Estruturas de Dados 2

Prof. Silvia Brandão

2024.1

Momento N2 – 25 pts

- Avaliação contínua de aprendizagem (todas as aulas) – 5pts
- Atividade avaliativa – 20pts
  - Implementação e avaliação comparativa dos métodos complexos de ordenação (15pts):
    - Quick Sort
    - Merge Sort
    - Heap Sort
    - Shell Sort
  - **Apresentação** (5pts): códigos e gráfico comparativo das curvas de desempenho dos n dados x tempo de ordenação
  - **Data de entrega e apresentação**: 22/05, postagem dos arquivos e gráfico no diário de bordo (basta um aluno postar, com o nome dos integrantes da equipe)

Momento N3 – 25 pts

- Projeto Prático
  - Ver arquivo com descrição do projeto (tema, avaliação e apresentação)

Algoritmo Heap Sort

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- **Funcionamento:**
  - 1. Selecione o menor item do vetor.
  - 2. Troque-o com o item da primeira posição do vetor.
  - 3. Repita estas operações com os **n − 1** itens restantes, depois com os **n − 2** itens, e assim sucessivamente.
- O custo para encontrar o menor (ou o maior) item entre **n** itens é **n − 1** comparações.
  - Isso pode ser reduzido utilizando uma **fila de prioridades**.

https://youtu.be/Xw2D9aJRBY4

Fila de Prioridades

É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de elementos rapidamente.

Aplicações:

- Sistemas Operacionais usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
- Alguns métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
- Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

Fila de Prioridades

Tipo Abstrato de Dados

- **Operações:**
  1. Constrói uma fila de prioridades a partir de um conjunto com **n** itens.
  2. Retira o item com maior prioridade.
  3. Restaura a filas de prioridades em uma única.

- **Forma de implementação**
  - Árvore binária

Árvore Binária

- Considerando as características de uma árvore binária de busca, poderíamos entender que:
  - As chaves poderiam ser inseridas, uma a uma, em uma árvore binária de busca;
  - Após a inserção de todas as chaves a árvore poderia ser percorrida, por exemplo, em **in-ordem** e as chaves seriam obtidas em ordem crescente.

- **Desvantagens da utilização de uma árvore binária de busca**:
  - Necessidade de área de memória adicional para o armazenamento da árvore;
  - O que aconteceria se as chaves já se encontrarem em ordem ou em ordem inversa?
    - Seria gerada um árvore degenerada. O que significa que para a inserção do **i-ésimo** elemento seriam requeridas **i-1** comparações, o que, praticamente, elimina a vantagem de se utilizar uma árvore no processo.

Heap Sort

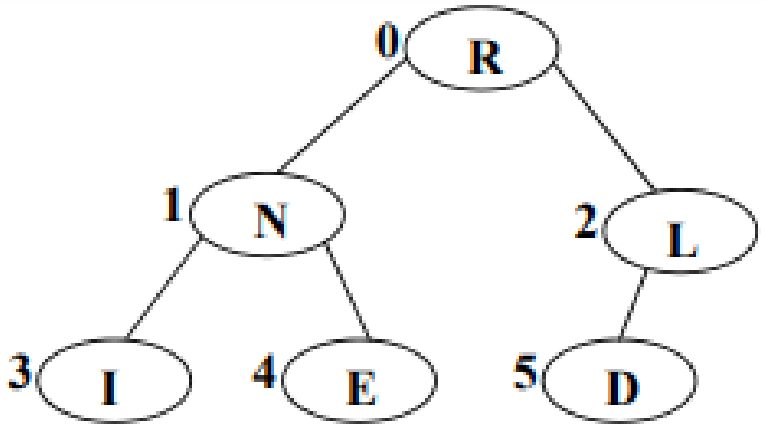
- As deficiências da classificação utilizando árvore binária ordenada são eliminadas no método heap sort.

- O heap sort é um método *in situ (*no local) de complexidade constante, independente da ordem da entrada.

- O Heap é uma estrutura de dados implementada como uma árvore binária.
  - No caso, é uma árvore binária, onde cada nó tem no máximo dois nós filhos e cada novo item é sempre adicionado de cima para baixo, e da esquerda para a direita. Mas não são árvores binárias de pesquisa.
  - Duas propriedades definem o Heap:
    1. O valor de um nó é maior ou igual ao valor de seus filhos;
    2. O Heap é uma árvore binária completa ou quase-completa da esquerda para a direita.

Heap Sort

- **Árvore Binária quase completa** é uma árvore binária onde:
  1. Cada folha na árvore está no nível **d** ou no nível **d−1**;
  2. Para todo nó **nd** que possui um descendente direito no nível **d**, todo descendente esquerdo de **nd** é folha no nível **d** ou tem 2 filhos.
- **Exemplo**: Árvore binária quase completa



Índices	0	1	2	3	4	5
Valores	R	N	L	I	E	D

Relação: info[j] ≤ info[(j-1)/2]
para 0 ≤ ((j-1)/2) < j ≤ n-1

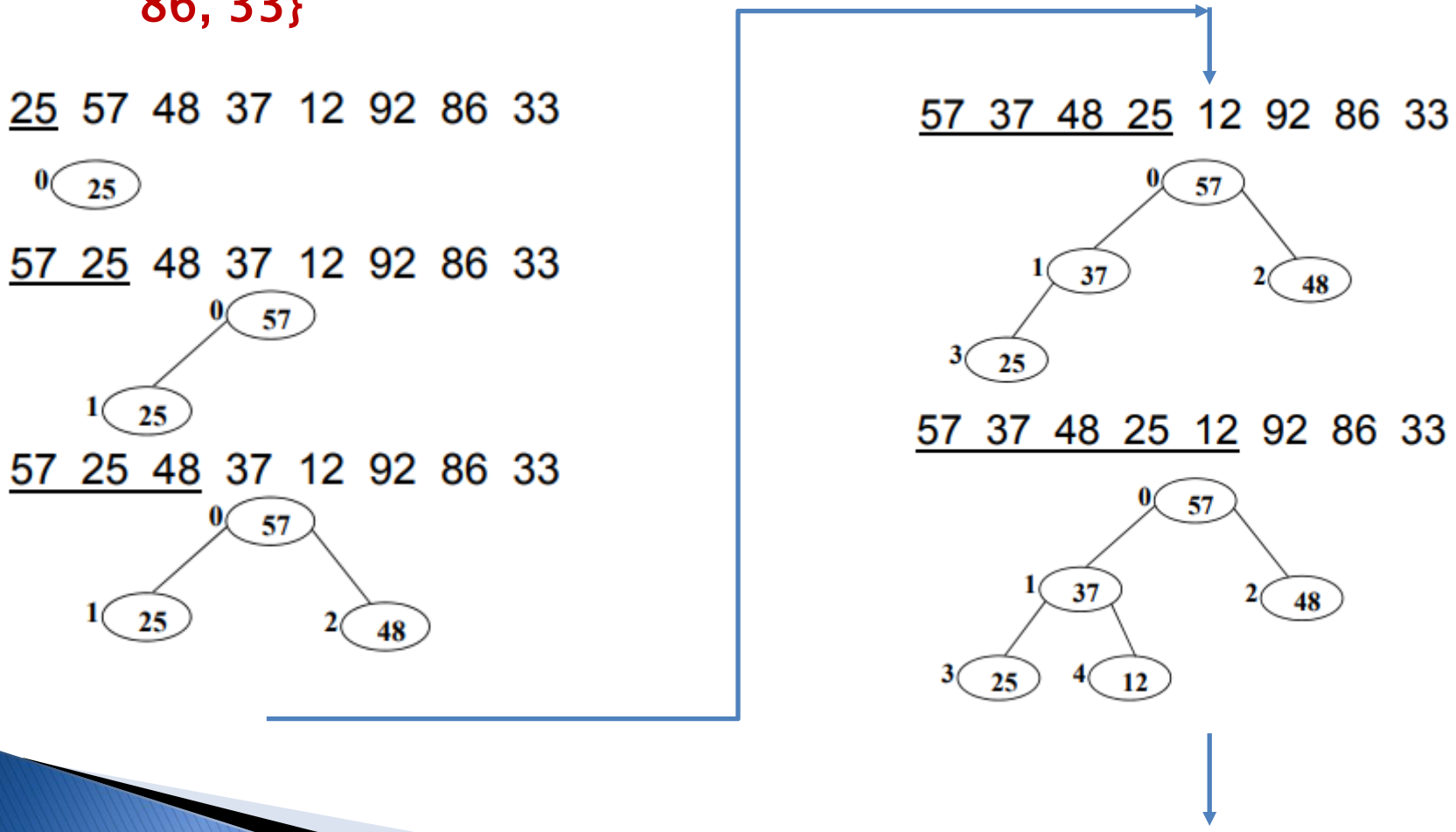
Algoritmo Heap Sort

Construção do heap binário:

```
def heapSort(vetor):
    for i in range(1,len(vetor)):
        e = vetor[i]
        s = i
        f = int((s-1)/2)
        while s>0 and vetor[f]<e:
            vetor[s] = vetor[f]
            s = f
            f = int((s-1)/2)
        vetor[s] = e;
```

Heap Sort – passo a passo

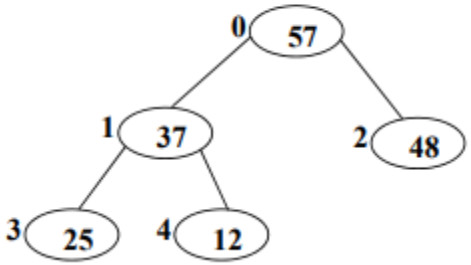
- **Exemplo**: construção de um heap partindo de um conjunto de n elementos (chaves). **Vetor = {25, 57, 48, 37, 12, 92, 86, 33}**



Heap Sort – passo a passo

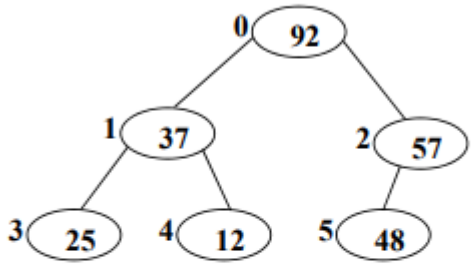
- **Exemplo**: construção de um heap partindo de um conjunto de n elementos (chaves). Vetor = {25, 57, 48, 37, 12, 92, 86, 33}

57 37 48 25 12 92 86 33

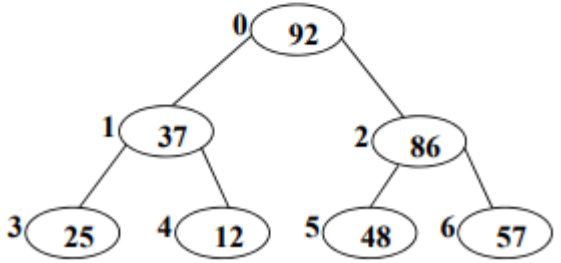


57 37 92 25 12 48 86 33

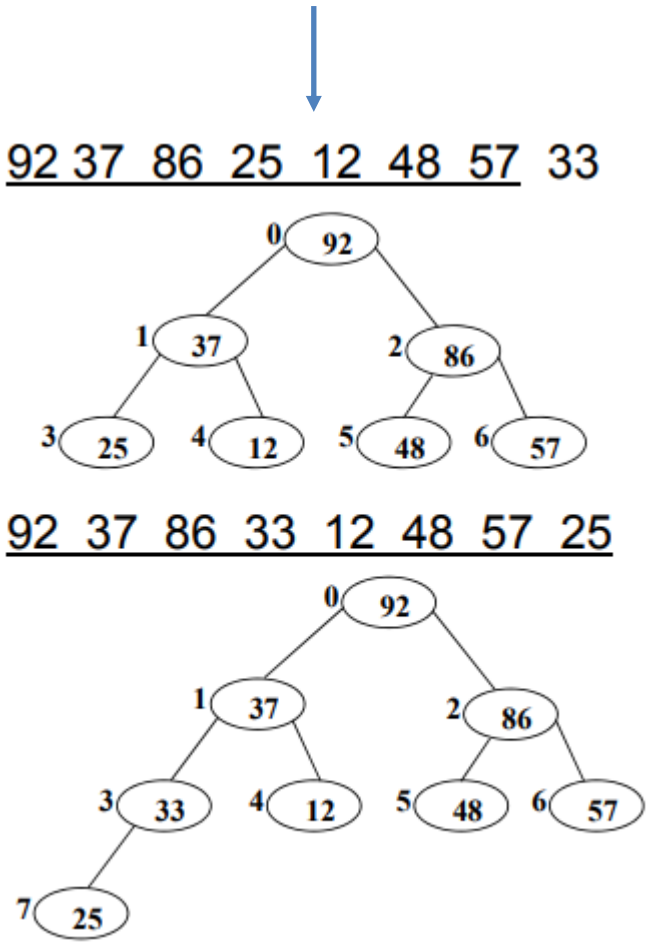
92 37 57 25 12 48 86 33



92 37 86 25 12 48 57 33



Heap Sort – passo a passo

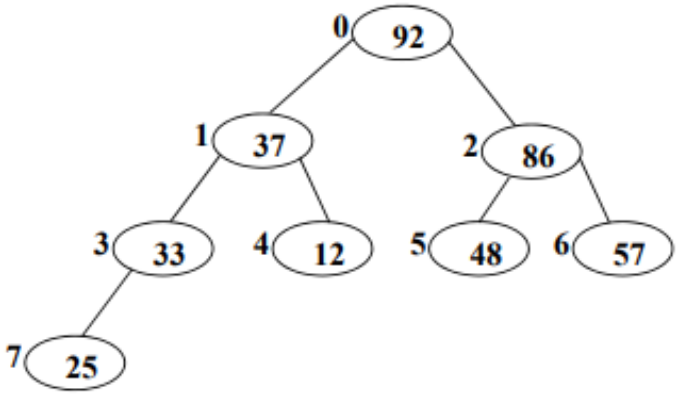


OBS.:

1. Recebemos um conjunto de chaves (elementos) e o transformamos em um heap binário.
2. Observamos no heap binário que **a raiz contém o elemento de maior valor** do conjunto de chaves.
3. Agora, podemos removê-lo e posicioná-lo no final do vetor que armazena o heap. Contudo, para isso temos que **reorganizar o heap**, mantendo sua propriedade e liberando espaço no final do vetor para colocar o elemento mencionado.

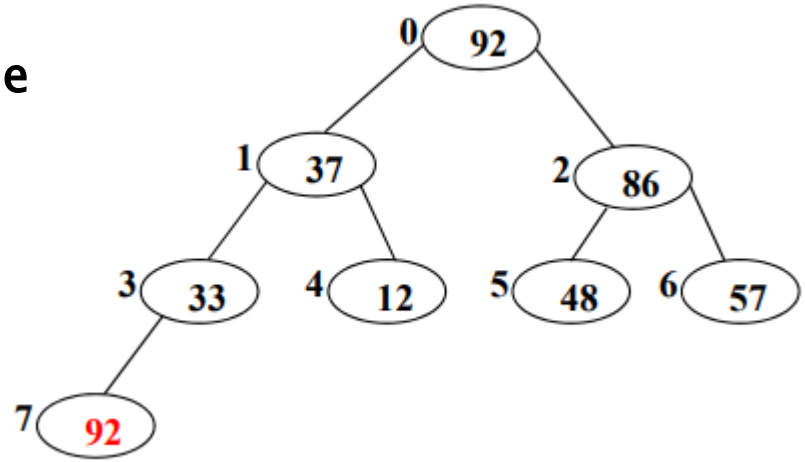
Heap Sort – passo a passo

Temos o vetor = [92, 37, 86, 33, 12, 48, 57, 25] representado no heap a seguir:



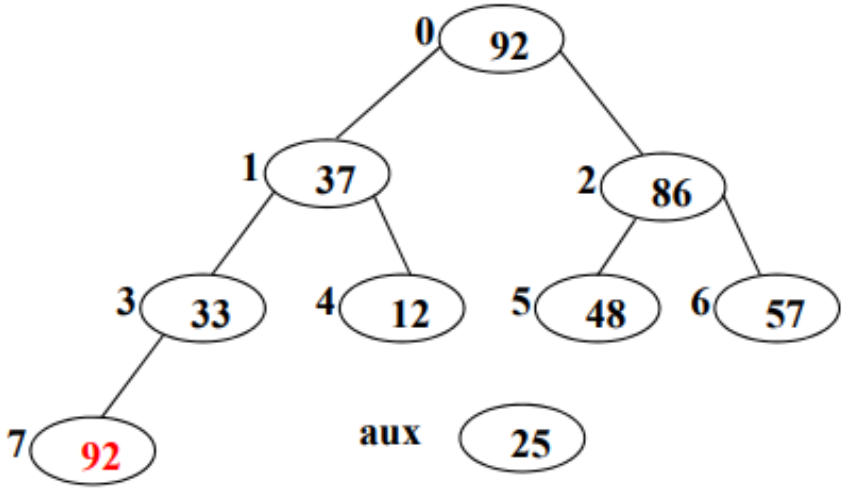
Armazenaremos o último elemento do nosso vetor em uma **variável auxiliar**.

Agora podemos **copiar o elemento de maior valor para aposição final do vetor**.

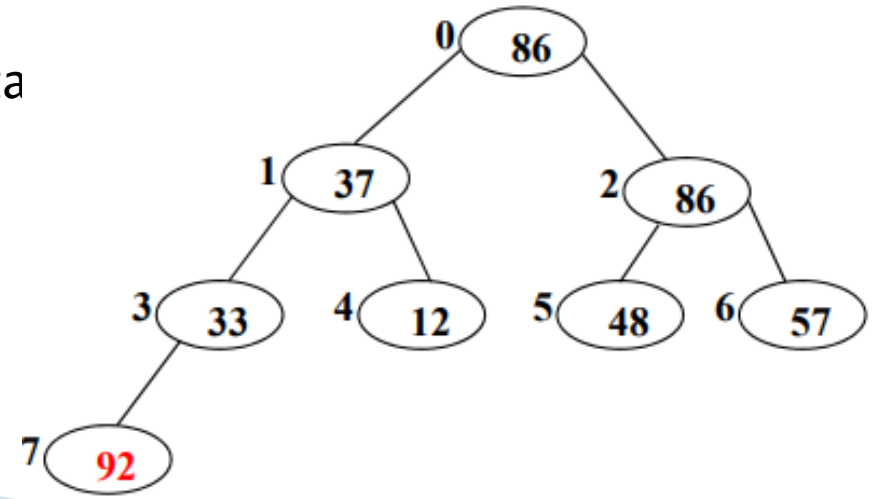


Heap Sort – passo a passo

Podemos considerar a **posição com índice zero como livre no vetor**, reorganizar o heap e posicionar o valor armazenado em **aux** na posição correta.

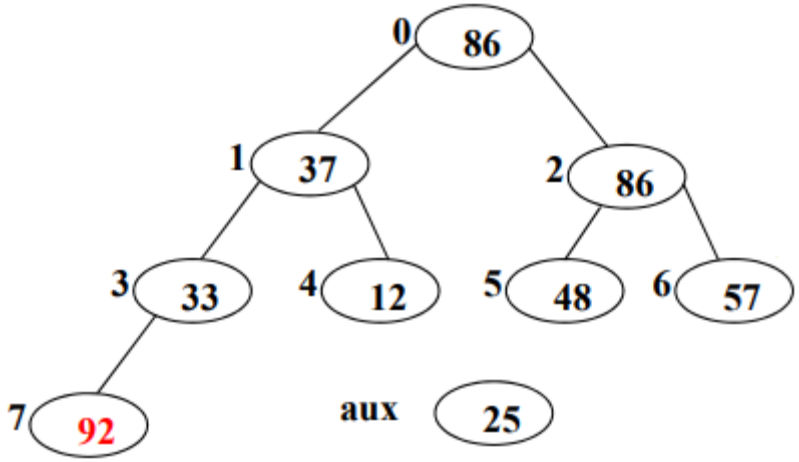


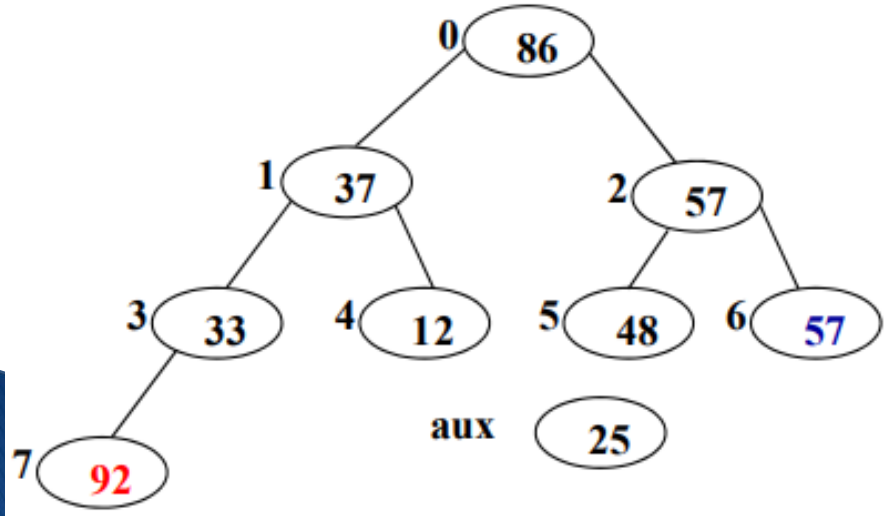
Como fazer isto?
Escolhendo o maior dentre os filhos da raiz e deslocá-lo para ta posição.



Heap Sort – passo a passo

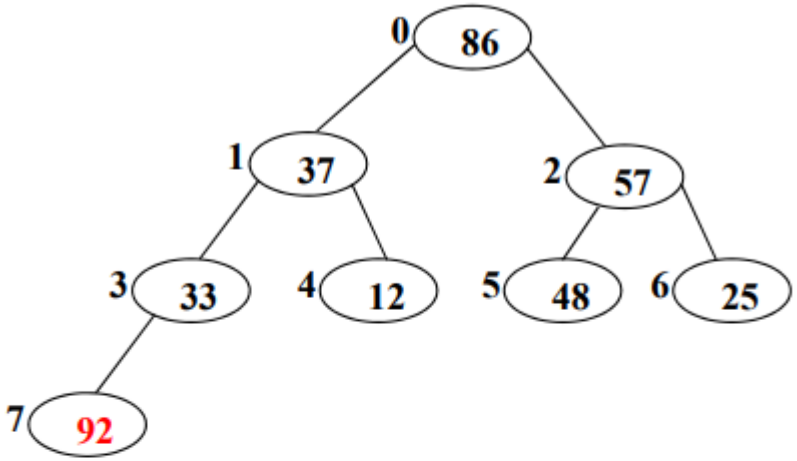
Mantendo este raciocínio, teremos **a posição com índice 2 livre** e a preencheremos com seu filho de maior valor.



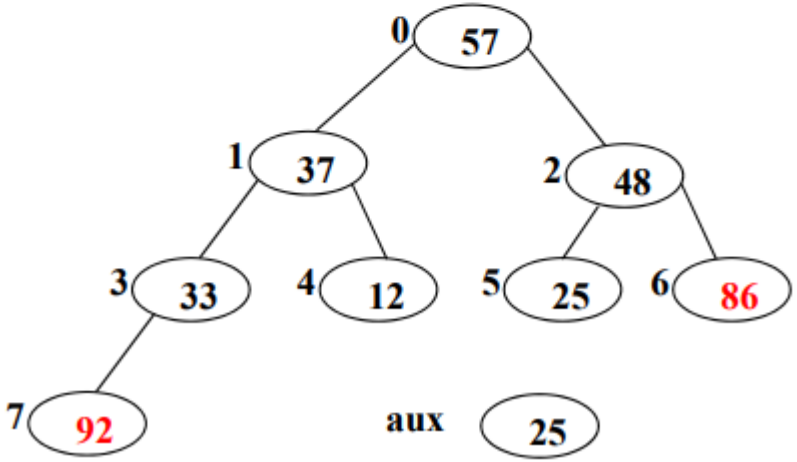


Quando chegarmos ao **último filho que foi movido teremos a posição de inserção do valor em aux**. Após sua inserção teremos o heap do próximo slide.

Heap Sort – passo a passo

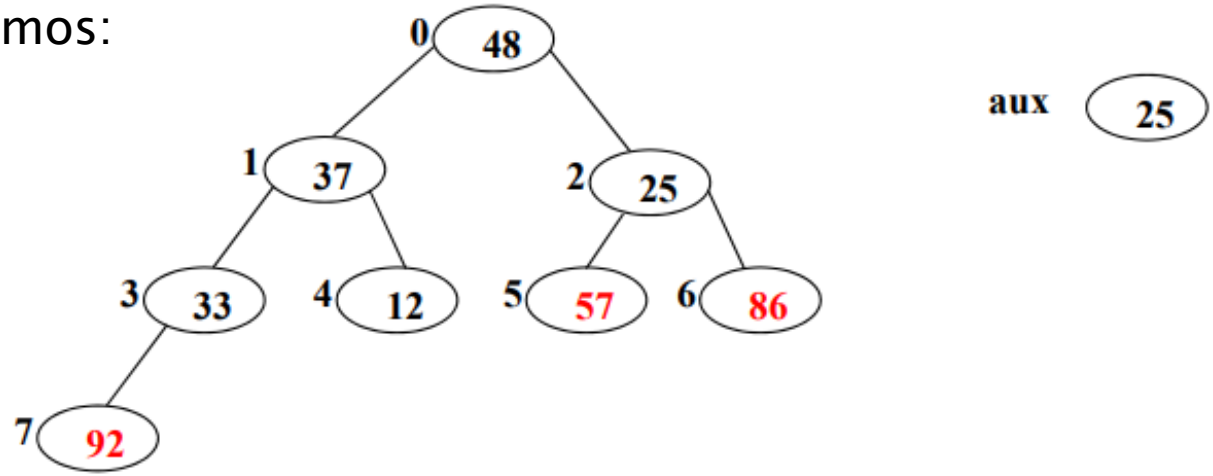


Aplicando este procedimento ao subvetor de **n-1** teremos:

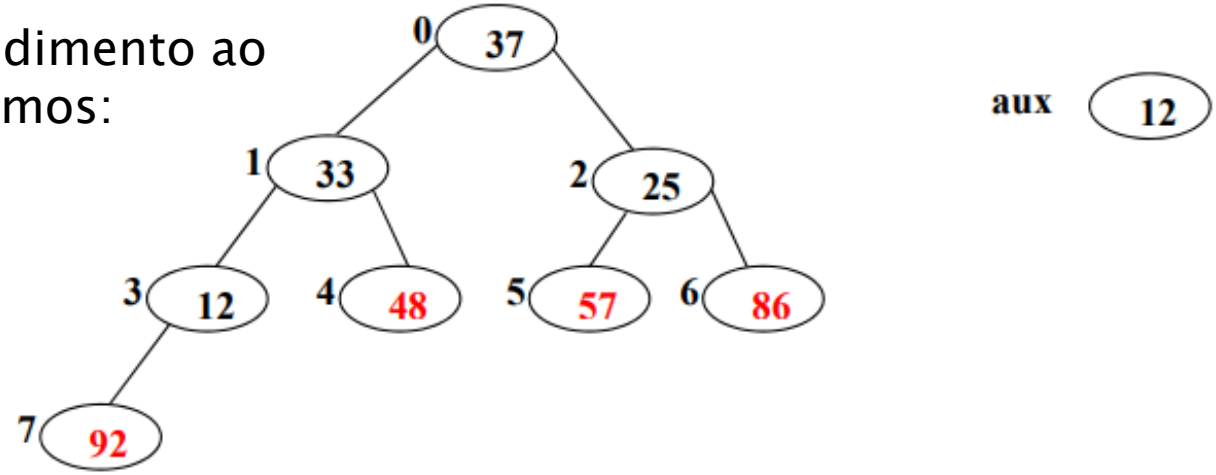


Heap Sort – passo a passo

Aplicando este procedimento ao subvetor de **n-2** teremos:

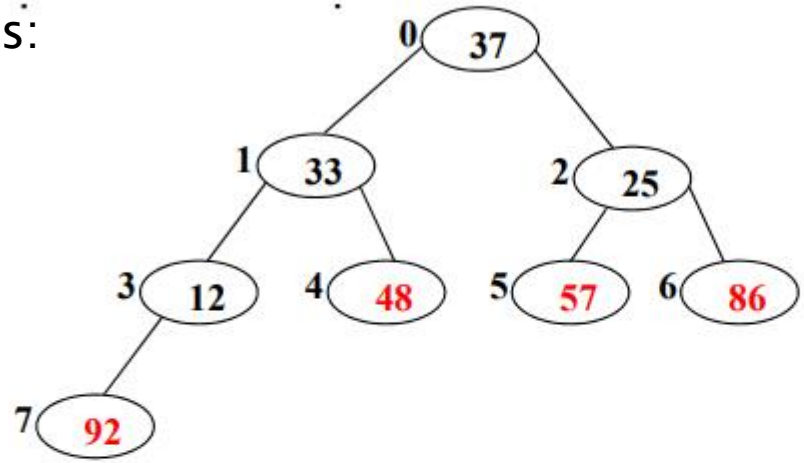


Aplicando este procedimento ao subvetor de **n-3** teremos:



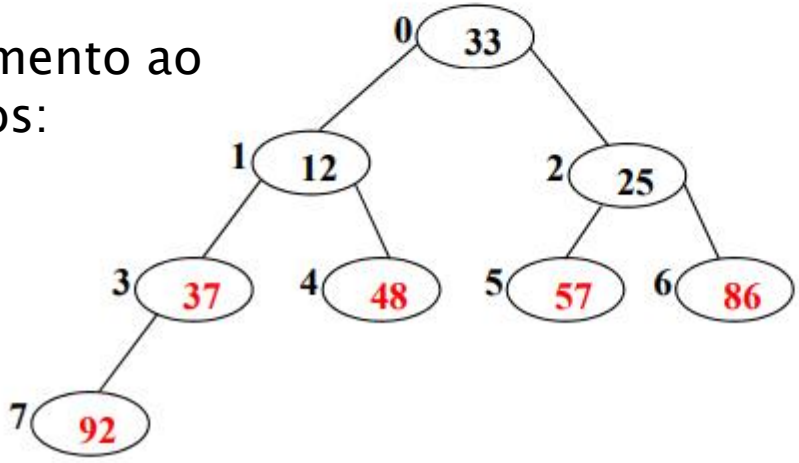
Heap Sort – passo a passo

Aplicando este procedimento ao subvetor de **n-3** teremos:



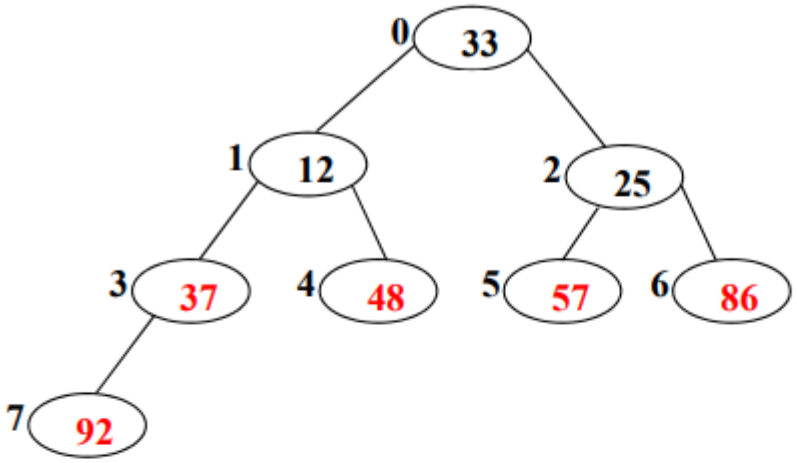
aux 12

Aplicando este procedimento ao subvetor de **n-4** teremos:

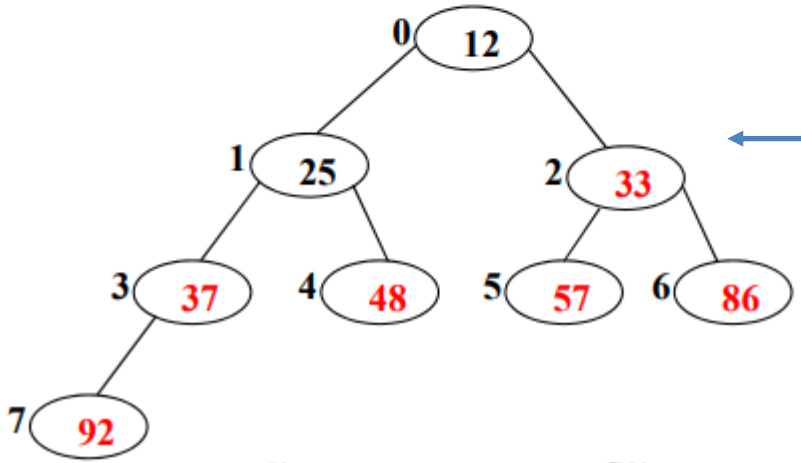


aux 25

Heap Sort – passo a passo



Aplicando este procedimento ao subvetor de **n-5** teremos:



Na última fase, **n-6**, não movemos diretamente o filho, analisamos seu valor e apenas se este for menor que aux o movemos.

Algoritmo Heap Sort

Com base no que foi apresentado, teremos dois métodos para que o vetor de chaves seja ordenado de forma crescente:

- **heapSort() modificado**– construção do heap binário
- **ordenaHeap()** – ordenação do vetor

```
def heapSort(vetor):
    d = len(vetor)
    e = int(d / 2)
    while e > 0:
        ordenaHeap(e, d, vetor)
        e -= 1
    while d >= 1:
        x = vetor[0]
        vetor[0] = vetor[d - 1]
        vetor[d - 1] = x
        d -= 1
        ordenaHeap(1, d, vetor)
```

```
def ordenaHeap(e, d, vetor):
    i = e
    j = 2 * i
    naoachou = 1
    x = vetor[i - 1]
    while j <= d and naoachou == 1:
        if (j < d):
            if (vetor[j - 1] < vetor[j]):
                j += 1
        if (x < vetor[j - 1]):
            vetor[i - 1] = vetor[j - 1]
            i = j
            j = 2 * i
        else:
            naoachou = 0
    vetor[i - 1] = x
```

Algoritmo Heap Sort

- **Complexidade:**
  - Para analisar o heap sort, observe que uma árvore binária completa com n nós tem **log (n+1)** níveis. Por conseguinte, se cada elemento no vetor fosse uma folha, exigindo que fosse filtrado pela árvore inteira durante a criação e o ajuste do heap, a classificação ainda seria **O(n log n).**
  - **No caso médio**, o heap sort não é tão eficiente quanto o quick sort. Experimentos indicam que o heap sort exige, aproximadamente, o dobro do tempo do quick sort para a entrada classificada aleatoriamente.
  - Entretanto, o heap sort é bem superior ao quick sort **no pior caso.** Na realidade, o heap sort permanece **O (n log n)** no pior caso.

Algoritmo Heap Sort

- **Vantagens**:
  - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
  - Não necessita de nenhuma memória adicional.
  - É bom para arquivos com grandes registros.
- **Desvantagens**:
  - O Heapsort não é recomendado para vetores de entrada com poucos elementos, devido a complexidade do heap.
  - O Heapsort não é estável.
  - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.

Algoritmo Heap Sort

Tempo de execução no pior caso:

Método	Complexidade
Inserção	O(n^2)
Seleção	O(n^2)
Bolha	O(n^2)
Shell sort	O(n lg(n)^2) ou O(n^(3/2))
Quick sort	O(n^2) ou O(n lg(n))
Merge sort	O(n lg(n))
Heap sort	O(n lg(n))

Referência

- Viana, Daniel. **Conheça os principais algoritmos de ordenação**. https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao

- Sorting. https://www.cs.usfca.edu/~galles/visualization/Algorithms.html