HeapSort





Heapsort

- O Heapsort, também chamado de **Classificação por Monte** ou **Ordenação com Árvore-Heap** é um método de ordenação que utiliza uma árvore-heap como estrutura intermediária para efetuar a ordenação.
- Heapsort **sempre** é **O(n log n)**
  - Quicksort **geralmente** é **O(n log n)** mas no pior caso fica **$O(n^2)$**
  - Quicksort geralmente é um pouco melhor, mas Heapsort é mais confiável em situações críticas

Heapsort

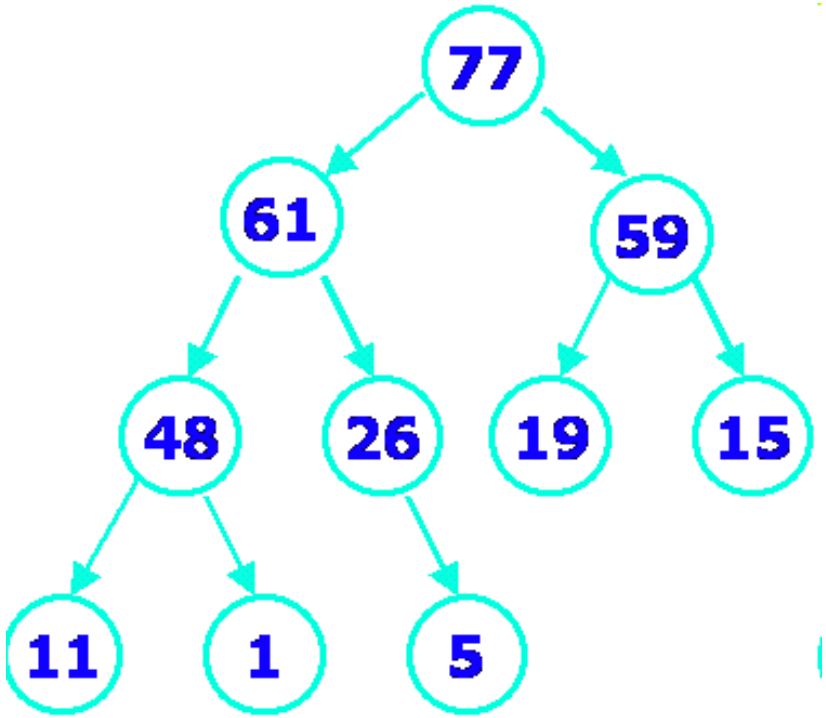
- Desvantagens:
  - construir a árvore-heap pode consumir **muita memória**.
- Em suma:
  - para dados bem bagunçados o Quicksort é mais vantajoso por ser **mais econômico**;
  - para dados imprevisíveis, pode ser mais vantajoso por ser previsível em termos de **tempo de execução**.

O que é “heap”?

- Definições de heap:
  - (**conhecida**) Uma grande quantiodade de memória de onde o programador poded alocar ou liberar blocos para uso dentro de um programa.
  - (**nova**) Uma árvore binária balanceada, justiifcada à esquerda, onde nenhum nodo possui um valor maior que seu pai.
- Não possuem nada em comum:
  - Heapsort usa a segunda

A Árvore-Heap

- Uma **árvore-heap** é uma árvore binária justificada à esquerda, organizada por níveis, onde o nodo pai de uma subárvore sempre é maior que todos os seus descendentes.



Método em Heapsort

É um método em três estágios:

- primeiramente criamos uma visão sobre o arquivo como árvore binária por níveis;
- Iterativamente:
  - reestruturamos o arquivo para que se transforme em um **monte**;
  - Geramos a seqüência de saída, repetidamente:
    - retirando a raiz do monte e
    - **reestruturando** o monte para que retorne a satisfazer a condição-heap.

Origem da Metáfora:

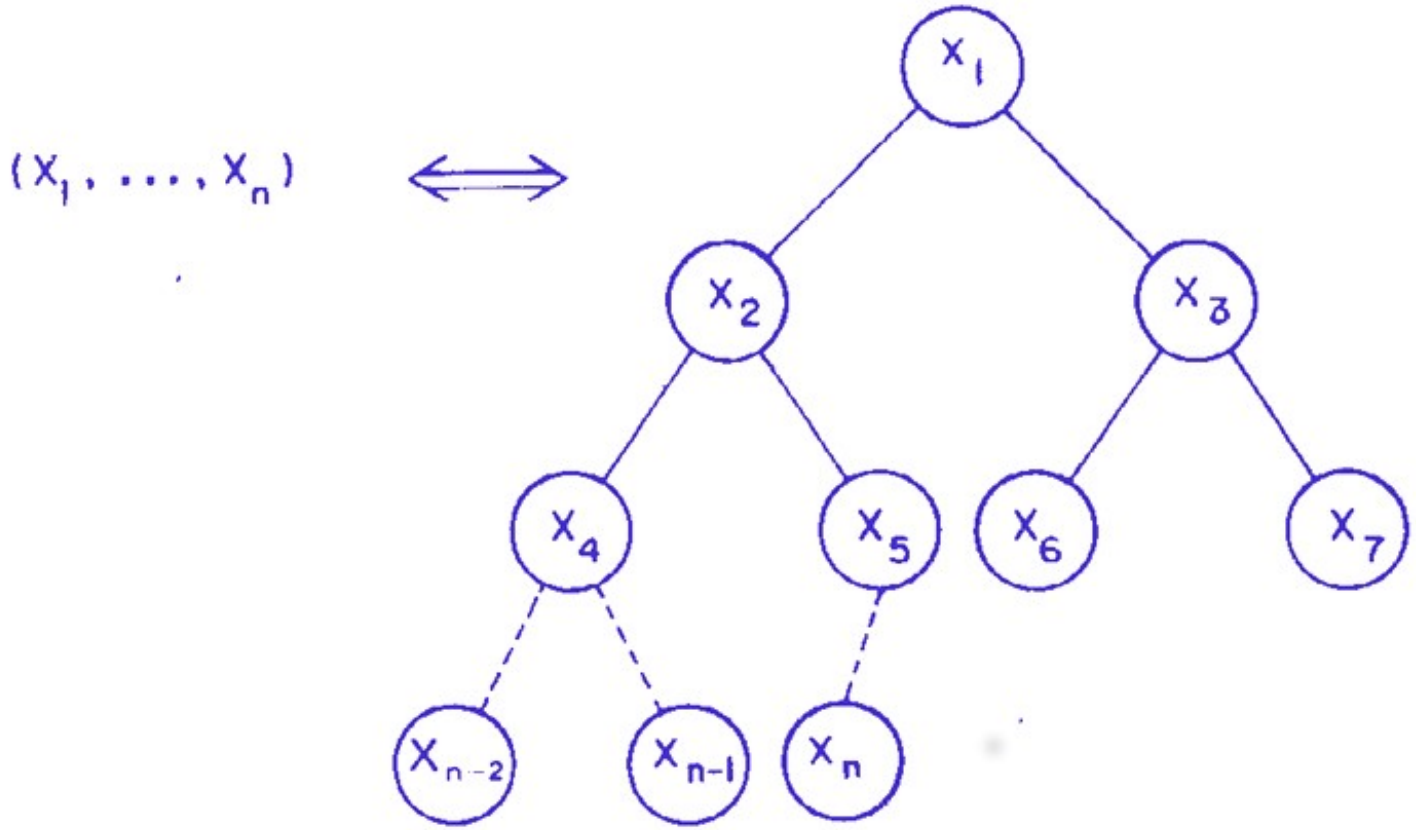
- Efeito Nóz-do-Pará (*Brazil Nut Effect*)

  Fenômeno da flutuação reversa em material granular



- Veja alguns vídeos...

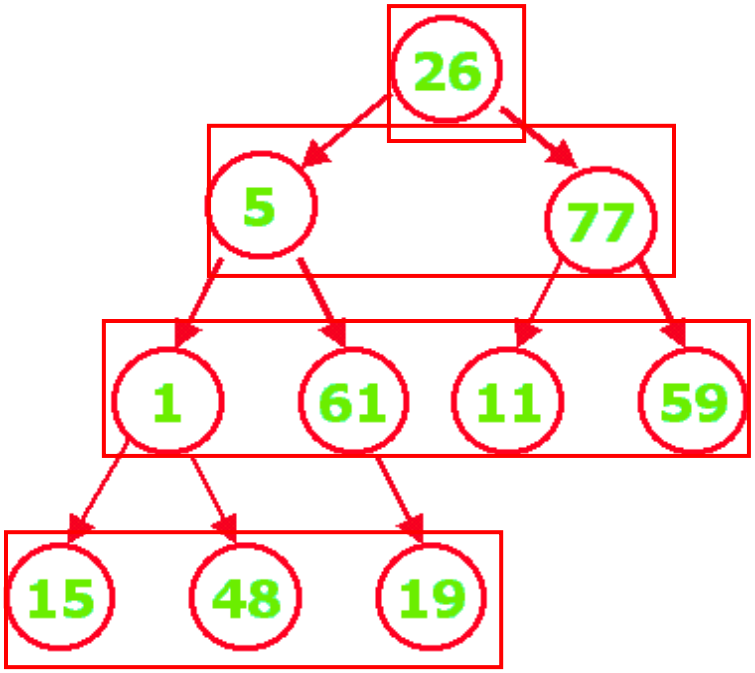
Visão do arquivo $x_1, ..., x_n$ por níveis



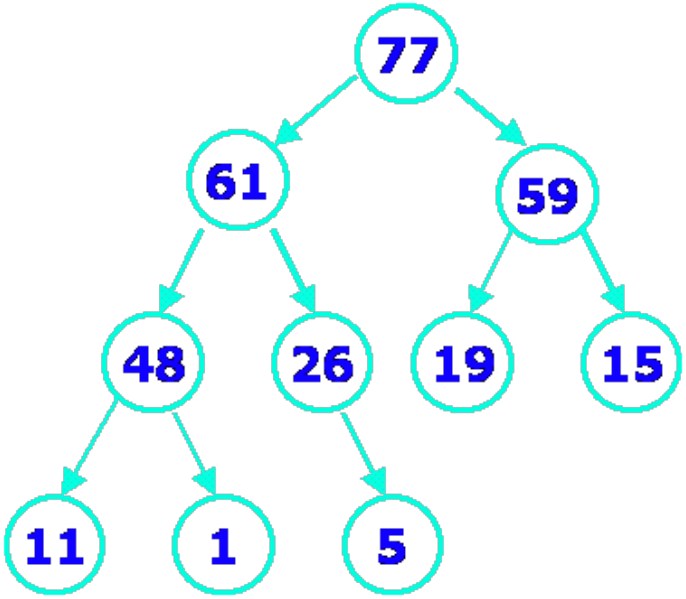
Exemplo

(26 5 77 1 61 11 59 15 48 19)

Arquivo original por níveis

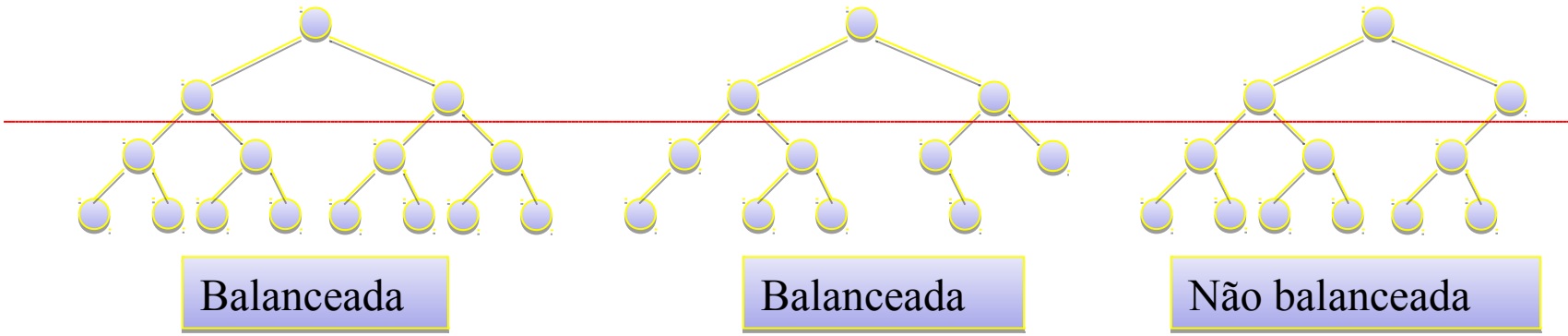


Árvore-Heap resultante



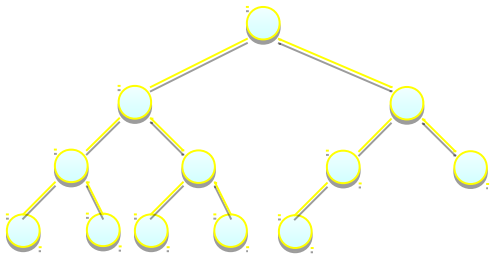
Árvores binárias balanceadas

- Lembre:
  - A **profundidade de um nodo** é sua distância da raiz
  - A **profundidade de uma árvore** é a profundidade de seu nodo mais profundo
- Uma árvore binária de profundidade n está balanceada se todos os nodos da profundidade **0** até **n-2** possuírem dois filhos

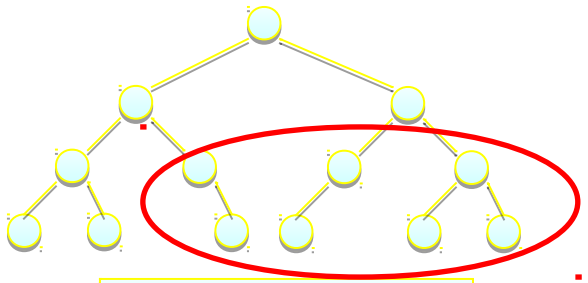


Árvores binárias justificadas à esquerda

- Uma árvore binária balanceada de profundidade n é justificada à esquerda se:
  - possui 2^n nodos de profundidade **n** (***cheia***), ou
  - possui 2^k nodos de profundidade **k**, para todo **k < n**, *e* todas as folhas no nível **n** estão mais à esquerda possível



Justificada à esquerda



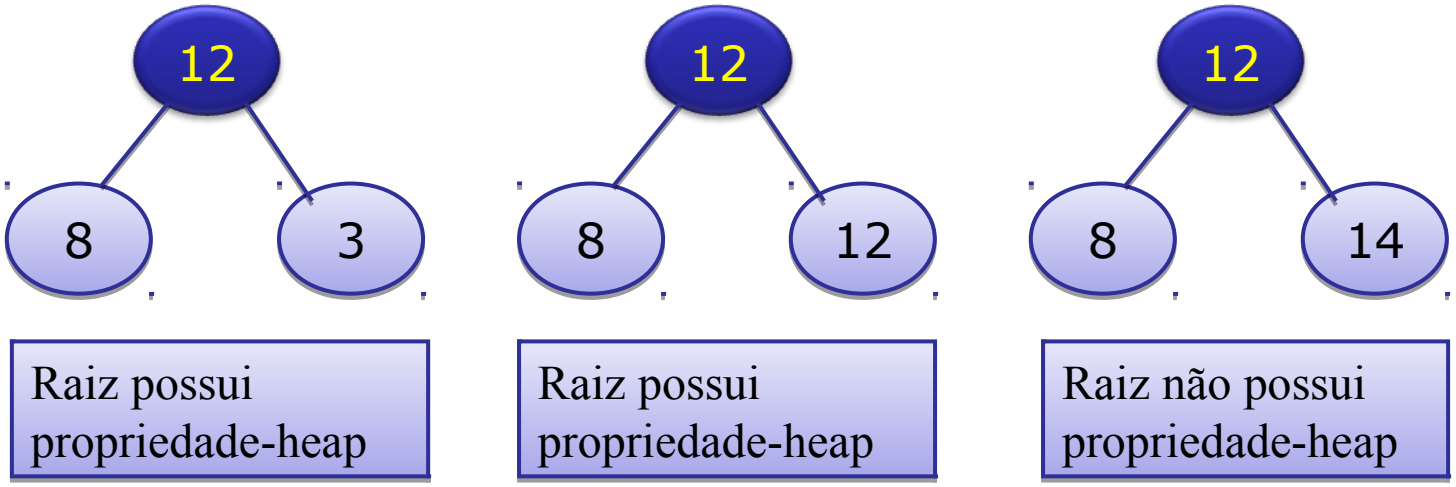
Não-justificada à esquerda

Passos para Dominar HeapSort

- Aprender a transformar uma árvore binária em um **heap**
- Aprender a transformar uma árvore binária **de volta** em um heap após ela ter sofrido **modificações**
- Aprender como estas idéias servem para **ordenar** um vetor

A propriedade-heap

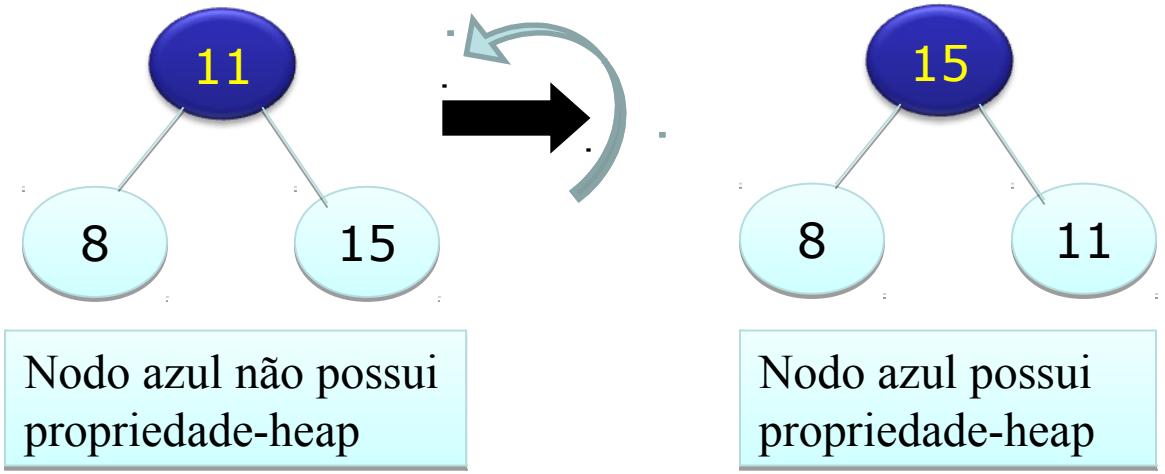
- Um nodo possui a propriedade-heap se nenhum de seus descendentes é maior que ele.



- Todas as folhas possuem propriedade-heap
- Uma árvore binária é um heap se todos os seus nodos possuem a propriedade-heap

Peneirar para cima (siftUp)

- Dado um nodo não-heap, ele pode adquirir a propriedade-heap trocando seu valor com seu maior filho.



- Isto é chamado sifting up
- Observe que o filho pode ter perdido a propriedade-heap

Construindo um monte I

- Uma árvore unitária é um heap
- Construímos um heap adicionando nodos um a um:
  - Adicione o nodo imediatamente à direita do nodo mais à direita no nível mais profundo.
  - Se o nível mais profundo estiver completo, inicie um novo nível
- Exemplos:

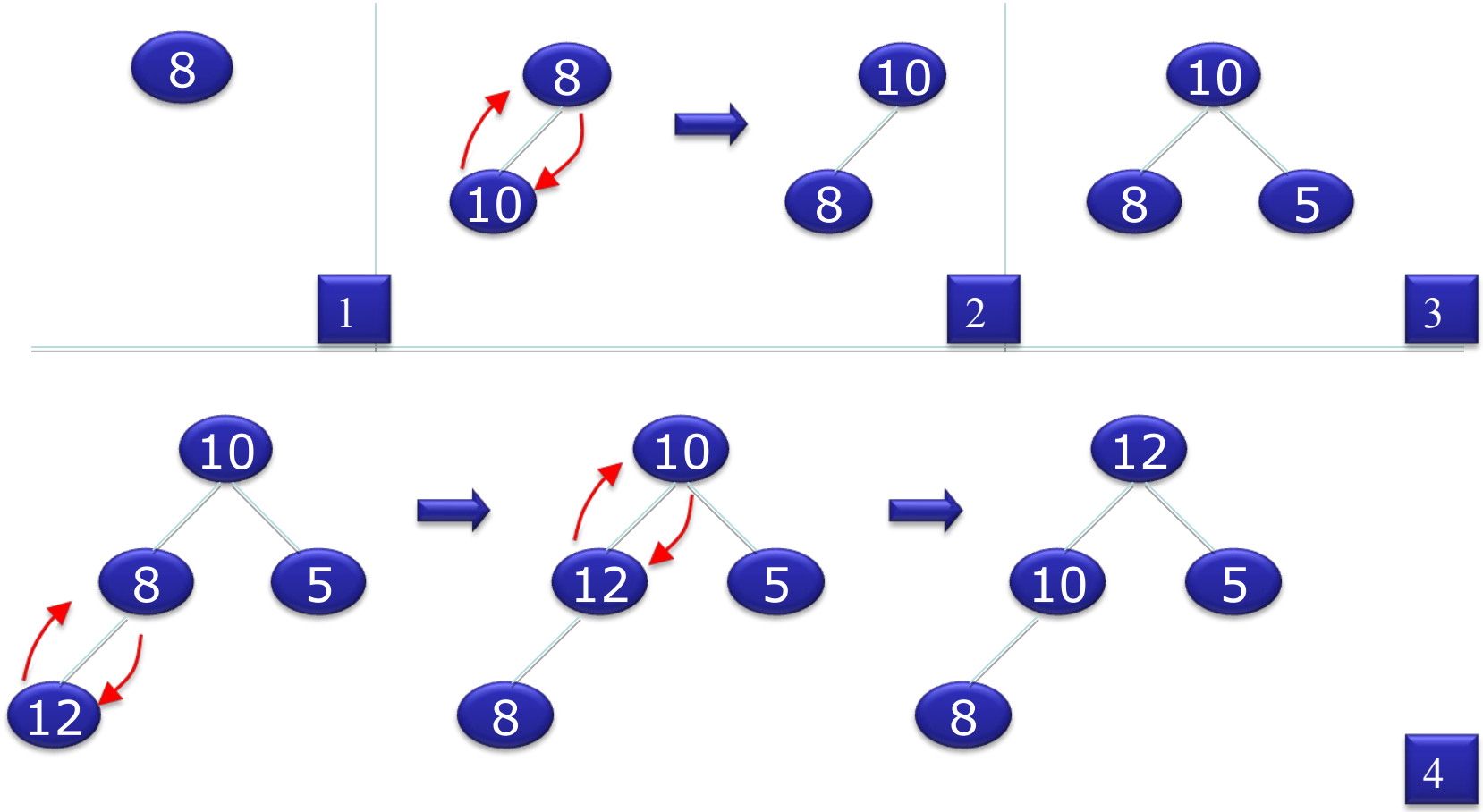




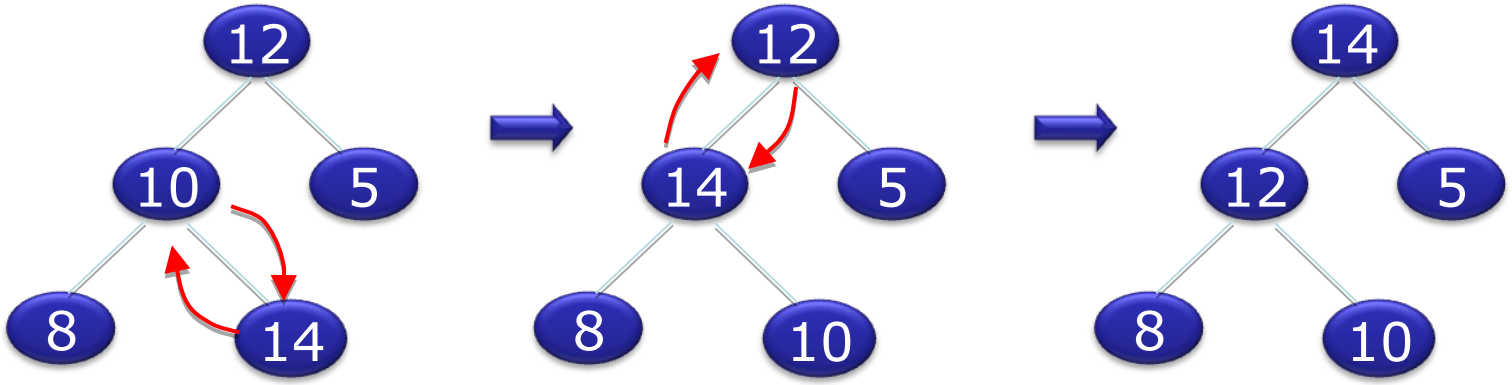
Construindo um monte II

- Toda vez que adicionamos um nodo, poderemos estar destruindo a propriedade-heap de seu pai.
  - Para arrumar isto, peneiramos para cima.
- Como o valor do nodo pai se altera, isto pode alterar a propriedade-heap em *seu* pai.
- Repetimos o sifting up, subindo na árvore até que
  - ou
    - Atinjamos nodos cujos valores não tenham de ser trocados
      - o pai ainda é maior que o filho
  - ou
    - Atinjamos a raiz

Construindo um monte III



Outros filhos não são afetados



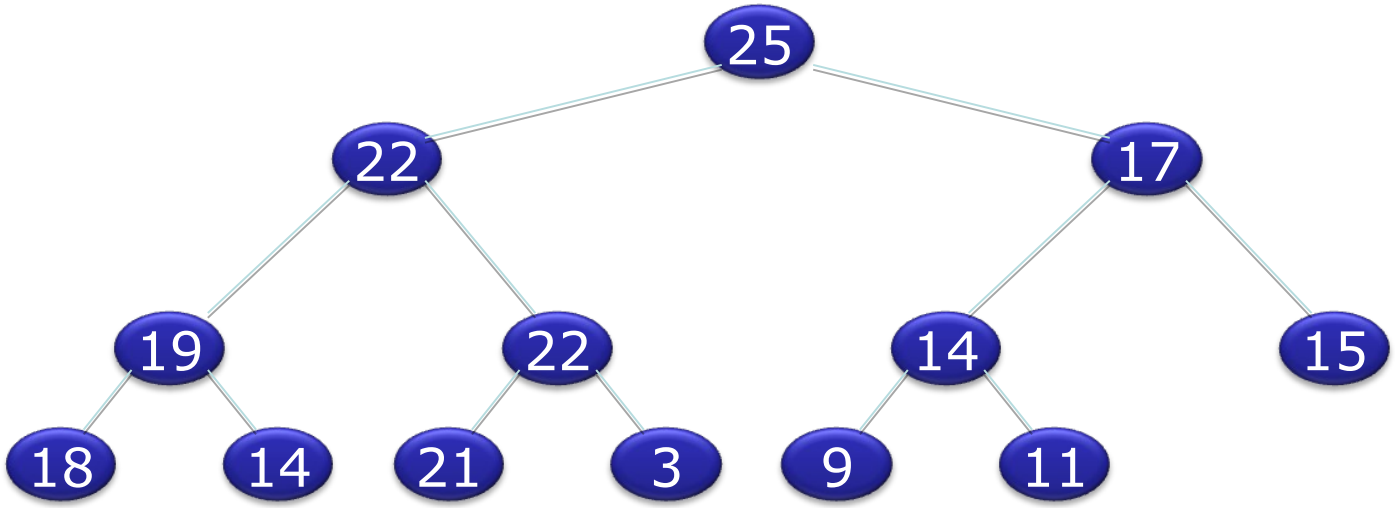
O nodo contendo 8 não é afetado porque seu pai fica maior, não menor

O nodo contendo 5 não é afetado porque seu pai fica maior, não menor

O nodo contendo 8 continua não sendo afetado porque seu pai, embora tenha diminuído, continua sendo mairo do que era antes

Um heap-exemplo

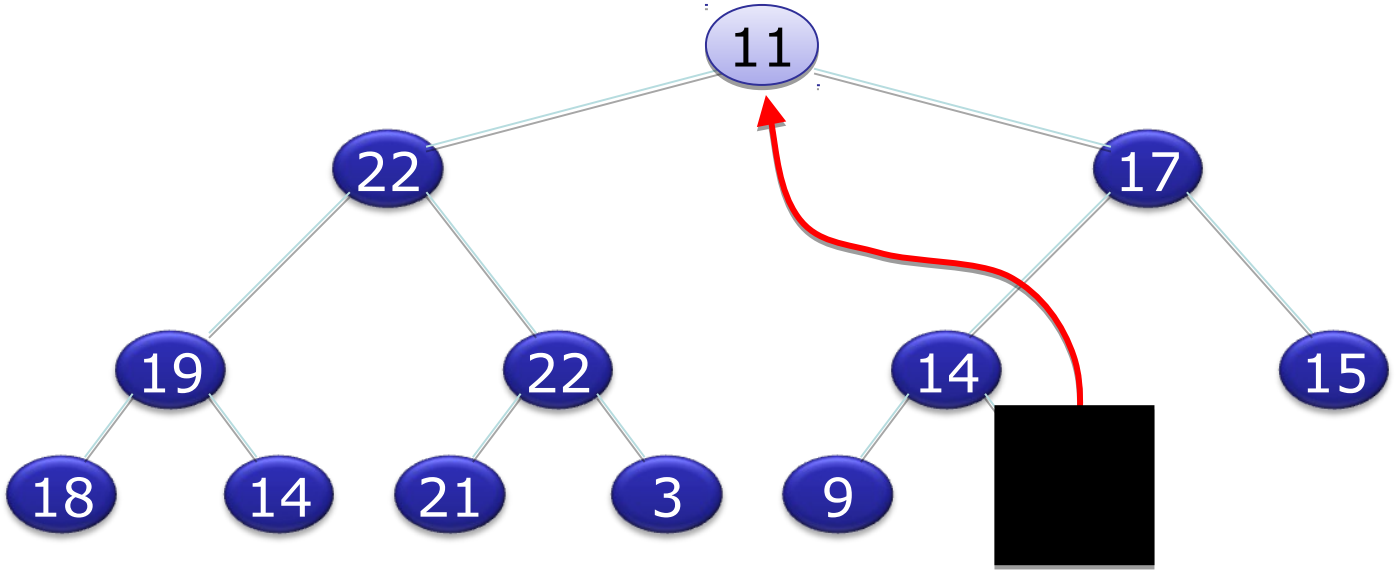
- Aqui uma árvore binária-exemplo depois que ela foi amontoada (*heapificada*…)



- Observe que amontoar não significa ordenar
- Amontoar também não altera a forma da árvore. Se ela começou justificada à esquerda, continua assim.

Removendo a raiz

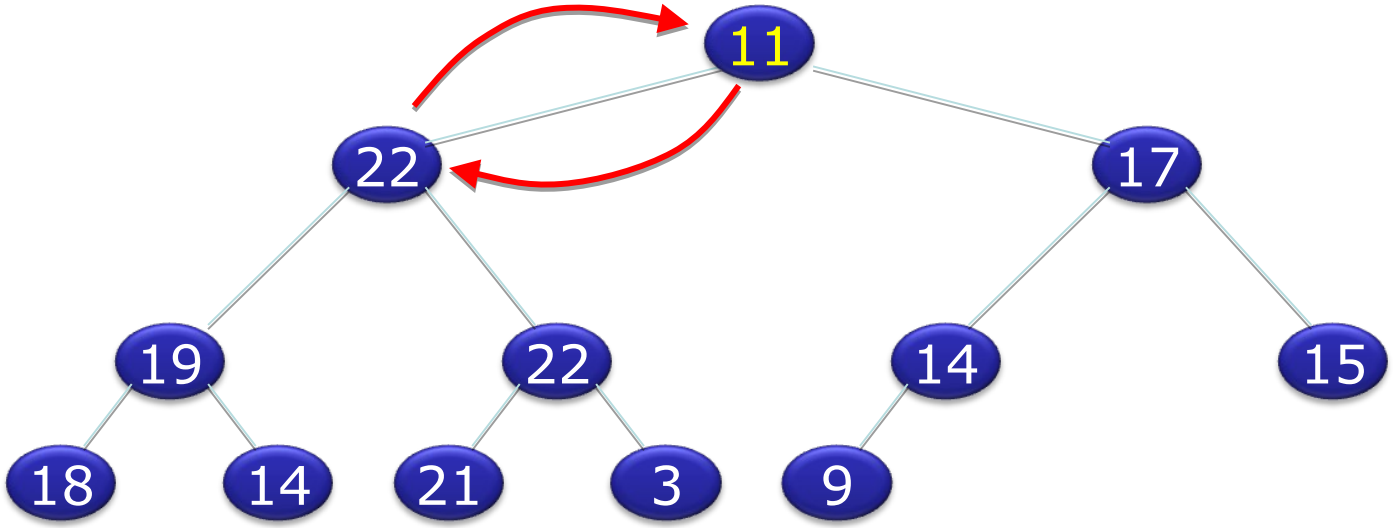
- Observe que o maior valor agora está na raiz
- Suponha que descartemos a raiz:



- Como consertamos a árvore para voltar a estar balanceada e justificada à esquerda?
- Solução: Retire o elemento mais à direita do nível mais inferior e use como raiz.

O método reamontoar I

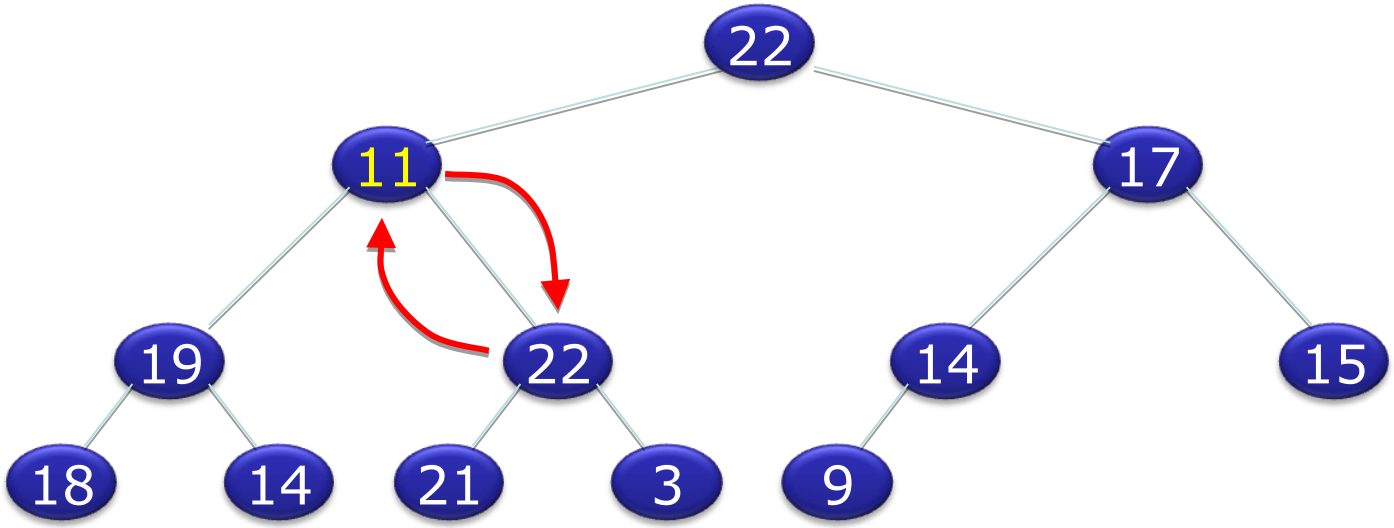
- A árvore está balanceada e justificada à esquerda, mas não é mais um heap
- Porém, *somente a raiz* não possui a propriedade-heap



- Peneiramos para cima a raiz (sift up)
- Feito isto, somente um filho perdeu a propriedade-heap

O método reamontoar II

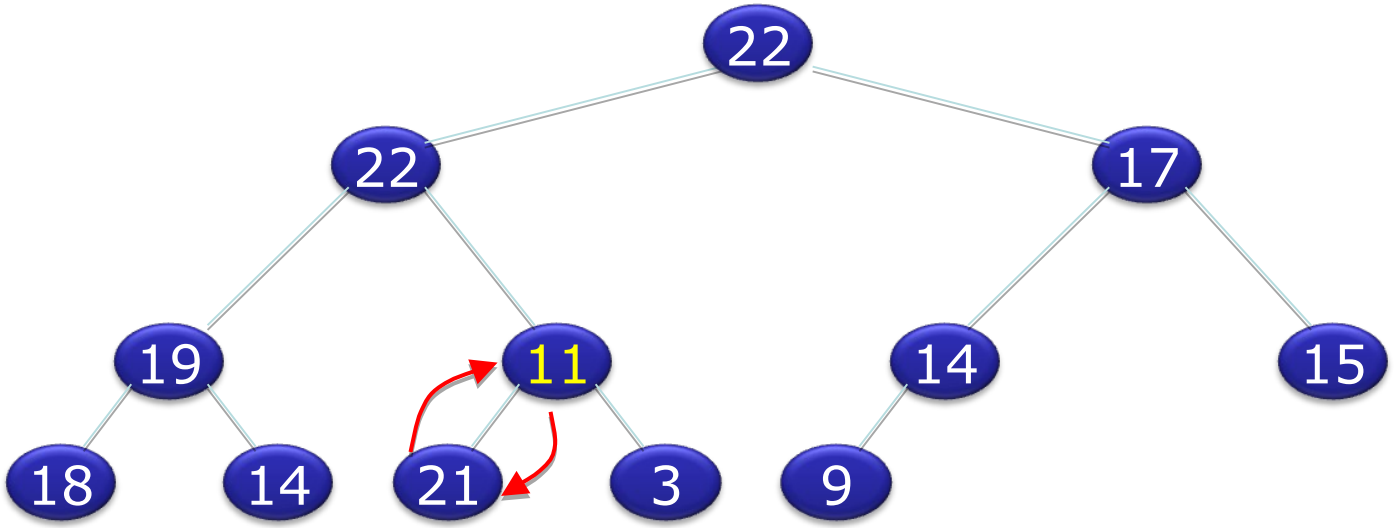
- Agora o filho à esquerda da raiz perdeu a propriedade-heap (também um 22)



- Este nodo podemos também peneirar para cima
- Novamente somente um filho perdeu a condição-heap

O método reamontoar III

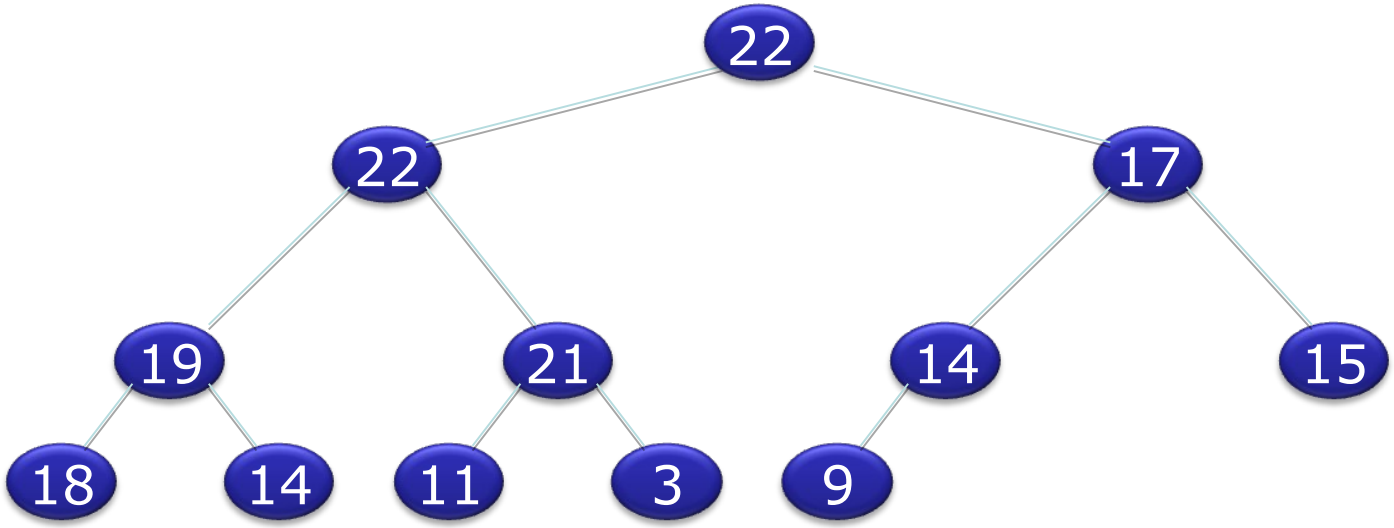
- Agora o filho à direita do filho à esquerda da raiz (11) é não-heap:



- Podemos peneirar para cima este nodo
- Depois, somente um de seus filhos poderia ter perdido a propriedade-heap, mas todos são folha!

O método reamontoar IV

- A árvore torna a ser um monte porque todos os nodos possuem a propriedade-heap



- O maior valor voltou a estar na raiz
- Podemos repetir isto até a árvore estar vazia
- Assim retiramos do maior ao menor

Ordenação usando um Monte

- A árvore é balanceada e justificada à esquerda, pode ser representada em um vetor
  - *Somente* funciona com árvores balanceadas justificadas è esquerda
  - Podemos representar tudo como operações sobre *arrays*

Para ordenar:

Passo 1: amontoe o vetor;

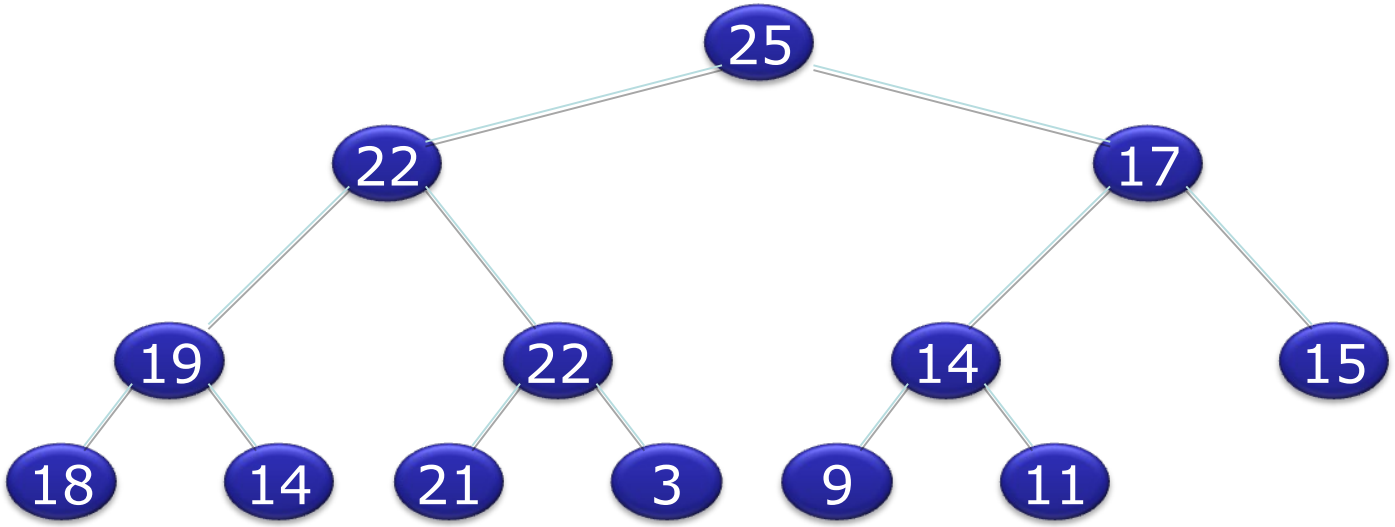
ENQUANTO houver elementos {

Passo 2: remova e substitua a raiz;

Passo 3: reamontoe o novo nodo;

}

Mapeando em um vetor



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Observe:

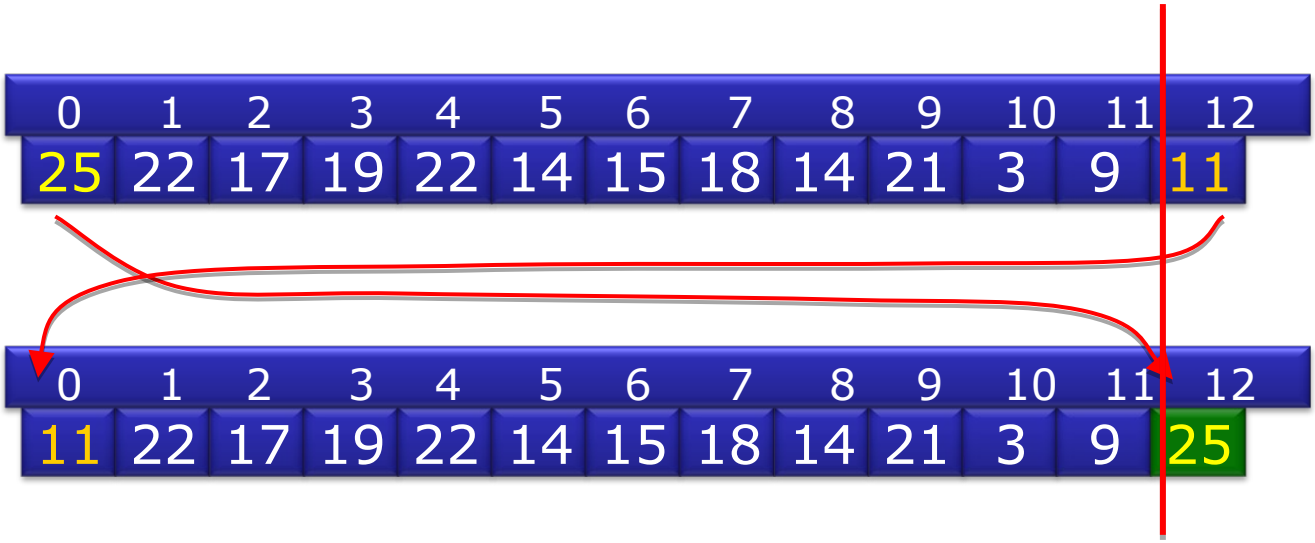
  Filho da esquerda de i está em 2*i+1

  Filho da direita de i está em 2*i+2

  Exemplo: filhos de 3 (19) são 7 (18) e 8 (14)

Removendo e substituindo a raiz

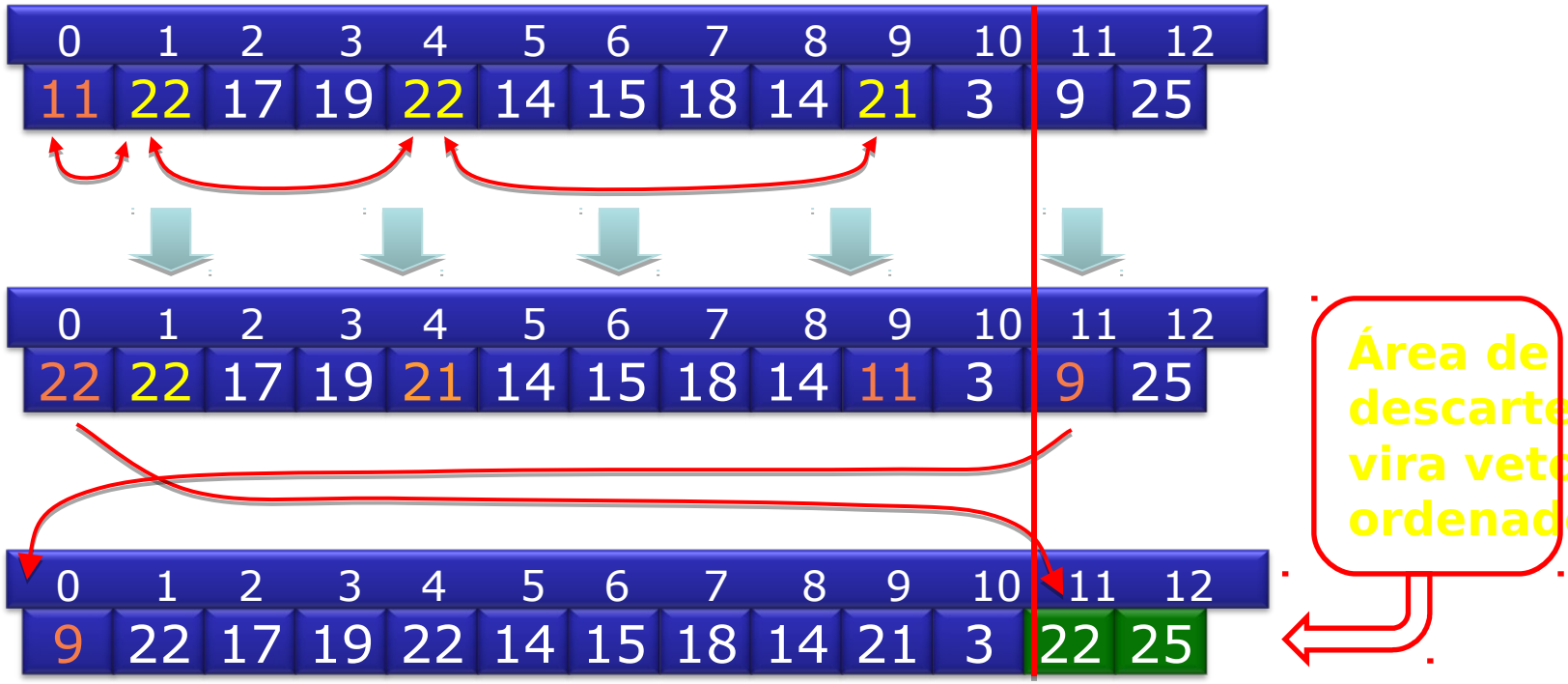
- A “raiz” é o primeiro elemento no vetor
- O “nodo mais à direita no nível mais profundo” é o último elemento
- Troque-os



- ...e faça de conta que o ultimo elemento do vetor não existe mais, sendo o último índice `11` (contendo 9)

Reamontoe e repita

- *Reamontoe* a raiz (índice 0, contendo 11)...



- Lembre-se que o “ultimo” mudou.
- Repita até o “ultimo” ser o primeiro e está ordenado!

Análise de Complexidade I

- Inicialização: Passo 1: Amontoe (*heapifique*) o vetor
- Para amontoar adicionamos os *n* nodos
  - Cada nodo tem de ser peneirado para cima
    - Como a árvore binária está perfeitamente balanceada, peneirar um nodo toma O(log n)
  - Como fazemos isto *n* vezes, amontoar toma n*O(log n), que é O(n log n) tempo

Análise de Complexidade II

- O laço principal:

```
ENQUANTO houver elementos {
    Passo 2: remova e substitua a raiz;
    Passo 3: reamontoe o novo nodo;
}
```

- Repetimos o laço *n* – 1 vezes, uma para cada remoção de um elemento
- Remover e substituir a raiz toma O(1) tempo
- O tempo total é n vezes o tempo de reamontoar

Análise de Complexidade III

- Para reamontoar a raiz, temos de, no pior caso, seguir apenas *um* caminho da raiz até um nodo folha
  - Eventualmente parando antes
    - Este caminho tem comprimento máximo O(log n) pois a árvore é perfeitamente balanceada
  - Logo, reamontoar um nodo tem complexidade O(log n)
- Como reamontoamos num laço repetido n vezes, a complexidade do laço é n*O(log n), ou O(n log n)

Análise de Complexidade IV

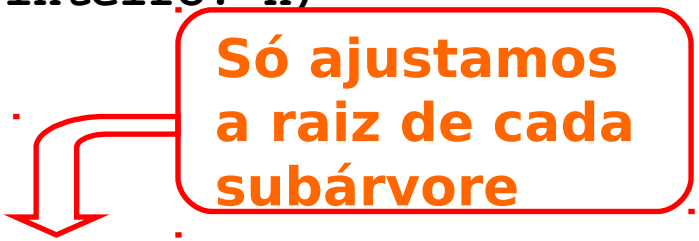
- Olhe o algoritmo novamente:

```
Passo 1: amontoe o vetor;
ENQUANTO houver elementos {
    Passo 2: remova e substitua a raiz;
    Passo 3: reamontoe o novo nodo;
}
```

- Amontoar tem complexidade O(n log n)
- O laço tem complexidade O(n log n)
- A complexidade total é O(n log n) + O(n log n)
- O que é o mesmo que O(n log n)

Algoritmo Heapsort

```
heapsort(tInfo: a[], inteiro: n)
   variáveis
        inteiro: i;
        tInfo: temp;
   início
        para i de (n div 2) até 1 passo -1 faça
               ajuste(a, i, n);  // Amontoe.
        fim para
        para i de (n-1) até 1 passo -1 faça
               temp <- a[i+1];
               a[i+1] <- a[1];  // Posição inicial = 1.
               a[1] <- temp;
               ajuste(a, 1, i);  // Reamontoe.
        fim para
   fim
```



Algoritmo do ajuste

```
ajuste(tInfo: a[], inteiro: i, n)
   variáveis
   inteiro: j;
   tInfo: aAux;
   início
   aAux <- a[i];  // Supõe posição inicial = 1.
   j <- 2*i;
   enquanto j <= n faça
   se j < n E a[j] < a[j+1] então
   incremente j;
   fim se
   se aAux >= a[j] então
   retorne
   fim se
   a[j div 2] <- a[j];
   a[j] <- aAux;  // Falta no algoritmo dado em
                  // Horowitz & Sahni.
   j <- 2*j;
   fim enquanto
   a[j div 2] <- aAux;
   fim
```

creative commons

COMMONS DEED

Atribuição-Uso Não-Comercial-Compartilhamento pela Licença 2.5 Brasil

Você pode:
- copiar, distribuir, exibir e executar a obra
- criar obras derivadas

Sob as seguintes condições:
Atribuição — Você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante.
Uso Não-Comercial — Você não pode utilizar esta obra com finalidades comerciais.
Compartilhamento pela mesma Licença — Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-sa/2.5/br/ ou mande uma carta para Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.