



UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

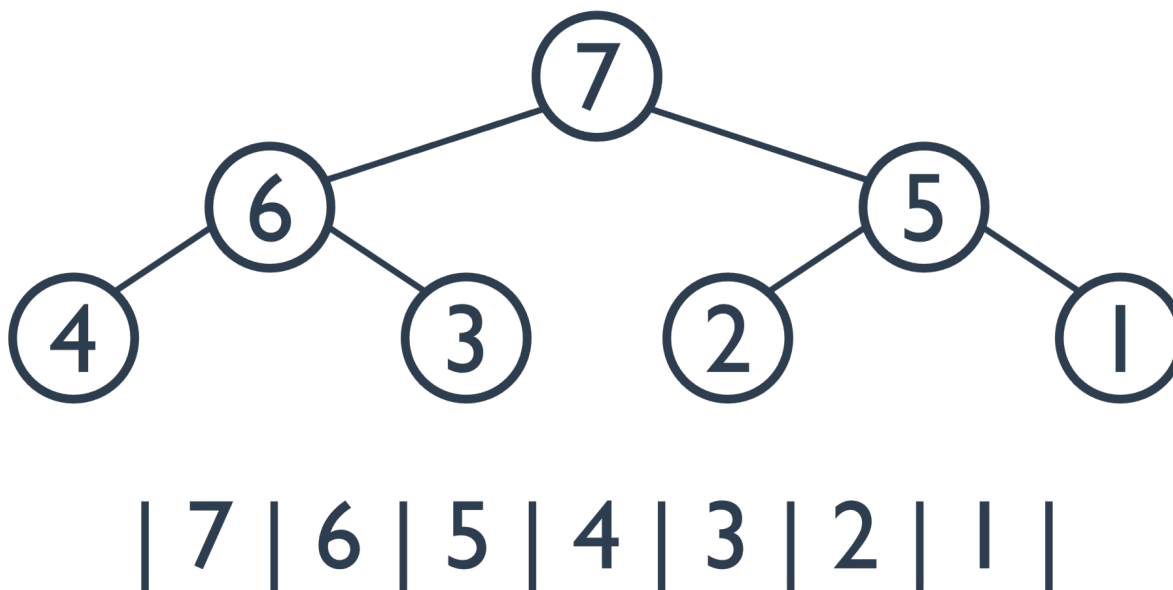
PROJETO E ANÁLISE DE ALGORITMOS

Cairé Carneiro Rocha
Larissa Feliciano
Lucas Cordeiro

Heapsort



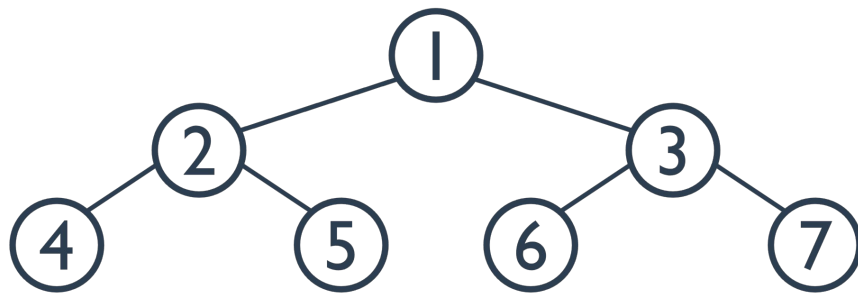
Algoritmo de ordenação que organiza um vetor em uma estrutura de Heap.



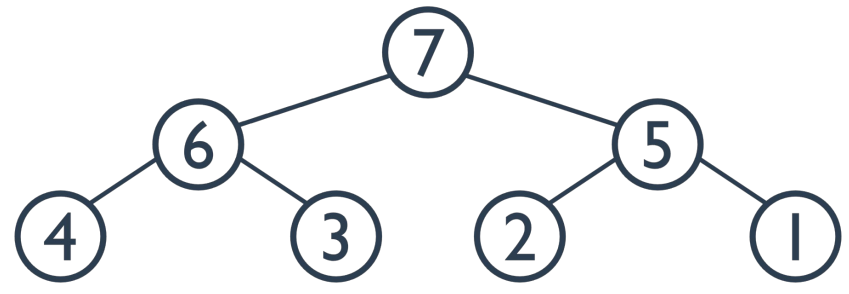
Heapsort



Há dois tipos de heap: heap mínimo e heap máximo



| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

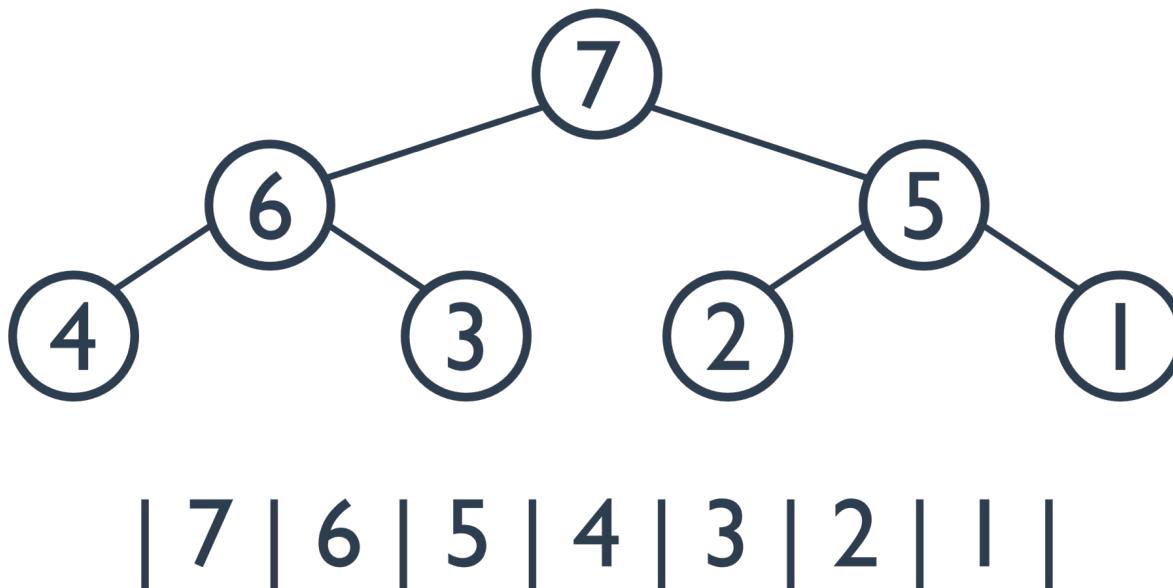


| 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Heapsort



Um Heap máximo pode ser representado por um array unidimensional de modo que a raiz ocupe a posição 1 e os demais elementos obedeça uma regra $esq.i = 2i$ e $dir.i = 2i + 1$



Heapsort



Há duas formas de ordenar a fila de prioridade, um elemento que sobe o heap, ou um elemento que desce o heap.

```
17 def desce(L, a, f):
18     e = filhoE(a)
19     d = filhoD(a)
20     m = a
21     if(d <= f):
22         if(L[e] >= L[d] and L[e] > L[a]):
23             m = e
24         elif(L[d] > L[e] and L[d] > L[a]):
25             m = d
26     elif(e <= f):
27         if(L[e] > L[a]):
28             m = e
29     if(m != a):
30         trocaAB(L, a, m)
31         desce(L, m, f)
```

```
33 def sobe(L, a):
34     p = pai(a)
35     if p >= 0:
36         if L[a] > L[p]:
37             trocaAB(L, a, p)
38             sobe(L, p)
```

A complexidade do subir e do descer é da ordem $O(\log n)$. Pois a cada nível da árvore, temos metade do resto para trabalhar

Heapsort



Esse é o heapsort em python

```
17 def desce(L, a, f):
18     e = filhoE(a)
19     d = filhoD(a)
20     m = a
21     if(d <= f):
22         if(L[e] >= L[d] and L[e] > L[a]):
23             m = e
24         elif(L[d] > L[e] and L[d] > L[a]):
25             m = d
26     elif(e <= f):
27         if(L[e] > L[a]):
28             m = e
29     if(m != a):
30         trocaAB(L, a, m)
31         desce(L, m, f)
48 def constroiMax(L):
49     m = int(len(L)/2)
50     for i in range(m, -1, -1):
51         desce(L, i, len(L)-1)
52
53 def heapSort(L):
54     constroiMax(L)
55     for i in range(len(L)-1, 0, -1):
56         print(L)
57         trocaAB(L, 0, i)
58         desce(L, 0, i-1)
```

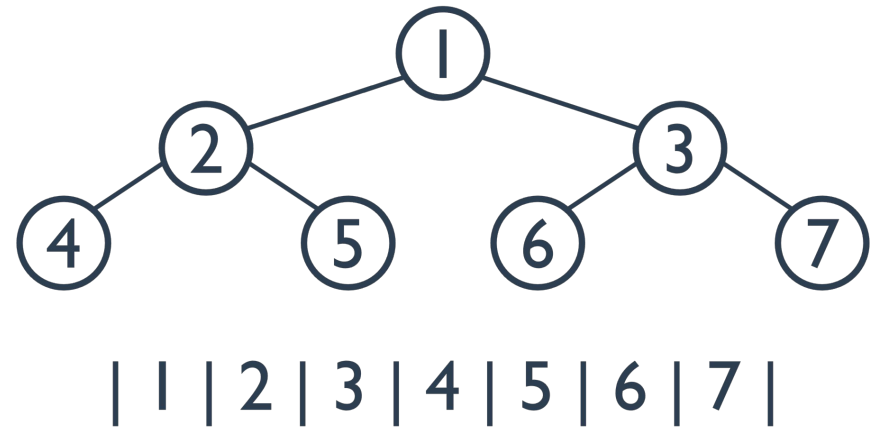
Heapsort



O primeiro elemento da lista ordenada é o pior caso para o heapfy.

Pois nessa circunstância ele passaria mais tempo chamando ele mesmo onde mostra a seta

```
17 def desce(L, a, f):
18     e = filhoE(a)
19     d = filhoD(a)
20     m = a
21     if(d <= f):
22         if(L[e] >= L[d] and L[e] > L[a]):
23             m = e
24         elif(L[d] > L[e] and L[d] > L[a]):
25             m = d
26     elif(e <= f):
27         if(L[e] > L[a]):
28             m = e
29     if(m != a):
30         trocaAB(L, a, m)
31         desce(L, m, f)
```



Heapsort



Observamos que a cada vez que a lista dobra, precisamos executar esse procedimento mais uma vez

Tamanho da entrada	Máximo de chamadas
2 ou 2^{11}	1
4 ou 2^{22}	2
8 ou 2^{33}	3
16 ou 2^{44}	4

Heapsort



Concluimos que o loop tem tamanho $\log(n)/\log(2)$ e como o loop é a única função que não tem tempo constante, então dizemos que o algoritmo roda na classe de $O(\log(n))$.

Tamanho da entrada n	$\log(n)$ $\log(2)$
2 ou $2^{^1}$	$\log(2) = 1$
4 ou $2^{^2}$	$\log(4) = 2$
8 ou $2^{^3}$	$\log(8) = 3$
16 ou $2^{^4}$	$\log(16) = 4$

Heapsort

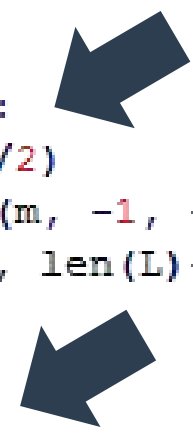


A função `constroiMax` chama a função `heapfy` $n/2$ vezes. Por isso dizemos que ela tem classe de $O(n \log(n))$

1 $O(n \cdot \log(n))$

2

```
48  def constroiMax(L):  
49      m = int(len(L)/2)  
50      for i in range(m, -1, -1):  
51          desce(L, i, len(L)-1)  
52  
53  def heapSort(L):  
54      constroiMax(L)  
55      for i in range(len(L)-1, 0, -1):  
56          print(L)  
57          trocaAB(L, 0, i)  
58          desce(L, 0, i-1)
```




Heapsort



A função `heapSort` chama `constroiMax` uma vez, chama a função `troca` e `desce` n vezes. `Troca` é da classe constante, `desce` é da classe $\log(n)$

```
48 def constroiMax(L):
49     m = int(len(L)/2)
50     for i in range(m, -1, -1):
51         desce(L, i, len(L)-1)
52
53 def heapSort(L):
54     constroiMax(L)
55     for i in range(len(L)-1, 0, -1):
56         print(L)
57         trocaAB(L, 0, i)
58         desce(L, 0, i-1)
```



Heapsort



Então a função que descreve o custo de tempo dessa função é:

$$HS(n) = \frac{1}{2} * n * \log(n) + n * \log(n) + n + c$$

Heapsort



Então a função que descreve o custo de tempo dessa função é:

$$HS(n) = \frac{1}{2}n \log(n) + n \log(n) + n + c$$

O big O é uma análise grosseira do pior caso, onde as constantes são ignoradas. Por isso dizemos que: $O(n \log(n))$

Heapsort



Então a função que descreve o custo de tempo dessa função é:

$$HS(n) = \frac{1}{2}n \log(n) + n \log(n) + n + c$$

O big O é uma análise grosseira do pior caso, onde as constantes são ignoradas. Por isso dizemos que: $O(n \log(n))$

Porém a análise Ω que é o melhor caso também diz que a ordem do algoritmo é de: $\Omega(n \log(n))$

Por isso dizemos que a complexidade dele é: $\Theta(n \log(n))$

<https://goo.gl/e1g8AZ>