



Heap Sort

Algoritmos e Estruturas de Dados

Cátia Vaz



Árvores

- As árvores são estruturas de dados usadas em diversas aplicações:
 - Bases de dados de grande dimensão.
 - Reconhecimento de frases geradas por linguagens (ex: programas, expressões aritméticas,...).
 - Modelação de sistemas organizacionais (ex: famílias, directórios de um computador, hierarquia de comando de uma empresa,...).
 - Determinação do caminho mais curto entre dois computadores de uma rede.



Árvores

- **Definição:** Uma **árvore** é um par (V, E) de dois conjuntos não vazios em que V é um conjunto finito e E é uma relação binária em V (isto é, $E \subseteq V \times V$), que satisfazem duas condições:
 - entre dois nós existe um único caminho (um **caminho** - *path* - é uma sequência de arestas entre dois nós);
 - um único nó, denominado **raiz** (*root*), só existe como primeiro elemento nos pares de E ; os restantes nós são um segundo elemento dos pares de E (podendo também ser primeiro elemento de alguns pares).
- **Nota:** o conjunto V é designado pelo conjuntos dos **nós** ou **vértices** (vertex) e E é o conjunto dos **arcos** (edges).
- **Nota:** uma árvore é um caso especial de um grafo.



Árvores

- **Definição:** se $(v1, v2) \in E$, $v1$ é nó **ascendente** (*parent*) e $v2$ é nó **filho** (*child*).
 - A raiz é o único nó sem ascendente.
 - Nós sem filhos são designados por **folhas** (*leaves*).
 - Nós com filhos são designados **não-terminais**. Nós não-terminais, distintos da raiz, são designados **intermédios**.
- **Definição:** A árvore é de tipo K , sse todos os nós intermédios tiverem K filhos. Quando $K=2$, a árvore diz-se **binária**.
- **Definição:** O **nível** (*level*) de um nó é o número de arcos entre a raiz e o nó (**raiz tem nível 0**).

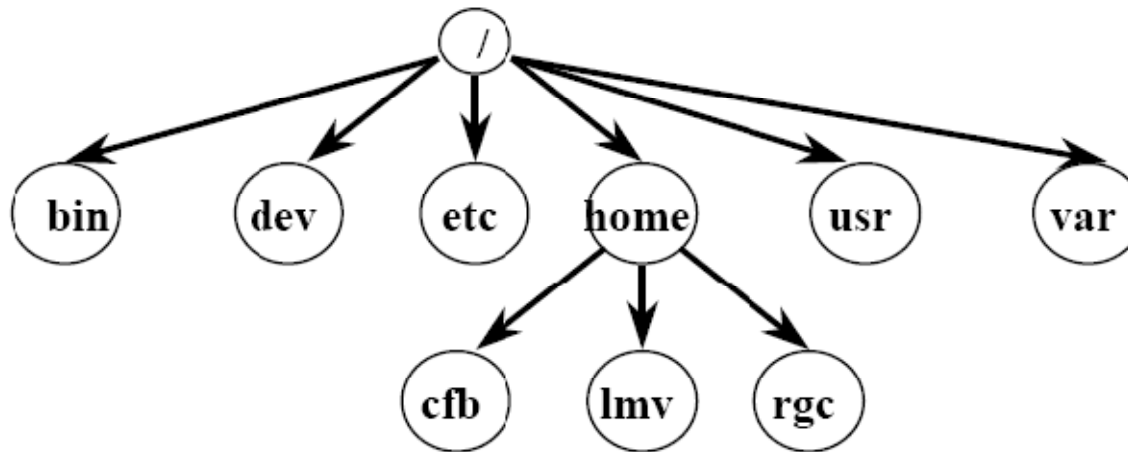


Árvores

- **Definição:** A **altura** (*height*) de uma árvore é o máximo dos níveis das folhas (uma árvore só com a raiz tem altura 0).
- O **percurso** (*path*) é um caminho possível entre dois nós.
- **Representação gráfica das árvores:**
 - raiz no topo,
 - nós representados por círculos,
 - arestas representadas por linhas (ou setas no sentido da raiz para a folha).

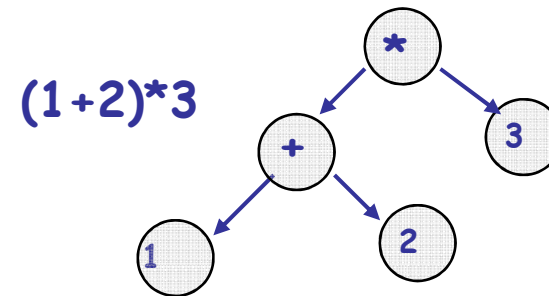
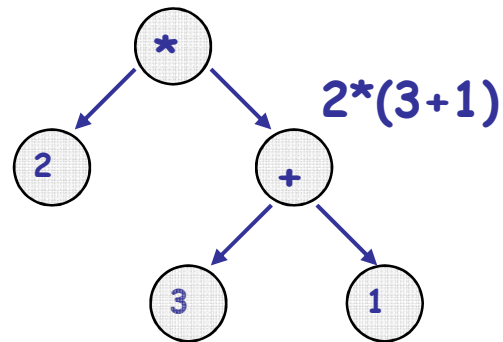
Árvores-Exemplos

- directórios do sistema operativo Unix-like



Árvores-Exemplos

expressões aritméticas



- árvore é avaliada de forma ascendente (*bottom-up*): $2*(3+1) = 8$
- Parêntesis, usados nas expressões aritméticas para ultrapassar as prioridades dos operadores, são indicados na árvore através da posição dos nós



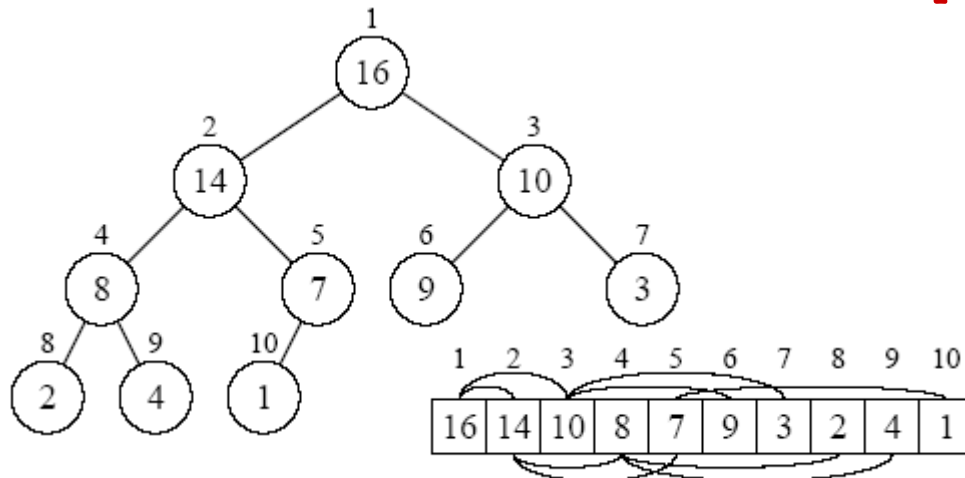
Árvores

- Define-se árvore binária **balanceada**, quando a diferença de **altura** das duas sub-árvores de qualquer nó, é no máximo 1.
- Define-se árvore binária **perfeitamente balanceada**, se relativamente a qualquer nó, a diferença entre o **número** de nós das suas sub-árvores for no máximo 1.
- Define-se **árvore binária completa** se estiver totalmente preenchida, isto é se todos os nós folhas estão no mesmo nível.

Heap (Amontoadado)

- Um *heap* (amontoadado) é uma estrutura de dados representada numa **árvore binária completa** ou quase completa.
- Um *heap* pode ter uma representação num *array* A .
 - A raiz da árvore é $A[1]$
 - O ascendente de $A[i]$ é $A[\lfloor i/2 \rfloor]$
 - O descendente esquerdo de $A[i]$ é $A[2i]$
 - O descendente direito de $A[i]$ é $A[2i+1]$

$$\begin{aligned} \Rightarrow A[0] \\ A[(i-1)/2] \\ A[2i+1] \\ A[2i+2] \end{aligned}$$



```
static int parent(int i ) {  
    return (i-1)>>1;}  
static int left(int i ) {  
    return (i<<1)+1;}  
static int right(int i ) {  
    return (i<<1)+2;}  
}
```

Cátia Vaz

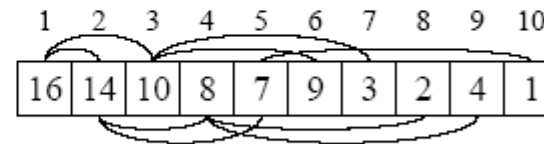
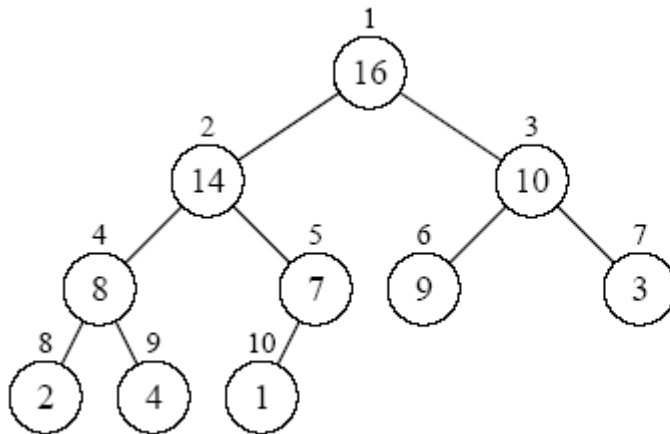


Heap

- Existem dois tipos de *binary heaps*: **max-heaps** e **min-heaps**.
- Em ambos os tipos, os valores nos nós satisfazem uma **propriedade do heap**, a qual depende do tipo de *heap*:
 - num **max-heap**, a propriedade é que para cada nó i excepto a raiz $A[\text{parent}(i)] \geq A[i]$
 - num **min-heap**, a propriedade é que para cada nó i excepto a raiz $A[\text{parent}(i)] \leq A[i]$
- No algoritmo *heap sort* utiliza-se **max-heap**.

Heap Sort

- Tempo de execução: $O(n \lg n)$ - como o *merge sort*.
- Ordena no local - como o *insertion sort*
- Combina o melhor dos dois algoritmos



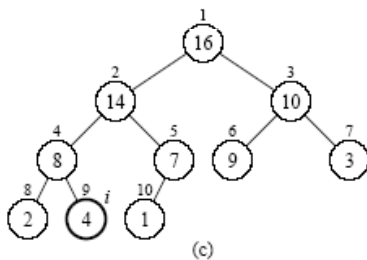
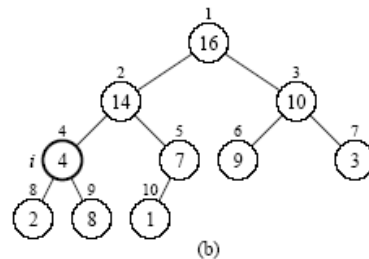
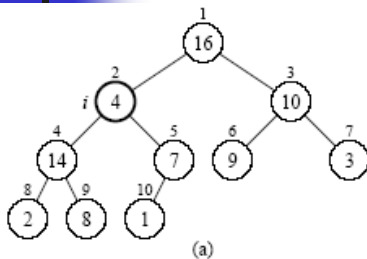


Manter a propriedade do *heap*

- **MAX-HEAPIFY** é uma sub-rotina importante para manipular *max-heaps*.
- **inputs:** um array A e um index i no array.
- quando MAX-HEAPIFY é invocado:
 - assume-se que as arvores com raiz em $\text{LEFT}(i)$ e $\text{RIGHT}(i)$ são *max-heaps*
 - mas $A[i]$ pode ser menor do que os seus filhos, violando a propriedade *max-heap*.

```
MAX-HEAPIFY( $A, i, n$ )  
   $l \leftarrow \text{LEFT}(i)$   
   $r \leftarrow \text{RIGHT}(i)$   
  if  $l \leq n$  and  $A[l] > A[i]$   
    then  $largest \leftarrow l$   
  else  $largest \leftarrow i$   
  if  $r \leq n$  and  $A[r] > A[largest]$   
    then  $largest \leftarrow r$   
  if  $largest \neq i$   
    then exchange  $A[i] \leftrightarrow A[largest]$   
    MAX-HEAPIFY( $A, largest, n$ )
```

Manter a propriedade do *heap*



```

MAX-HEAPIFY( $A, i, n$ )
 $l \leftarrow \text{LEFT}(i)$ 
 $r \leftarrow \text{RIGHT}(i)$ 
if  $l \leq n$  and  $A[l] > A[i]$ 
    then  $\text{largest} \leftarrow l$ 
else  $\text{largest} \leftarrow i$ 
if  $r \leq n$  and  $A[r] > A[\text{largest}]$ 
    then  $\text{largest} \leftarrow r$ 
if  $\text{largest} \neq i$ 
    then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 
    MAX-HEAPIFY( $A, \text{largest}, n$ )
    
```

$$T(n) \leq T(2n/3) + \Theta(1).$$

$$T(n) = O(\lg n)$$



Manter a propriedade do *heap*

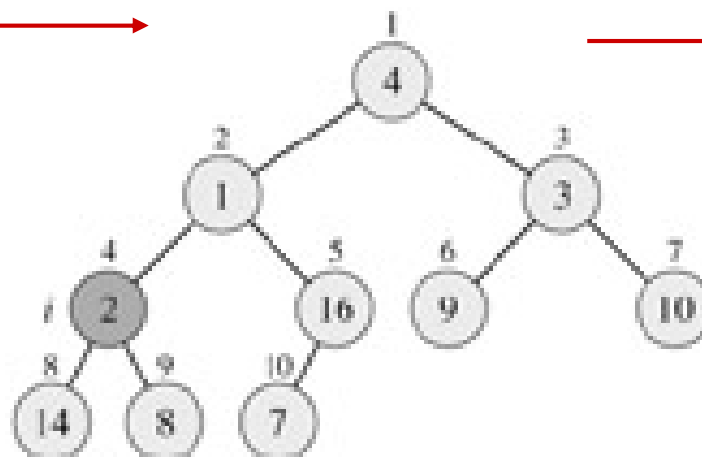
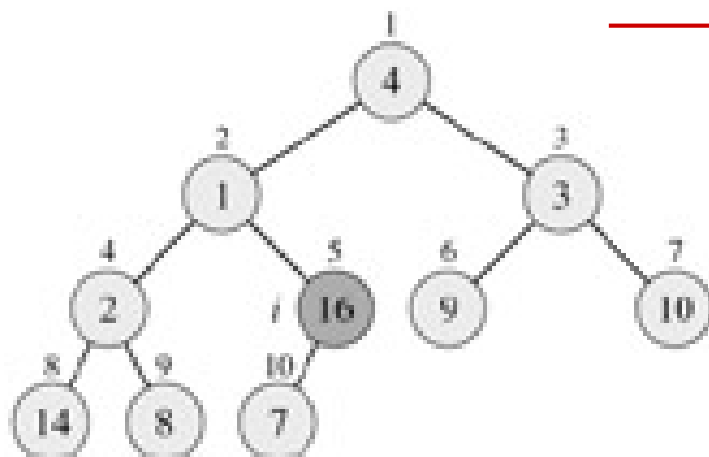
```
public static void heapify(int[] v, int p, int hSize) {
    int l, r, largest;
    l = left(p);
    r = right(p);
    largest=p;
    if(l < hSize && v[l] > v[p]) largest=l;
    if ( r < hSize && v[r] > v[largest]) largest = r;
    if ( largest == p ) return;
    exchange(v, p, largest);
    heapify(v, largest, hSize);
}
public static void exchange(int[] v, int i, int j){
    int tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
}
```

Construir um *heap*

- MAX-HEAPIFY pode ser usado para converter um array $A[1;n]$ num max-heap, onde $n = \text{length}[A]$.
- Nota: os elementos no sub-array $A[(\lfloor n/2 \rfloor + 1); n]$ são todas as folhas da árvore

Exemplo

A [4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7]



```
BUILD-MAX-HEAP( $A, n$ )
  for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1
    do MAX-HEAPIFY( $A, i, n$ )
```

```
static void buildHeap(int[] v, int n){
    int p = parent(n-1);
    for ( ; p >= 0 ; --p)
        heapify(v, p, n);
}
```

Cátia Vaz





Build-MAX-Heap - análise

- O número de nós numa árvore binária que têm altura h é $\leq \lceil n/2^{h+1} \rceil$
- O tempo requerido pelo max-Heapify quando chamado sobre um nó de altura h é $O(h)$
- O custo total do build-max-heap é:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O(n)$$

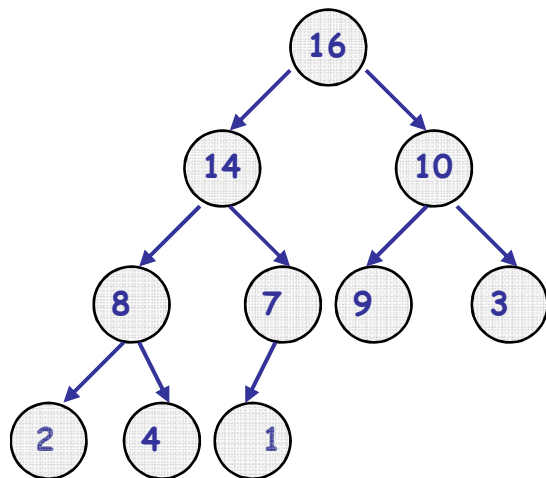
Sabendo que $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ para $x = 1/2$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2 \quad \text{logo} \quad \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} < 2$$

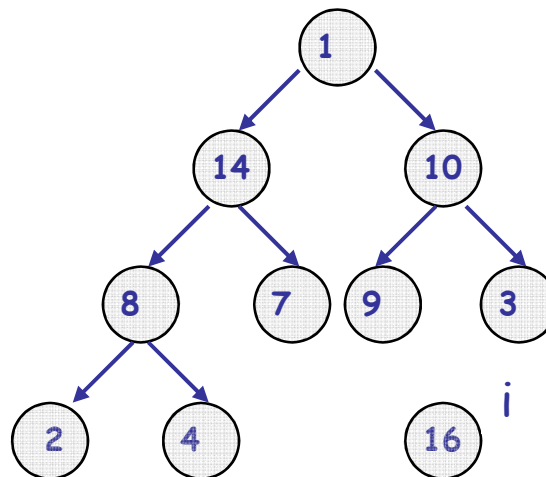
Heap - Sort

```
HEAPSORT( $A, n$ )  
  BUILD-MAX-HEAP( $A, n$ )  
  for  $i \leftarrow n$  downto 2  
    do exchange  $A[1] \leftrightarrow A[i]$   
       MAX-HEAPIFY( $A, 1, i - 1$ )
```

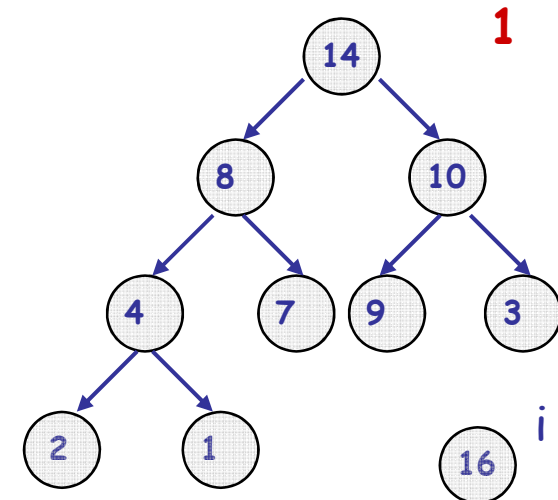
- Como o máximo elemento está armazenado na raiz, pode ser colocado na sua posição correcta.
- Agora, ao descartar o nó n o heap, sabes-se que as sub-árvores da raiz são heaps, mas a nova raiz pode violar a propriedade, logo tem que se invocar o MAX-HEAPIFY.



Após a execução de
Build-Max-Heap



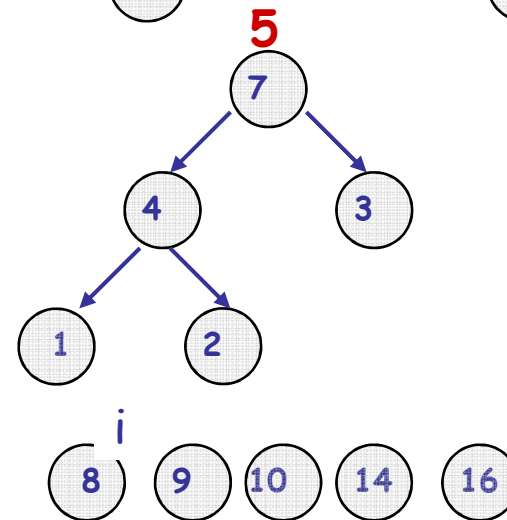
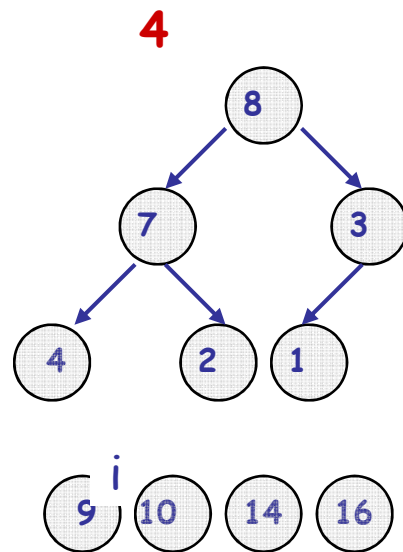
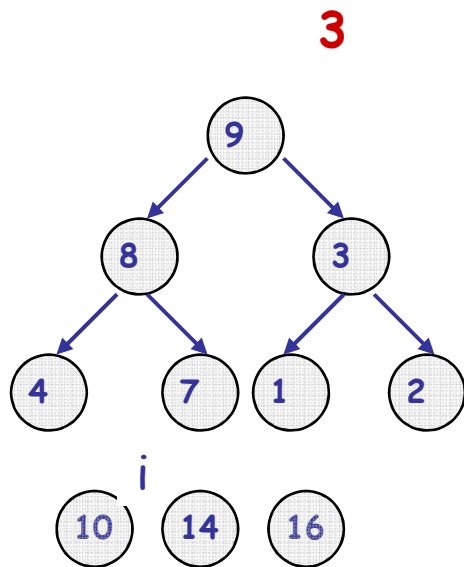
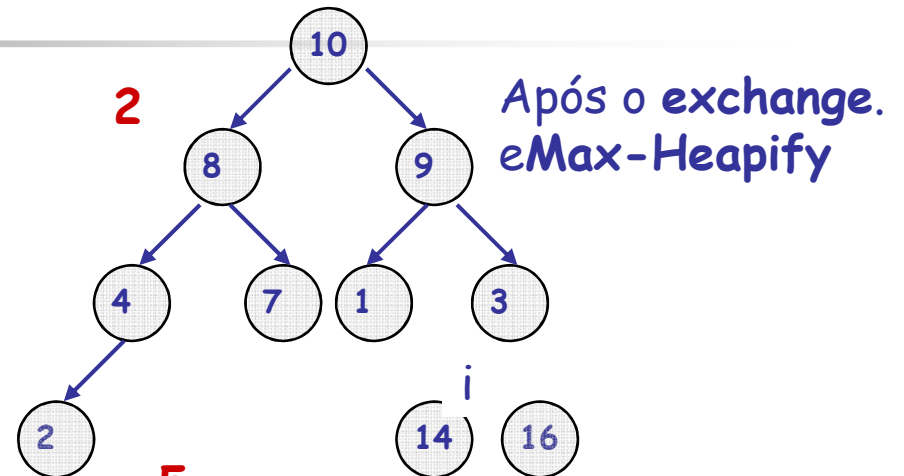
Após o **exchange**.
(deixou de ser heap)



Após o **Max-Heapify**

Heap - Sort

```
static void heapSort(int[] v, int n) {
    buildHeap(v,n);
    for ( int i= n-1; i> 0; --i) {
        exchange(v, 0, i);
        heapify(v, 0, i);
    }
}
```



Heap - Sort

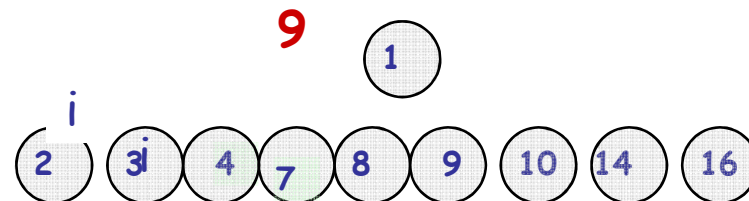
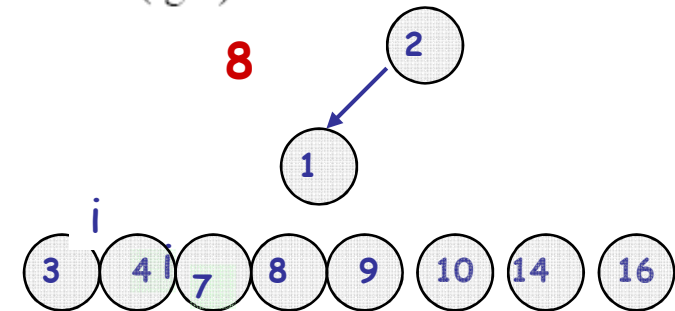
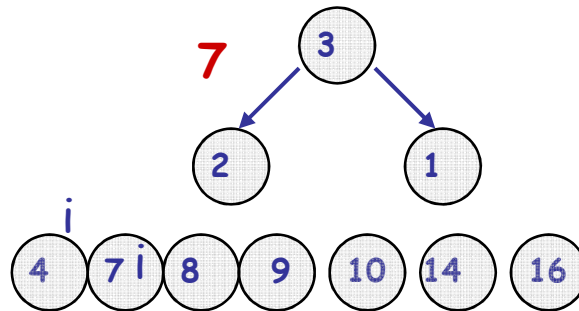
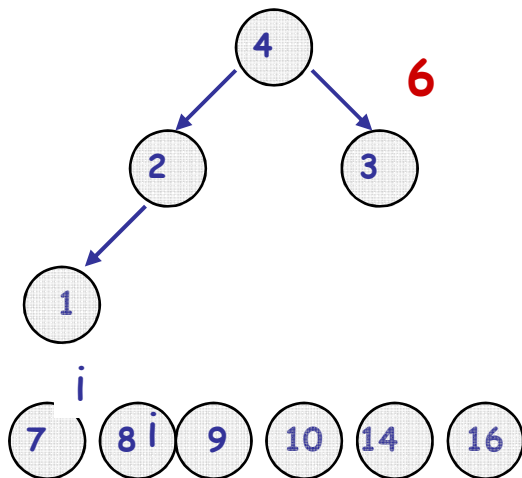
```

HEAPSORT( $A, n$ )
  BUILD-MAX-HEAP( $A, n$ )
  for  $i \leftarrow n$  downto 2
    do exchange  $A[1] \leftrightarrow A[i]$ 
      MAX-HEAPIFY( $A, 1, i - 1$ )
  
```

Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$





Priority Queue

- Uma *Priority Queue* (fila prioritária) é uma estrutura de dados para manter um conjunto S de elementos, cada um com uma *chave* associada. Uma *max-priority queue* suporta as seguintes operações:
 - $\text{INSERT}(S, x)$, insere o elemento x no conjunto S ;
 - $\text{MAXIMUM}(S)$, retorna o elemento de S com a maior chave;
 - $\text{EXTRACT-MAX}(S)$, remove e retorna o elemento de S com a maior chave;
 - $\text{INCREASE-KEY}(S, x, k)$, aumenta o chave do elemento x para o novo valor k , que é assumido ser maior ou igual ao valor actual da chave de x .
- Uma *min-priority queue* suporta as operações INSERT , MINIMUM , EXTRACT-MIN e DECREASE-KEY .

Implementar uma max-priority queue sobre max-heaps

```
HEAP-MAXIMUM(A)  
return A[1]
```

O maior elemento está na 1ª posição

Time: $\Theta(1)$

```
HEAP-EXTRACT-MAX(A, n)  
if n < 1  
    then error "heap underflow"  
max ← A[1]  
A[1] ← A[n]  
MAX-HEAPIFY(A, 1, n - 1) ▷ remakes heap  
return max
```

Time: $O(\lg n)$.

Retira-se o maior elemento;

Ao descartar o nó 1 o heap, sabe-se que as sub-árvores da raiz são heaps, mas a nova raiz pode violar a propriedade, logo tem que se invocar o MAX-HEAPIFY.

```
static int heapExtractMax(int[] v,  
                           int hSize){  
    int max = v[ 0 ];  
    v[ 0 ] = v[hSize-1];  
    heapify(v, 0, hSize-1);  
    return max;  
}
```

v é um max-heap

Cátia Vaz

Implementar uma max-priority queue sobre max-heaps

```
HEAP-INCREASE-KEY( $A, i, key$ )  
if  $key < A[i]$   
    then error "new key is smaller than current key"  
 $A[i] \leftarrow key$   
while  $i > 1$  and  $A[PARENT(i)] < A[i]$   
    do exchange  $A[i] \leftrightarrow A[PARENT(i)]$   
     $i \leftarrow PARENT(i)$ 
```

Time: $O(\lg n)$.

```
static void heapIncreaseKey(int[] v,  
                           int i, int key) {  
    v[i] = key;  
    while(i > 0 && v[i] > v[parent(i)]) {  
        exchange(v, i, parent(i));  
        i = parent(i);  
    }  
}
```

```
MAX-HEAP-INSERT( $A, key, n$ )  
 $A[n + 1] \leftarrow -\infty$   
HEAP-INCREASE-KEY( $A, n + 1, key$ )
```

Time: $O(\lg n)$.

```
static void maxHeapInsert(int[] v,  
                          int hSize, int key) {  
    v[hSize] = key;  
    heapIncreaseKey(v, hSize, key);  
}
```

v é um max-heap

Expando o max-heap;
Altera a chave do novo elemento para o valor correcto e invoca o HEAP-INCREASE-KEY para manter a propriedade do heap.



Heap Sort genérico

```
public static <T> void heapify(T[] v, int p, int hSize, Comparator<T> cmp){
    int l, r, largest;
    l = left(p);
    r = right(p);
    largest=p;
    if(l < hSize && cmp.compare(v[l],v[p])>0) largest=l;
    if ( r < hSize && cmp.compare(v[r],v[largest])>0) largest = r;
    if ( largest == p ) return;
    exchange(v, p, largest);
    heapify(v, largest,hSize,cmp);
}

public static <T> void exchange(T[] v, int i, int j){
    T tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
}
```