

Algoritmos e Estruturas de Dados II

# HeapSort

Prof. Tiago Eugenio de Melo

[tmelo@uea.edu.br](mailto:tmelo@uea.edu.br)

[www.tiagodemelo.info](http://www.tiagodemelo.info)

# Observações

- As palavras com a fonte Courier indicam as palavras-reservadas da linguagem de programação.

# Referências

- **Algorithms in a Nutshell.** George T. Heineman, Gary Pollice, Stanley Selkow. O'Reilly Media, 2009.
- **Projetos de Algoritmos – com implementações em Pascal e C.** Nivio Ziviani. 2ª edição. Thomson, 2005.

# HeapSort

# HeapSort

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.
  - Troque com o elemento da primeira posição.



# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.
  - Troque com o elemento da primeira posição.
  - Repita as duas primeiras operações com os  $n-1$  elementos restantes.

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.
  - Troque com o elemento da primeira posição.
  - Repita as duas primeiras operações com os  $n-1$  elementos restantes.
- Qual é o custo para encontrar o maior (ou menor) item?

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.
  - Troque com o elemento da primeira posição.
  - Repita as duas primeiras operações com os  $n-1$  elementos restantes.
- Qual é o custo para encontrar o maior (ou menor) item?
  - Custa  $n-1$  comparações.

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.
  - Troque com o elemento da primeira posição.
  - Repita as duas primeiras operações com os  $n-1$  elementos restantes.
- Qual é o custo para encontrar o maior (ou menor) item?
  - Custa  $n-1$  comparações.
  - Como reduzir esse custo?

# HeapSort

- Utiliza o mesmo princípio de ordenação por seleção
  - Selecione o menor item da lista.
  - Troque com o elemento da primeira posição.
  - Repita as duas primeiras operações com os  $n-1$  elementos restantes.
- Qual é o custo para encontrar o maior (ou menor) item?
  - Custa  $n-1$  comparações.
  - Como reduzir esse custo?
    - Fila de prioridades.

# Fila de Prioridades

# Fila de Prioridades

# Fila de Prioridades

- É uma fila onde cada elemento possui uma prioridade.



# Fila de Prioridades

- É uma fila onde cada elemento possui uma prioridade.
- Essa prioridade determina a posição de um elemento na fila.

# Fila de Prioridades

- É uma fila onde cada elemento possui uma prioridade.
- Essa prioridade determina a posição de um elemento na fila.
- Numa fila comum, o elemento do início sempre é removido (FIFO – *First In, First Out*).

# Fila de Prioridades

- É uma fila onde cada elemento possui uma prioridade.
- Essa prioridade determina a posição de um elemento na fila.
- Numa fila comum, o elemento do início sempre é removido (FIFO – *First In, First Out*).
- Na fila com prioridades, o elemento removido é determinado pela prioridade.

# Fila de Prioridades

- É uma fila onde cada elemento possui uma prioridade.
- Essa prioridade determina a posição de um elemento na fila.
- Numa fila comum, o elemento do início sempre é removido (FIFO – *First In, First Out*).
- Na fila com prioridades, o elemento removido é determinado pela prioridade.
- A fila de prioridades possui o critério de ordenação de acordo com a prioridade.

# Fila de Prioridades

# Fila de Prioridades

- Aplicações:

# Fila de Prioridades

- Aplicações:
  - Fila de pacientes que aguardam por algum órgão.

# Fila de Prioridades

- Aplicações:
  - Fila de pacientes que aguardam por algum órgão.
  - Caminhos mínimos (algoritmo de Dijkstra).



# Fila de Prioridades

- Aplicações:
  - Fila de pacientes que aguardam por algum órgão.
  - Caminhos mínimos (algoritmo de Dijkstra).
  - Escalonamento de processos (SO).

# Fila de Prioridades

# Fila de Prioridades

- Existem vários tipos de implementação:

# Fila de Prioridades

- Existem vários tipos de implementação:
  - Lista encadeada.

# Fila de Prioridades

- Existem vários tipos de implementação:
  - Lista encadeada.
  - Heap.

# Fila de Prioridades

- Existem vários tipos de implementação:
  - Lista encadeada.
  - Heap.
  - Lista desordenada.

# Fila de Prioridades

- Existem vários tipos de implementação:
  - Lista encadeada.
  - Heap.
  - Lista desordenada.
  - Lista ordenada.

# Fila de Prioridades

- Existem vários tipos de implementação:
  - Lista encadeada.
  - Heap.
  - Lista desordenada.
  - Lista ordenada.
- A escolha do tipo de implementação depende da aplicação.



# Heap

# HeapSort

# HeapSort

- Fila de prioridades

# HeapSort

- Fila de prioridades
  - Representação inicial

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$
      - Retirar:  $O(1)$



# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$
      - Retirar:  $O(1)$
    - Lista linear não-ordenada

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$
      - Retirar:  $O(1)$
    - Lista linear não-ordenada
      - Construir:  $O(n)$

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$
      - Retirar:  $O(1)$
    - Lista linear não-ordenada
      - Construir:  $O(n)$
      - Inserir:  $O(1)$

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$
      - Retirar:  $O(1)$
    - Lista linear não-ordenada
      - Construir:  $O(n)$
      - Inserir:  $O(1)$
      - Retirar:  $O(n)$

# HeapSort

- Fila de prioridades
  - Representação inicial
    - Lista linear ordenada.
      - Construir:  $O(n \log n)$
      - Inserir:  $O(n)$
      - Retirar:  $O(1)$
    - Lista linear não-ordenada
      - Construir:  $O(n)$
      - Inserir:  $O(1)$
      - Retirar:  $O(n)$
    - Outra alternativa: *heaps*

# HeapSort

# HeapSort

- Algoritmo criado em 1964 por John Williams.

# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:



# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:
  - Sequência de elementos com chaves

# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:
  - Sequência de elementos com chaves  $c[1], c[2], \dots, c[n]$

# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:
  - Sequência de elementos com chaves  $c[1], c[2], \dots, c[n]$  tal que

# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:
  - Sequência de elementos com chaves  
 $c[1], c[2], \dots, c[n]$   
tal que  
 $c[i] \geq c[2.i],$

# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:
  - Sequência de elementos com chaves  
 $c[1], c[2], \dots, c[n]$   
tal que  
 $c[i] \geq c[2.i],$   
 $c[i] \geq c[2.i + 1],$

# HeapSort

- Algoritmo criado em 1964 por John Williams.
- Definição de *heap*:
  - Sequência de elementos com chaves  
 $c[1], c[2], \dots, c[n]$   
tal que  
 $c[i] \geq c[2.i],$   
 $c[i] \geq c[2.i + 1],$   
para todo  $i = 1, 2, \dots, n/2.$

# HeapSort

- Exemplo:

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 50 | 40 | 30 | 20 | 25 | 10 | 15 |
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |

# HeapSort

- Exemplo:

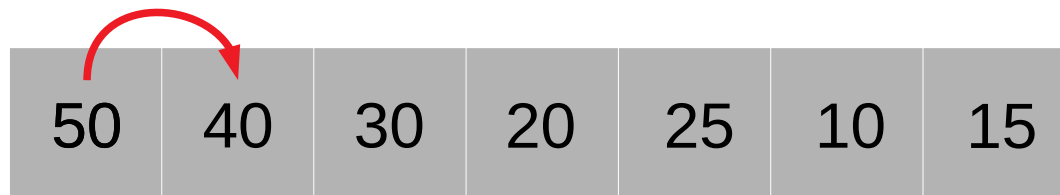
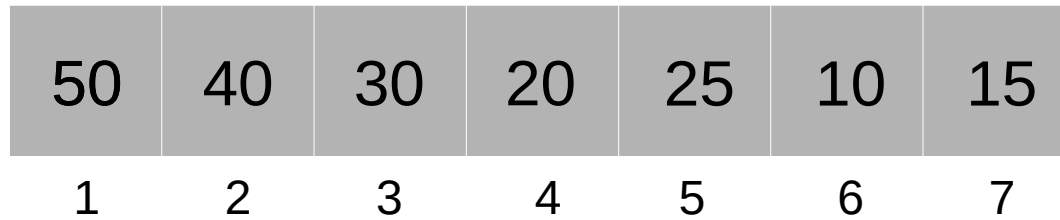
|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 50 | 40 | 30 | 20 | 25 | 10 | 15 |
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 50 | 40 | 30 | 20 | 25 | 10 | 15 |
|----|----|----|----|----|----|----|



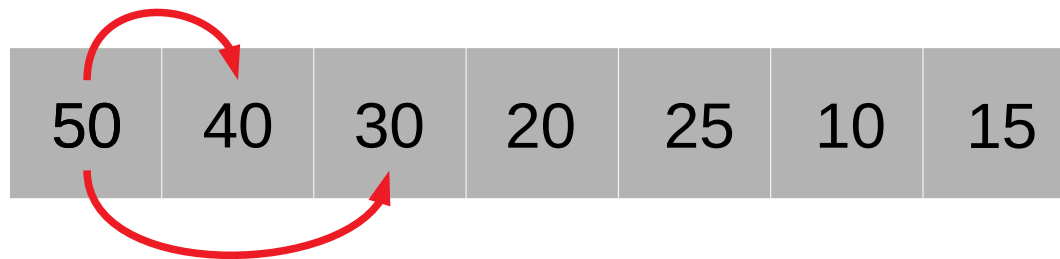
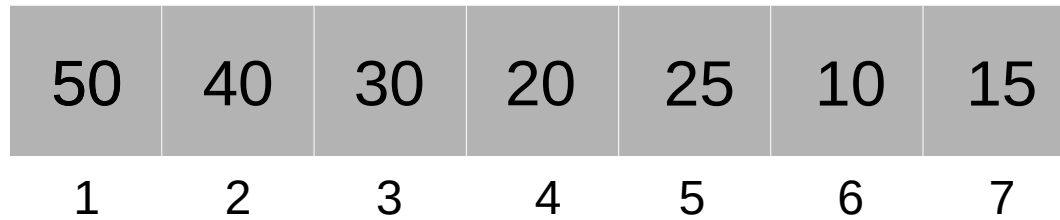
# HeapSort

- Exemplo:



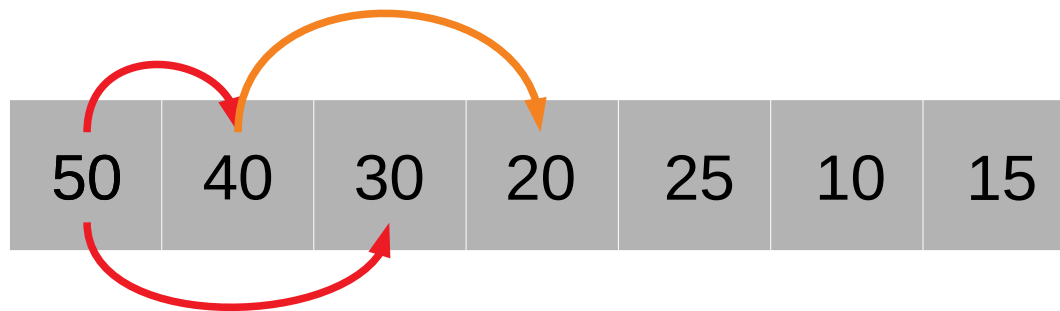
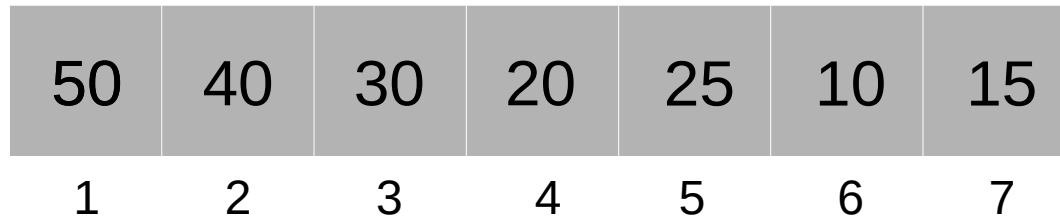
# HeapSort

- Exemplo:



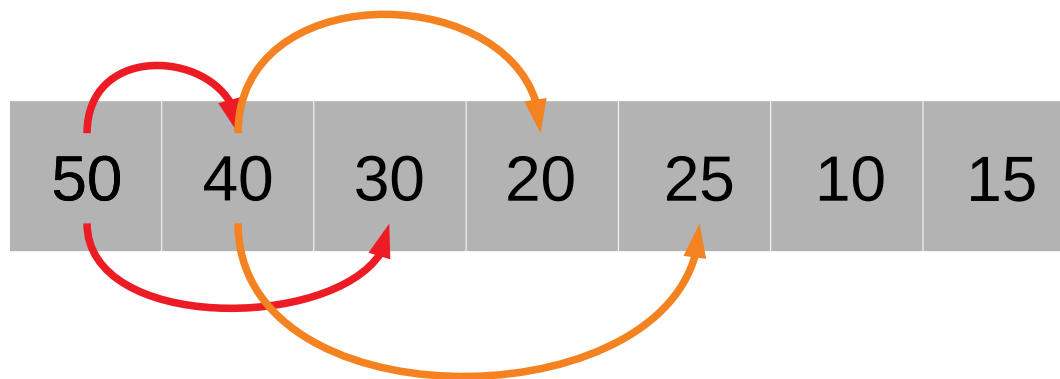
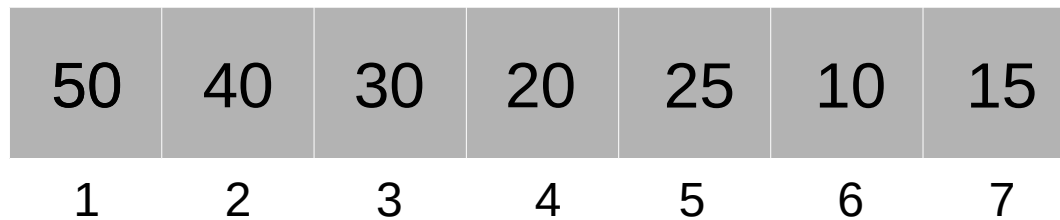
# HeapSort

- Exemplo:



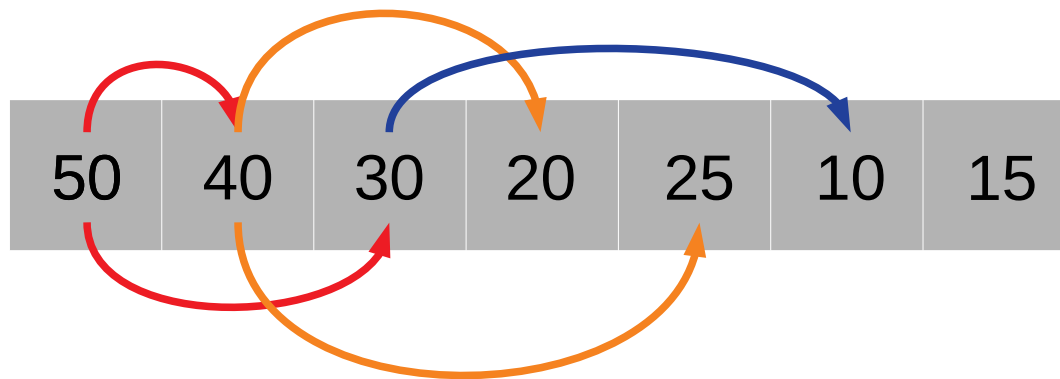
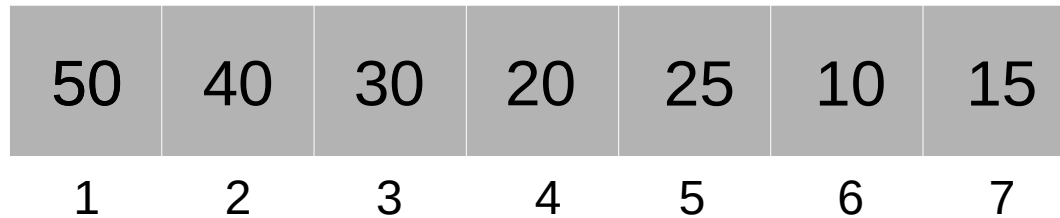
# HeapSort

- Exemplo:



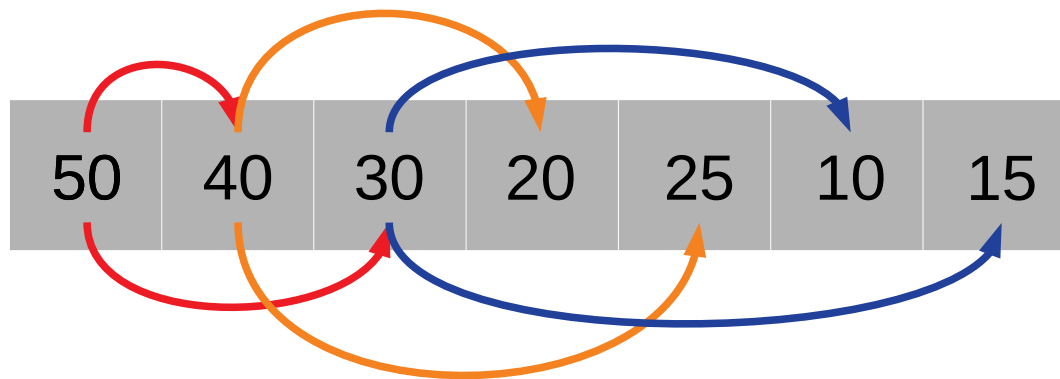
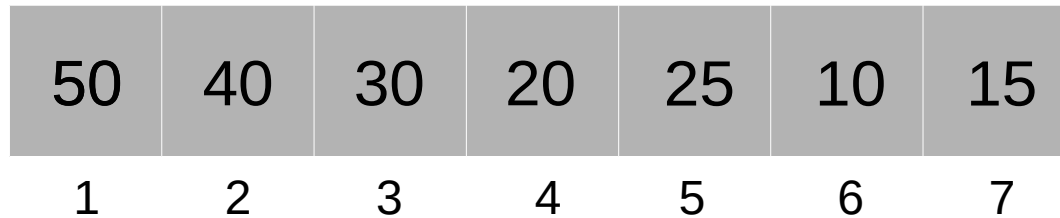
# HeapSort

- Exemplo:



# HeapSort

- Exemplo:

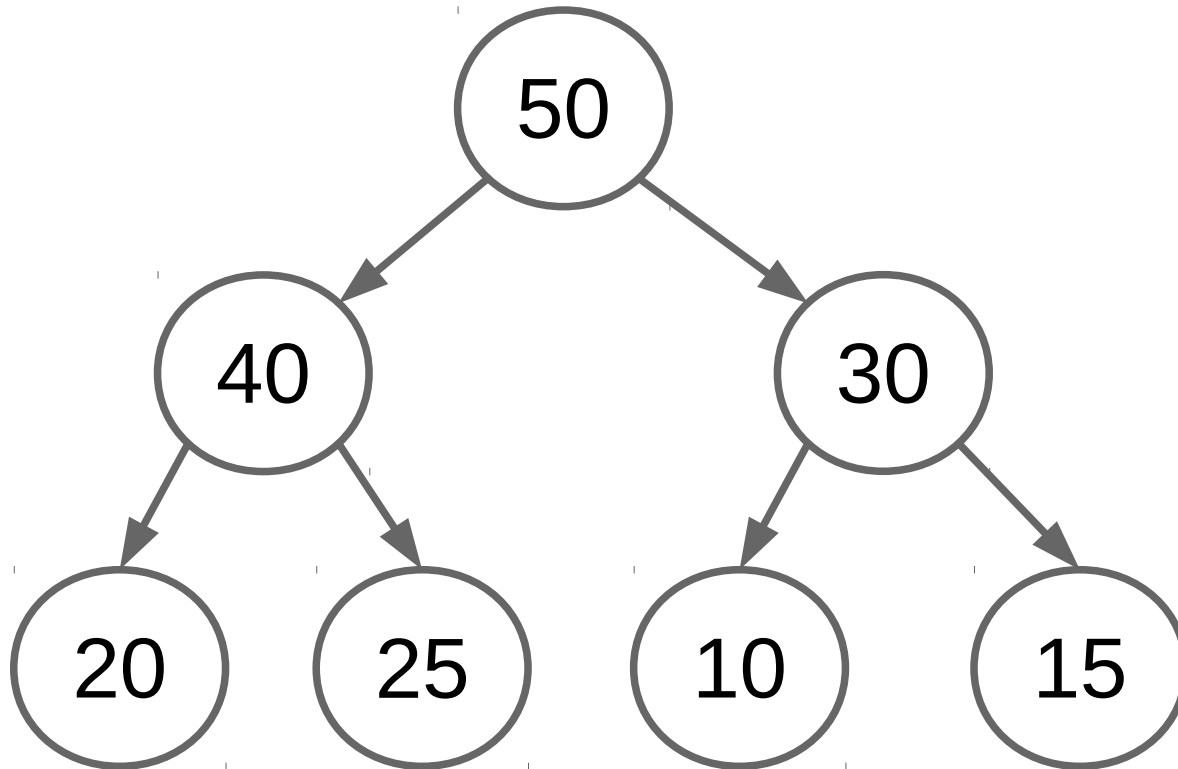


# HeapSort

- Essa ordenação pode ser visualizada como uma árvore binária completa:

# HeapSort

- Essa ordenação pode ser visualizada como uma árvore binária completa:





# HeapSort

# HeapSort

- Portanto, um *heap* é uma árvore binária completa na qual cada nó satisfaz a condição do *heap* apresentada na sua definição.

# HeapSort

- Portanto, um *heap* é uma árvore binária completa na qual cada nó satisfaz a condição do *heap* apresentada na sua definição.
- Os elementos no *heap* não estão perfeitamente ordenados.

# HeapSort

- Portanto, um *heap* é uma árvore binária completa na qual cada nó satisfaz a condição do *heap* apresentada na sua definição.
- Os elementos no *heap* não estão perfeitamente ordenados.
- Sabe-se apenas que o maior elemento está no nó raiz e que, para cada nó, todos os seus descendentes não são maiores que o elemento raiz.

# HeapSort

# HeapSort

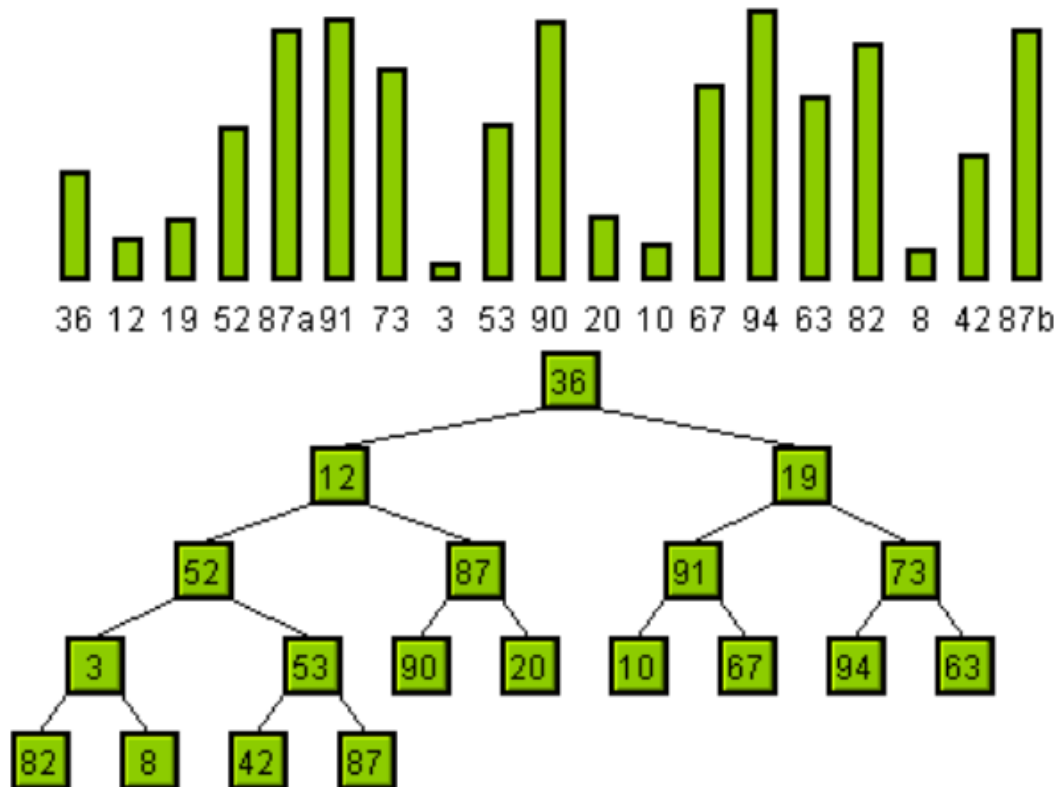
- Tenta-se evitar a utilização real de uma árvore.

# HeapSort

- Tenta-se evitar a utilização real de uma árvore.
- A ideia é utilizar a abordagem de *heap* representando uma árvore como uma lista.

# HeapSort

- Tenta-se evitar a utilização real de uma árvore.
- A ideia é utilizar a abordagem de *heap* representando uma árvore como uma lista.





# HeapSort

# HeapSort

- Por que usar um *heap* é importante?

# HeapSort

- Por que usar um *heap* é importante?
  - Descobrir quem é o maior elemento é  $O(1)$ .

# HeapSort

- Por que usar um *heap* é importante?
  - Descobrir quem é o maior elemento é  $O(1)$ .
  - Se o valor da raiz for alterado, o heap pode ser refeito rapidamente  $O(\log n)$ .

# HeapSort

# HeapSort

- Algoritmo

# HeapSort

- Algoritmo
  - Transformar uma lista  $L$  em um heap  $H$ .

# HeapSort

- Algoritmo
  - Transformar uma lista  $L$  em um heap  $H$ .
  - Pegar o maior elemento de  $H$  e trocar com a posição  $L[\text{máx}]$ .



# HeapSort

- Algoritmo
  - Transformar uma lista  $L$  em um heap  $H$ .
  - Pegar o maior elemento de  $H$  e trocar com a posição  $L[\text{máx}]$ .
  - Repetir os passos anteriores com os demais elementos.

# HeapSort

# HeapSort

- Análise:

# HeapSort

- Análise:
  - O algoritmo aparenta não ser eficiente.

# HeapSort

- Análise:
  - O algoritmo aparenta não ser eficiente.
  - Porém, refazer o *heap* gasta cerca de  $\log n$  operações (pior caso).

# HeapSort

- Análise:
  - O algoritmo aparenta não ser eficiente.
  - Porém, refazer o *heap* gasta cerca de  $\log n$  operações (pior caso).
  - Portanto, *heapsort* é proporcional a  $n \log n$  no pior caso.

# HeapSort

- Análise:
  - O algoritmo aparenta não ser eficiente.
  - Porém, refazer o *heap* gasta cerca de  $\log n$  operações (pior caso).
  - Portanto, *heapsort* é proporcional a  $n \log n$  no pior caso.
  - O Quicksort é, em média, cerca de duas vezes mais rápido que o HeapSort.

# HeapSort

- Análise:
  - O algoritmo aparenta não ser eficiente.
  - Porém, refazer o *heap* gasta cerca de  $\log n$  operações (pior caso).
  - Portanto, *heapsort* é proporcional a  $n \log n$  no pior caso.
  - O Quicksort é, em média, cerca de duas vezes mais rápido que o HeapSort.
  - HeapSort é melhor que o ShellSort para grandes arquivos.



# HeapSort

# HeapSort

- Pontos positivos

# HeapSort

- Pontos positivos
  - $O(n \log n)$  qualquer que seja a entrada.

# HeapSort

- Pontos positivos
  - $O(n \log n)$  qualquer que seja a entrada.
- Pontos negativos

# HeapSort

- Pontos positivos
  - $O(n \log n)$  qualquer que seja a entrada.
- Pontos negativos
  - Método não estável.

# Exercícios

- Qual é a diferença entre heap máximo e heap mínimo? Existe diferença de eficiência entre eles?
- Implemente em Python uma função para reconstruir o heap, de tal maneira que as suas propriedades sejam mantidas.

# Exercícios

- Dado o seguinte conjunto de elementos  $X = \{ 8, 6, 7, 4, 5, 3, 2, 1 \}$ , desenhe o heap (árvore).