UNISINOS

Estruturas Avançadas de Dados I (Heapsort)

Professor Gilberto Irajá Müller

Última atualização 16/10/2017

O que é Heap?

- É uma **"Estrutura de prioridades"** na forma de árvore binária completa ou cheia que representa uma ordem parcial entre os elementos do conjunto;
- Uma árvore completa é aquela em que os nós com menos de dois filhos estão no último ou penúltimo nível;
- Maior nível é preenchido a partir da esquerda para a direita;
- Dois tipos de heaps (propriedades):
  - Máximo
  - Mínimo

Heap Máximo

- Para todo nó diferente da raiz, o pai é maior ou igual aos filhos:
  - $A[i] \geq A[2i + 1]$
  - $A[i] \geq A[2i + 2]$

- O maior elemento de um heap máximo está armazenado na raiz;

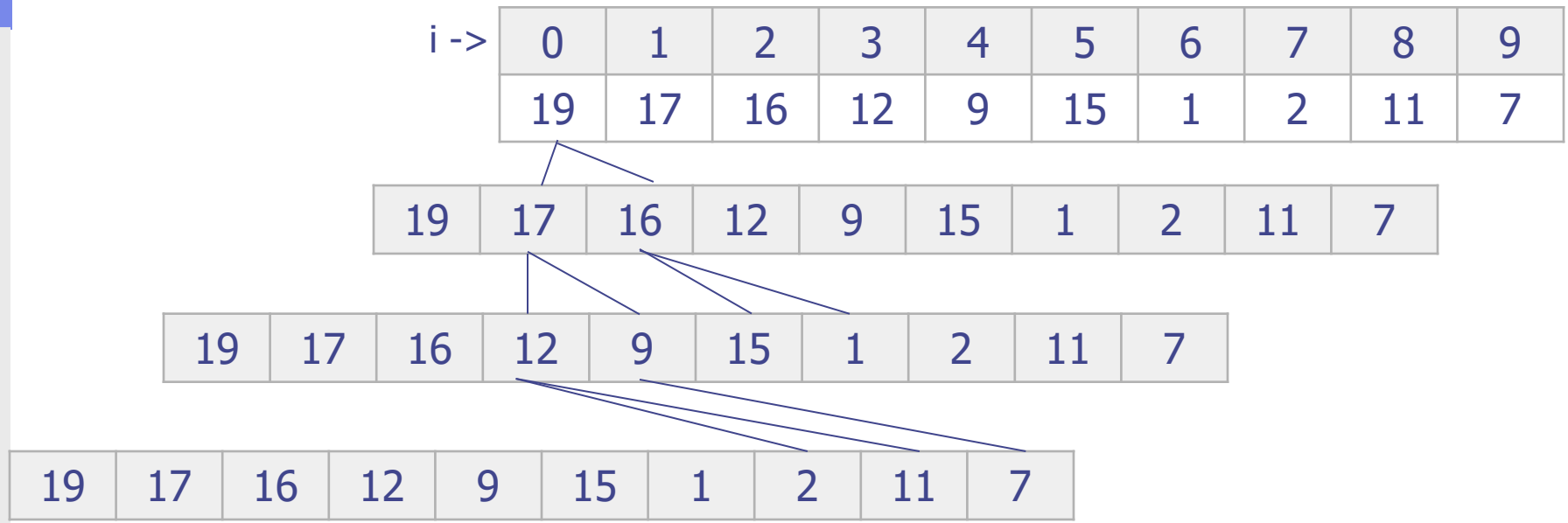
- **Uso do Heap Máximo em algoritmos de classificação como o Heapsort.**

Heap Mínimo

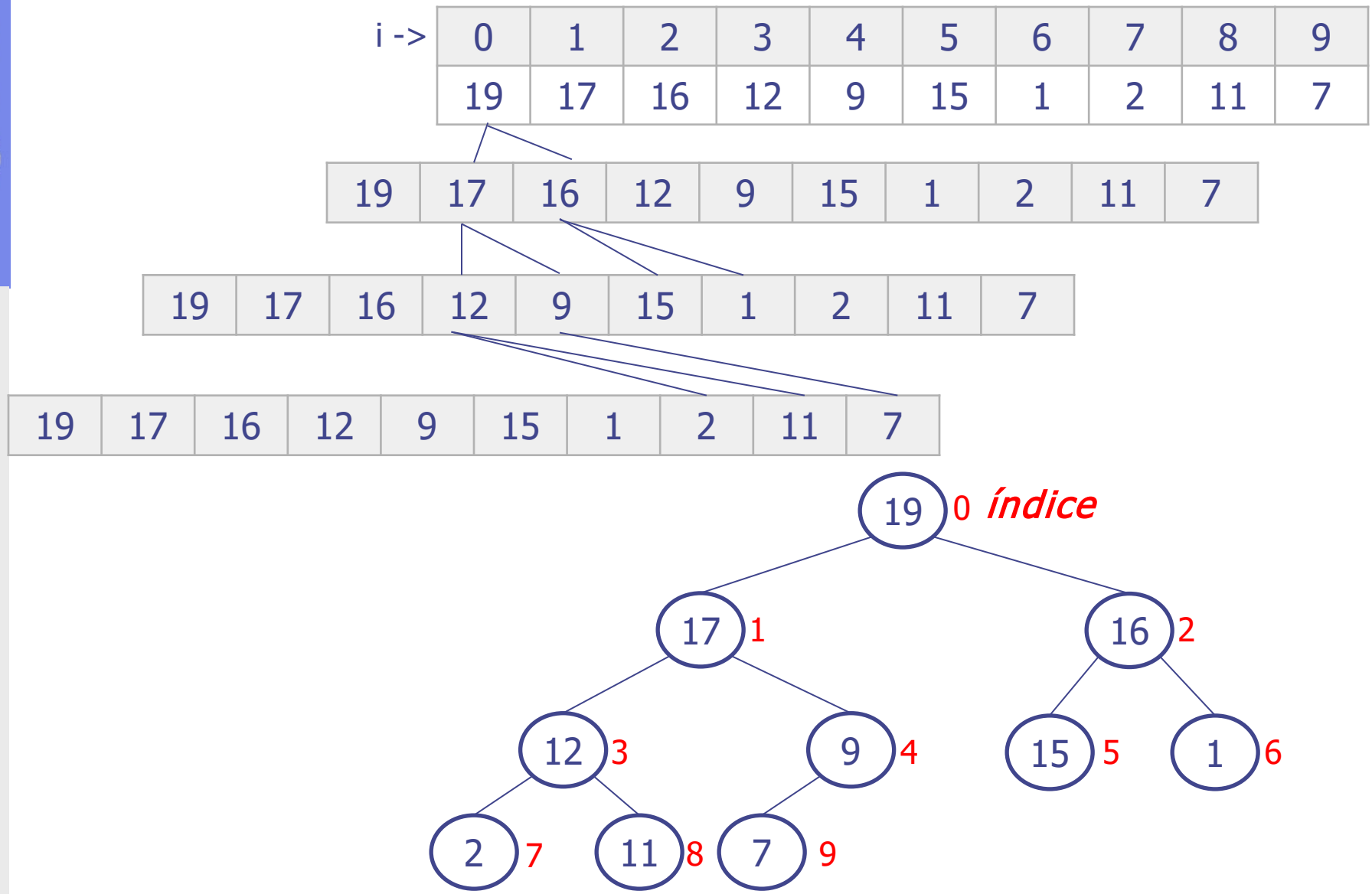
- Para todo nó diferente da raiz, o pai é menor ou igual aos filhos:
  - $A[i] \leq A[2i + 1]$
  - $A[i] \leq A[2i + 2]$
- O menor elemento de um heap mínimo está armazenado na raiz;
- **Uso do heap mínimo em filas de prioridades.**

Representação

- Um Heap pode ser representado por um array unidimensional de modo que a raiz ocupa a posição 0, o pai = $\lfloor (i - 1) / 2 \rfloor$ e os demais elementos:
  - Esquerda de i = 2i + 1;
  - Direita de i = 2i + 2.



Representação (cont.)



Procedimentos sobre Heaps

- **Heapify**
  - Garante a manutenção da propriedade do *Heap.* Complexidade *O(log(n))*.
- **Build-Heap**
  - Produz um *heap* a partir de um vetor não ordenado. Complexidade *O(n)*.
- **Heapsort**
  - Procedimento de ordenação. Complexidade *O(nlog(n))*.

Heapsort

- Algoritmo de ordenação sofisticado;
- Desenvolvido em 1964 por Robert W. Floyd (New York, USA) e J.W.J. Williams;
- Consiste em um método de seleção em árvore binária do tipo ***heap*** de forma ordenada em relação aos valores de suas chaves**;**
- Consiste em duas fases:
  - Fase 1: construção do heap (build-Heap)
  - Fase 2: seleção dos elementos na ordem desejada (Heapsort)

Implementação do Heapsort

```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
    buildMaxHeap(a);
    for (int i = a.length - 1; i > 0; i--) {
        exchange(a, 0, i);
        maxHeapify(a, 0, i);
    }
}
```

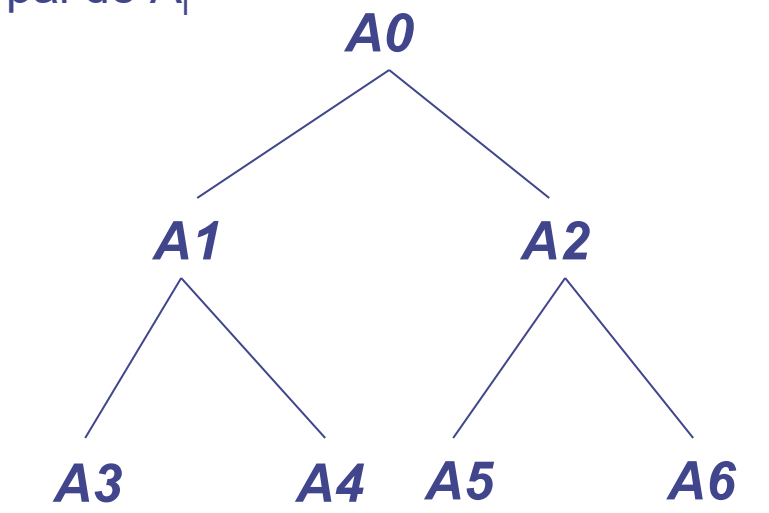
```
private static <T extends Comparable<? super T>> void buildMaxHeap(T[] a) {
    for (int i = a.length / 2 - 1; i >= 0; i--) {
        maxHeapify(a, i, a.length);
    }
}
```

```
private static <T extends Comparable<? super T>> void maxHeapify(T[] a, int i, int n) {
  int max = 2 * i + 1;
  if (max + 1 < n && a[max].compareTo(a[max + 1]) < 0) max++;
  if (max < n && a[max].compareTo(a[i]) > 0) {
     exchange(a, i, max);
     maxHeapify(a, max, n);
  }
}
```

```
private static <T extends Comparable<? super
T>> void exchange(T[] a, int i, int j) {
    T tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

Heapsort – Fase 1

- Fase 1: Construindo o ***heap***
  - consideremos o array $A_{0..6}$ como sendo a estrutura de dados de representação de uma árvore binária com a seguinte interpretação dos índices das chaves:
    - A_0 é a raiz da árvore
    - A_{2i+1} = subárvore à esquerda de A_i
    - A_{2i+2} = subárvore à direita de A_i
    - $A_{(i-1)/2}$ = pai de A_i



Árvore representada pelo array $A_{0..6}$

Heapsort – Fase 1 (cont.)

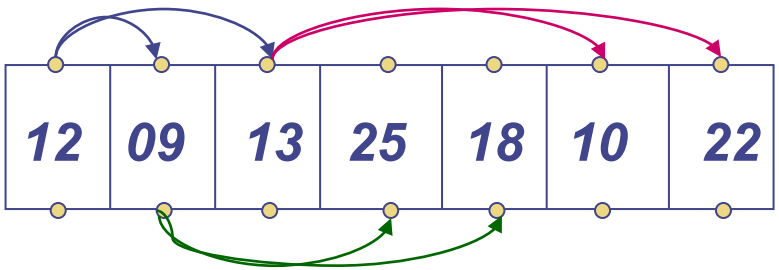
- Fase 1: Construindo o ***heap***
  - a partir da estrutura de dados vista anteriormente, o passo seguinte consiste em trocar as chaves de posição no array, de forma que a árvore representada passe a ser um ***heap***, ou seja, cada raiz da árvore satisfaça as seguintes condições:

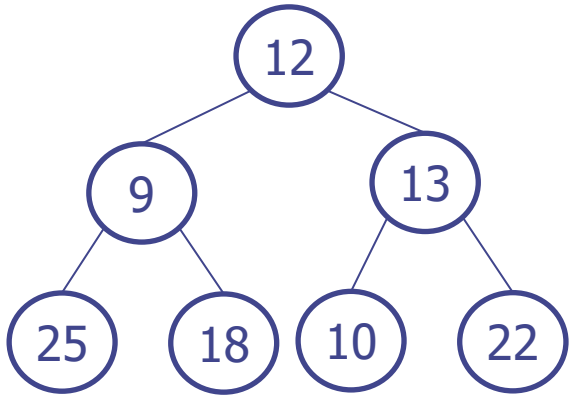
$$A_i \geq A_{2i+1}$$
$$A_i \geq A_{2i+2}$$

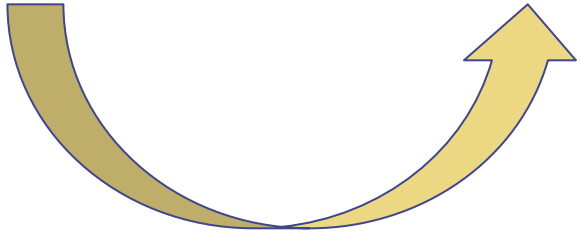
  - os testes das chaves iniciam pela subárvore à direita (ou à esquerda) analisando recursivamente se a condição acima é satisfeita. Caso não seja, então ocorre o processo de *heapify*.

Heapsort – Fase 1 (cont.)

- Exemplo de construção do ***heap***:
  - Seja o seguinte array de chaves:







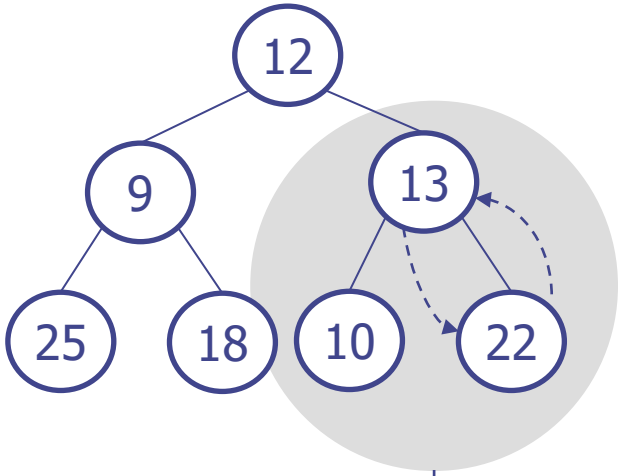
Heapsort – Fase 1 (cont.)

```
private static <T extends Comparable<? super T>> void buildMaxHeap(T[] a) {
   for (int i = a.length / 2 - 1; i >= 0; i--) {
      maxHeapify(a, i, a.length);
   }
}
```

Transformação da subárvore de raiz 13 (i=2) para 22 (i=6) em heap

12	9	13	25	18	10	22

12	9	22	25	18	10	13




```
private static <T extends Comparable<? super T>> void maxHeapify(T[] a, int i, int n) {
   int max = 2 * i + 1;
   if (max + 1 < n && a[max].compareTo(a[max + 1]) < 0) max++;
   if (max < n && a[max].compareTo(a[i]) > 0) {
      exchange(a, i, max);
      maxHeapify(a, max, n);
   } }
```

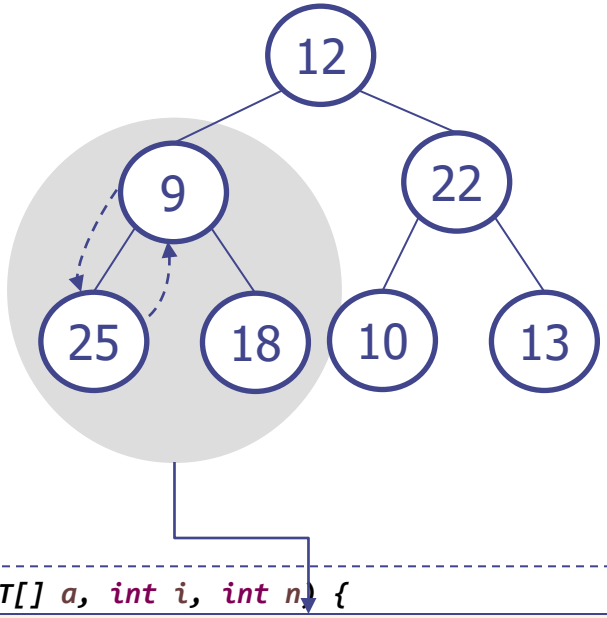
Heapsort – Fase 1 (cont.)

```
private static <T extends Comparable<T>> void buildMaxHeap(T[] a) {
    for (int i = a.length / 2 - 1; i >= 0; i--) {
        maxHeapify(a, i, a.length);
    }
}
```

Transformação da subárvore de raiz 9 (i=1) para 25 (i=3) em heap

12	9	22	25	18	10	13

12	25	22	9	18	10	13




```
private static <T extends Comparable<? super T>> void maxHeapify(T[] a, int i, int n) {
    int max = 2 * i + 1;
    if (max + 1 < n && a[max].compareTo(a[max + 1]) < 0) max++;
    if (max < n && a[max].compareTo(a[i]) > 0) {
        exchange(a, i, max);
        maxHeapify(a, max, n);
    } }
```

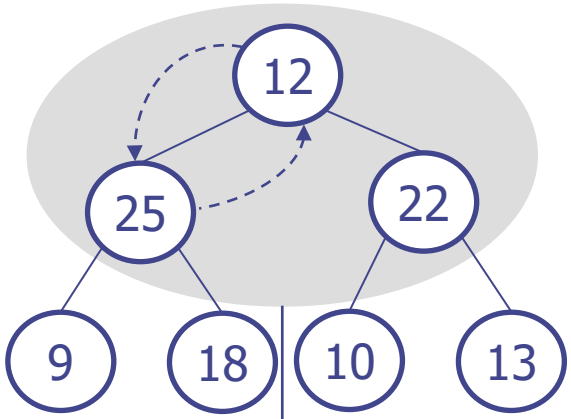
Heapsort – Fase 1 (cont.)

```
private static <T extends Comparable<T>> void buildMaxHeap(T[] a) {
    for (int i = a.length / 2 - 1; i >= 0; i--) {
        maxHeapify(a, i, a.length);
    }
}
```

Transformação da árvore de raiz 12 (i=0) para 25 (i=1) em heap

12	25	22	9	18	10	13

25	12	22	9	18	10	13




```
private static <T extends Comparable<? super T>> void maxHeapify(T[] a, int i, int n) {
    int max = 2 * i + 1;
    if (max + 1 < n && a[max].compareTo(a[max + 1]) < 0) max++;
    if (max < n && a[max].compareTo(a[i]) > 0) {
        exchange(a, i, max);
        maxHeapify(a, max, n);
    } }
```

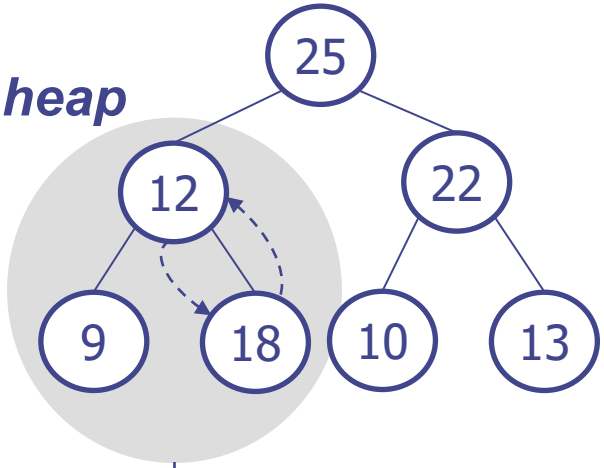
Heapsort – Fase 1 (cont.)

```
private static <T extends Comparable<? super T>> void buildMaxHeap(T[] a) {
   for (int i = a.length / 2 - 1; i >= 0; i--) {
      maxHeapify(a, i, a.length);
   }
}
```

*Transformação da árvore de raiz 12 (**cont. recursão**) para 18 (i=4) em heap*

25	12	22	9	18	10	13

25	18	22	9	12	10	13

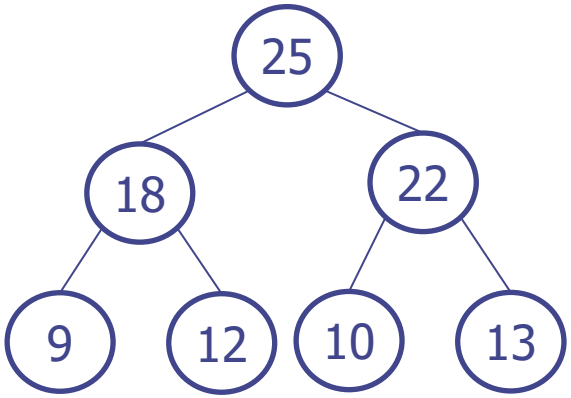



```
private static <T extends Comparable<? super T>> void maxHeapify(T[] a, int i, int n) {
   int max = 2 * i + 1;
   if (max + 1 < n && a[max].compareTo(a[max + 1]) < 0) max++;
   if (max < n && a[max].compareTo(a[i]) > 0) {
      exchange(a, i, max);
      maxHeapify(a, max, n);
   } }
```

Heapsort – Fase 1 (cont.)

```
private static <T extends Comparable<? super T>> void buildMaxHeap(T[] a) {
   for (int i = a.length / 2 - 1; i >= 0; i--) {
      maxHeapify(a, i, a.length);
   }
}
```

25	18	22	9	12	10	13



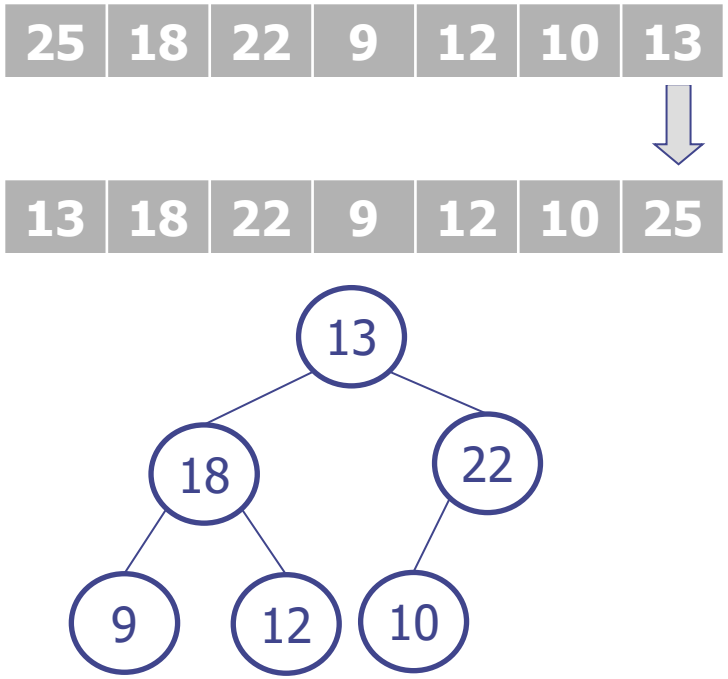
É um heap!

```
private static <T extends Comparable<? super T>> void maxHeapify(T[] a, int i, int n) {
  int max = 2 * i + 1;
  if (max + 1 < n && a[max].compareTo(a[max + 1]) < 0) max++;
  if (max < n && a[max].compareTo(a[i]) > 0) {
     exchange(a, i, max);
     maxHeapify(a, max, n);
  }
}
```

Heapsort – Fase 2

- Fase 2: seleção dos elementos na ordem desejada
  - se a chave que está na raiz é a maior de todas, então sua posição definitiva, na ordem crescente, é na última posição do array;
  - então, esta maior chave é colocada na última posição do array por troca com a chave que está ocupando aquela posição.

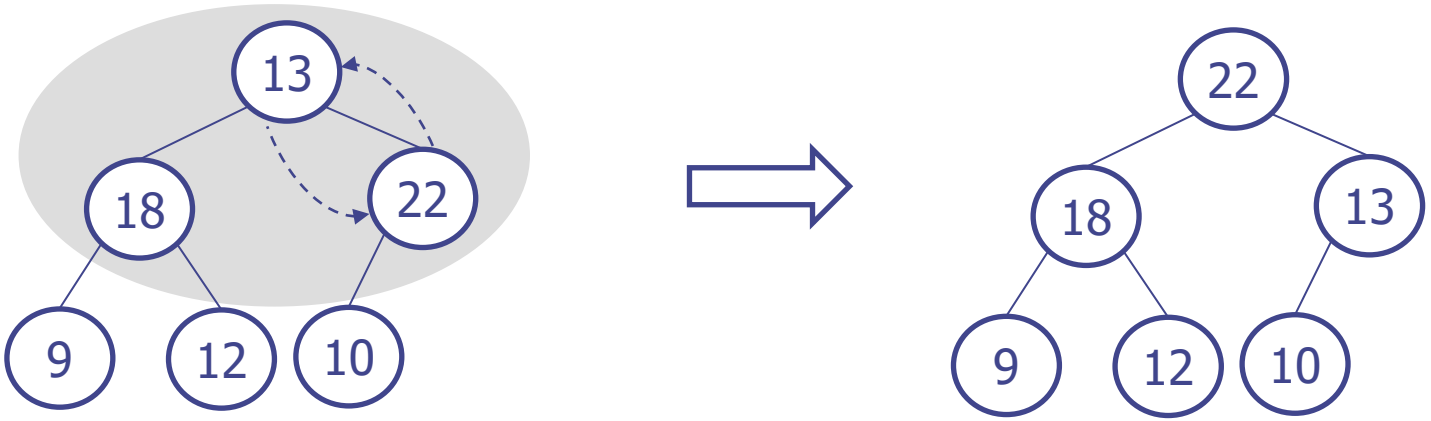
Seleção da primeira maior chave:




```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
    buildMaxHeap(a); // FASE 1
    for (int i = a.length - 1; i > 0; i--) { // FASE 2
        exchange(a, 0, i);
        maxHeapify(a, 0, i);
    }
}
```

Heapsort – Fase 2 (cont.)

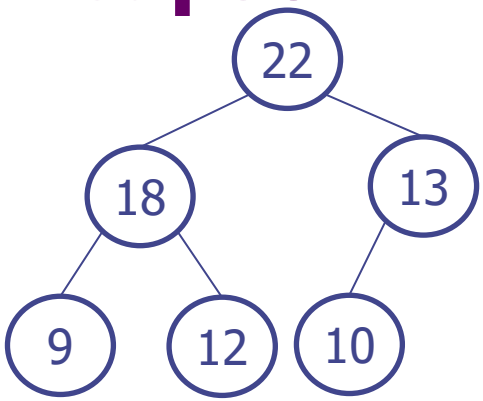
Reajando



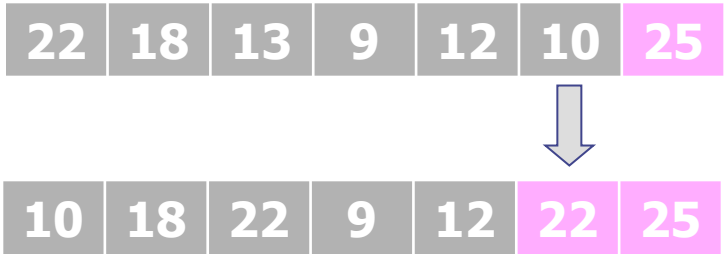
22	18	13	9	12	10	25

```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
    buildMaxHeap(a); // FASE 1
    for (int i = a.length - 1; i > 0; i--) { // FASE 2
        exchange(a, 0, i);
        maxHeapify(a, 0, i);
    }
}
```

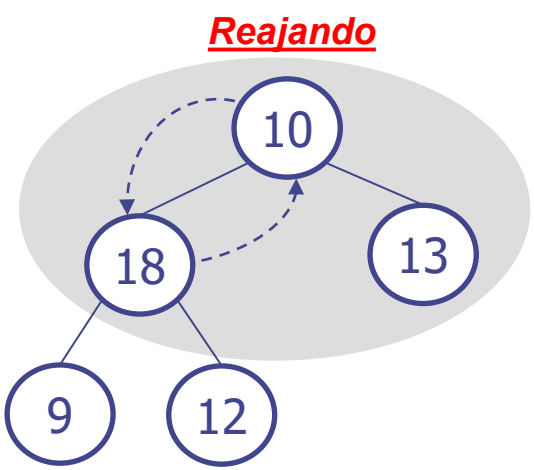
Heapsort – Fase 2 (cont.)

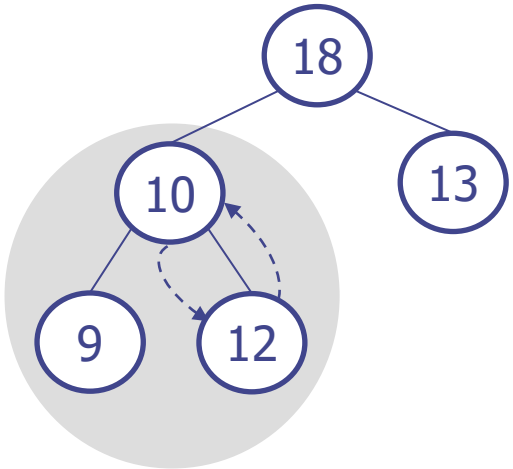


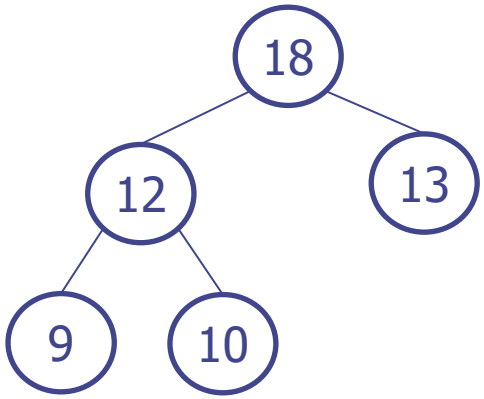
Seleção da segunda maior chave:



Reajando

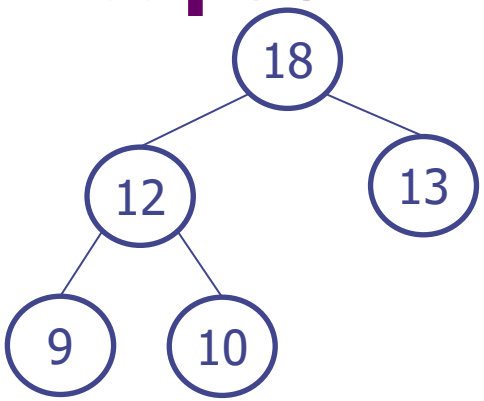




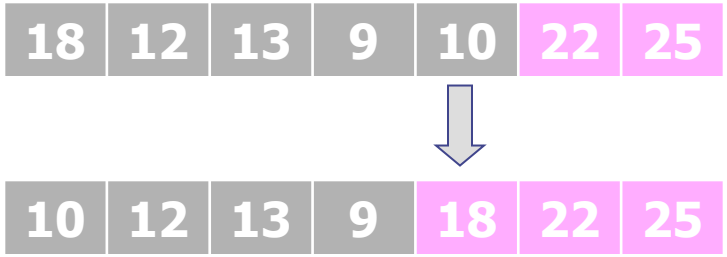



```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
   buildMaxHeap(a); // FASE 1
   for (int i = a.length - 1; i > 0; i--) { // FASE 2
      exchange(a, 0, i);
      maxHeapify(a, 0, i);
   } }
```

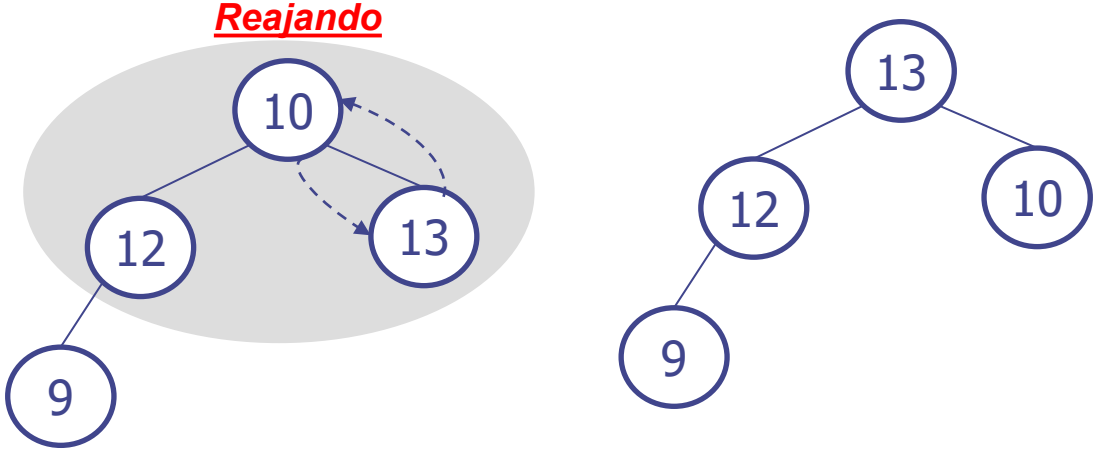
Heapsort – Fase 2 (cont.)



Seleção da terceira maior chave:

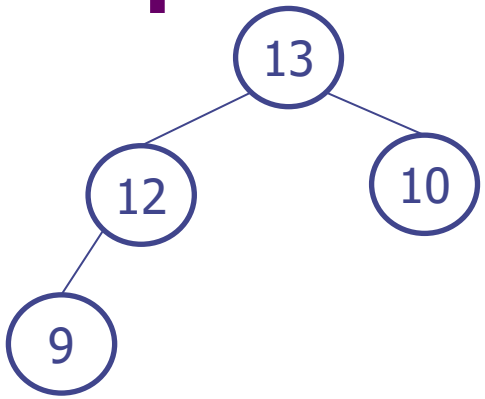


Reajando

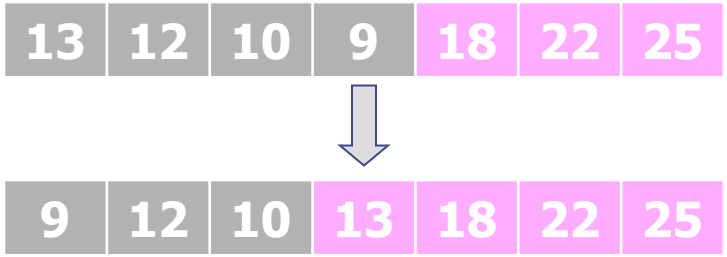



```
public static <T extends Comparable<T>> void heapSort(T[] a) {
   buildMaxHeap(a); // FASE 1
   for (int i = a.length - 1; i > 0; i--) { // FASE 2
      exchange(a, 0, i);
      maxHeapify(a, 0, i);
   } }
```

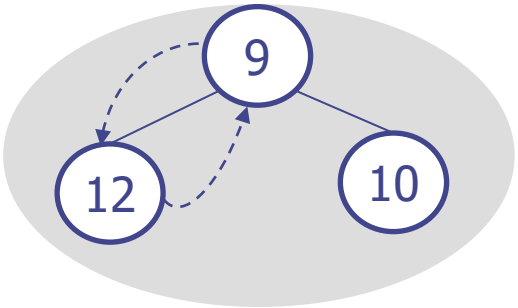
Heapsort – Fase 2 (cont.)

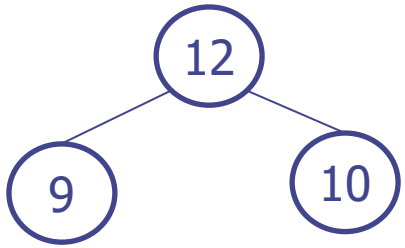


Seleção da quarta maior chave:



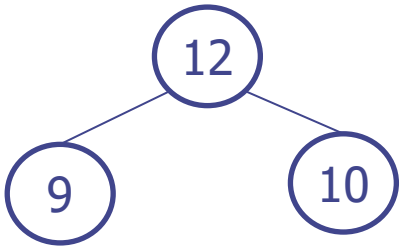
Reajando



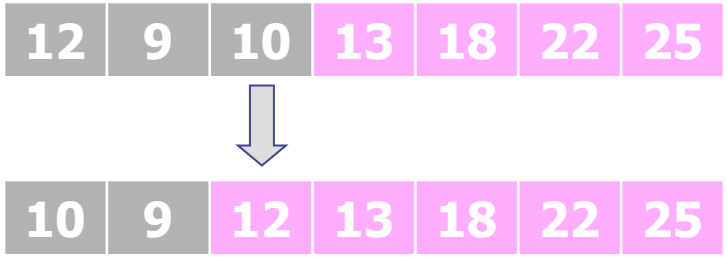



```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
   buildMaxHeap(a); // FASE 1
   for (int i = a.length - 1; i > 0; i--) { // FASE 2
      exchange(a, 0, i);
      maxHeapify(a, 0, i);
   } }
```

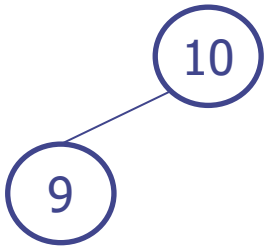
Heapsort – Fase 2 (cont.)



Seleção da quinta maior chave:

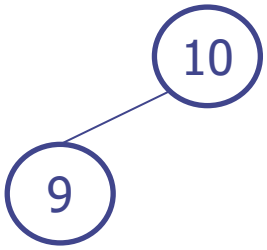


Não tem reajando

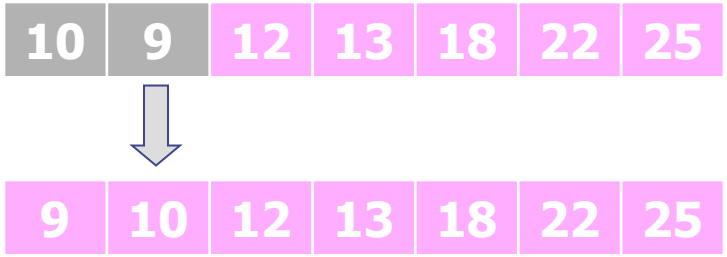



```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
   buildMaxHeap(a); // FASE 1
   for (int i = a.length - 1; i > 0; i--) { // FASE 2
      exchange(a, 0, i);
      maxHeapify(a, 0, i);
   } }
```

Heapsort – Fase 2 (cont.)



Seleção da sexta maior chave:



9	10	12	13	18	22	25

Array Ordenado!

```
public static <T extends Comparable<? super T>> void heapSort(T[] a) {
   buildMaxHeap(a); // FASE 1
   for (int i = a.length - 1; i > 0; i--) { // FASE 2
      exchange(a, 0, i);
      maxHeapify(a, 0, i);
   } }
```

Complexidade

- Tanto no melhor como no pior caso, o desempenho do Heapsort é igual a **O(nlog(n))**, pois:
  - A construção do heap O(n);
  - A troca é O(1);
  - O Heapify é O(log(n)).

- É um algoritmo in-place (uso constante de memória O(1) – não usa estrutura adicional);

- Além disso, é um algoritmo instável, pois o processo de ordenação ocorre a partir da troca de itens criados basicamente utilizando a chave e não considera a sua posição origem.

Complexidade (cont.)

Método	Caso médio	Melhor caso	Pior caso	Complexidade de Espaço	Estável	Interno	Recursivo	Comparação
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	In-place = O(1)	Sim	Sim	Não	Sim
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	In-place = O(1)	Sim	Sim	Não	Sim
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	In-place = O(1)	Sim	Sim	Não	Sim
Shell Sort	$O(n^{7/6})$ – depende do gap	O(nlog(n))	O(nlog(n)) a $O(n^{3/2})$	In-place = O(1)	Não	Sim	Não	Sim
Heap Sort	**O(nlog(n))**	**O(nlog(n))**	**O(nlog(n))**	**In-place = O(1)**	**Não**	**Sim**	**Sim**	**Sim**

Exercícios Teóricos

- **Exercício 1.** Considerando o seguinte array:

11	1	5	7	6	12	17	8

a) Construa o heap (passo-a-passo)
b) Aplique o heapsort (passo-a-passo)

Bibliografia

- Silberchatz, A; Korth, H. F., Sudarshan, S. Sistema de Banco de Dados. 3ª. Edição, Makron Books, 1999.
- Lafore, Robert. Estruturas de Dados & Algoritmos em Java. Editora Ciência Moderna, 2004.
- Lâminas do Prof. Alexandre Parra Carneiro da Silva. Métodos de Classificação por Seleção: HeapSort.