SCC-601– Introdução à Ciência da Computação II

Ordenação e Complexidade – Parte 3

Lucas Antiqueira

Ordenação: HeapSort

HeapSort

- Utiliza uma estrutura de dados chamada *heap* para ordenar.
- Um *heap* é um vetor (array) que representa uma árvore binária.

HeapSort

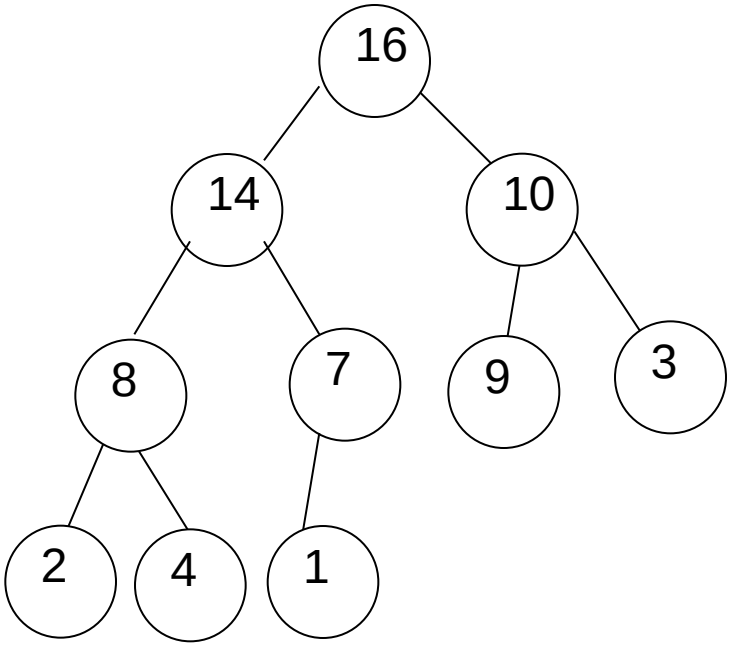
- Heap:
  - Estrutura de dados para armazenar dados segundo uma regra particular. Tradução usual: monte.

ou

  - Espaço de memória variável onde são criados objetos.

HeapSort

- No caso da ordenação, um heap é uma estrutura de dados em que há uma ordenação entre elementos representados via árvore binária

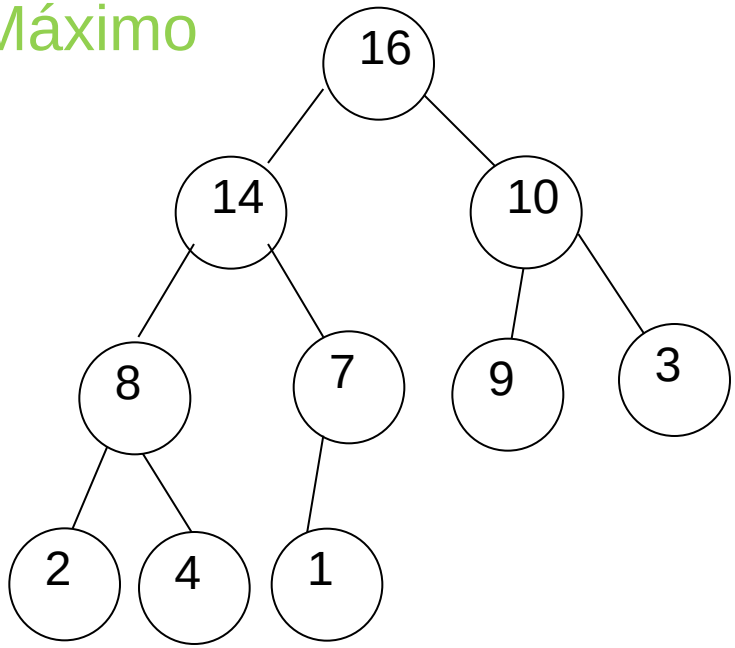


HeapSort

- Um heap observa conceitos de ordem e de forma
  - Ordem: o item de qualquer nó deve satisfazer uma relação de ordem com os itens dos nós filhos
    - Heap máximo (ou descendente): pai >= filhos, sendo que a raiz é o maior elemento
    - Heap mínimo (ou heap ascendente): pai <= filhos, sendo que a raiz é o menor elemento

HeapSort

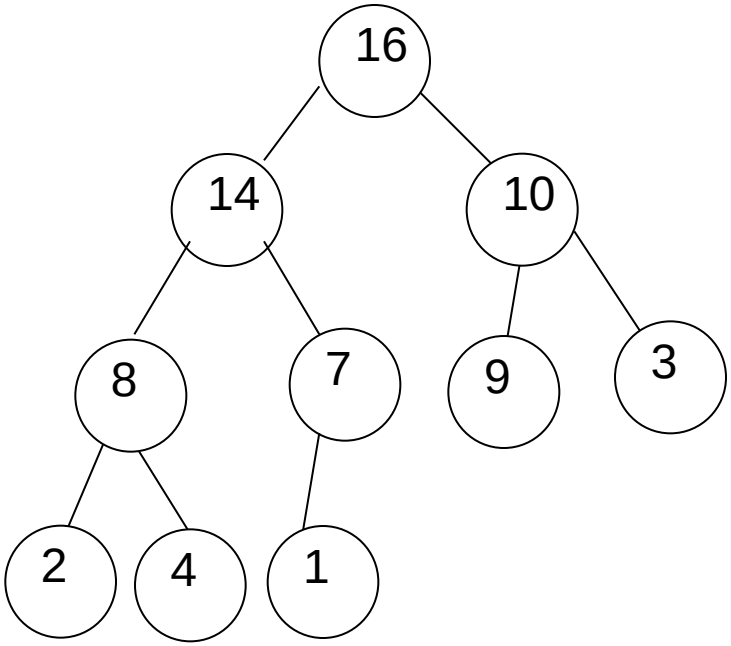
Heap Máximo



HeapSort

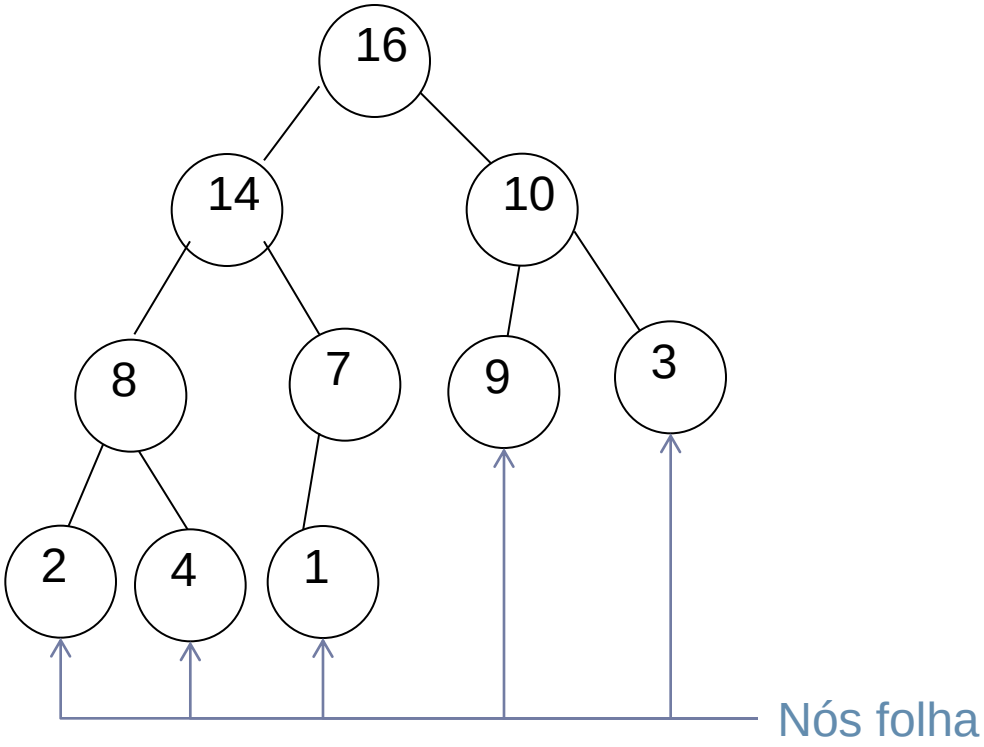
- Um heap observa conceitos de ordem e de forma
  - Forma: a árvore binária tem seus nós folha, no máximo, em dois níveis (ou seja, somente o último nível pode estar incompleto), sendo que as folhas devem estar o mais à esquerda possível

HeapSort



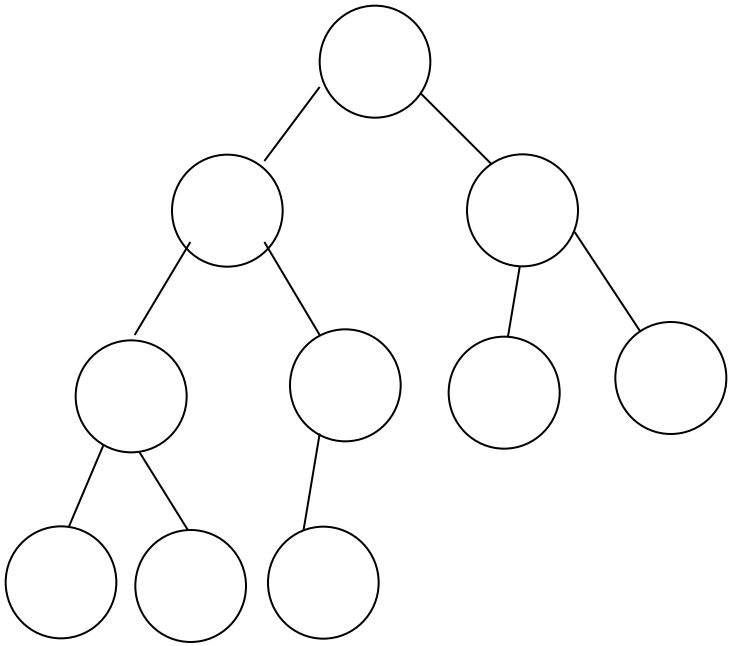
Nós folha?

HeapSort

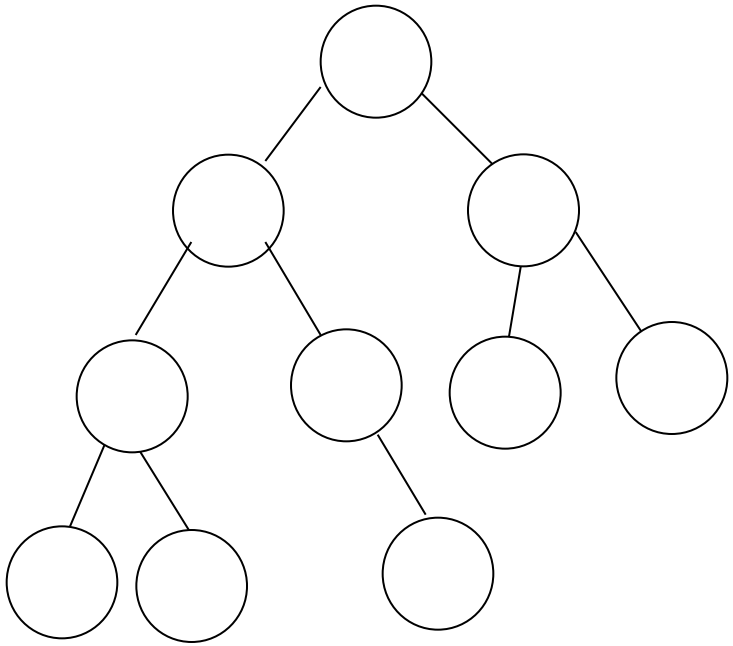


HeapSort

- Exemplos



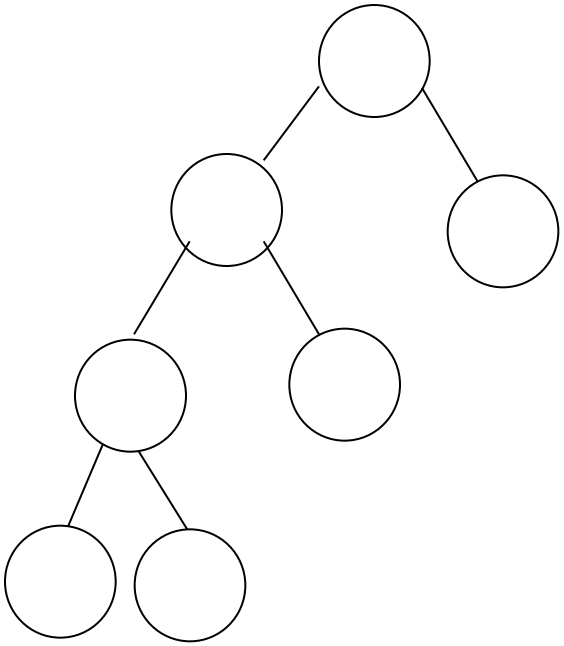
OK

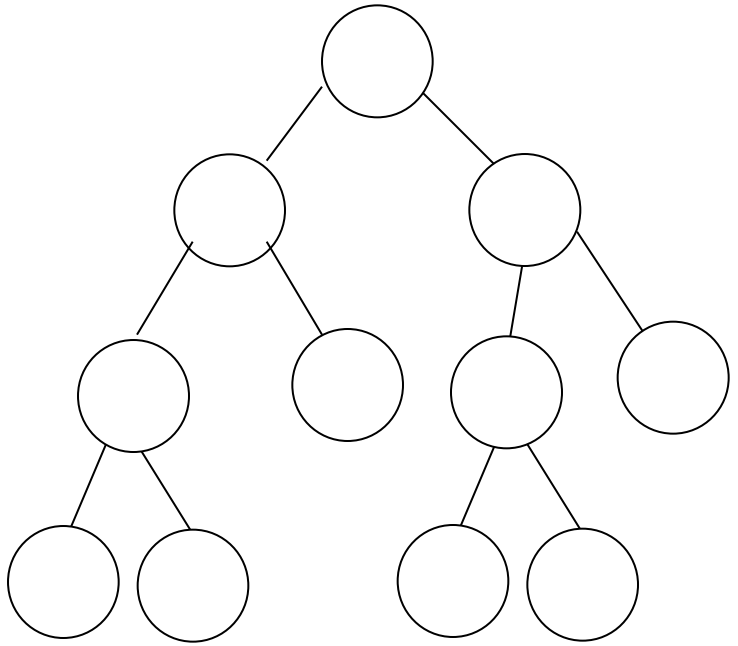


Não!

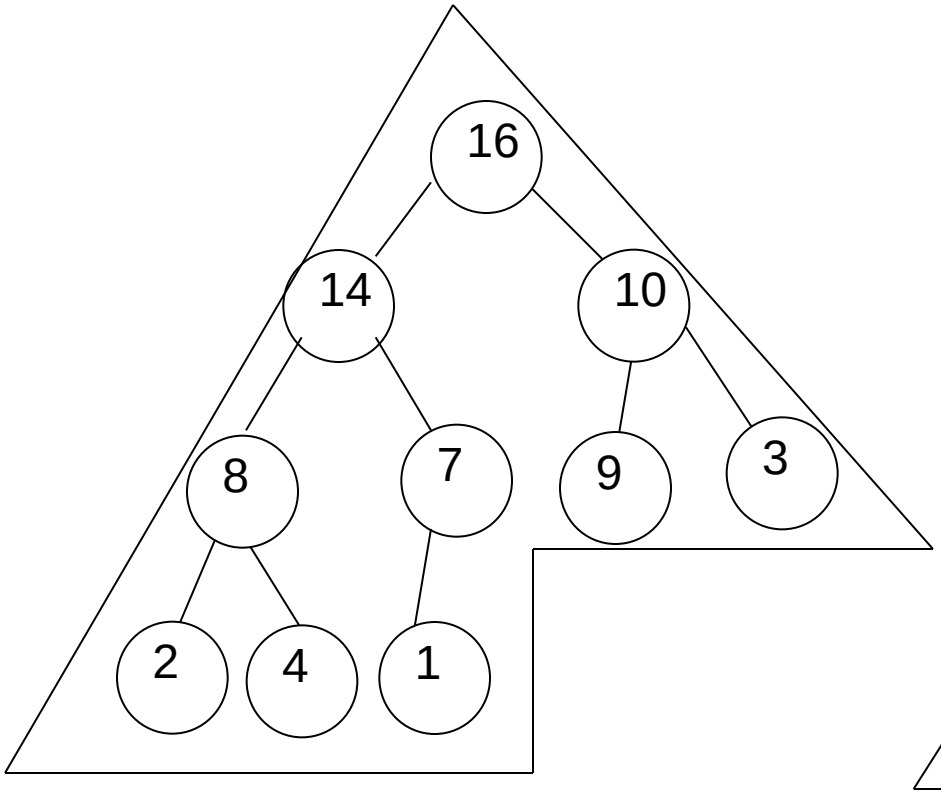
HeapSort

- Exemplos de árvores binárias que não são heaps
  - Por quê?

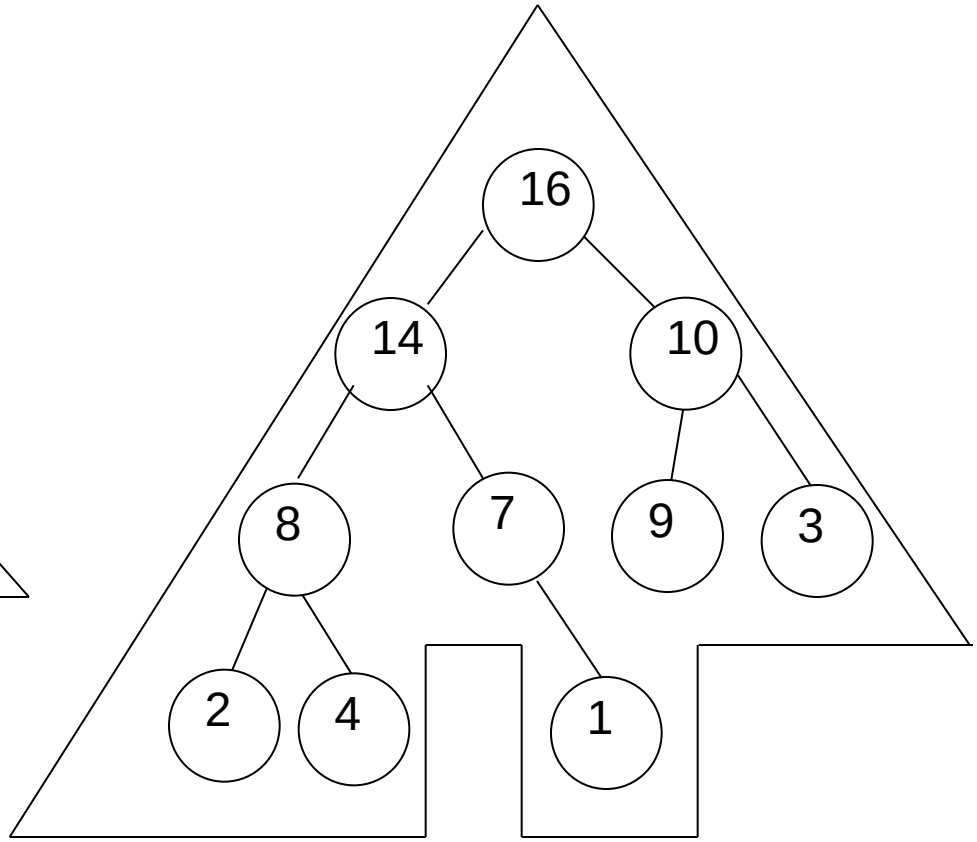




HeapSort

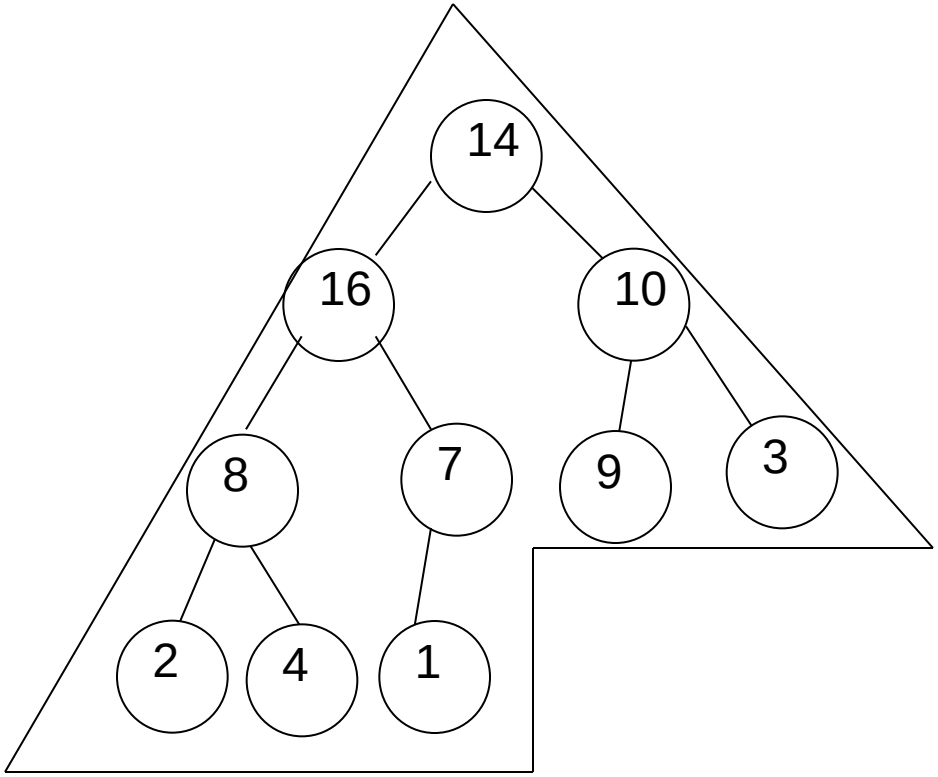


É um heap máximo

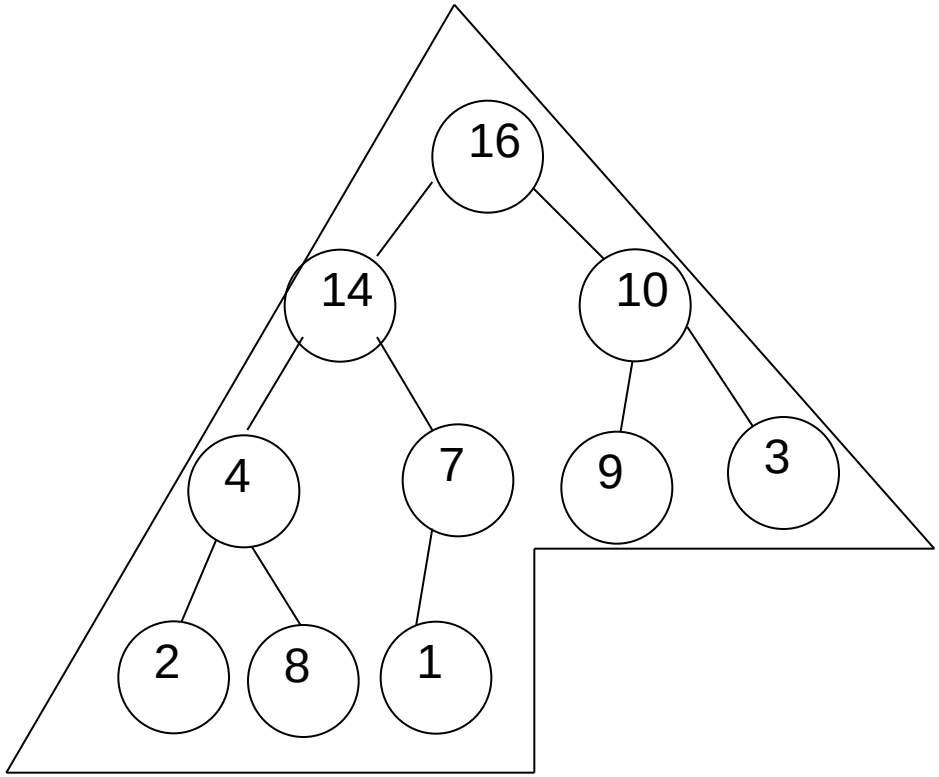


Não é um heap máximo

HeapSort

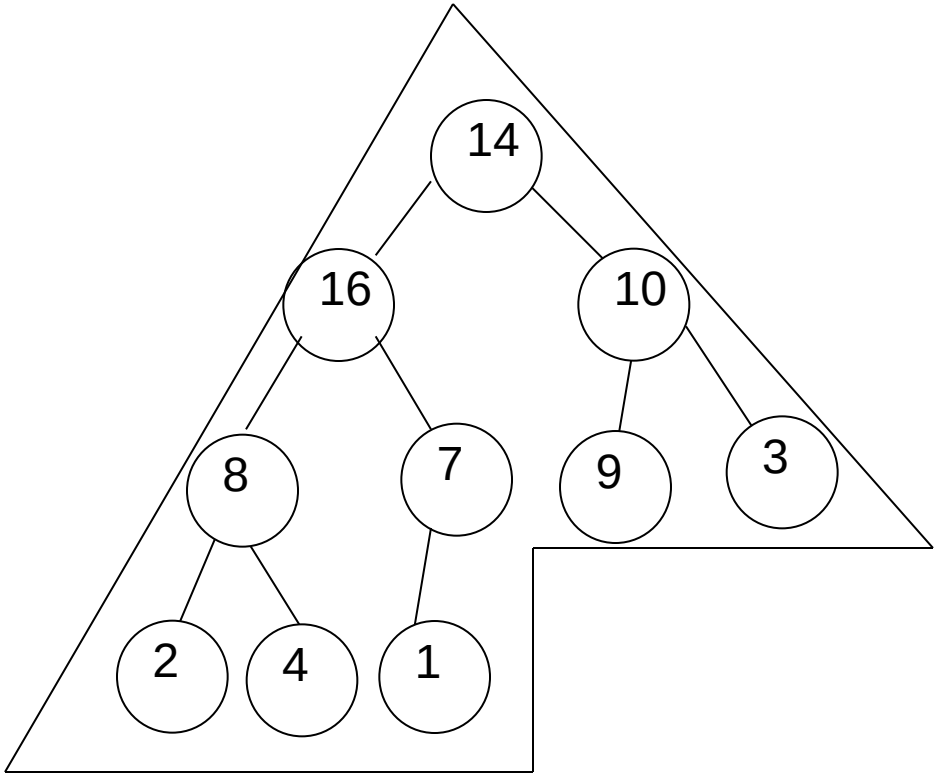


?

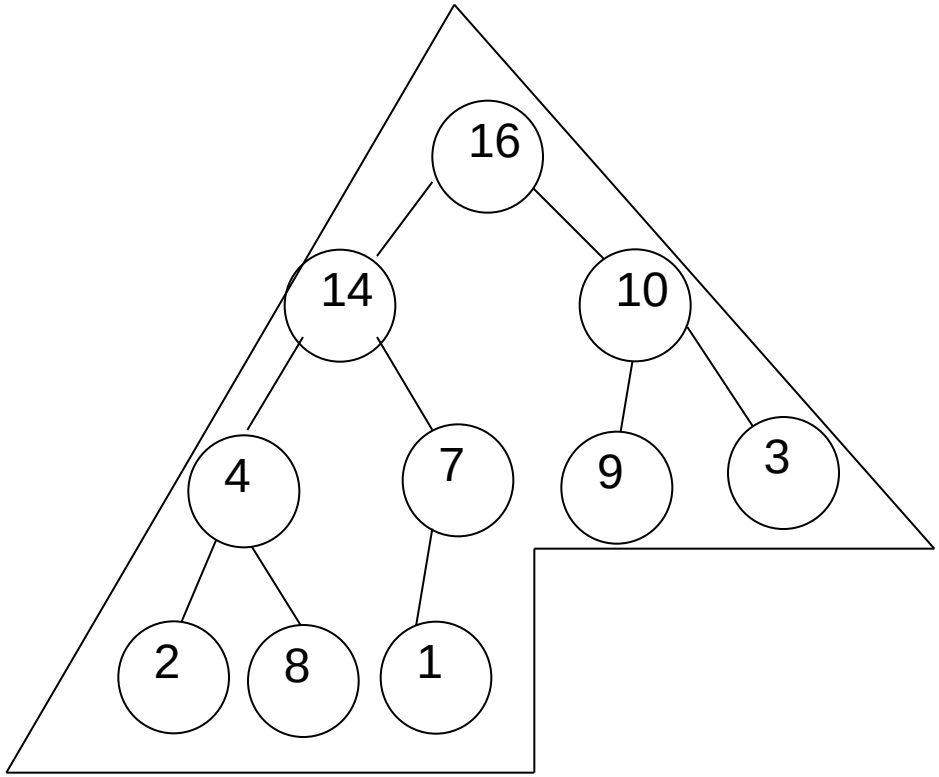


?

HeapSort



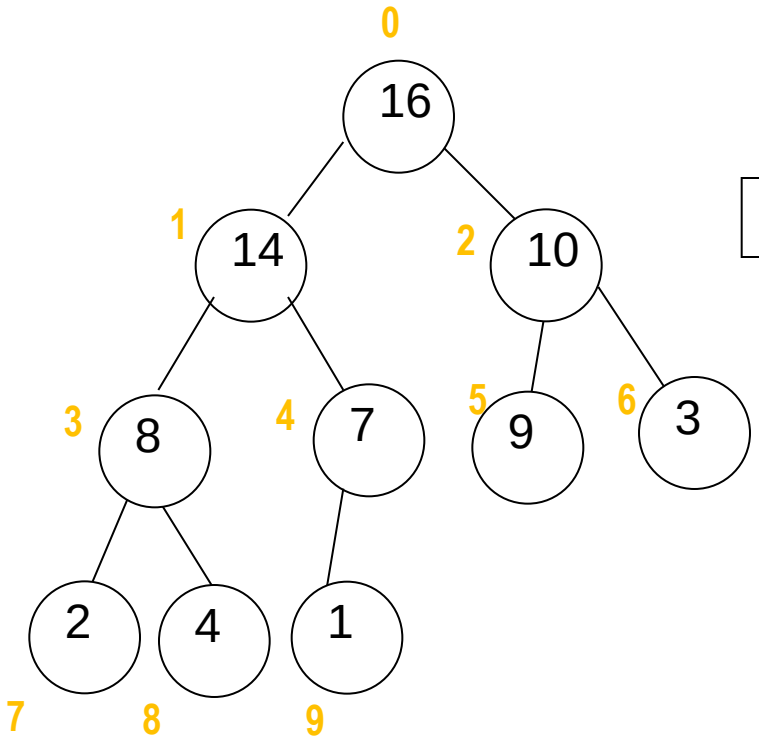
Não é um heap máximo



Não é um heap máximo

HeapSort

- Como armazenar um heap?

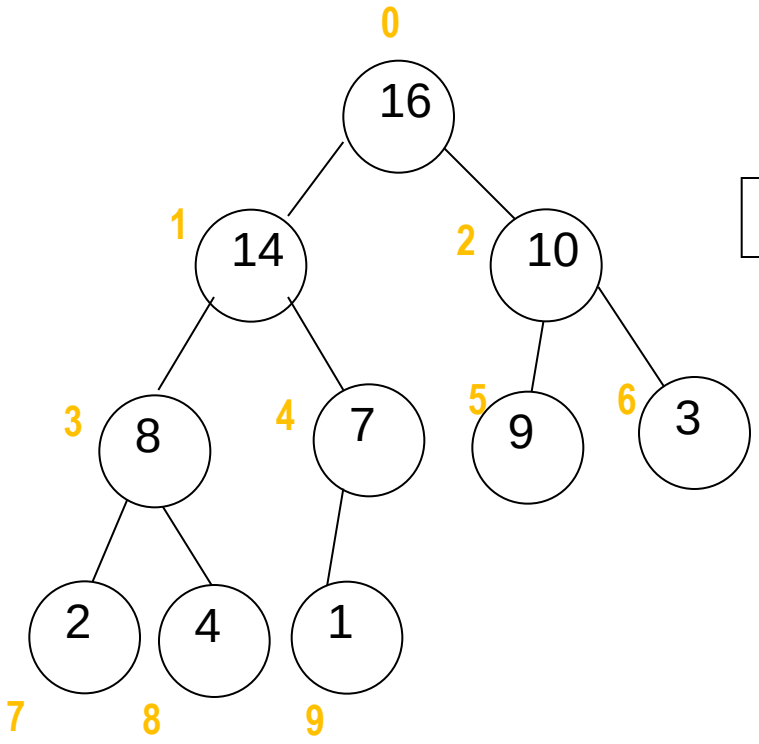


0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Filhos do nó **i** ?

HeapSort

- Como armazenar um heap?



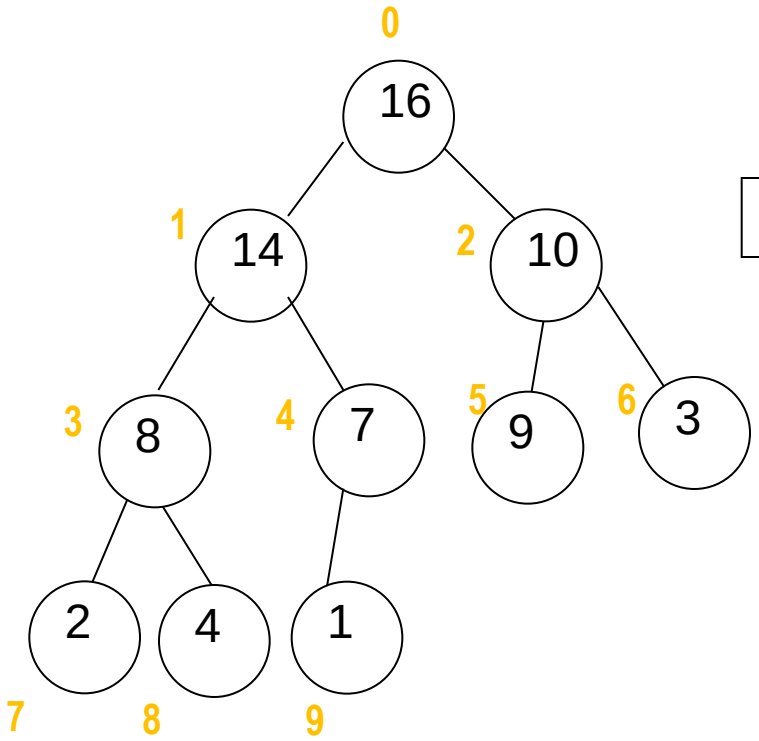
0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Filhos do nó **i**:

- $2i + 1$ = filho esquerdo
- $2i + 2$ = filho direito

HeapSort

- Como armazenar um heap?

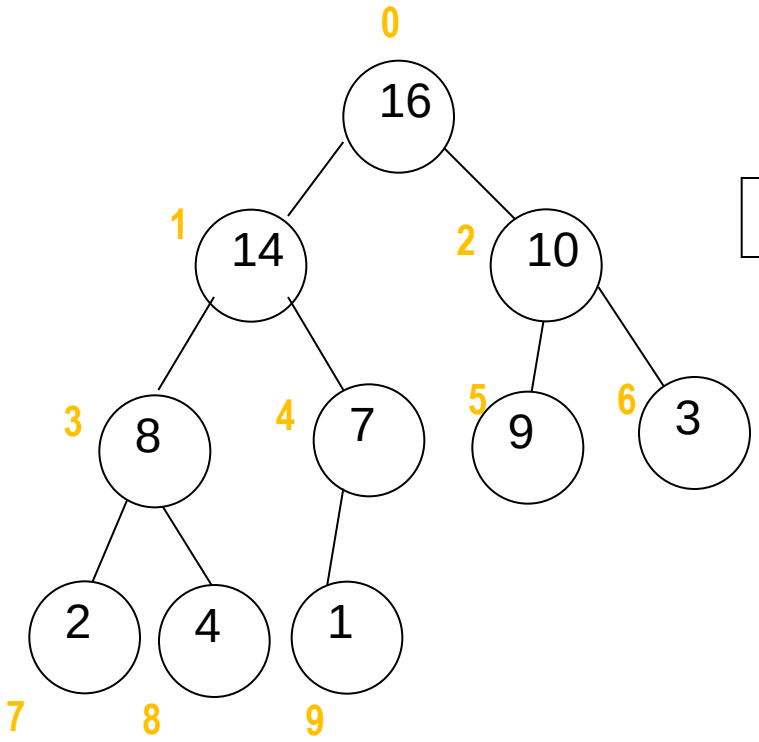


0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Pai de um nó **k** ?

HeapSort

- Como armazenar um heap?



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Pai de um nó **k** ?

- Parte inteira de (k-1)/2

HeapSort

Assume-se que:

- A raiz está sempre na posição 0 do vetor

- Teremos que armazenar o número de elementos do vetor

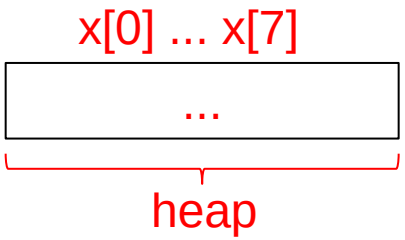
e

- Teremos que armazenar o número de elementos no heap armazenado dentro do vetor

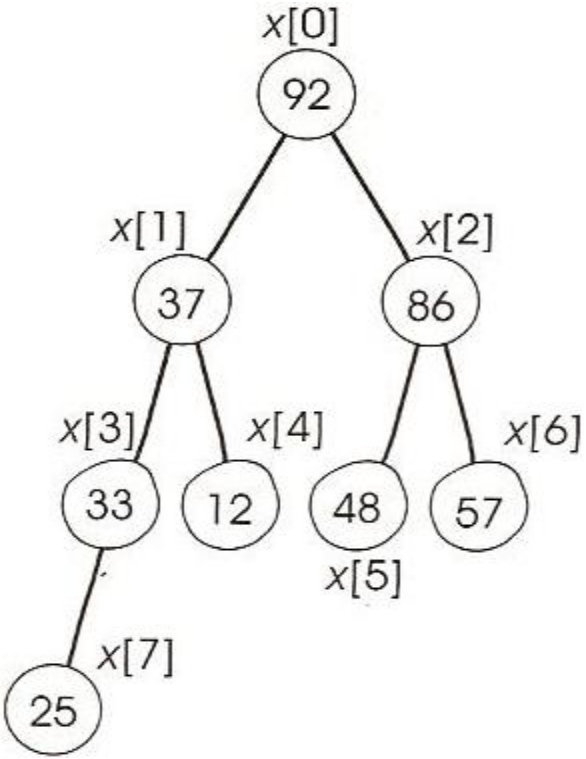
HeapSort

- A idéia para ordenar usando um heap é:
  - Construir um heap máximo
  - Trocar a raiz – o maior elemento – com o elemento da última posição do vetor
  - Diminuir o tamanho do heap em 1
  - Rearranjar o heap máximo (agora menor), se necessário
  - Repetir o processo n-1 vezes

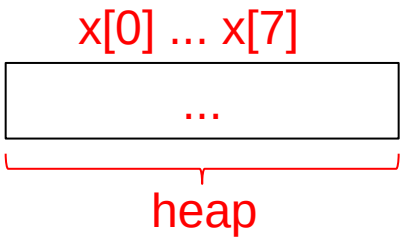
HeapSort: exemplo

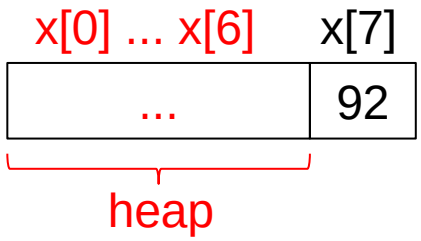


1) Monta-se o heap com base no vetor desordenado

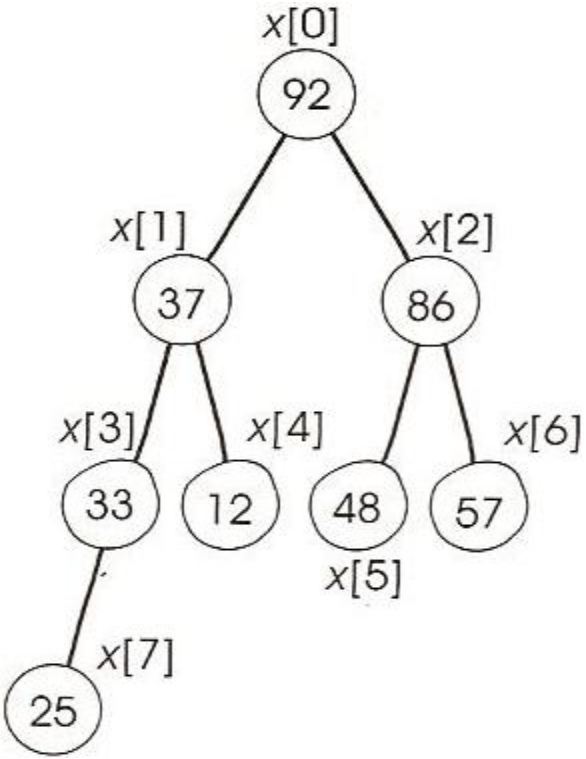


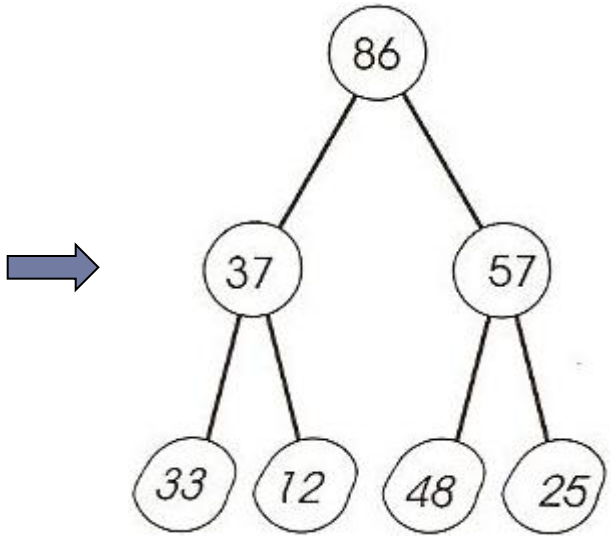
HeapSort: exemplo





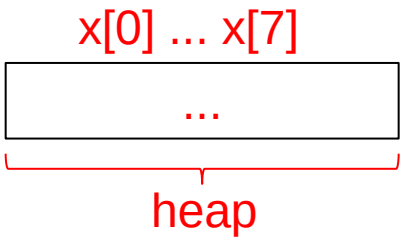
1) Monta-se o heap com base no vetor desordenado

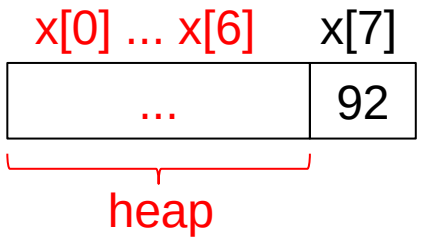


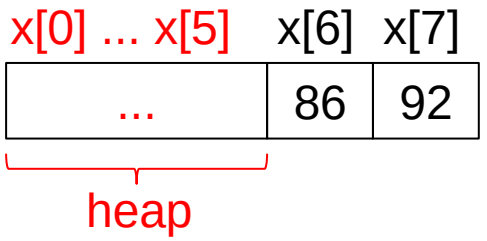


2) Troca-se a raiz (maior elemento) com o último elemento (x[7]) e rearranja-se o heap

HeapSort: exemplo

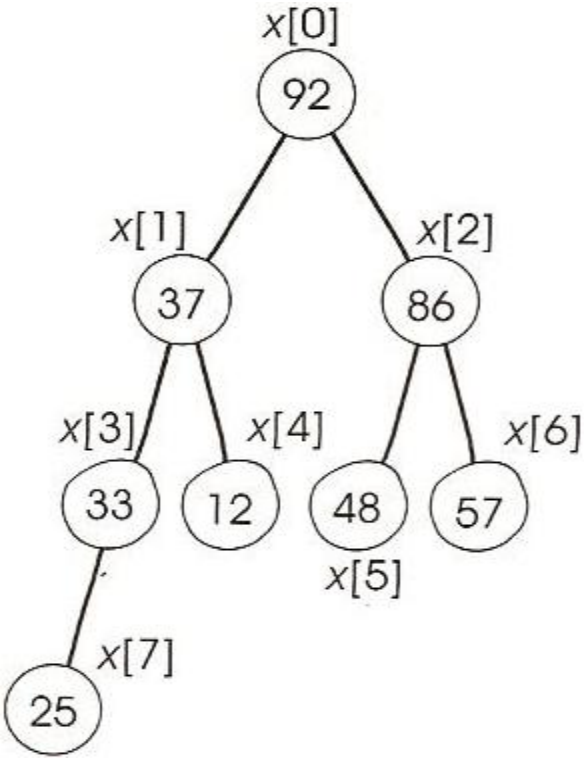


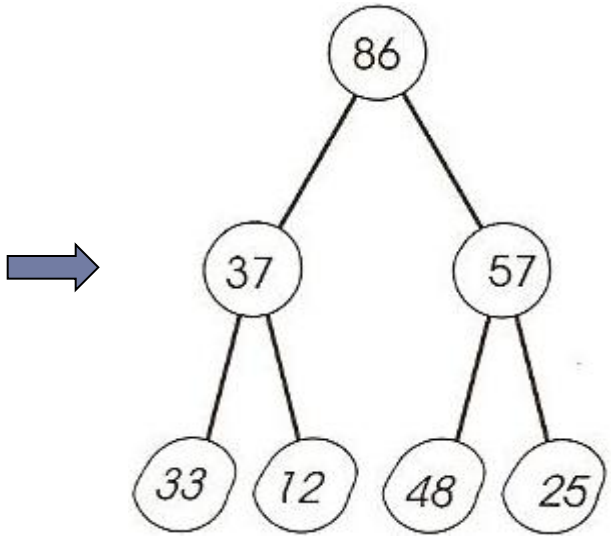


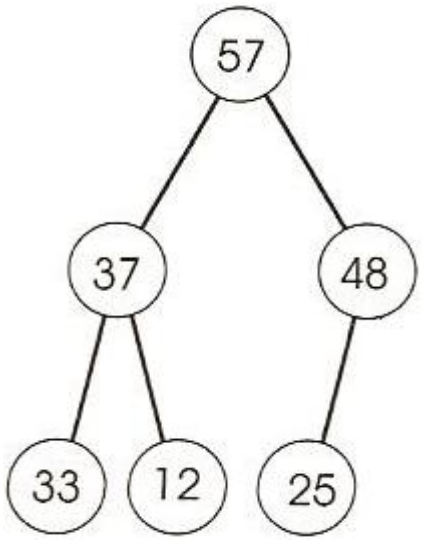


1) Monta-se o heap com base no vetor desordenado

3) Troca-se a raiz com o último elemento (x[6]) e rearranja-se o heap

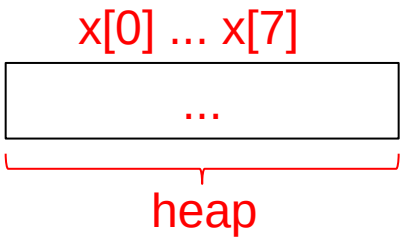


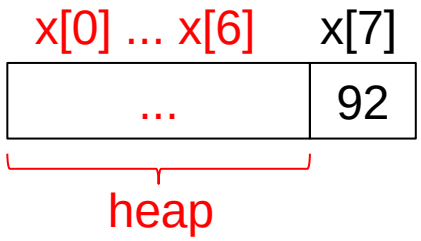


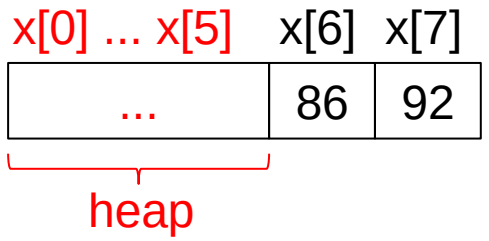


2) Troca-se a raiz (maior elemento) com o último elemento (x[7]) e rearranja-se o heap

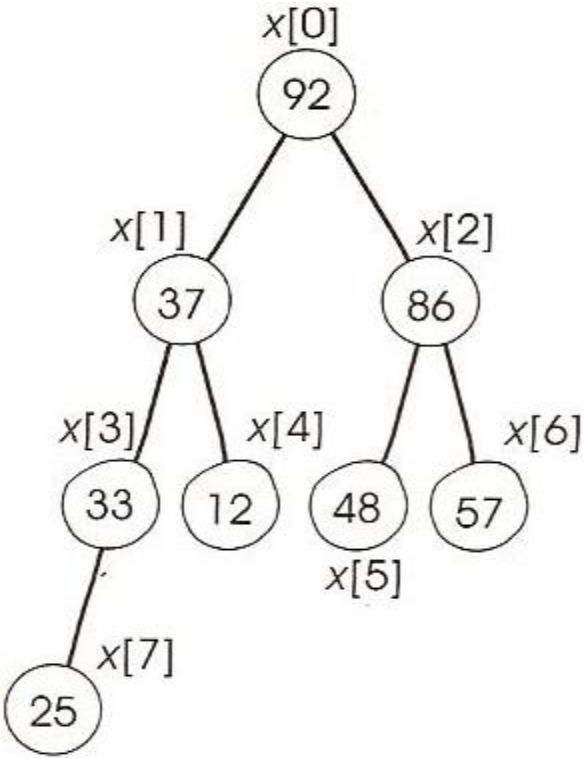
HeapSort: exemplo

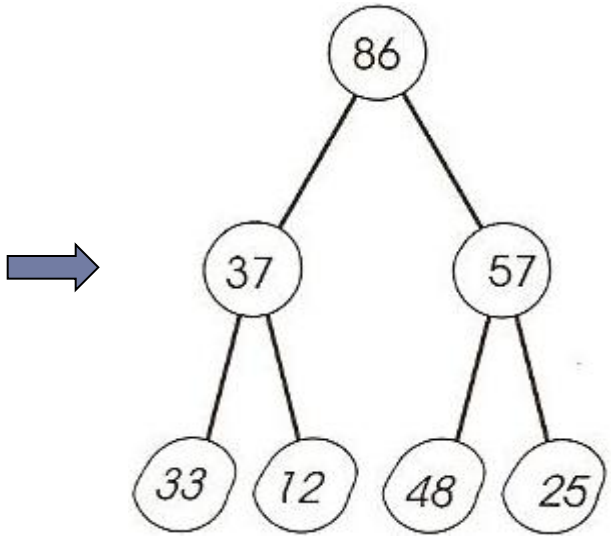






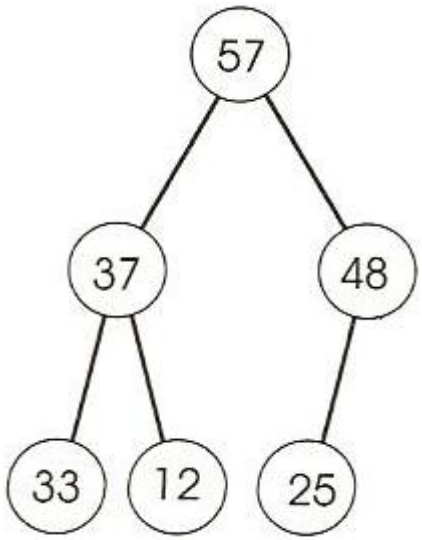
1) Monta-se o heap com base no vetor desordenado





2) Troca-se a raiz (maior elemento) com o último elemento (x[7]) e rearranja-se o heap

3) Troca-se a raiz com o último elemento (x[6]) e rearranja-se o heap



HeapSort

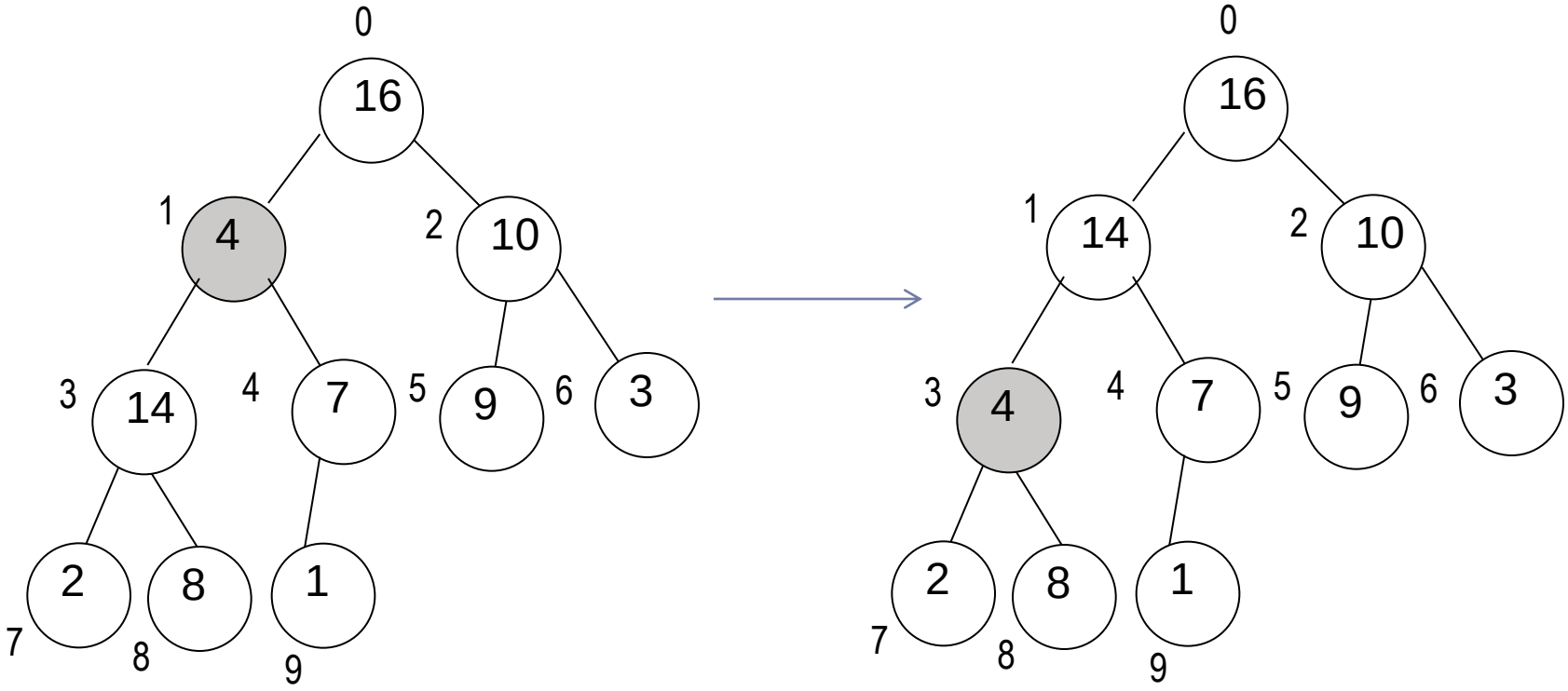
- O processo continua até todos os elementos terem sido incluídos no vetor de forma ordenada

- É necessário:
  - Saber construir um heap a partir de um vetor qualquer
    - Procedimento *construir_heap*
  - Saber como rearranjar o heap, ou seja, manter a propriedade de heap máximo
    - Procedimento *rearranjar_heap*

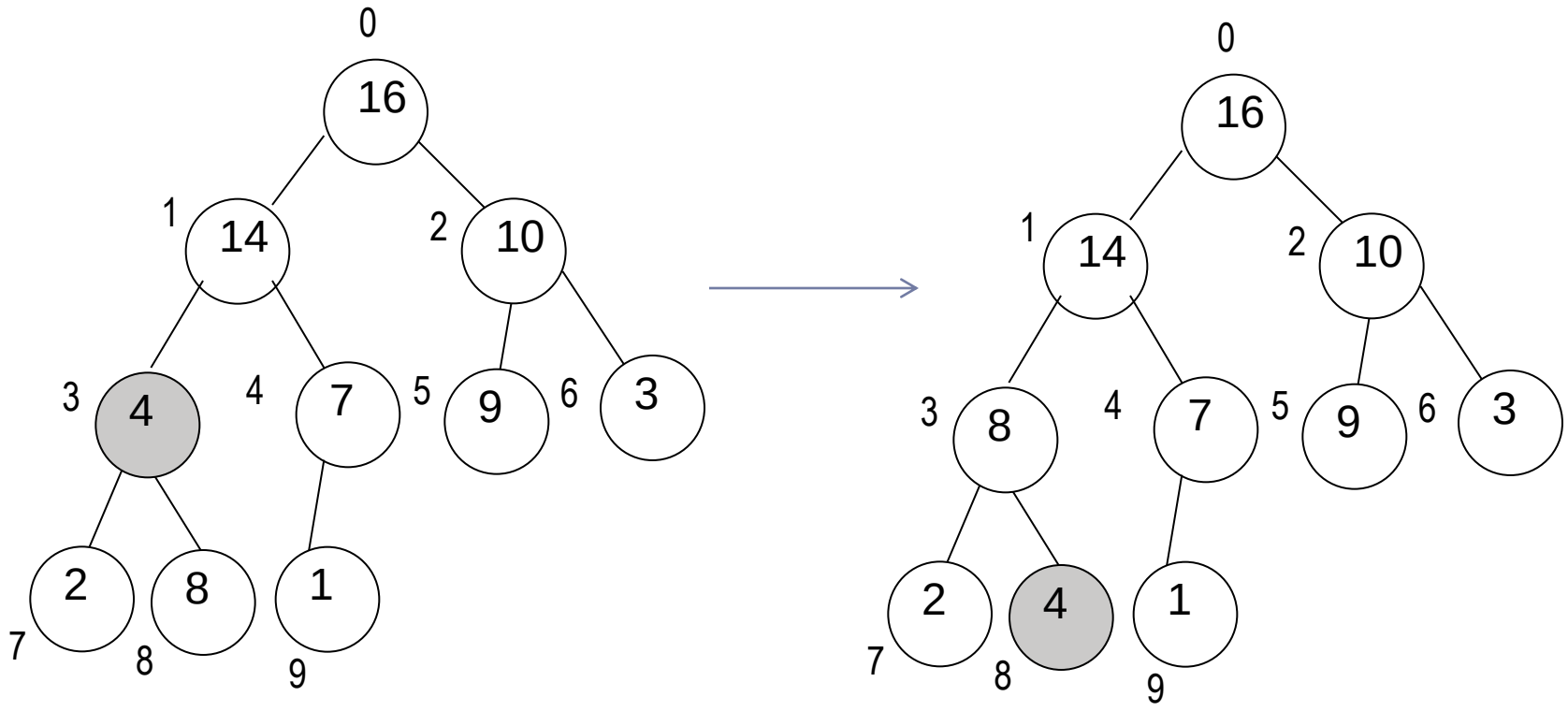
HeapSort

- Procedimento *rearranjar_heap*: manutenção da propriedade de heap máximo
  - Assume que as árvores binárias com raízes nos filhos esquerdo e direito de i são heap máximos, mas que A[i] pode ser menor que seus filhos, violando a propriedade de heap máximo
  - A função do procedimento *rearranjar_heap* é deixar A[i] "escorregar" para a posição correta, de tal forma que a subárvore com raiz em i torne-se um heap máximo

HeapSort

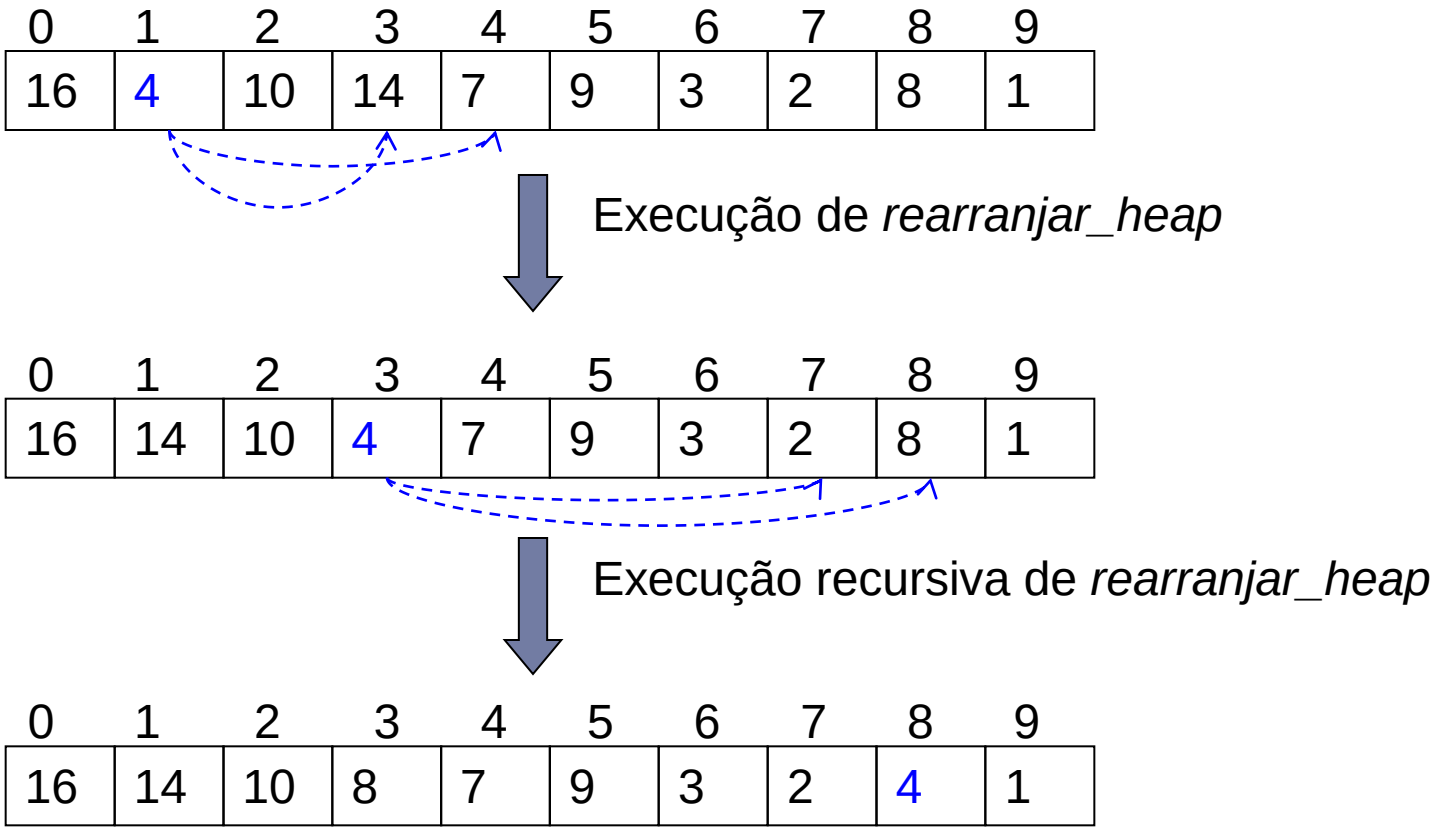


HeapSort



HeapSort

- Na prática, trabalha-se com o vetor



HeapSort

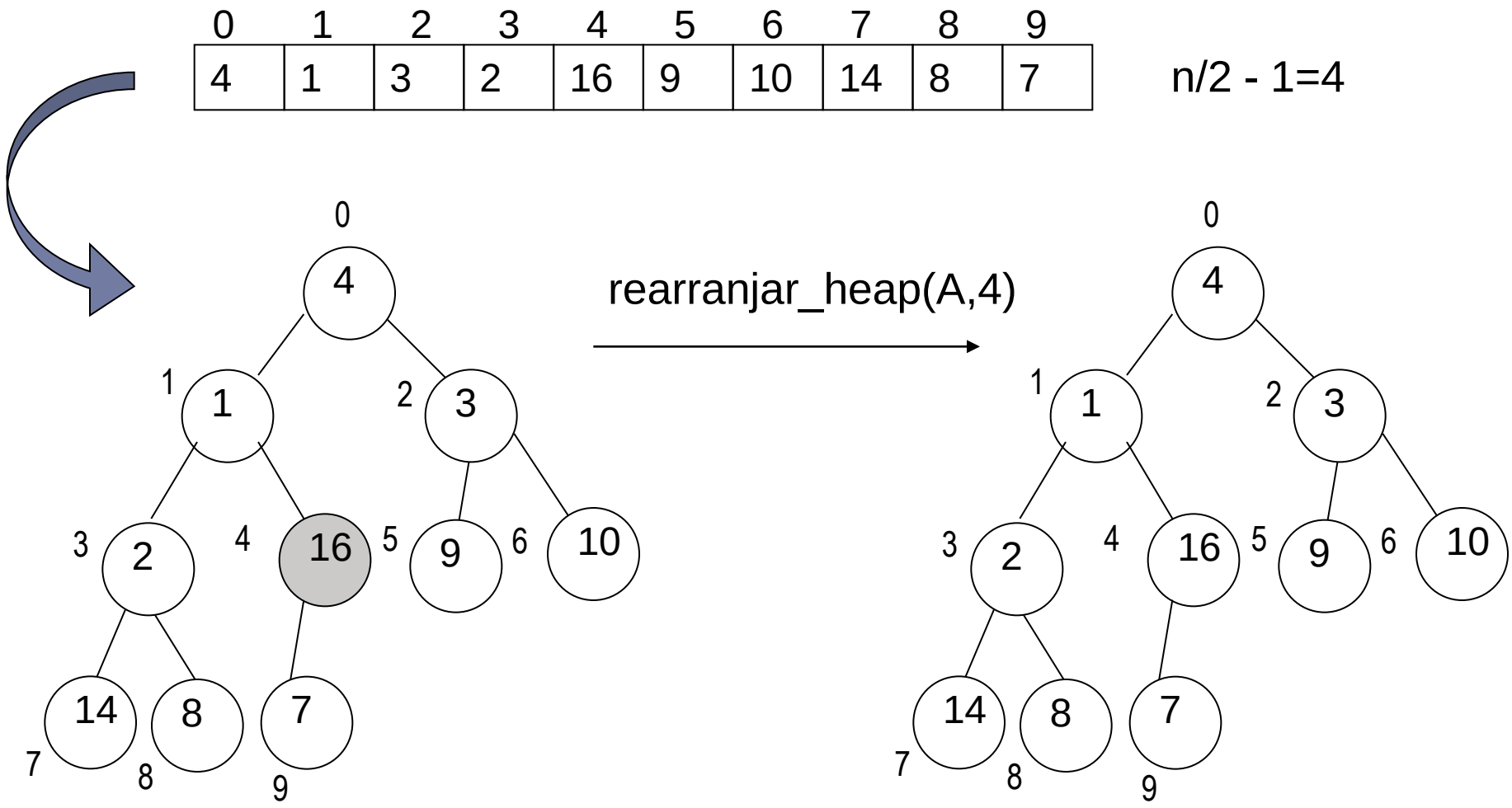
```
void rearranjar_heap(int v[], int i, int tamanho_do_heap) {
   int esq, dir, maior, aux;
   esq = 2 * i + 1;
   dir = 2 * i + 2;
   if ((esq < tamanho_do_heap) && (v[esq] > v[i]))
      maior = esq;
   else
      maior = i;
   if ((dir < tamanho_do_heap) && (v[dir] > v[maior]))
      maior = dir;
   if (maior != i) {
      aux = v[i];
      v[i] = v[maior];
      v[maior] = aux;
      rearranjar_heap(v, maior, tamanho_do_heap);
   }
}
```

nó a partir do qual é necessário rearranjar

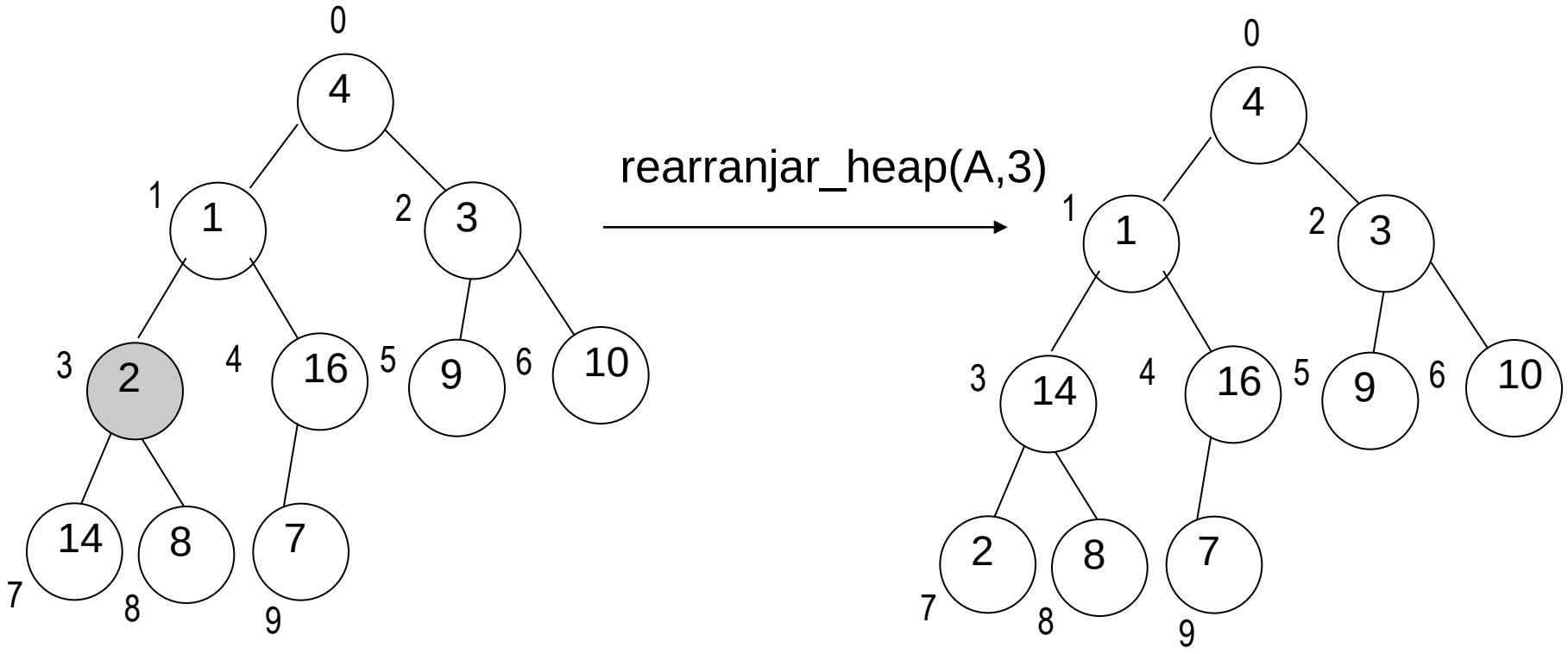
HeapSort

- Procedimento *construir_heap*
  - Percorre de forma ascendente os primeiros n/2 - 1 nós (que não são folhas) e executa o procedimento *rearranjar_heap*
  - A cada chamada do *rearranjar_heap* para um nó, as duas árvores com raiz neste nó tornam-se heaps máximos
  - Ao chamar o *rearranjar_heap* para a raiz, o heap máximo completo é obtido

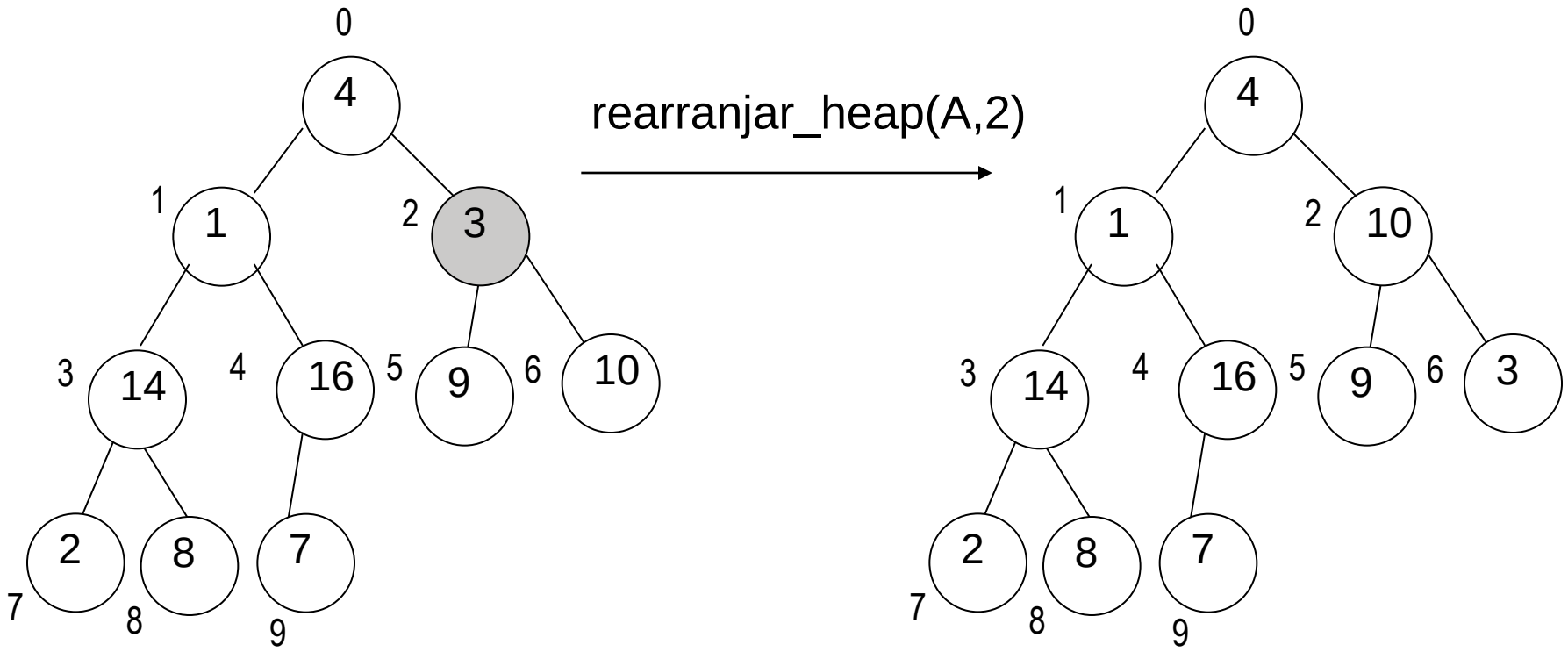
HeapSort


0 1 2 3 4 5 6 7 8 9
4 1 3 2 16 9 10 14 8 7
n/2 - 1=4
rearranjar_heap(A,4)
0 4
1 1
2 3
3 2
4 16
5 9
6 10
7 14
8 8
9 7
0 4
1 1
2 3
3 2
4 16
5 9
6 10
7 14
8 8
9 7

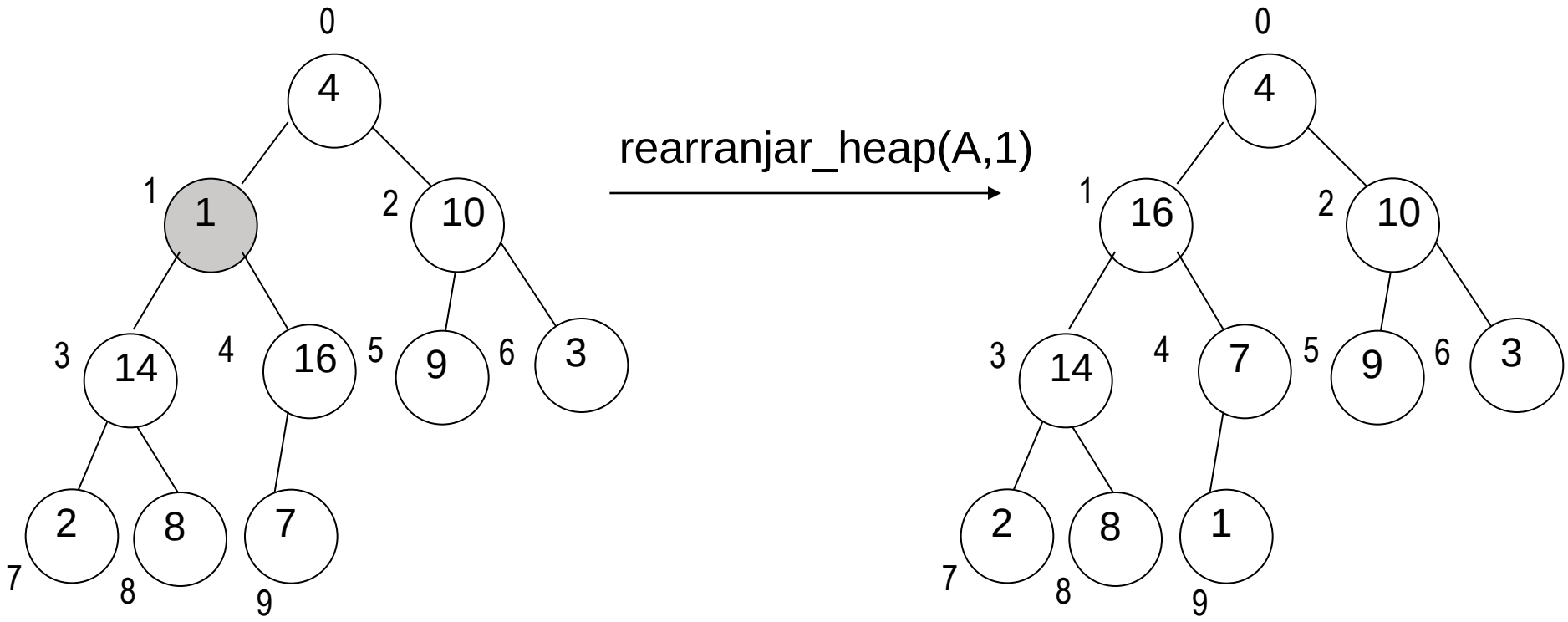
HeapSort


0
4
1
1
2
3
3
2
4
16
5
9
6
10
14
8
7
7
8
9
rearranjar_heap(A,3)
0
4
1
1
2
3
3
14
4
16
5
9
6
10
2
8
7
7
8
9

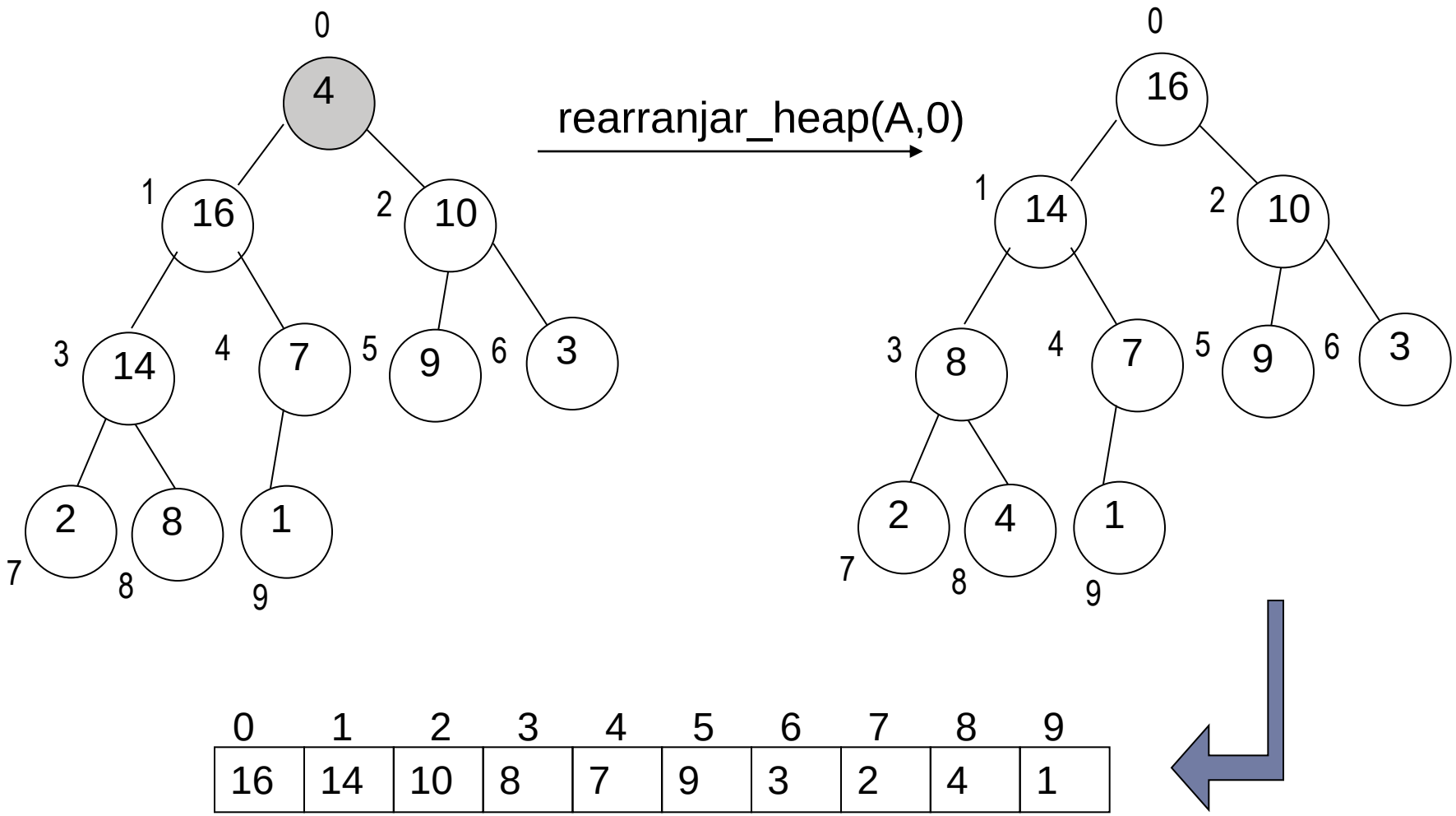
HeapSort


0
4
1
1
2
3
3
14
4
16
5
9
6
10
7
2
8
7
8
9
rearranjar_heap(A,2)
0
4
1
1
2
10
3
14
4
16
5
9
6
3
7
2
8
7
8
9

HeapSort


0
4
1
1
2
10
3
14
4
16
5
9
6
3
7
2
8
7
8
9
rearranjar_heap(A,1)
0
4
1
16
2
10
3
14
4
7
5
9
6
3
7
2
8
1
8
9

HeapSort


0
4
1
16
2
10
3
14
4
7
5
9
6
3
7
2
8
8
1
9
rearranjar_heap(A,0)
0
16
1
14
2
10
3
8
4
7
5
9
6
3
7
2
4
8
1
9
0 1 2 3 4 5 6 7 8 9
16 14 10 8 7 9 3 2 4 1

HeapSort

```
void construir_heap(int v[], int n) {
  int i;
  for (i = n/2-1; i >= 0; i--)
       rearranjar_heap(v, i, n);
}
```

Simulação (applet Java) em:
http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/heap/heapen.htm

HeapSort

- Procedimento **heap-sort**
  1. Construir um heap máximo (via **construir_heap**)
  2. Trocar a raiz – o maior elemento – com o elemento da última posição do vetor
  3. Diminuir o tamanho do heap em 1
  4. Ajustar heap, se necessário (via **rearranjar_heap**)
  5. Repetir o processo n-1 vezes

HeapSort

- Dado o vetor:

0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

- Chamar construir_heap e obter:

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

- Executar os passos de 2 a 4 n – 1 vezes

HeapSort

```
void heapsort(int v[], int n) {
  int i, aux, tamanho_do_heap;
  construir_heap(v, n);
  tamanho_do_heap = n;
  for (i = n - 1; i > 0; i--) {
       aux = v[0];
       v[0] = v[i];
       v[i] = aux;
       tamanho_do_heap--;
       rearranjar_heap(v, 0, tamanho_do_heap);
  }
}
```

HeapSort

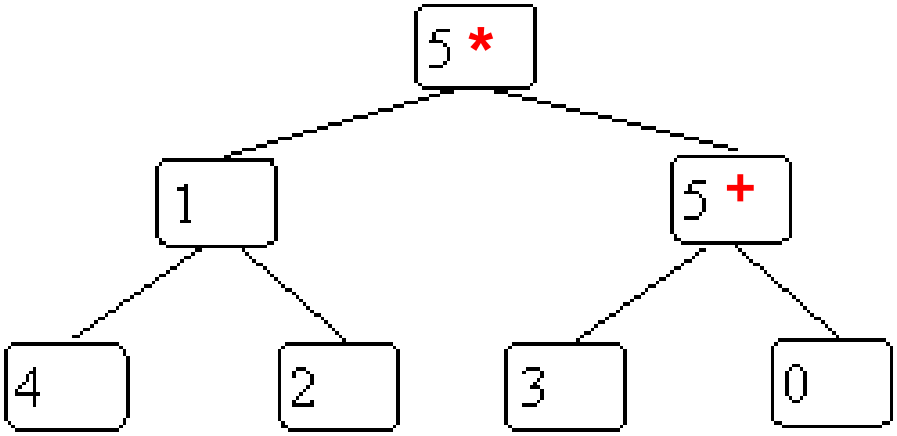
- Executar o processo de ordenação completo para o vetor abaixo:

(44 , 55 , 12 , 42 , 94)

HeapSort

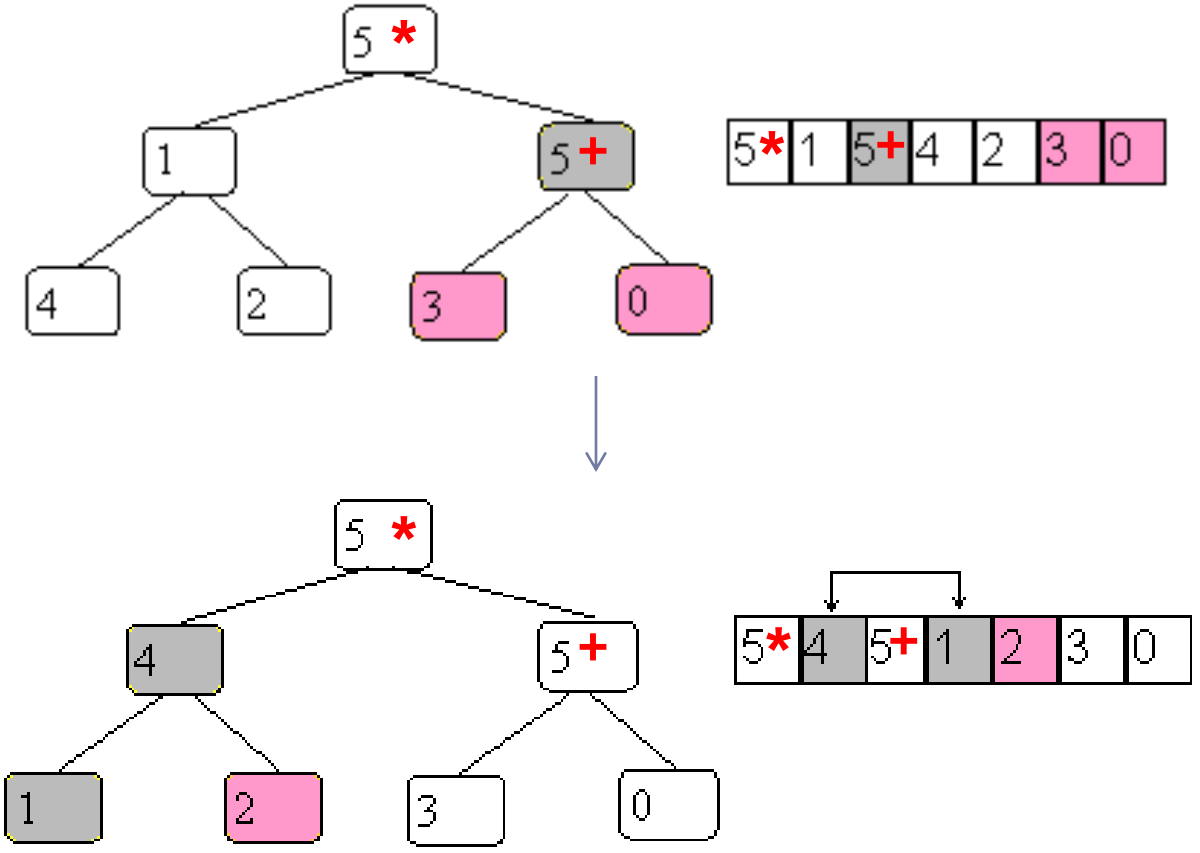
HeapSort é estável ?

HeapSort

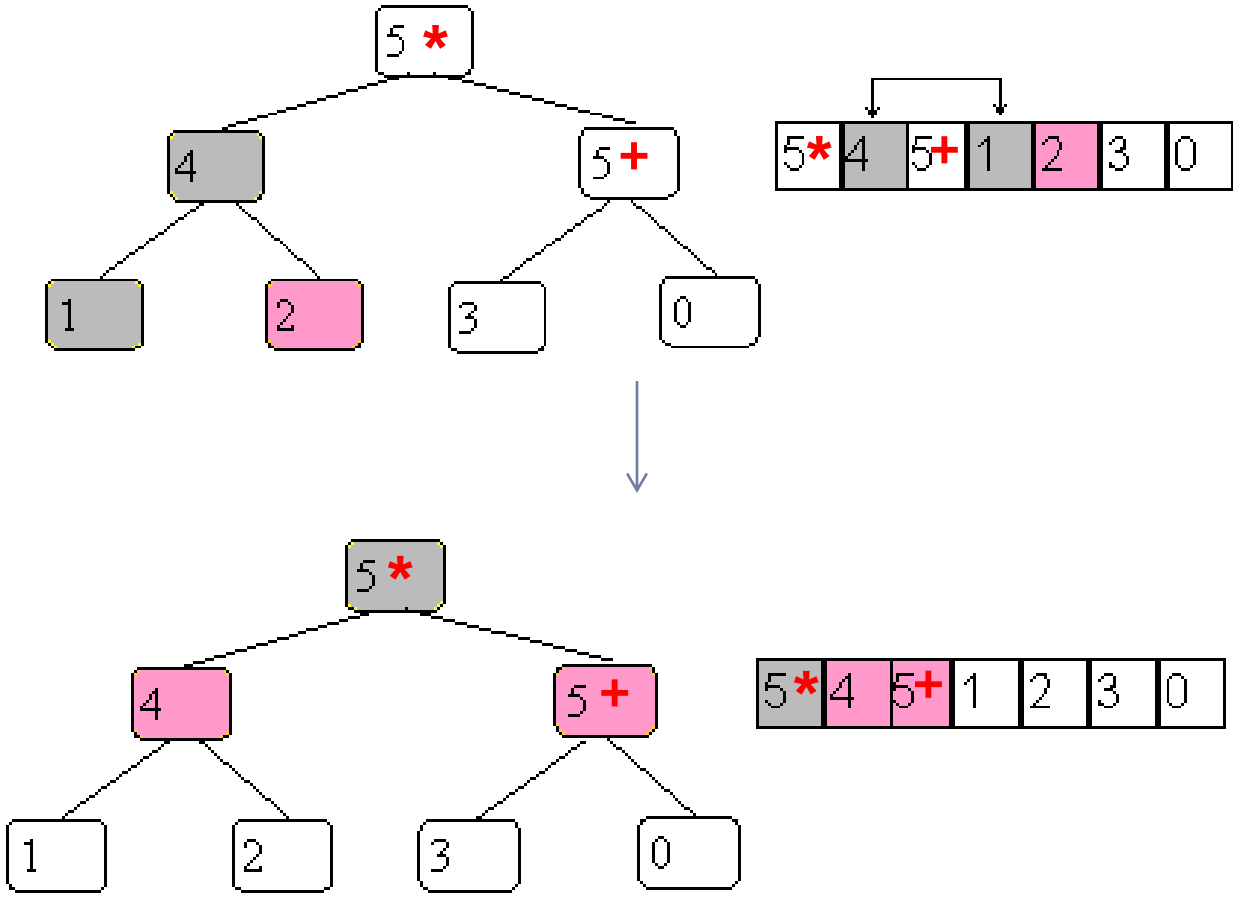


5*	1	5+	4	2	3	0

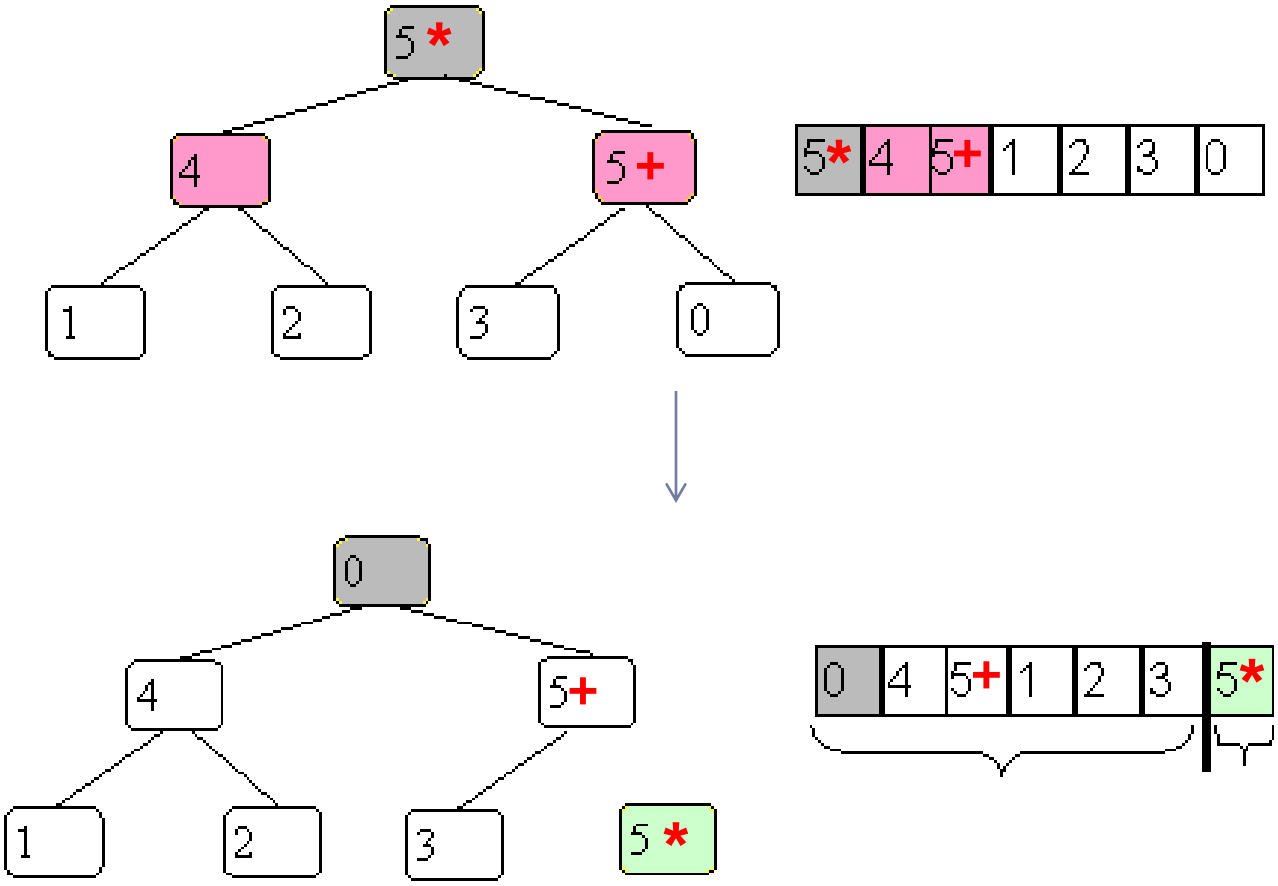
HeapSort



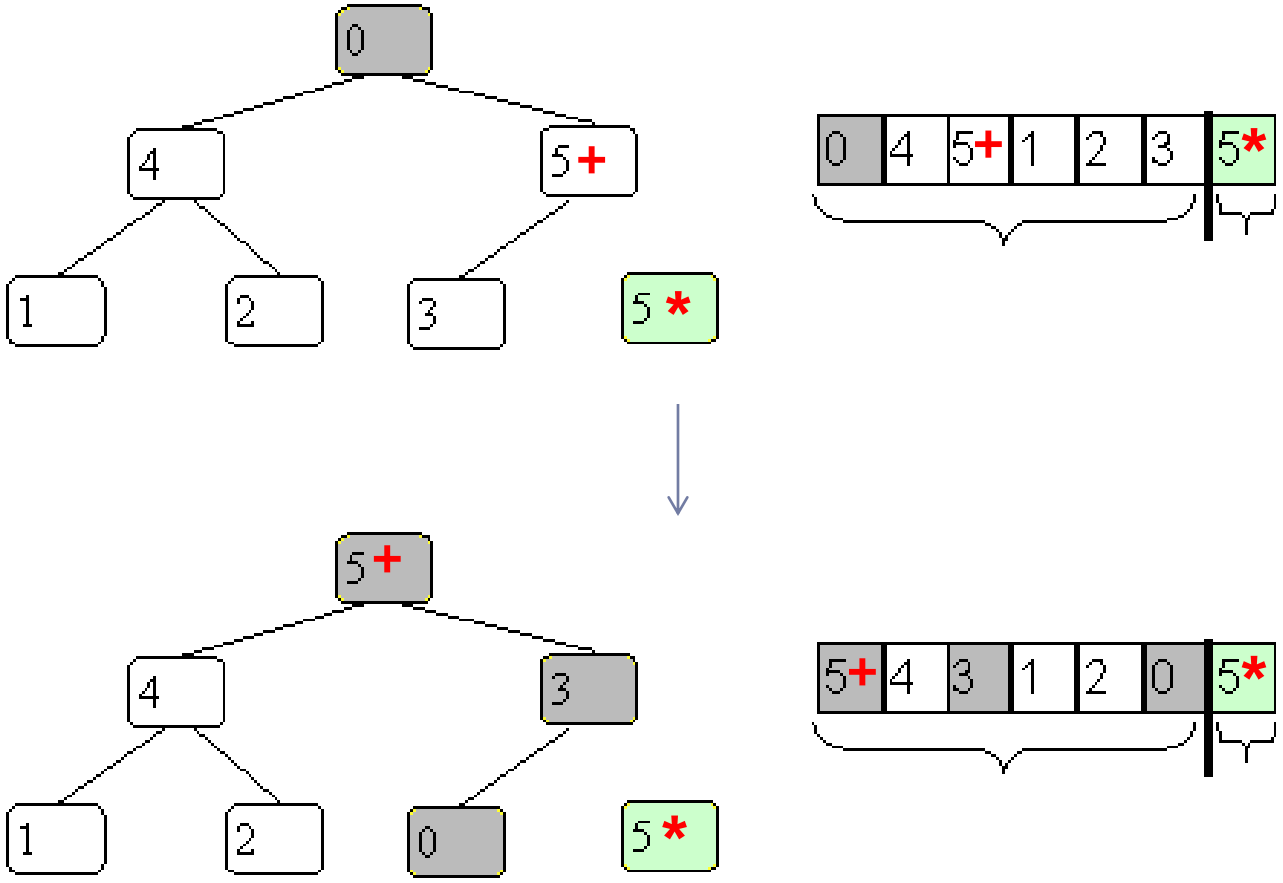
HeapSort



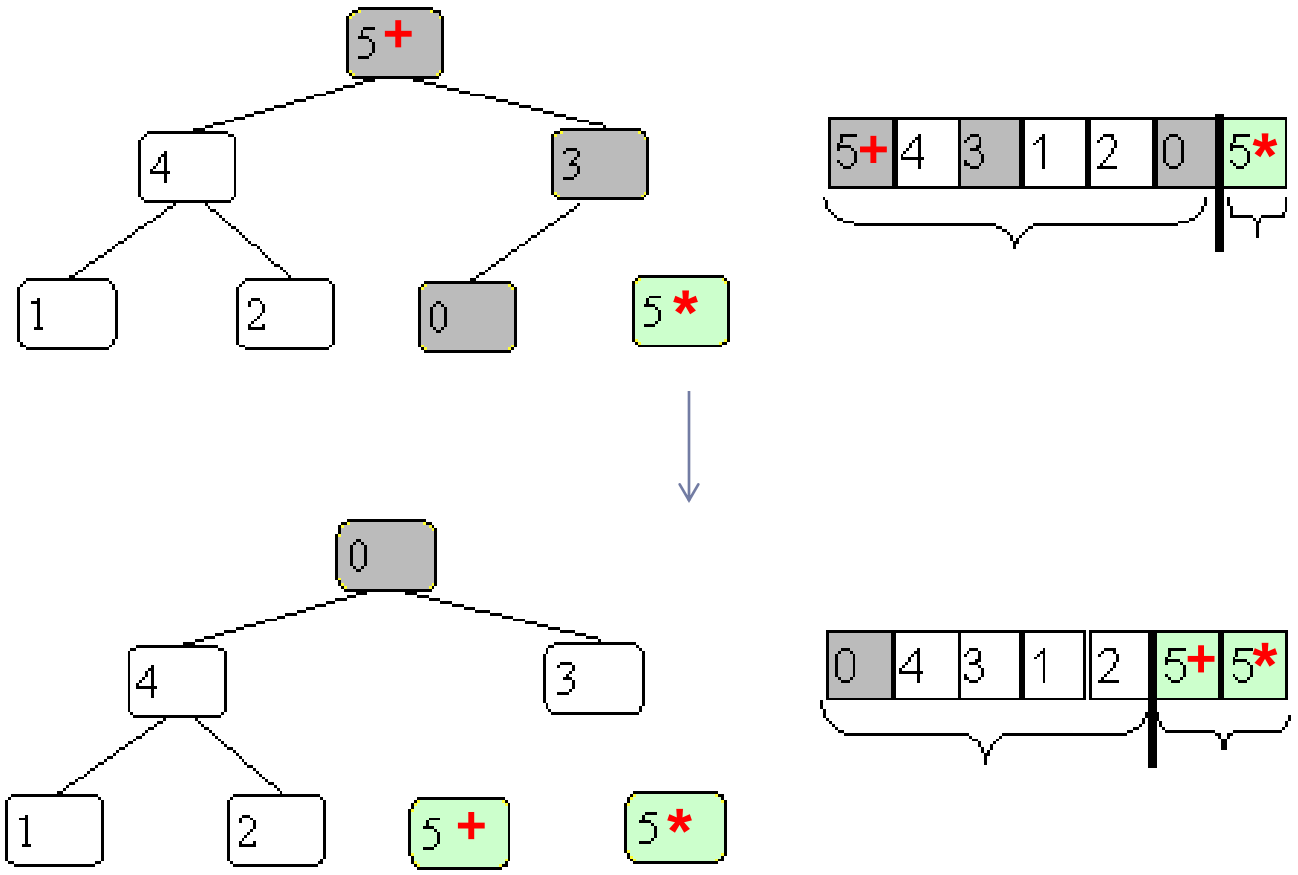
HeapSort



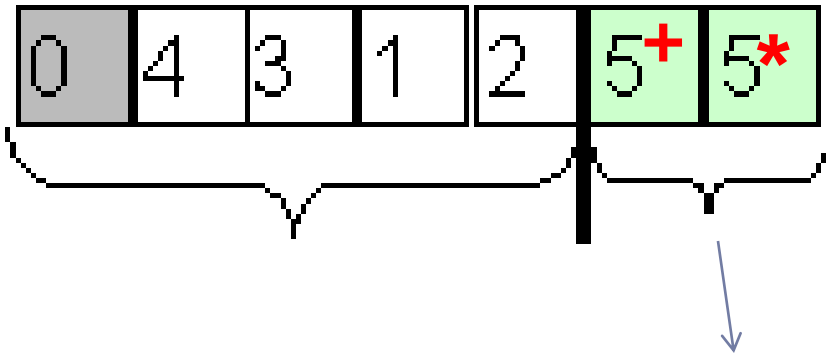
HeapSort



HeapSort



HeapSort



Parte já ordenada, não vai mudar.
A posição relativa desses dois números foi alterada.

Heapsort é instável

HeapSort

Qual a complexidade do HeapSort?

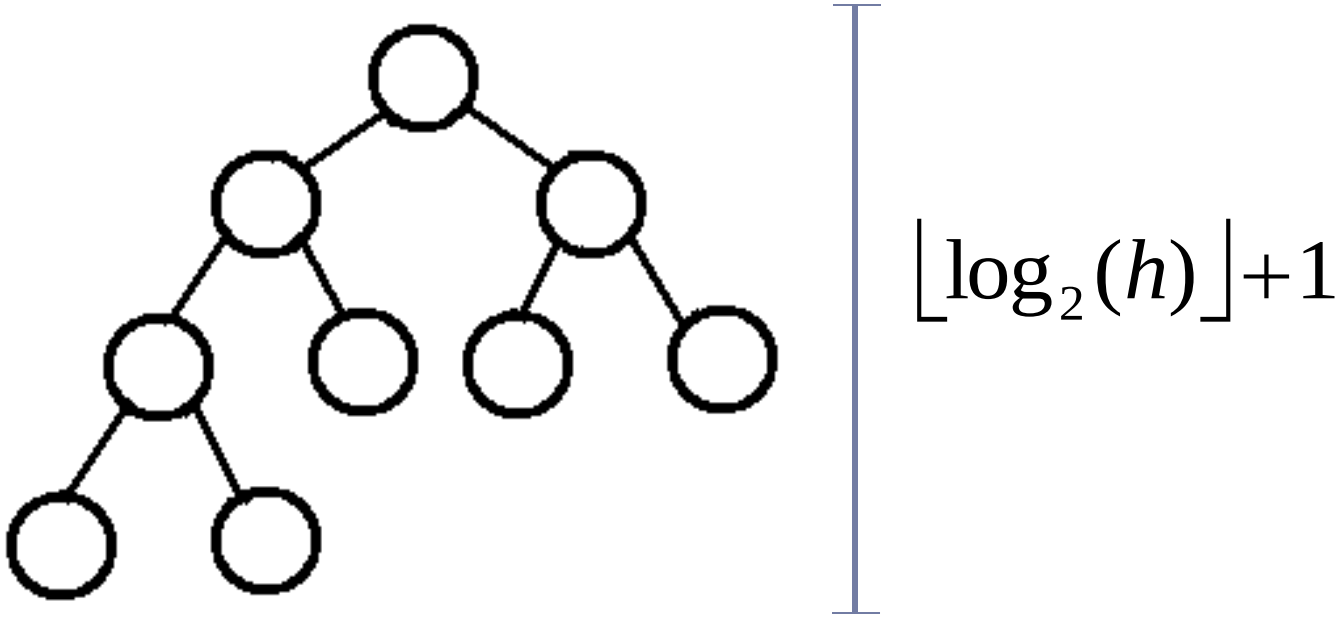
HeapSort

```
void rearranjar_heap(int v[], int i, int tamanho_do_heap) {
   int esq, dir, maior, aux;
   esq = 2 * i + 1;
   dir = 2 * i + 2;
   if ((esq < tamanho_do_heap) && (v[esq] > v[i]))
      maior = esq;
   else
      maior = i;
   if ((dir < tamanho_do_heap) && (v[dir] > v[maior]))
      maior = dir;
   if (maior != i) {
      aux = v[i];
      v[i] = v[maior];
      v[maior] = aux;
      rearranjar_heap(v, maior, tamanho_do_heap);
   }
}
```

5 comparações

HeapSort

- rearranjar_heap
  - 5 comparações realizadas no máximo $\lfloor \log_2(h) \rfloor + 1$ vezes, onde *h* é o número de elementos no heap



HeapSort

```
void construir_heap(int v[], int n) {
  int i;
  for (i = n/2-1; i >= 0; i--)
      rearranjar_heap(v, i, n);
}
```

$\lfloor n/2 \rfloor$ vezes

$5(\lfloor \log_2(n-i) \rfloor + 1)$

HeapSort

- construir_heap

i=$\lfloor n/2 \rfloor - 1$ → $5(\lfloor \log_2(n - \lfloor n/2 \rfloor - 1) \rfloor + 1)$

...

i=2 → $5(\lfloor \log_2(n-2) \rfloor + 1)$

i=1 → $5(\lfloor \log_2(n-1) \rfloor + 1)$

i=0 → $5(\lfloor \log_2(n-0) \rfloor + 1)$

HeapSort

- construir_heap

$i=\lfloor n/2 \rfloor - 1 \rightarrow 5(\lfloor \log_2(n - \lfloor n/2 \rfloor - 1) \rfloor + 1)$

...

$i=2 \rightarrow 5(\lfloor \log_2(n-2) \rfloor + 1)$

$i=1 \rightarrow 5(\lfloor \log_2(n-1) \rfloor + 1)$

$i=0 \rightarrow 5(\lfloor \log_2(n-0) \rfloor + 1)$

Soma ?

HeapSort

Aproximação da soma:

$$5(\log_2(n)+1)+5(\log_2(n-1)+1)+5(\log_2(n-2)+1)+\ldots+5(\log_2(\lceil n/2\rceil-1)+1)=$$

$$=5\sum_{j=\lceil n/2\rceil-1}^{n}\log_2(j)+1=5\left(\sum_{j=\lceil n/2\rceil-1}^{n}1+\sum_{j=\lceil n/2\rceil-1}^{n}\log_2(j)\right)=5\left((\lceil n/2\rceil+2)+\sum_{j=\lceil n/2\rceil-1}^{n}\log_2(j)\right)=$$

$$=\lceil 5n/2\rceil+10+5\left(\sum_{j=\lceil n/2\rceil-1}^{n}\log_2(j)\right)=\lceil 5n/2\rceil+10+5\left(\sum_{j=1}^{n}\log_2(j)-\sum_{j=1}^{\lceil n/2\rceil-2}\log_2(j)\right)=$$

$$=\lceil 5n/2\rceil+10+5(\log_2 n!-\log_2(\lceil n/2\rceil-2)!)=\quad ?$$

HeapSort

$\log_2 n! \approx n \log_2 n - n$ *Aproximação de Stirling*

$$= \lceil 5n/2 \rceil + 10 + 5(\log_2 n! - \log_2 (\lceil n/2 \rceil - 2)!) =$$

$$= \lceil 5n/2 \rceil + 10 + 5(n \log_2 n - n - (\lceil n/2 \rceil - 2)\log_2 (\lceil n/2 \rceil - 2) + (\lceil n/2 \rceil - 2)) =$$

$$= \lceil 5n/2 \rceil + 10 + 5n \log_2 n - 5n - 5(\lceil n/2 \rceil - 2)\log_2 (\lceil n/2 \rceil - 2) + \lceil 5n/2 \rceil - 10 =$$

$$= \lceil 10n \rceil + 5n \log_2 n - 5n - (\lceil 5n/2 \rceil - 10)\log_2 (\lceil n/2 \rceil - 2)$$

$$T_{construir\_heap}(n) \approx 5n \log_2 n - \lceil 5n/2 \rceil \log_2 (\lceil n/2 \rceil - 2) + 10 \log_2 (\lceil n/2 \rceil - 2) + 5n$$

HeapSort

$\log_2 n! \approx n \log_2 n - n$ *Aproximação de Stirling*

$$= \lceil 5n/2 \rceil + 10 + 5(\log_2 n! - \log_2 (\lceil n/2 \rceil - 2)!) =$$

$$= \lceil 5n/2 \rceil + 10 + 5(n \log_2 n - n - (\lceil n/2 \rceil - 2)\log_2 (\lceil n/2 \rceil - 2) + (\lceil n/2 \rceil - 2)) =$$

$$= \lceil 5n/2 \rceil + 10 + 5n \log_2 n - 5n - 5(\lceil n/2 \rceil - 2)\log_2 (\lceil n/2 \rceil - 2) + \lceil 5n/2 \rceil - 10 =$$

$$= \lceil 10n \rceil + 5n \log_2 n - 5n - (\lceil 5n/2 \rceil - 10)\log_2 (\lceil n/2 \rceil - 2)$$

$$T_{construir\_heap}(n) \approx 5n \log_2 n - \lceil 5n/2 \rceil \log_2 (\lceil n/2 \rceil - 2) + 10 \log_2 (\lceil n/2 \rceil - 2) + 5n$$

$T_{construir\_heap}(n) \leq 5n \log_2 n$, para $n_0 >= 269$

HeapSort

$$\log_2 n! \approx n\log_2 n - n$$

Aproximação de Stirling

$$= \lceil 5n/2 \rceil + 10 + 5(\log_2 n! - \log_2(\lceil n/2 \rceil - 2)!) =$$

$$= \lceil 5n/2 \rceil + 10 + 5(n\log_2 n - n - (\lceil n/2 \rceil - 2)\log_2(\lceil n/2 \rceil - 2) + (\lceil n/2 \rceil - 2)) =$$

$$= \lceil 5n/2 \rceil + 10 + 5n\log_2 n - 5n - 5(\lceil n/2 \rceil - 2)\log_2(\lceil n/2 \rceil - 2) + \lceil 5n/2 \rceil - 10 =$$

$$= \lceil 10n \rceil + 5n\log_2 n - 5n - (\lceil 5n/2 \rceil - 10)\log_2(\lceil n/2 \rceil - 2)$$

$$T_{construir\_heap}(n) \approx 5n\log_2 n - \lceil 5n/2 \rceil \log_2(\lceil n/2 \rceil - 2) + 10\log_2(\lceil n/2 \rceil - 2) + 5n$$

$T_{construir\_heap}(n) \leq 5n\log_2 n$, para $n_0 >= 269$

Portanto, $T_{construir\_heap}(n) = O(n\log_2 n)$

HeapSort

```
void heapsort(int v[], int n) {
  int i, aux, tamanho_do_heap;
  construir_heap(v, n);
  tamanho_do_heap = n;
  for (i = n - 1; i > 0; i--) {
       aux = v[0];
       v[0] = v[i];
       v[i] = aux;
       tamanho_do_heap--;
       rearranjar_heap(v, 0, tamanho_do_heap);
  }
}
```

n-1 vezes

$5(\lfloor \log_2(i) \rfloor + 1)$

Onde i=tamanho_do_heap

HeapSort

- n-1 vezes rearranjar_heap

i=n-1 → $5(\lfloor \log_2(n-1) \rfloor + 1)$

...

i=3 → $5(\lfloor \log_2 3 \rfloor + 1)$

i=2 → $5(\lfloor \log_2 2 \rfloor + 1)$

i=1 → $5(\lfloor \log_2 1 \rfloor + 1)$

Soma ?

HeapSort

Aproximação da soma:

$$5\left(\log_2(n-1)+1\right)+\ldots+5\left(\log_2 3+1\right)+5\left(\log_2 2+1\right)+5\left(\log_2 1+1\right)=$$

$$=5\sum_{j=1}^{n-1}\log_2(j)+1=5\sum_{j=1}^{n-1}\log_2(j)+\sum_{j=1}^{n-1}5=5\log_2(n-1)!+5(n-1)=\ ?$$

HeapSort

Aproximação da soma:

$$5(\log_2(n-1)+1)+\ldots+5(\log_2 3+1)+5(\log_2 2+1)+5(\log_2 1+1)=$$

$$=5\sum_{j=1}^{n-1}\log_2(j)+1=5\sum_{j=1}^{n-1}\log_2(j)+\sum_{j=1}^{n-1}5=5\log_2(n-1)!+5(n-1)=$$

$$=5(n-1)\log_2(n-1)-(n-1)+5(n-1)$$

$$\log_2 n!\approx n\log_2 n-n$$

$$T_{rearranjos\_heap}(n)=5(n-1)\log_2(n-1)+4n-4$$

$T_{rearranjos\_heap}(n)\leq 8n\log_2 n$, para $n_0 >= 108$

Portanto, $T_{rearranjos\_heap}(n) = O(n\log_2 n)$

HeapSort

$$T_{heapsort}(n) = T_{construir\_heap}(n) + T_{rearranjos\_heap}(n)$$

$$T_{heapsort}(n) = \mathrm{O}(n \log_2 n) + \mathrm{O}(n \log_2 n)$$

$$T_{heapsort}(n) = \max\{\mathrm{O}(n \log_2 n), \mathrm{O}(n \log_2 n)\}$$

$$T_{heapsort}(n) = \mathrm{O}(n \log_2 n)$$

HeapSort

$$T_{heapsort}(n) = T_{construir\_heap}(n) + T_{rearranjos\_heap}(n)$$

$$T_{heapsort}(n) = \mathrm{O}(n \log_2 n) + \mathrm{O}(n \log_2 n)$$

$$T_{heapsort}(n) = \max\{\mathrm{O}(n \log_2 n), \mathrm{O}(n \log_2 n)\}$$

$$\boxed{T_{heapsort}(n) = \mathrm{O}(n \log_2 n)}$$

Pior caso do HeapSort

HeapSort

Caso médio e melhor caso: $O(n \log_2 n)$

Uma análise detalhada em:
R. Schaffer, R. Sedgewick. The Analysis of Heapsort.
Journal of Algorithms, 15(1), 76-100, 1993.

HeapSort

Caso médio e melhor caso: $O(n \log_2 n)$

Uma análise detalhada em:
R. Schaffer, R. Sedgewick. The Analysis of Heapsort.
Journal of Algorithms, 15(1), 76-100, 1993.

Complexidade de espaço: $O(n)$