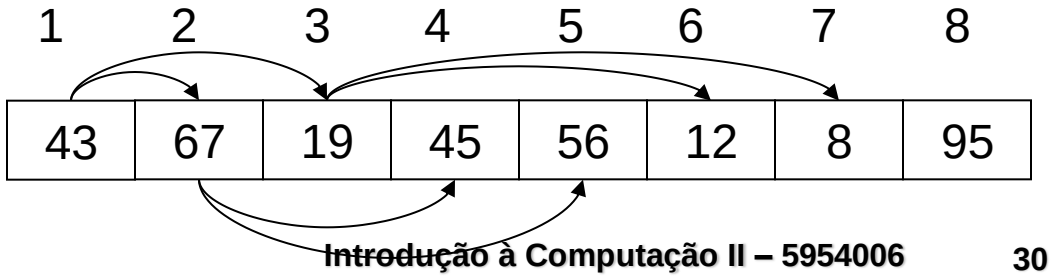
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

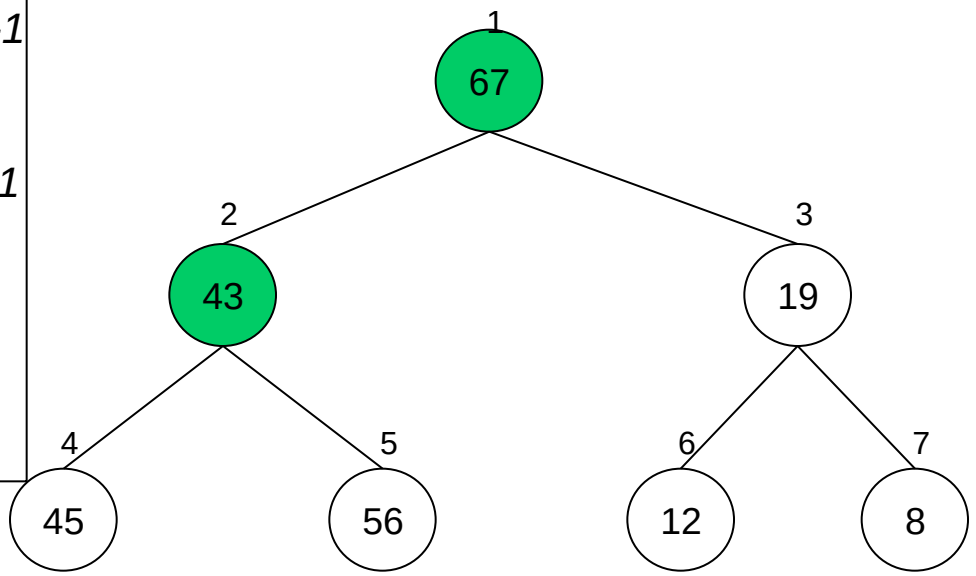
R = 8





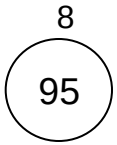
5.5.2. Ordenação por Heapsort

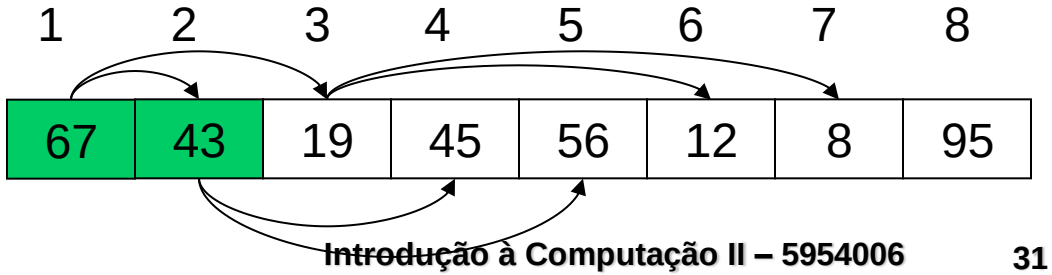
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

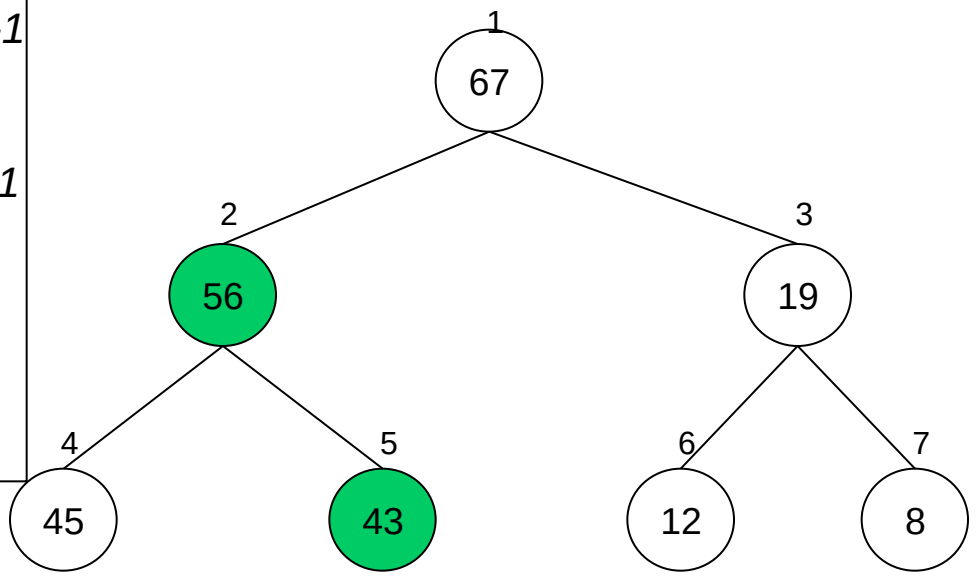
R = 8





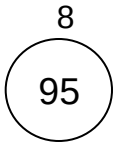
5.5.2. Ordenação por Heapsort

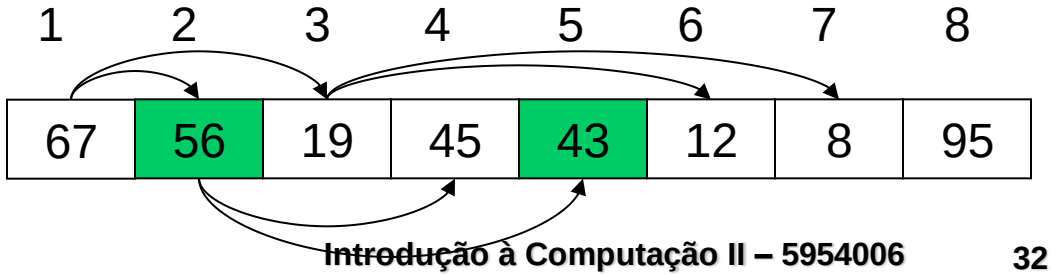
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

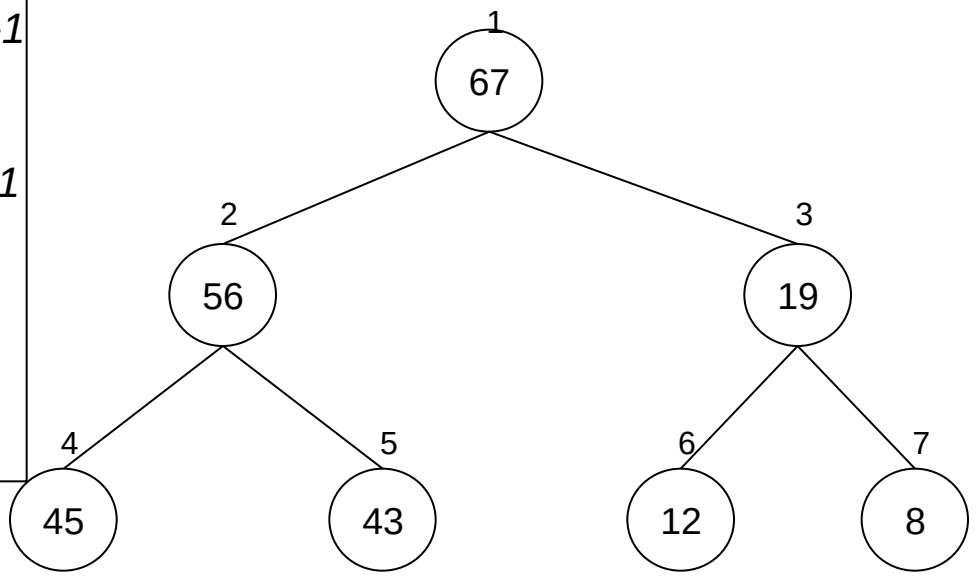
R = 8





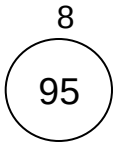
5.5.2. Ordenação por Heapsort

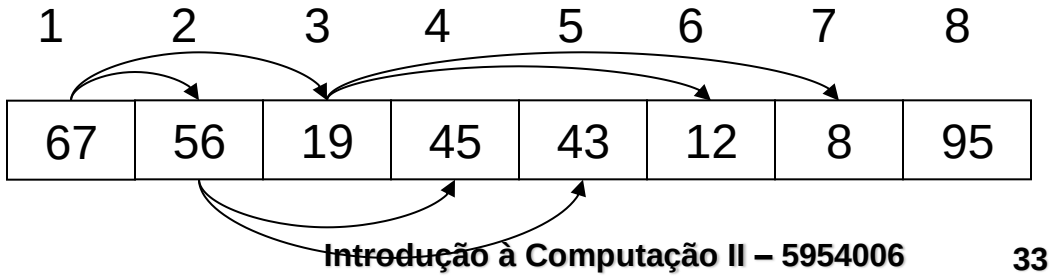
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

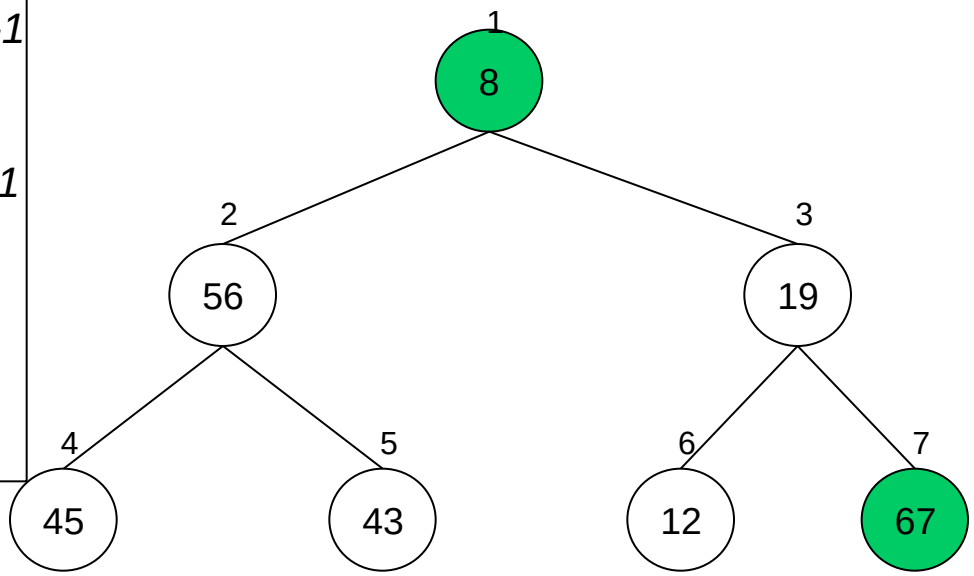
R = 7





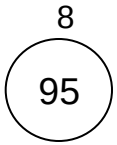
5.5.2. Ordenação por Heapsort

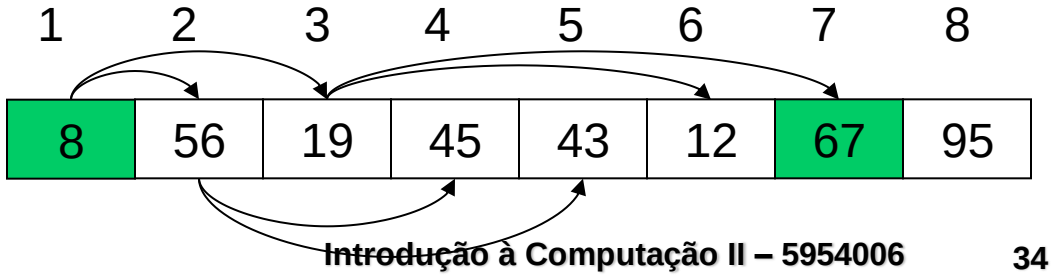
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

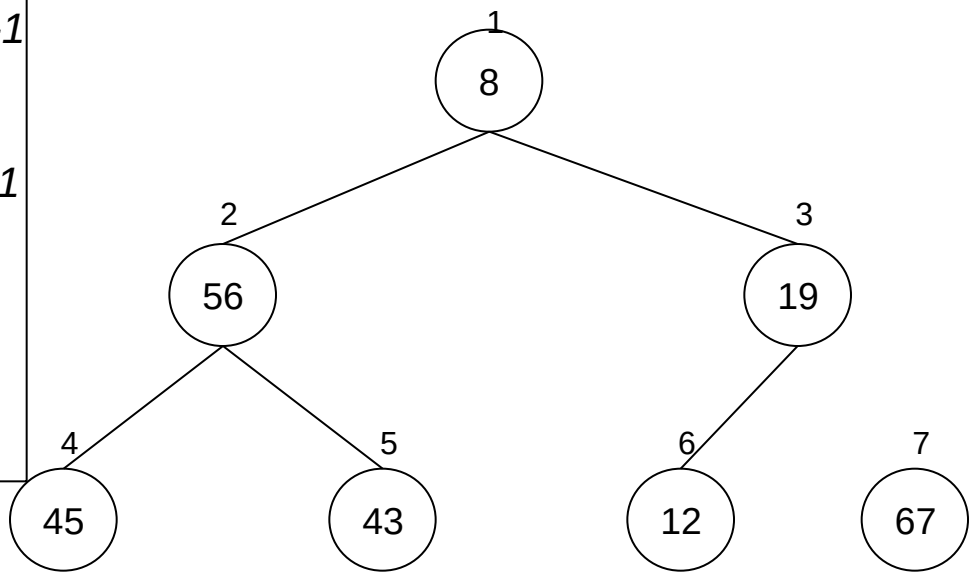
R = 7





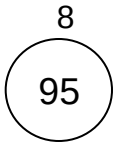
5.5.2. Ordenação por Heapsort

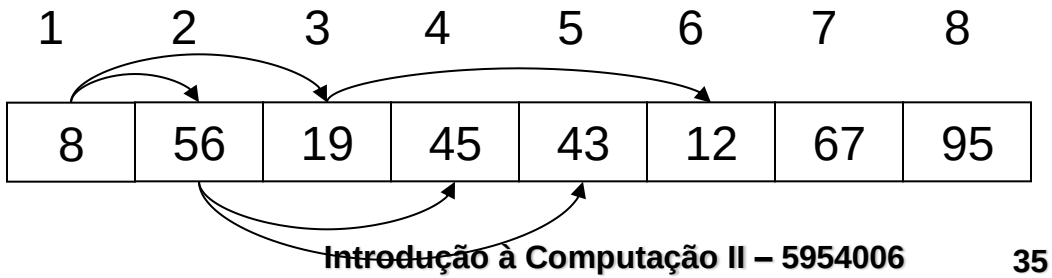
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

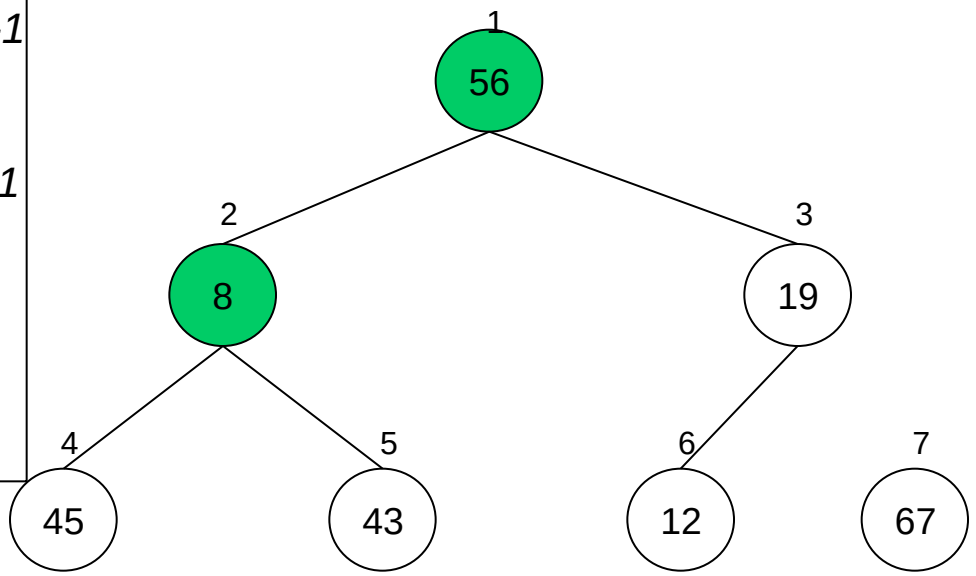
R = 7





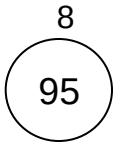
5.5.2. Ordenação por Heapsort

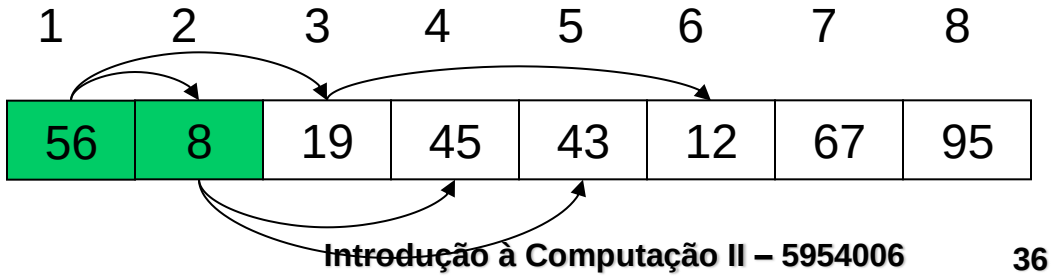
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

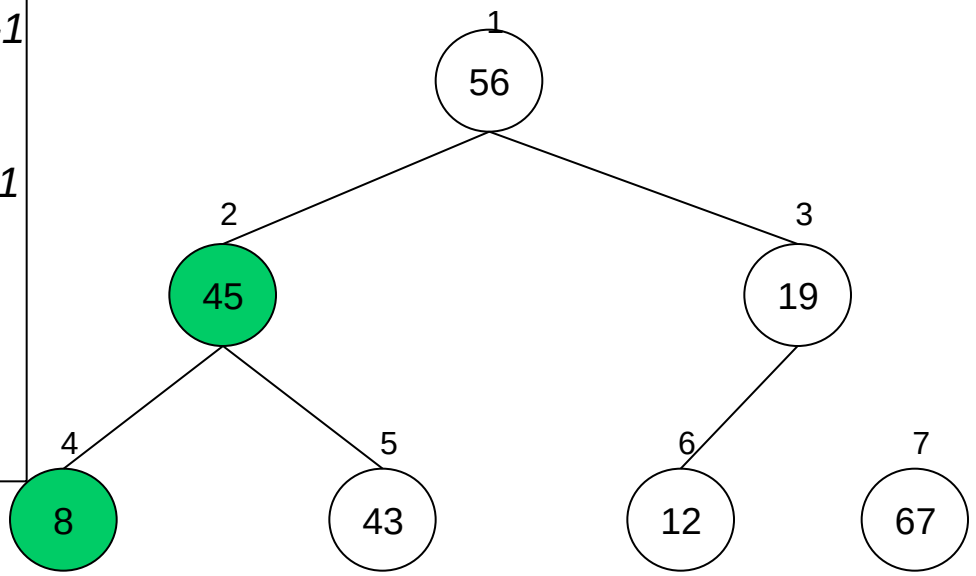
R = 7





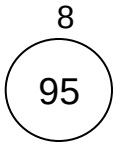
5.5.2. Ordenação por Heapsort

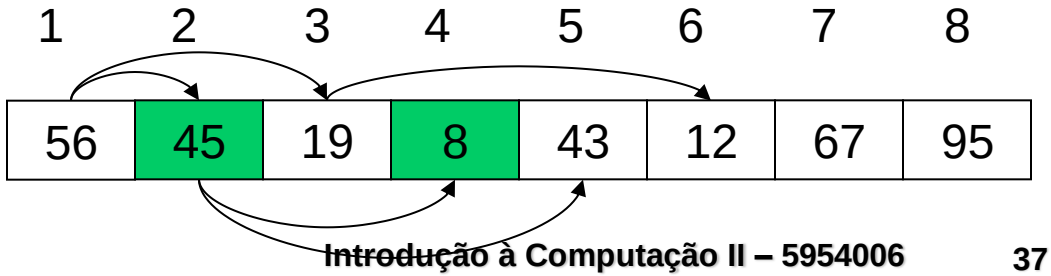
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

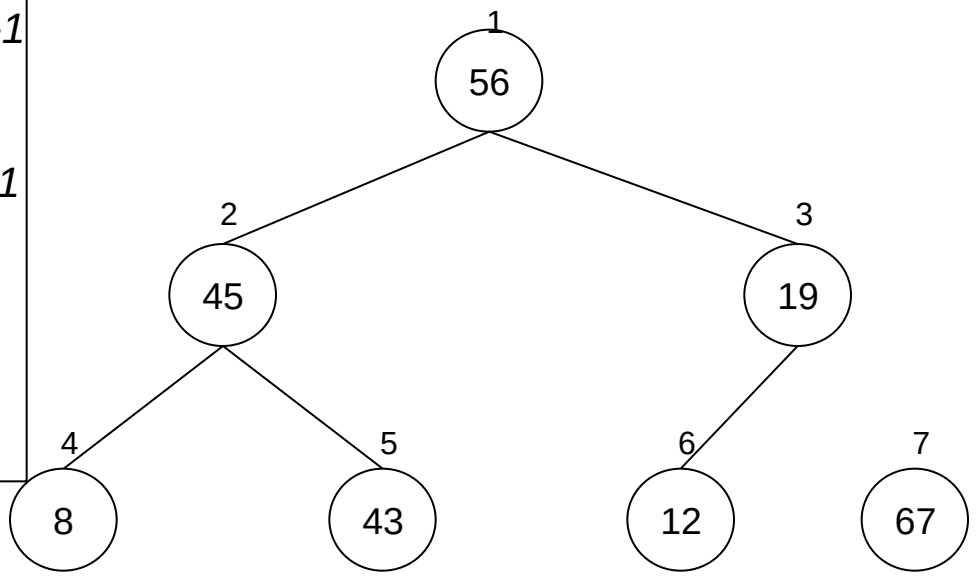
R = 7





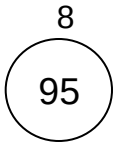
5.5.2. Ordenação por Heapsort

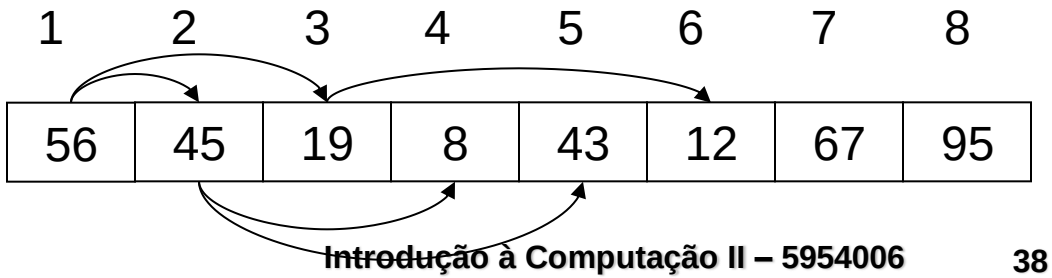
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

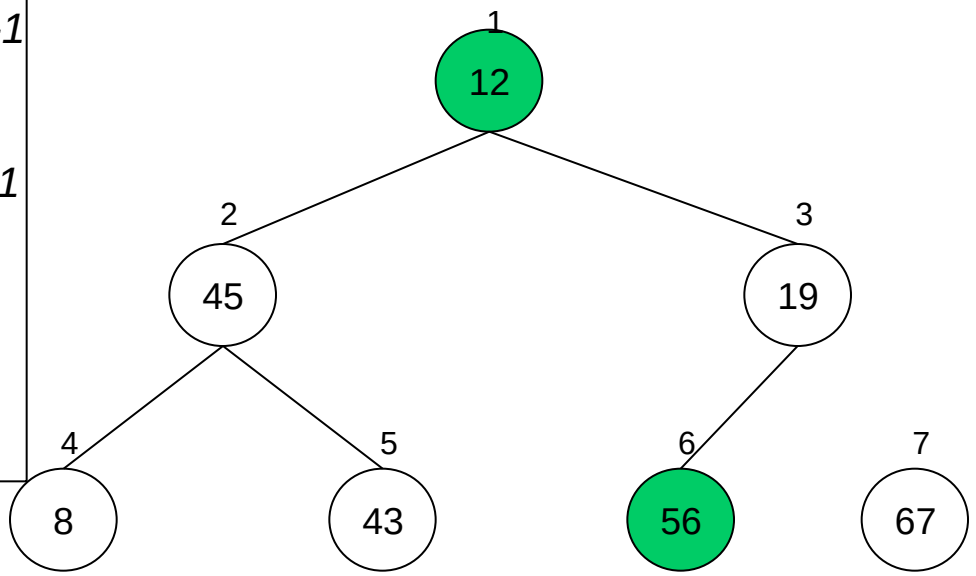
R = 6





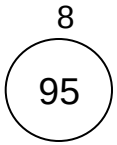
5.5.2. Ordenação por Heapsort

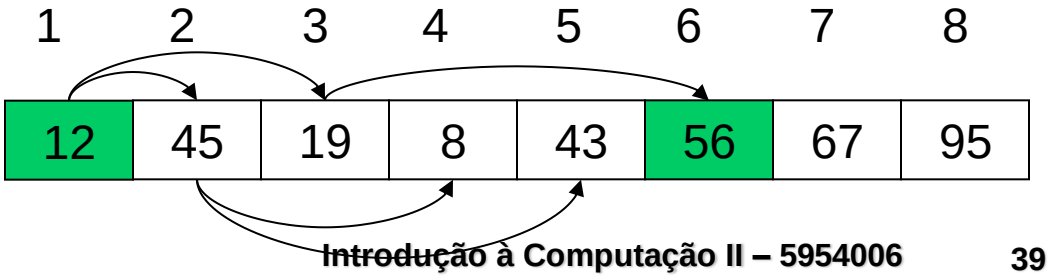
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

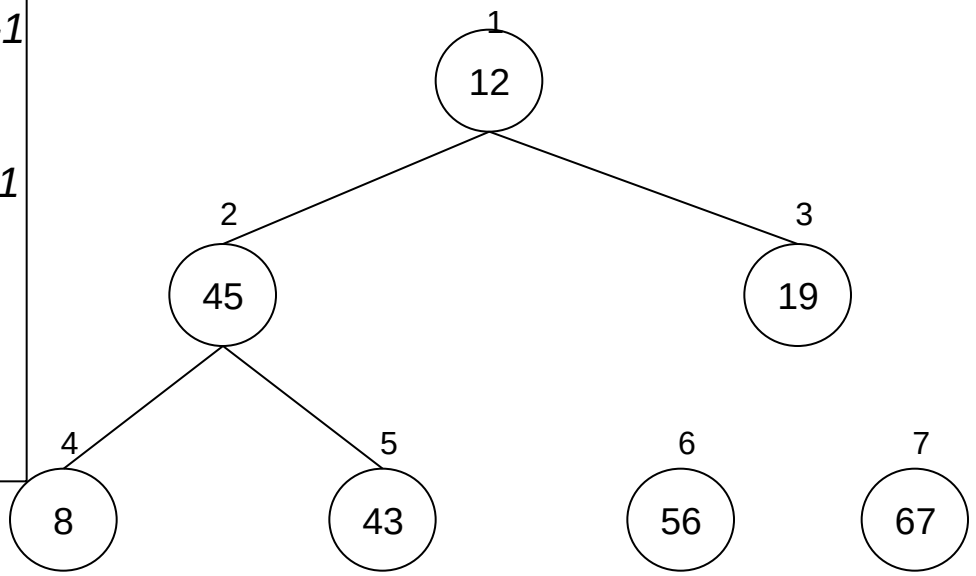
R = 6





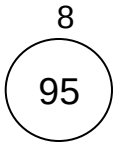
5.5.2. Ordenação por Heapsort

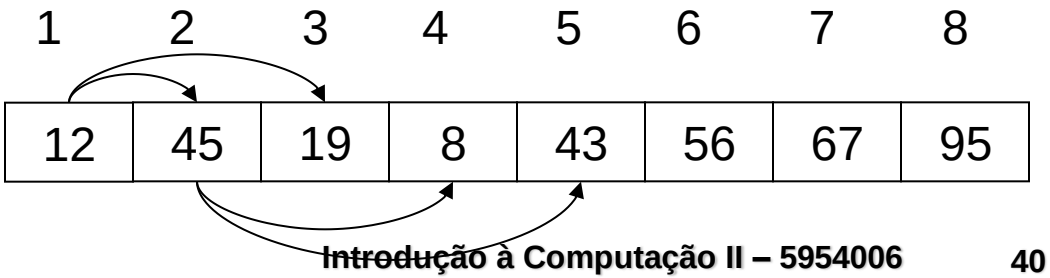
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

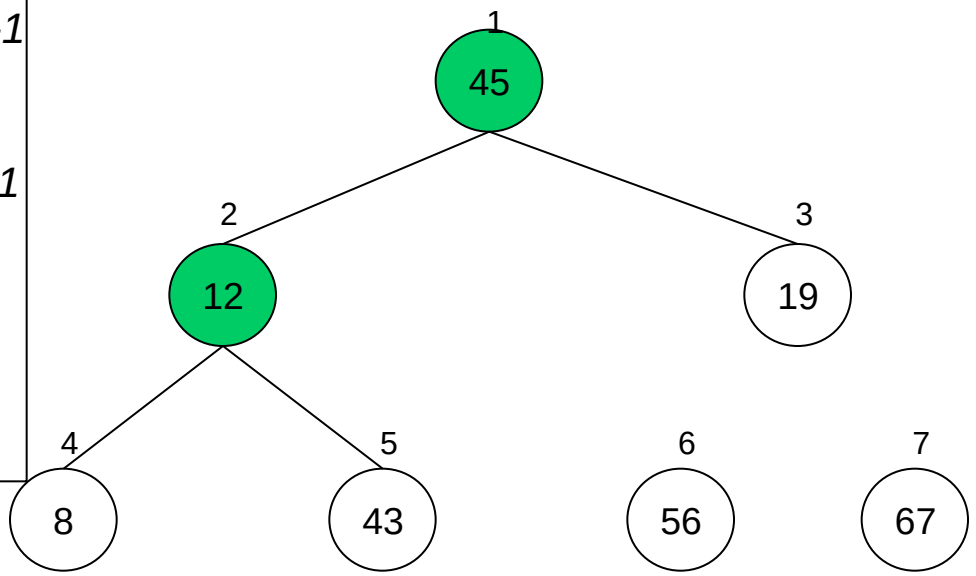
R = 6





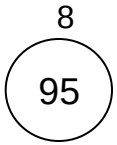
5.5.2. Ordenação por Heapsort

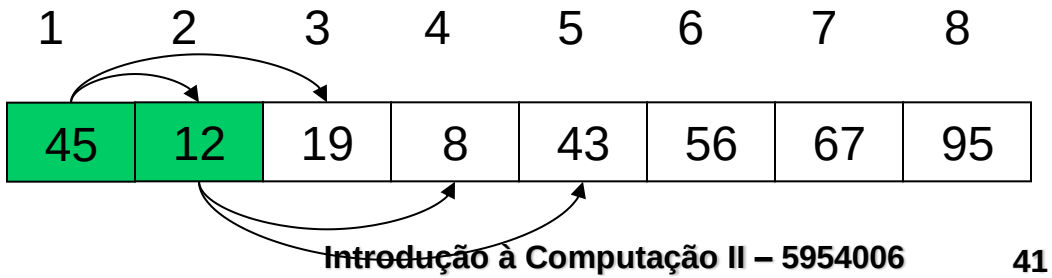
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

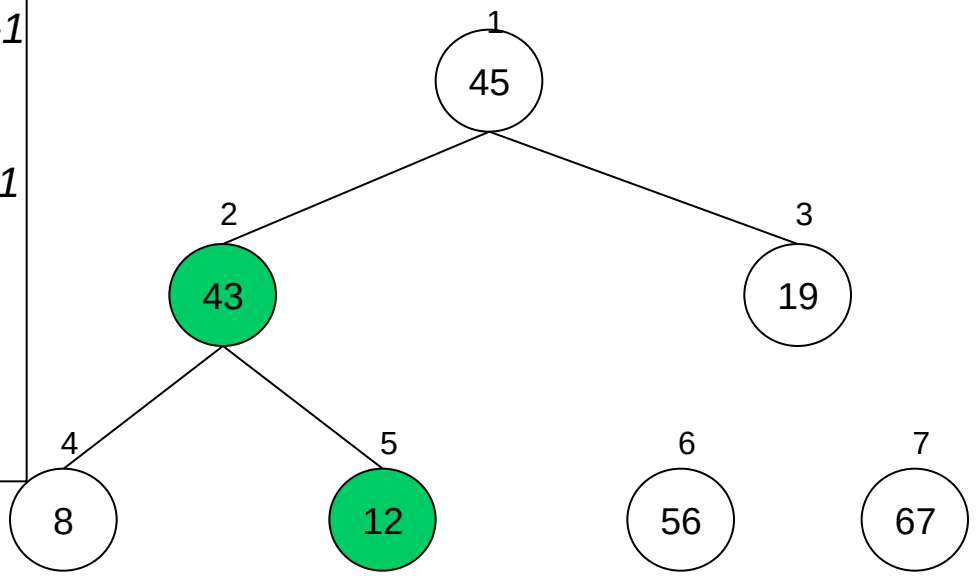
R = 6





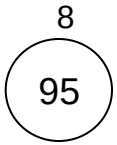
5.5.2. Ordenação por Heapsort

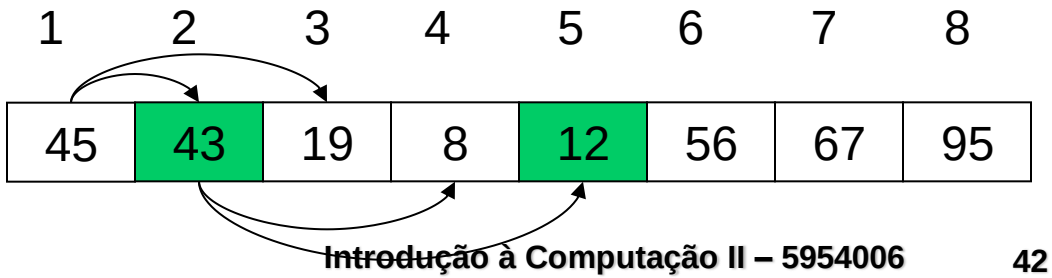
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

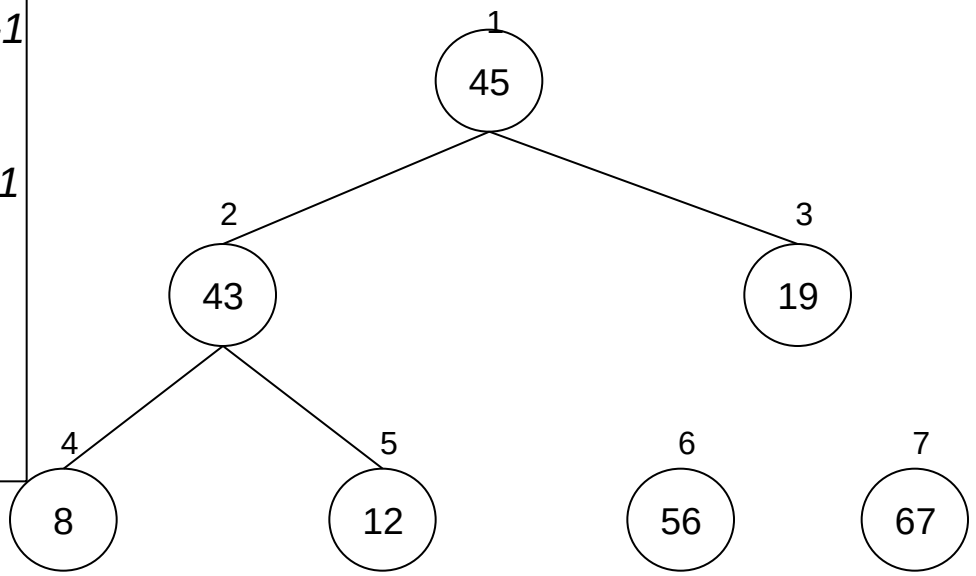
R = 6





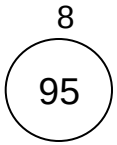
5.5.2. Ordenação por Heapsort

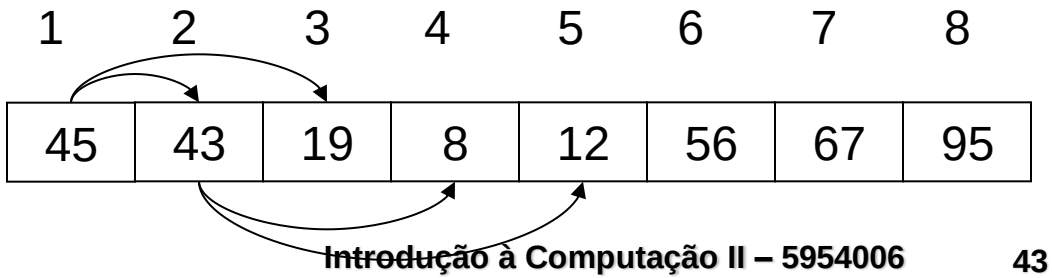
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

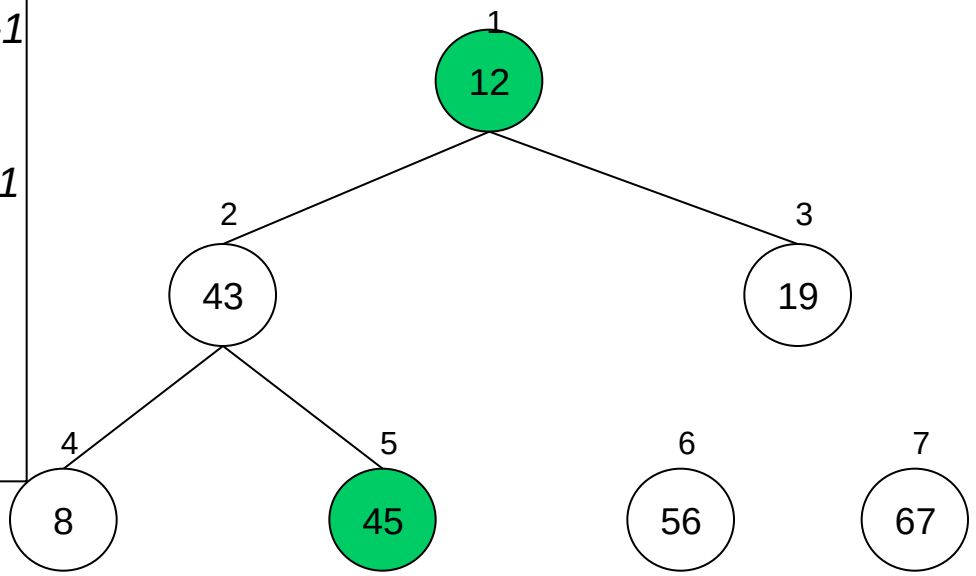
R = 5





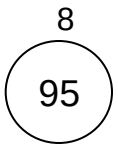
5.5.2. Ordenação por Heapsort

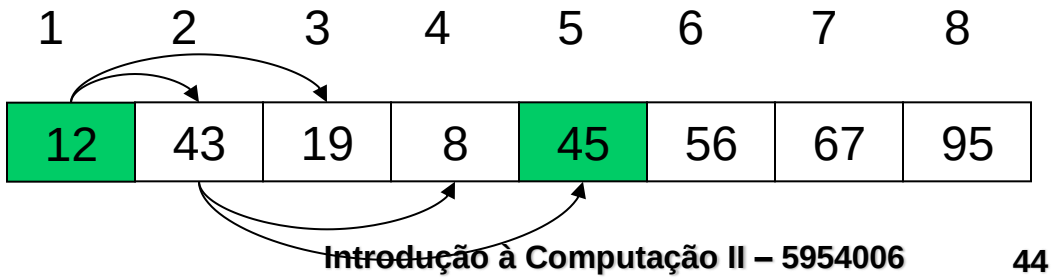
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

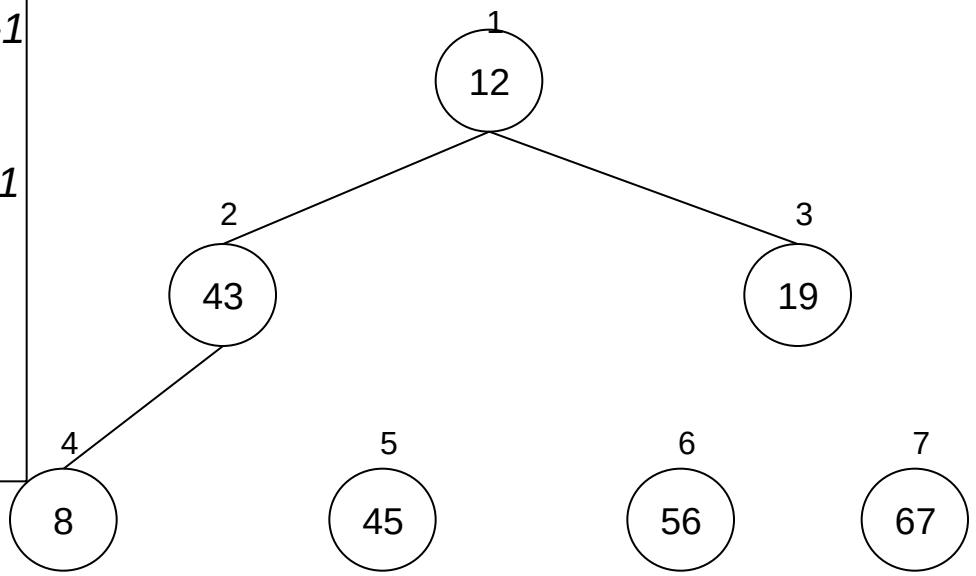
R = 5





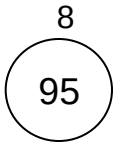
5.5.2. Ordenação por Heapsort

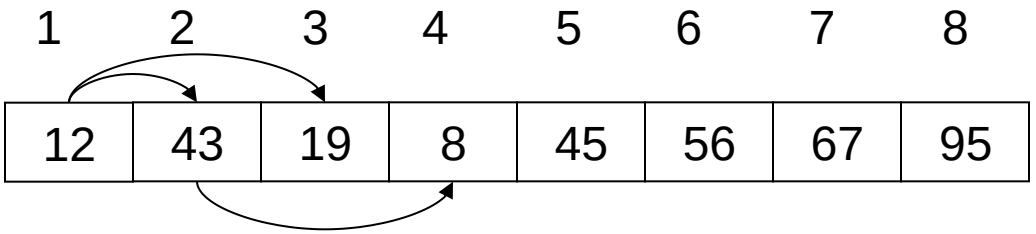
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

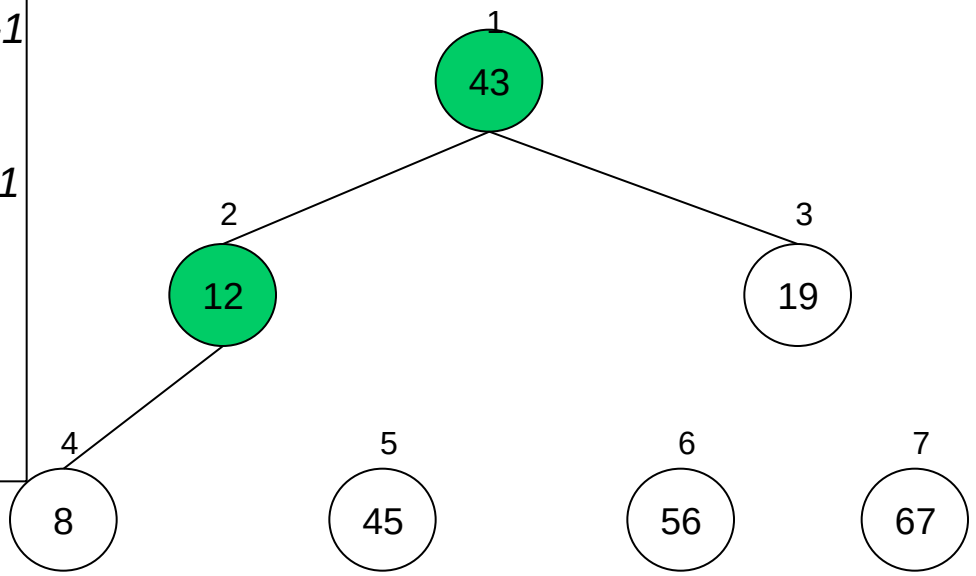
R = 5





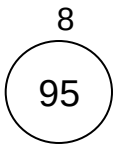
5.5.2. Ordenação por Heapsort

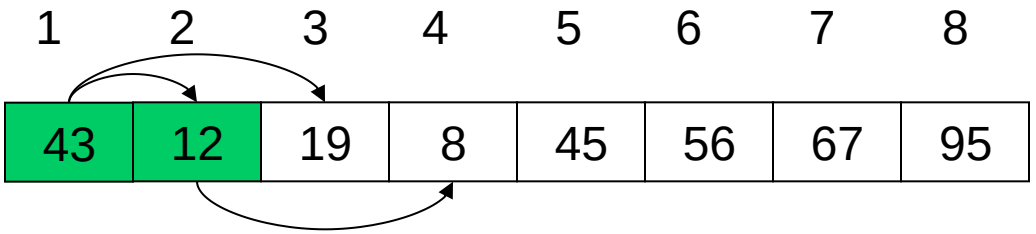
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

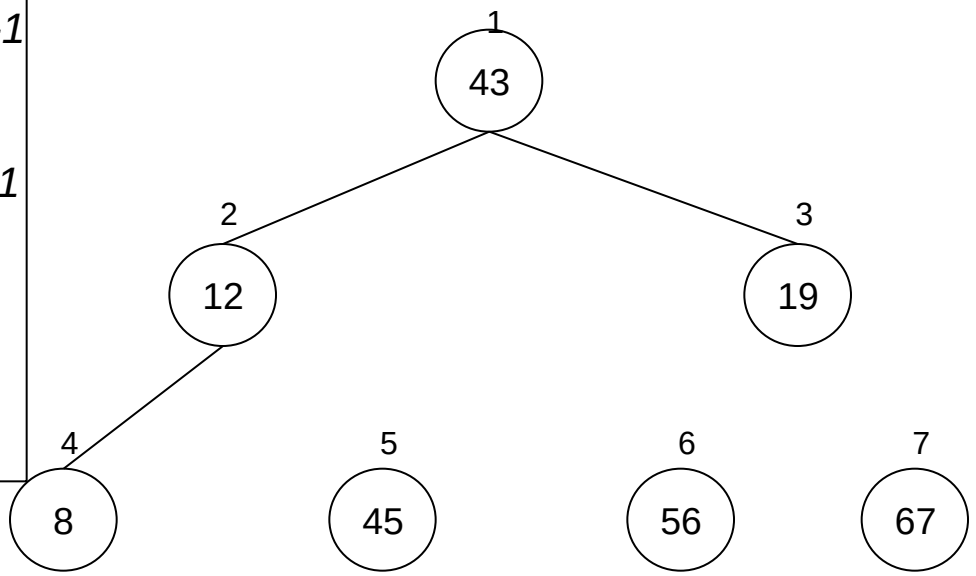
R = 5





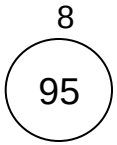
5.5.2. Ordenação por Heapsort

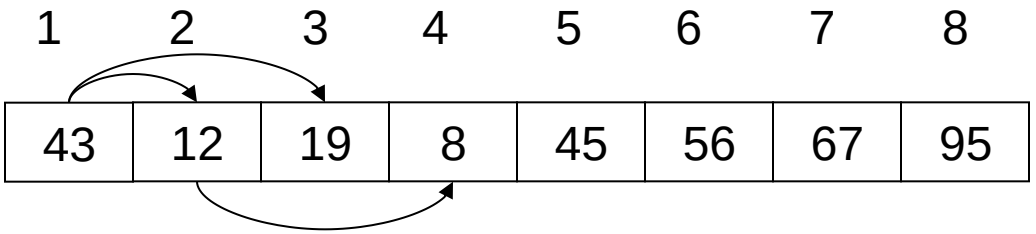
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

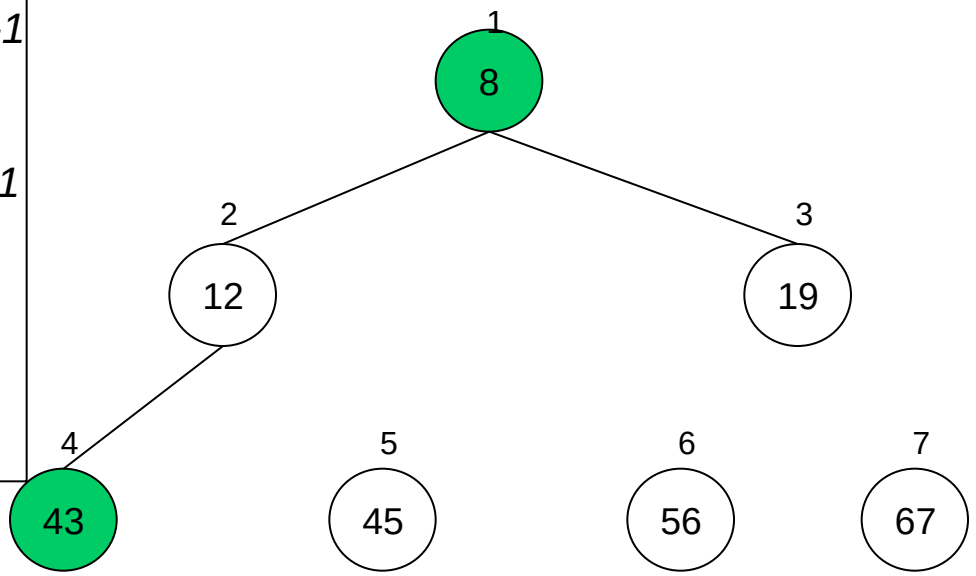
R = 4





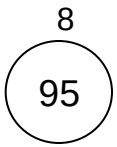
5.5.2. Ordenação por Heapsort

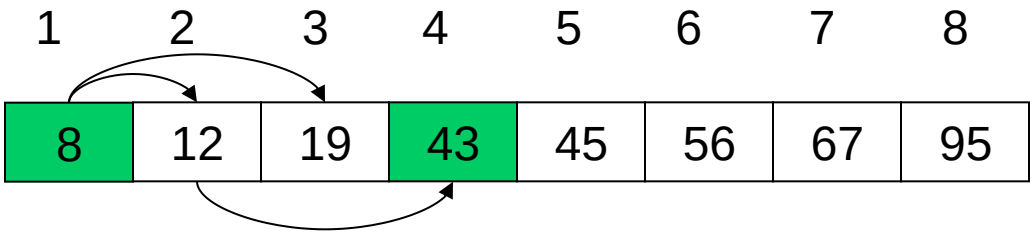
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

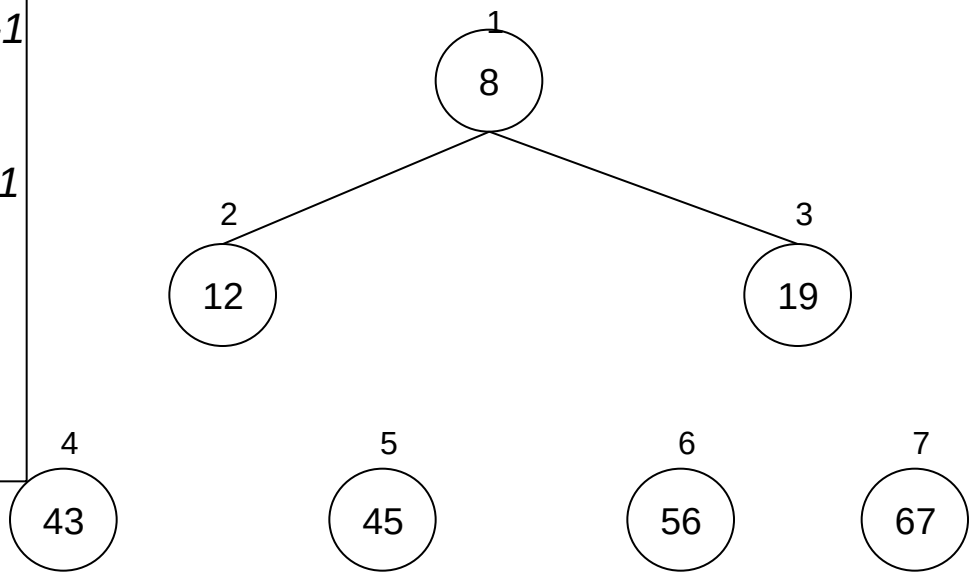
R = 4





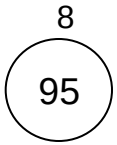
5.5.2. Ordenação por Heapsort

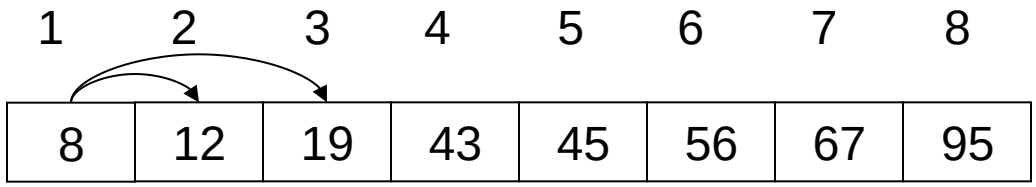
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

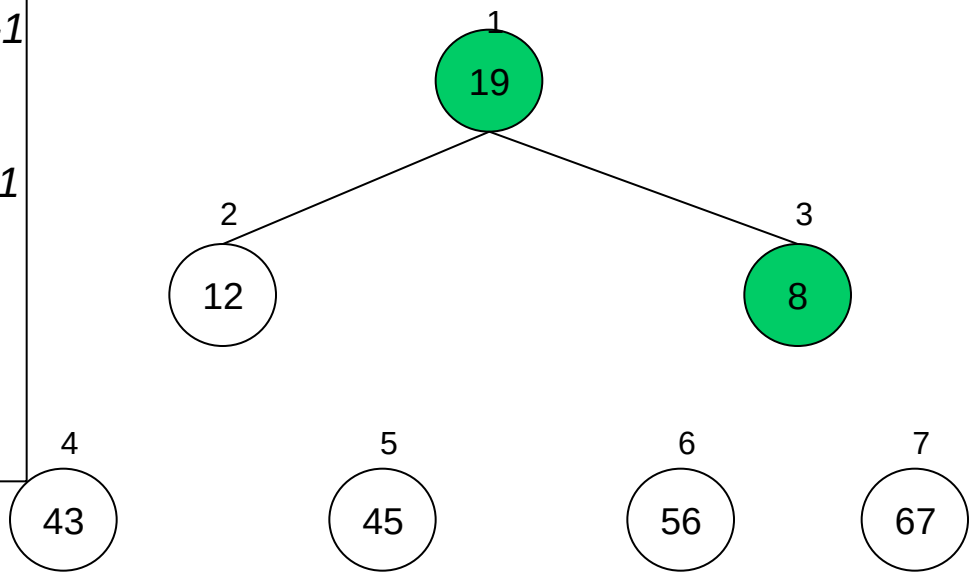
R = 4





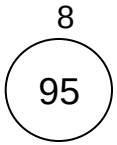
5.5.2. Ordenação por Heapsort

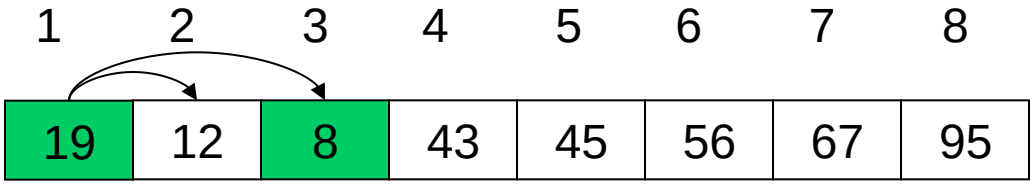
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

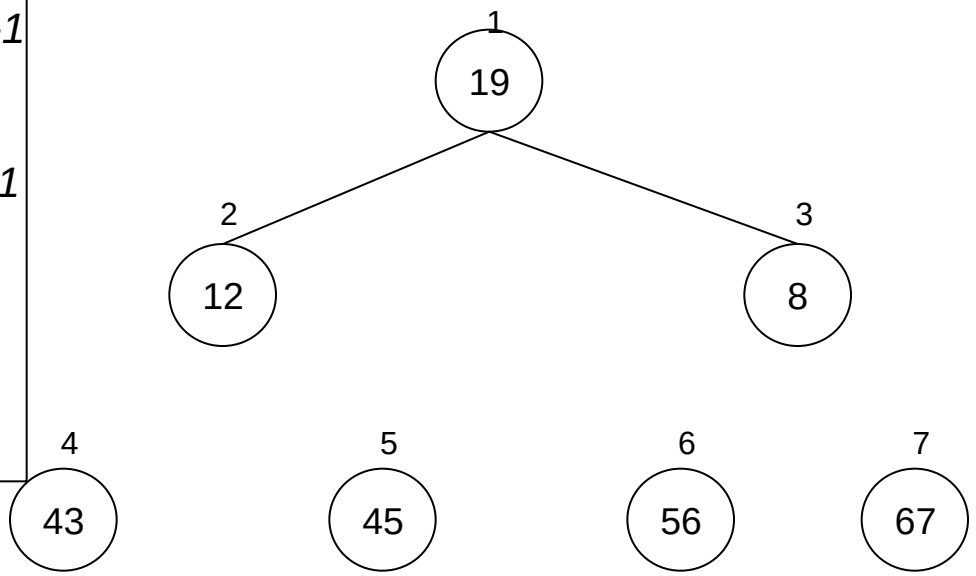
R = 4





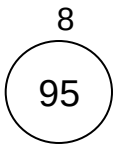
5.5.2. Ordenação por Heapsort

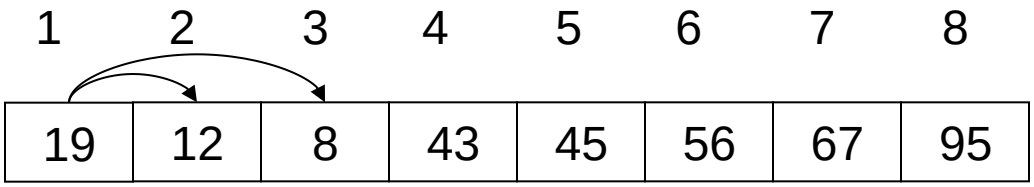
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

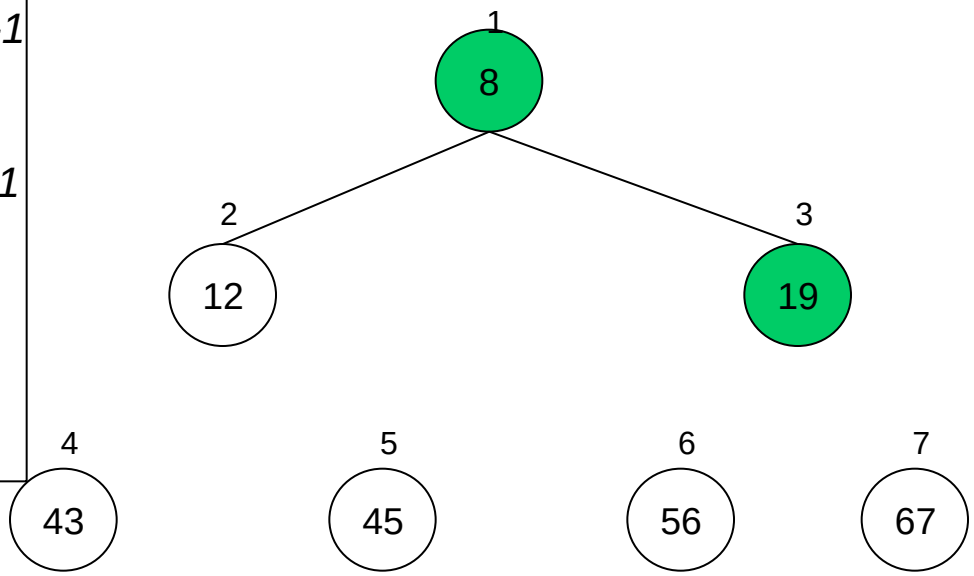
R = 3





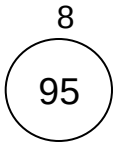
5.5.2. Ordenação por Heapsort

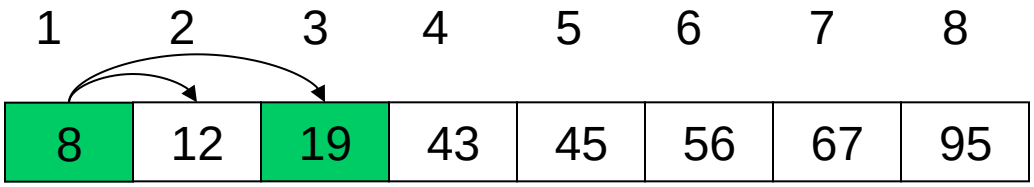
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

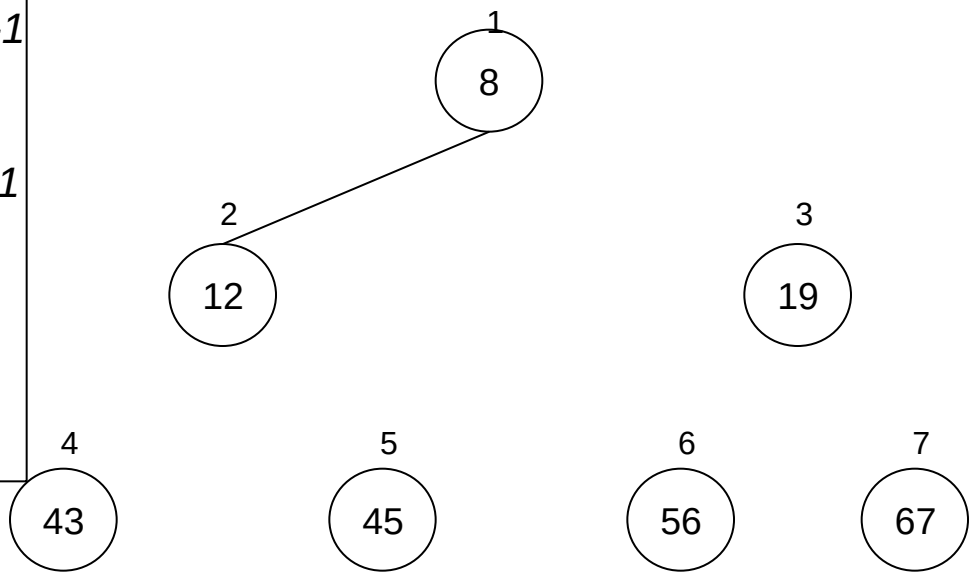
R = 3





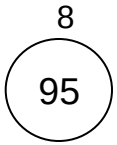
5.5.2. Ordenação por Heapsort

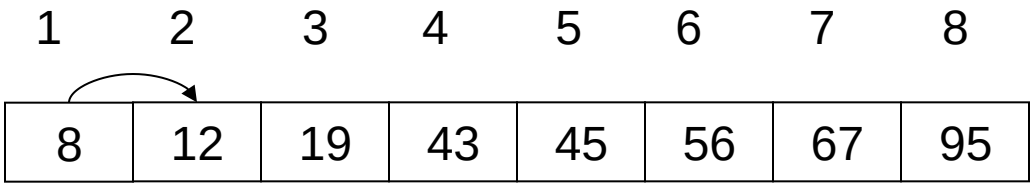
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

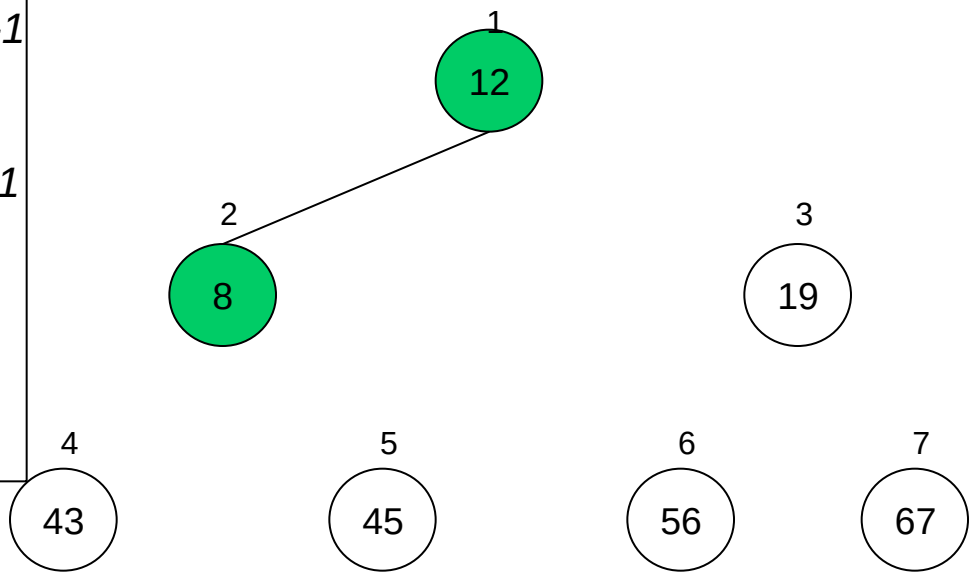
R = 3





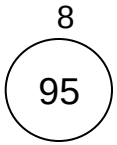
5.5.2. Ordenação por Heapsort

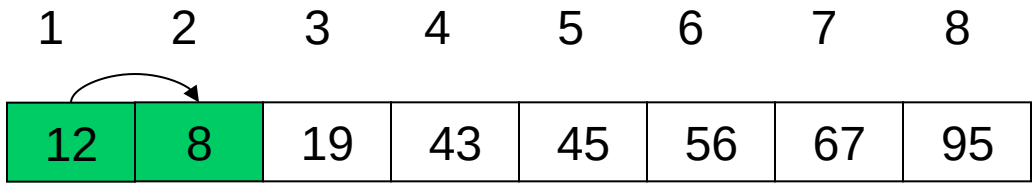
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

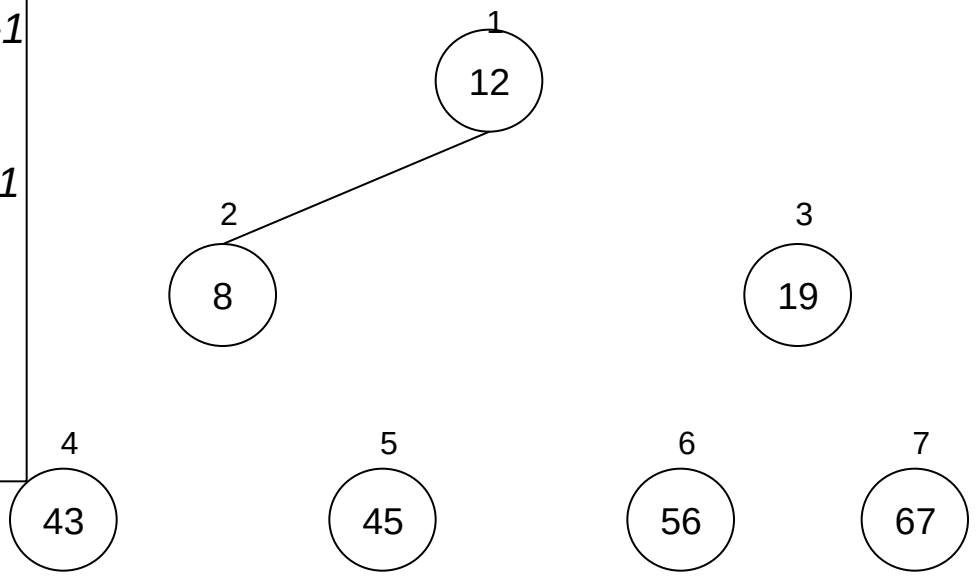
R = 3





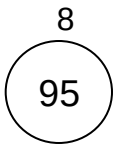
5.5.2. Ordenação por Heapsort

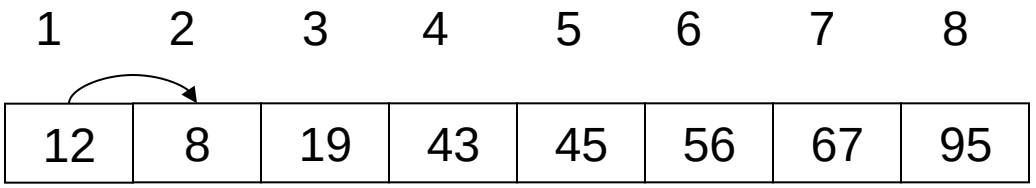
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

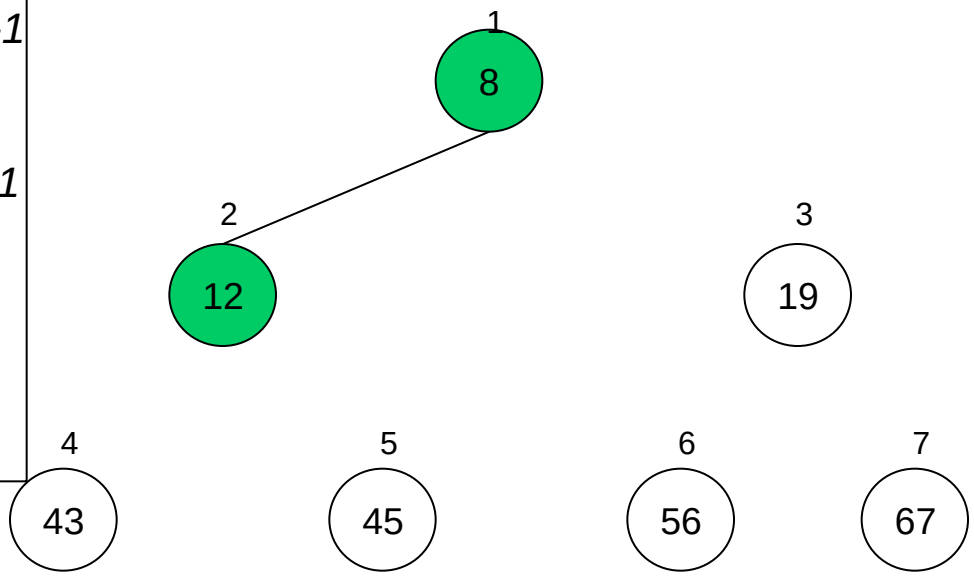
R = 2





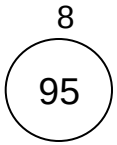
5.5.2. Ordenação por Heapsort

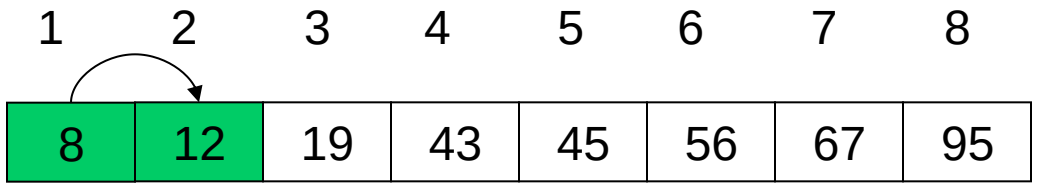
```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

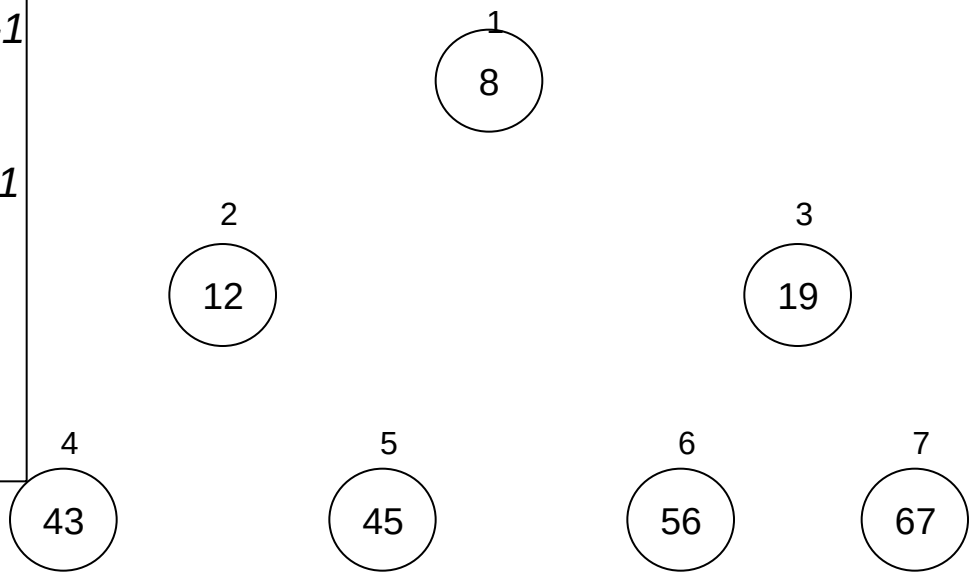
R = 2





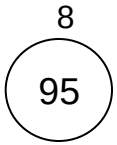
5.5.2. Ordenação por Heapsort

```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8

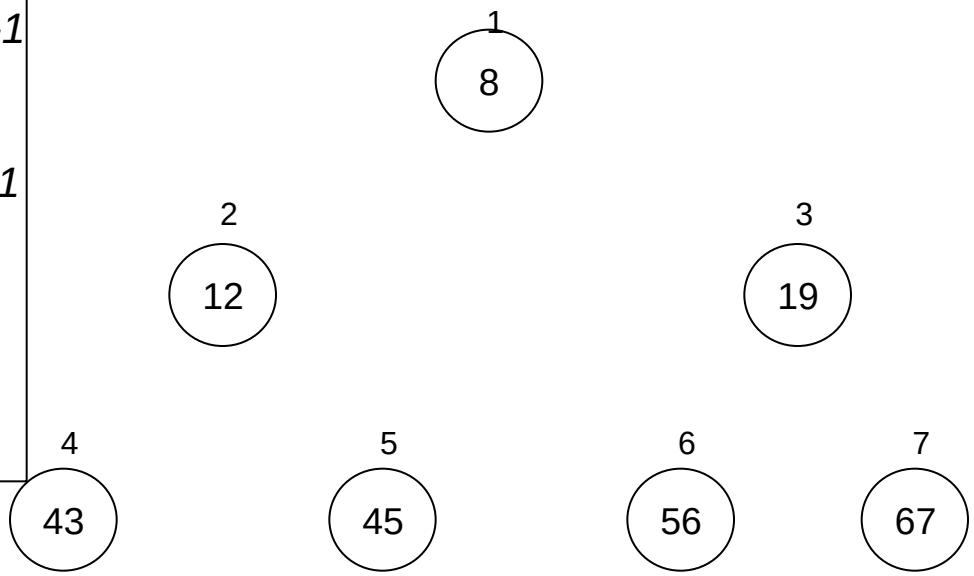
R = 2



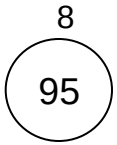
1	2	3	4	5	6	7	8
8	12	19	43	45	56	67	95

5.5.2. Ordenação por Heapsort

```
para L ← N/2 até L ← 1 com passo L ←L-1
        heapify( a , L , N )
fim para
para R ← N até R ← 2 com passo R ←R-1
        w ← a[1]
        a[1] ← a[R]
        a[R] ← w
        heapify( a , 1 , R-1 )
fim para
```



N = 8



1	2	3	4	5	6	7	8
8	12	19	43	45	56	67	95

Vetor ordenado!

5.5.2. Ordenação por Heapsort

- Análise
  - À primeira vista não é evidente que este método de ordenação ofereça bons resultados, pois os elementos escorregam para a esquerda em primeiro lugar antes de serem finalmente colocados na sua posição correta, na extremidade direita
  - De fato, o procedimento não é recomendado para pequenos valores de *N*
  - Todavia, para valores grandes de *N*, o método Heapsort é muito eficiente, e quanto maior for o valor de *N*, melhor será o seu desempenho
  - O número médio de movimentos é de aproximadamente (*N*/2) log *N* e os desvios relativos a este valor são relativamente pequenos
  - A complexidade (para melhor e pior caso e também para caso médio) é O(*N* log *N*)
    - Repare que o método aplica o procedimento heapify *N/2+N-1* vezes e que o procedimento heapify é O(log *N*)

5.5.2. Ordenação por Heapsort

Exercício 5.5.1. Utilizando ordenação pelo método heapsort, obtenha o número de comparações e movimentações em cada passo (*i* e *j*) para os seguintes vetores

- [45 56 12 43 95 19 8 67]
- [8 12 19 43 45 56 67 95]
- [95 67 56 45 43 19 12 8]
- [19 12 8 45 43 56 67 95]

5.5.2. Ordenação por Heapsort

Exercício 5.5.1. Solução

L/R	Ci	Mi	45	56	12	43	95	19	8	67
L=4	2	3	45	56	12	67	95	19	8	43
L=3	3	3	45	56	19	67	95	12	8	43
L=2	3	3	45	95	19	67	56	12	8	43
L=1	4	4	95	67	19	45	56	12	8	43
R=8	5	7	67	56	19	45	43	12	8	95
R=7	5	7	56	45	19	8	43	12	67	95
R=6	5	7	45	43	19	8	12	56	67	95
R=5	2	6	43	12	19	8	45	56	67	95
R=4	3	6	19	12	8	43	45	56	67	95
R=3	2	6	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	35	57								

L/R	Ci	Mi	8	12	19	43	45	56	67	95
L=4	2	3	8	12	19	95	45	56	67	43
L=3	3	3	8	12	67	95	45	56	19	43
L=2	4	4	8	95	67	43	45	56	19	12
L=1	5	4	95	45	67	43	8	56	19	12
R=8	5	7	67	45	56	43	8	12	19	95
R=7	2	6	56	45	19	43	8	12	67	95
R=6	5	7	45	43	19	12	8	56	67	95
R=5	4	7	43	12	19	8	45	56	67	95
R=4	3	6	19	12	8	43	45	56	67	95
R=3	2	6	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	36	58								

L/R	Ci	Mi	19	12	8	45	43	56	67	95
L=4	2	3	19	12	8	95	43	56	67	45
L=3	3	3	19	12	67	95	43	56	8	45
L=2	4	4	19	95	67	45	43	56	8	12
L=1	4	4	95	45	67	19	43	56	8	12
R=8	5	7	67	45	56	19	43	12	8	95
R=7	4	7	56	45	12	19	43	8	67	95
R=6	5	7	45	43	12	19	8	56	67	95
R=5	4	7	43	19	12	8	45	56	67	95
R=4	3	6	19	8	12	43	45	56	67	95
R=3	0	5	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	35	58								

L/R	Ci	Mi	95	67	56	45	43	19	12	8
L=4	0	2	95	67	56	45	43	19	12	8
L=3	1	2	95	67	56	45	43	19	12	8
L=2	1	2	95	67	56	45	43	19	12	8
L=1	1	2	95	67	56	45	43	19	12	8
R=8	5	7	67	45	56	8	43	19	12	95
R=7	4	7	56	45	19	8	43	12	67	95
R=6	5	7	45	43	19	8	12	56	67	95
R=5	2	6	43	12	19	8	45	56	67	95
R=4	3	6	19	12	8	43	45	56	67	95
R=3	2	6	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	25	52								

Comentários

- **Agradecimentos**

  - Parte do material desta apresentação foi obtida através de slides da disciplina de Introdução à Computação II ministrada pelo Prof. José Augusto Baranauskas