

# Árvores

*Heaps* binárias

---

Prof. Edson Alves - UnB/FGA

2019

1. Definição
2. Inserção
3. Identificação e extração do elemento máximo
4. Construção de uma heap binária em  $O(N)$

# Definição

---

# Definição de heap binária

- A *heap* binária é uma estrutura de dados que mantém um conjunto de elementos, ordenados por meio de uma chave, que permite a identificação rápida do menor dentro todos estes elementos
- Uma variante comum da *heap* binária é a troca da identificação do menor elemento para a identificação do maior dentre seus elementos
- As duas operações principais de uma *heap* binária é a inserção de novos elementos ou a identificação (e extração) do menor elemento
- Outra operação importante é a construção de uma *heap* a partir de uma sequência de elementos dados

# Propriedade fundamental de uma heap binária

- Uma *heap* binária pode ser implementada a partir de uma árvore binária ou de um vetor
- A primeira alternativa tem como vantagem a visualização mais natural da propriedade fundamental das *heaps*
- A segunda permite uma implementação eficiente em termos de memória

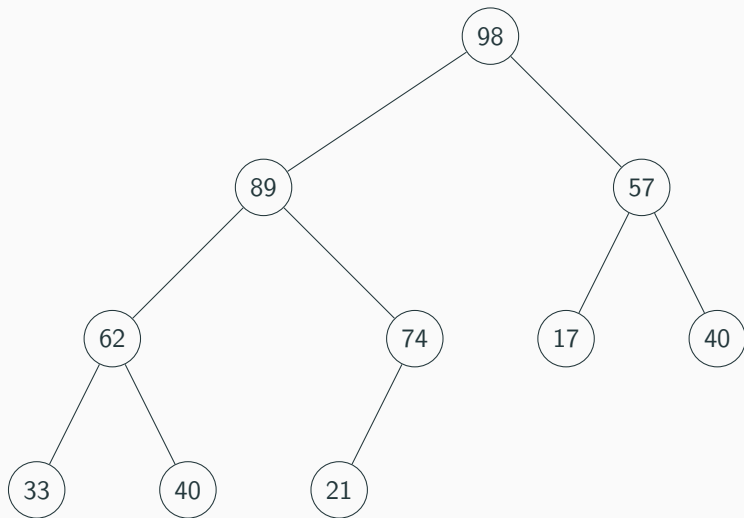
## Propriedade fundamental da heap binária

Para qualquer elemento  $x$  contido na *heap*, a chave de  $x$  é maior ou igual as chaves de todos os seus descendentes.

# Heaps como árvores

- A visualização de uma *heap* como uma árvore binária permite a identificação de propriedades consequentes da propriedade fundamental
- Primeiramente, a raiz da árvore será o menor dentre todos os elementos
- Em segundo lugar, a representação da *heap* não é única
- Além disso, a travessia de uma folha até a raiz leva a um caminho cujas chaves estão em ordem decrescente
- Esta propriedade é fundamental para a implementação das operações de inserção e remoção de elementos de um elemento

## Exemplo de max heap binária

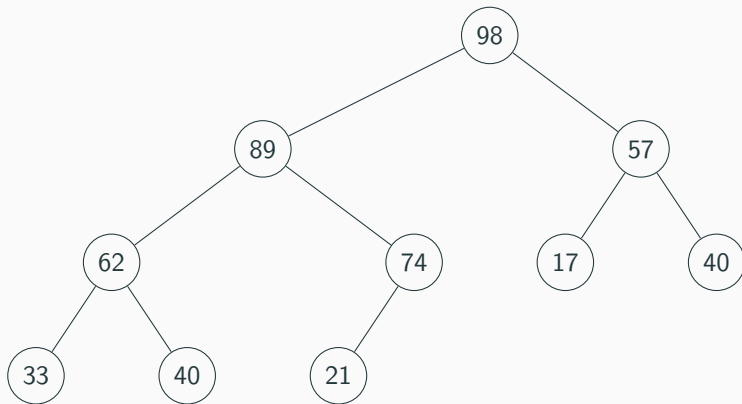


# Heaps binárias como vetores

- Uma *heap* binária pode ser armazenada como uma árvore cheia
- Esta propriedade permite o armazenamento de seus elementos em um vetor
- Se a raiz for armazenada no índice 1 (e não no zero), as relações de parentesco ficam simplificadas, através de operações binárias simples
- O pai de um elemento  $x$  que ocupa o índice  $i$  está no índice  $p = \lfloor i/2 \rfloor$
- O filho à esquerda de  $x$  tem índice  $l = 2i$
- O filho à direita de  $x$  tem índice  $r = 2i + 1$
- Mesmo que o índice zero fique inutilizado, a economia de memória em relação à representação com árvore é notável: com os parentes podem ser localizado em  $O(1)$ , não é necessário armazenar ponteiros
- Observe a relação entre a representação como vetor e uma travessia por largura da árvore binária



## Exemplo de heap binária como árvore e como vetor



1	2	3	4	5	6	7	8	9	10
98	89	57	62	74	17	40	33	40	21

## Exemplo de implementação de uma heap usando vector

```
1 #include <vector>
2 #include <iostream>
3 #include <stdexcept>
4
5 template<typename T>
6 class Heap {
7 private:
8     std::vector<T> xs;
9     size_t N;
10
11     size_t parent(int i) { return i/2; }
12     size_t left(int i) { return 2*i; }
13     size_t right(int i) { return 2*i + 1; }
14
15 public:
16     Heap() : xs(1), N(0) {}
17
18     size_t size() const { return N; }
19 }
```

# Insertão

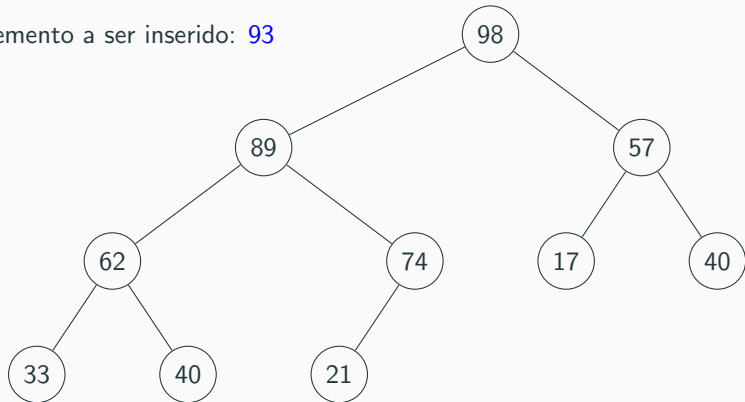
---

## Inserção em $O(\log N)$

- A inserção de um elemento em um *heap* binária pode ser feita em  $O(\log N)$ , onde  $N$  é o número de elementos armazenados na *heap*
- A inserção deve preservar a árvore cheia e a propriedade fundamental
- Manter a árvore cheia é simples: basta inserir o elemento na primeira posição desocupada do vetor, isto é, na posição do filho nulo mais à esquerda do último nível
- Isto pode violar a propriedade fundamental
- Para restaurar a propriedade da *heap*, basta trocar as informações do novo nó com o seu pai
- Esta troca pode levar a uma nova violação, entre o pai e o avô
- A violação sobe um nível por vez, de modo que são necessárias, no máximo,  $O(\log N)$  correções

## Exemplo de inserção em uma heap binária

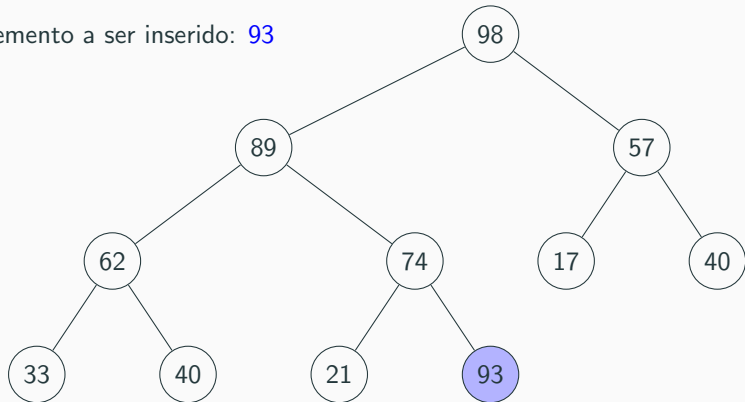
Elemento a ser inserido: 93



1	2	3	4	5	6	7	8	9	10	11
98	89	57	62	74	17	40	33	40	21	

## Exemplo de inserção em uma heap binária

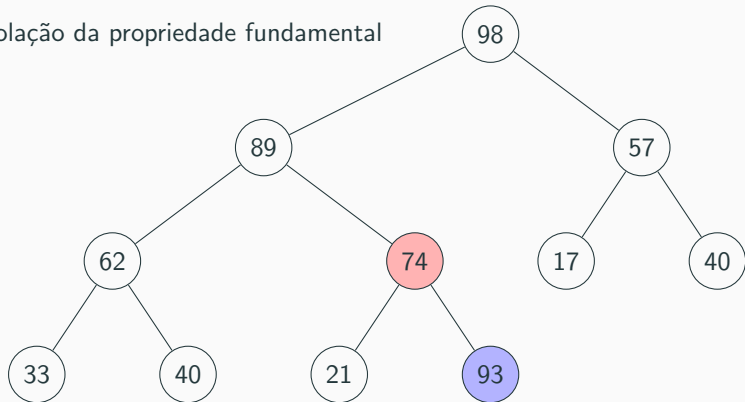
Elemento a ser inserido: 93



1	2	3	4	5	6	7	8	9	10	11
98	89	57	62	74	17	40	33	40	21	93

## Exemplo de inserção em uma heap binária

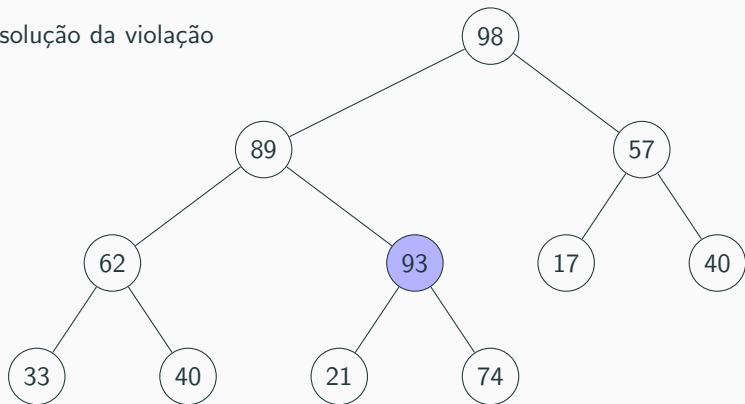
Violação da propriedade fundamental



1	2	3	4	5	6	7	8	9	10	11
98	89	57	62	74	17	40	33	40	21	93

## Exemplo de inserção em uma heap binária

Resolução da violação

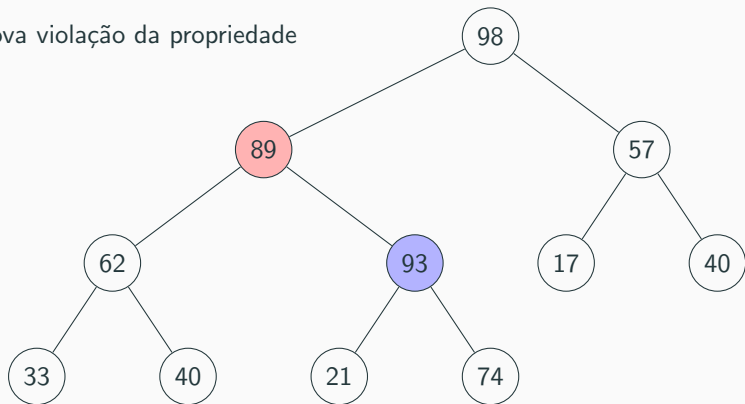


1	2	3	4	5	6	7	8	9	10	11
98	89	57	62	93	17	40	33	40	21	74



## Exemplo de inserção em uma heap binária

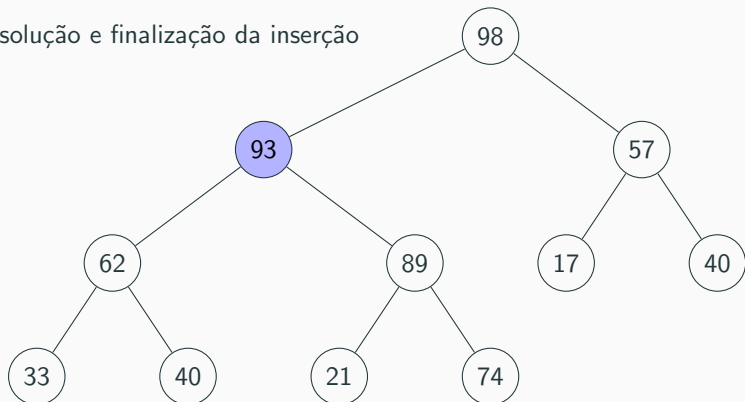
Nova violação da propriedade



1	2	3	4	5	6	7	8	9	10	11
98	89	57	62	93	17	40	33	40	21	74

## Exemplo de inserção em uma heap binária

Resolução e finalização da inserção



1	2	3	4	5	6	7	8	9	10	11
98	93	57	62	89	17	40	33	40	21	74

# Implementação da inserção em heap binária

```
20 void insert(int x)
21 {
22     if (N + 1 == xs.size())
23         xs.push_back(x);
24     else
25         xs[N + 1] = x;
26
27     int i = N + 1, p = parent(i);
28
29     while (p and xs[p] < xs[i])
30     {
31         std::swap(xs[p], xs[i]);
32         i = p;
33         p = parent(i);
34     }
35
36     ++N;
37 }
```

## **Identificação e extração do elemento máximo**

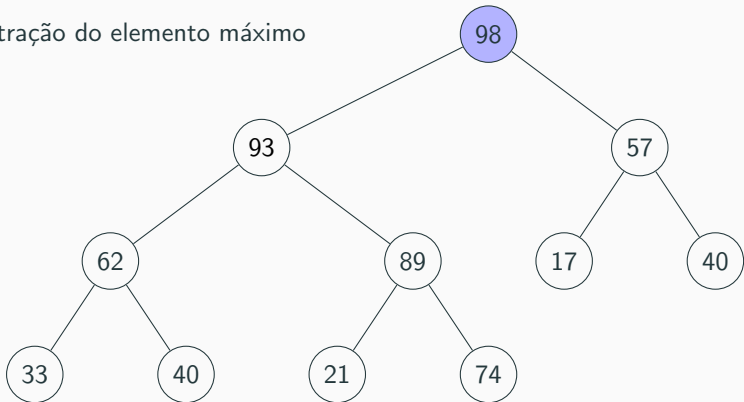
---

# Identificação e extração do elemento máximo

- O elemento máximo de um *heap* está localizado na raiz da árvore
- Portanto a identificação deste elemento pode ser feita em  $O(1)$
- A extração deste elemento é o processo reverso da inserção
- A raiz deve ser substituída pela folha mais à direita do último nível
- Esta substituição pode gerar uma violação da propriedade fundamentação do nó em relação aos seus filhos
- As possíveis violações devem ser resolvidas da raiz para as folhas
- Se a violação ocorrem com ambos filhos, deve-se escolher o que possui a maior informação para prosseguir com as trocas
- Esta extração tem complexidade  $O(\log N)$

## Exemplo de extração em uma heap binária

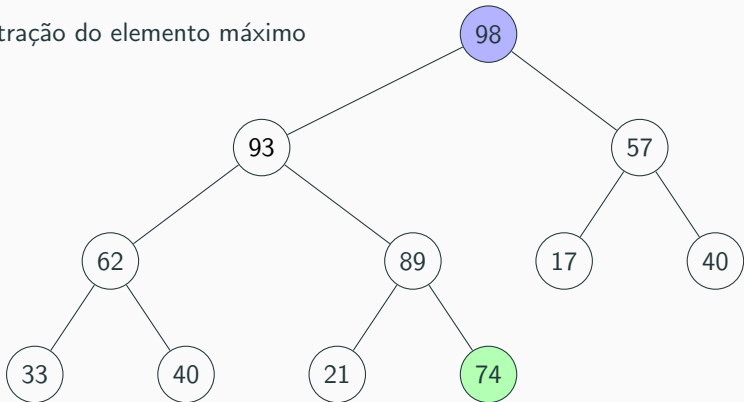
Extração do elemento máximo



1	2	3	4	5	6	7	8	9	10	11
98	93	57	62	89	17	40	33	40	21	74

## Exemplo de extração em uma heap binária

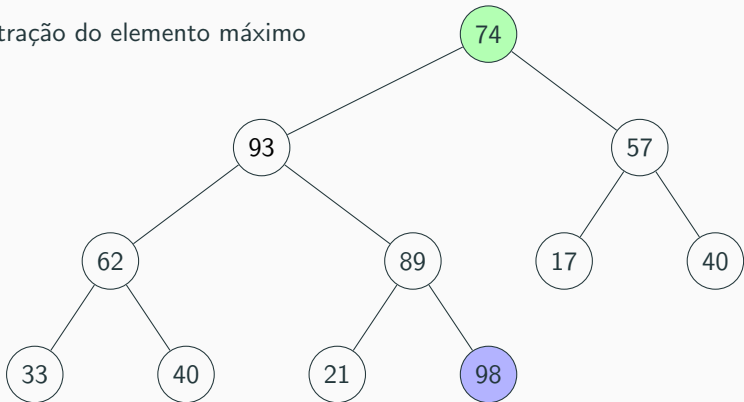
Extração do elemento máximo



1	2	3	4	5	6	7	8	9	10	11
98	93	57	62	89	17	40	33	40	21	74

## Exemplo de extração em uma heap binária

Extração do elemento máximo

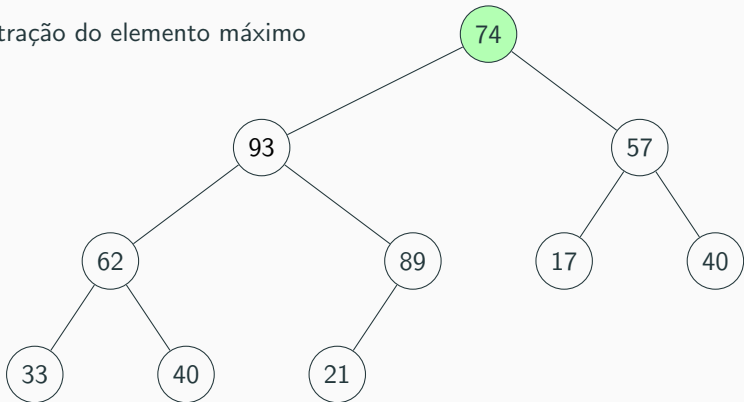


1	2	3	4	5	6	7	8	9	10	11
74	93	57	62	89	17	40	33	40	21	98



## Exemplo de extração em uma heap binária

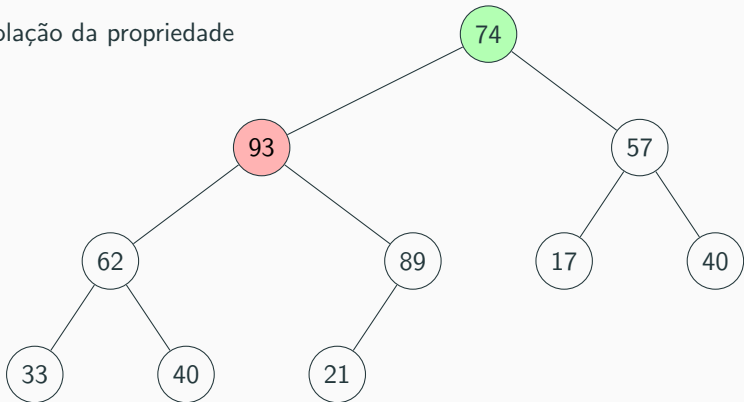
Extração do elemento máximo



1	2	3	4	5	6	7	8	9	10	11
74	93	57	62	89	17	40	33	40	21	

## Exemplo de extração em uma heap binária

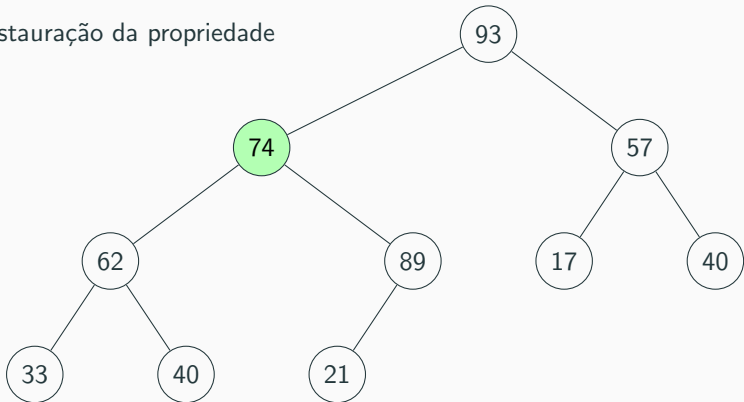
Violação da propriedade



1	2	3	4	5	6	7	8	9	10	11
74	93	57	62	89	17	40	33	40	21	

## Exemplo de extração em uma heap binária

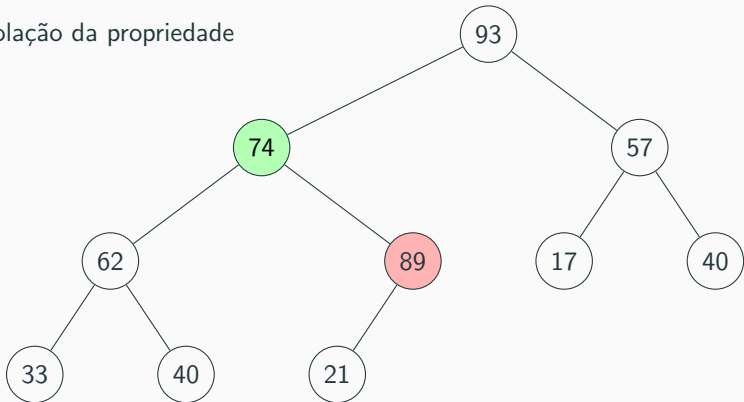
Restauração da propriedade



1	2	3	4	5	6	7	8	9	10	11
93	74	57	62	89	17	40	33	40	21	

## Exemplo de extração em uma heap binária

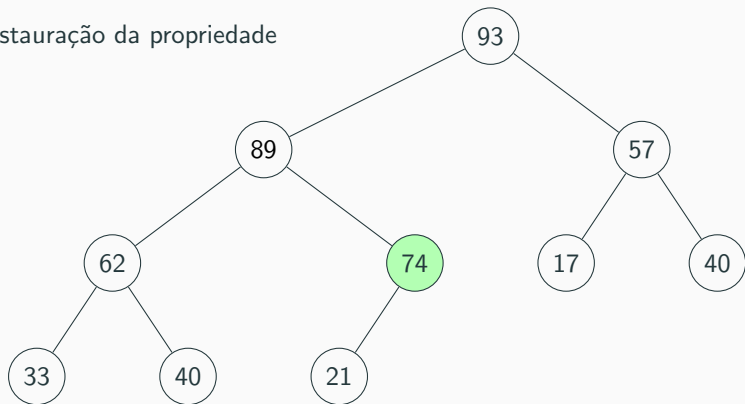
Violação da propriedade



1	2	3	4	5	6	7	8	9	10	11
93	74	57	62	89	17	40	33	40	21	

## Exemplo de extração em uma heap binária

Restauração da propriedade



1	2	3	4	5	6	7	8	9	10	11
93	89	57	62	74	17	40	33	40	21	

# Implementação da extração em heap binária

```
39     bool empty() const { return N == 0; }
40
41     T max() const
42     {
43         if (empty())
44             throw std::out_of_range("Empty heap");
45
46         return xs[1];
47     }
48
49     T extract_max()
50     {
51         if (empty())
52             throw std::out_of_range("Empty heap");
53
54         std::swap(xs[1], xs[N]);
55         auto x = xs[1];
56         --N;
57
58         int i = 1, n = left(i) > N ? 0 : left(i);
59
```

# Implementação da extração em heap binária

```
60     while (n)
61     {
62         auto r = right(i) > N ? 0 : right(i);
63
64         if (r and xs[r] > xs[n])
65             n = r;
66
67         if (xs[i] < xs[n])
68         {
69             std::swap(xs[i], xs[n]);
70             i = n;
71             n = left(i) > N ? 0 : left(i);
72         } else
73             n = 0;
74     }
75
76     return x;
77 }
```

## Construção de uma heap binária em $O(N)$

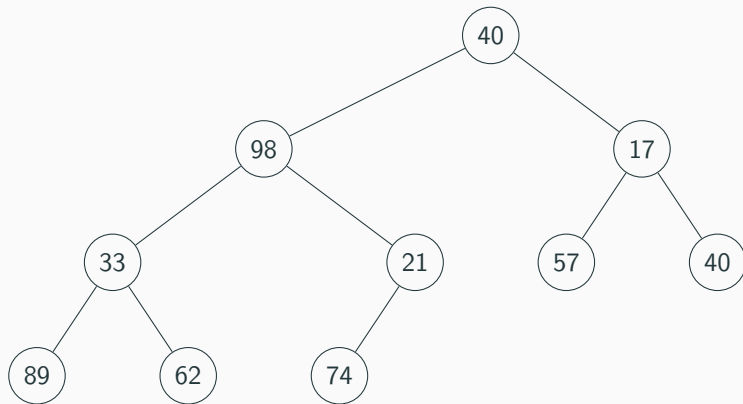
---



# Heapify

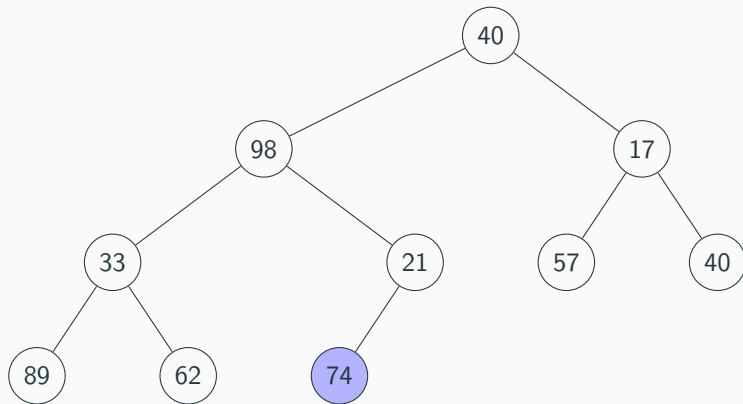
- Dado um vetor de elementos  $xs$ , uma *heap* binária pode ser construída em  $O(N \log N)$  através de  $N$  inserções
- Porém é possível construir a mesma *heap* em  $O(N)$
- Esta rotina, denominada *heapify*, foi proposta por Floyd
- Ela consiste em preencher uma árvore binária com os elementos de  $xs$ , na ordem em que foram informados
- Em seguida, em ordem reversa (da última folha para a raiz), as violações da propriedade fundamental devem ser corrigidas, utilizando a mesma estratégia da extração do elemento máximo

## Exemplo de execução da rotina heapify()



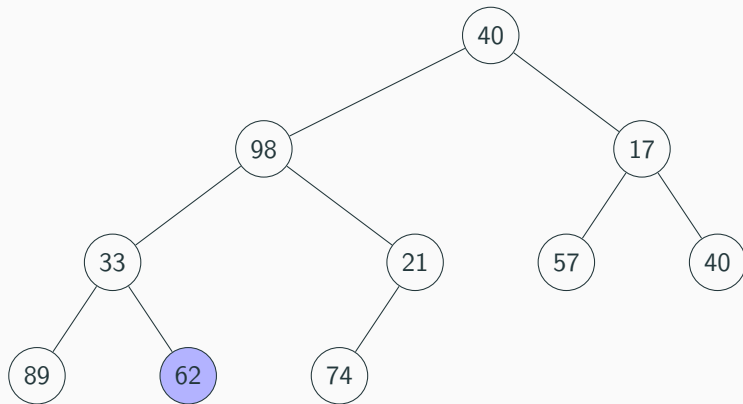
1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

## Exemplo de execução da rotina heapify()



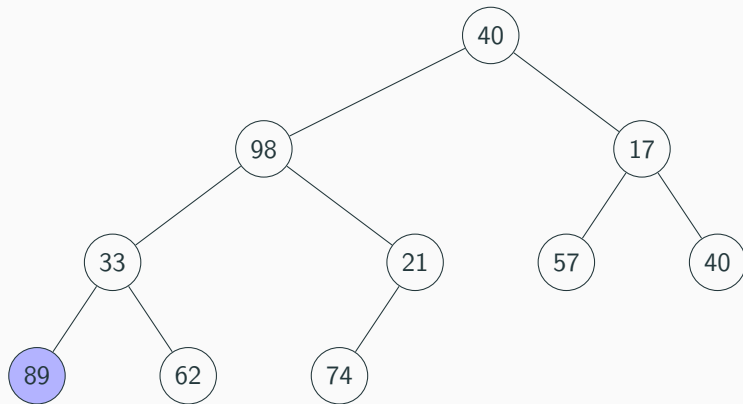
1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

## Exemplo de execução da rotina heapify()



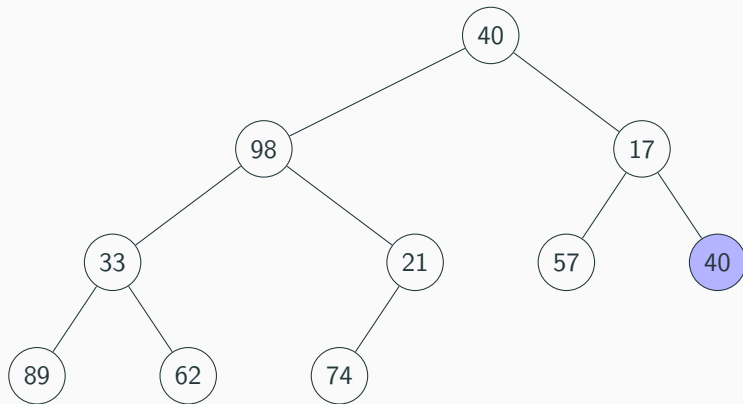
1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

## Exemplo de execução da rotina heapify()



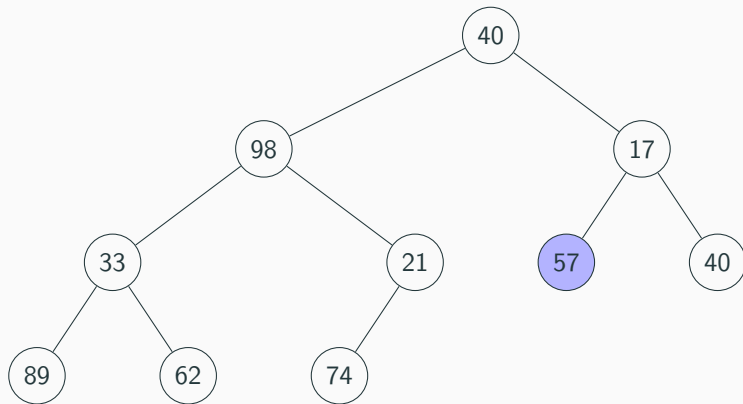
1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

## Exemplo de execução da rotina heapify()



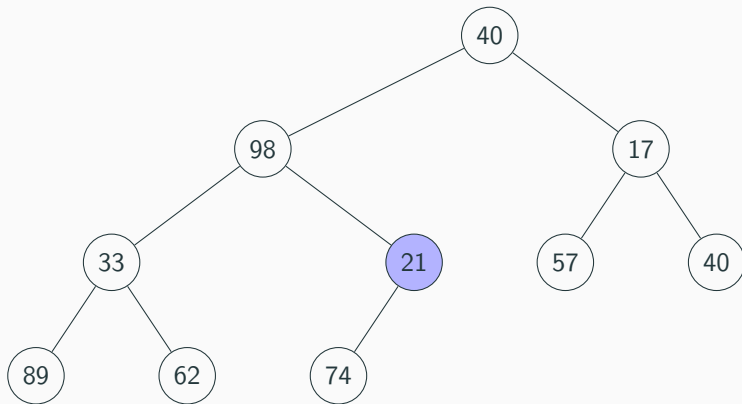
1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

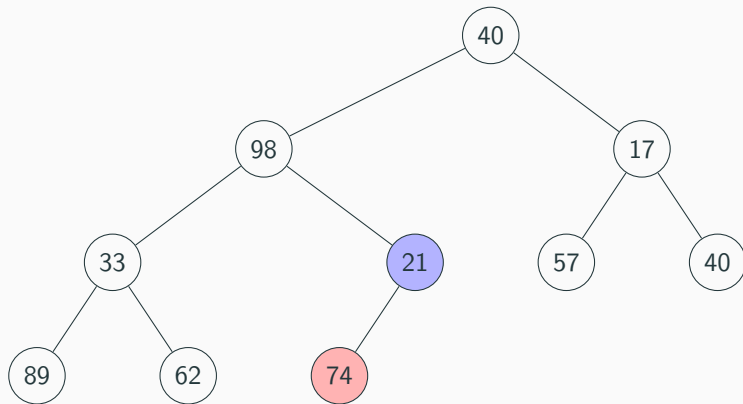
## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

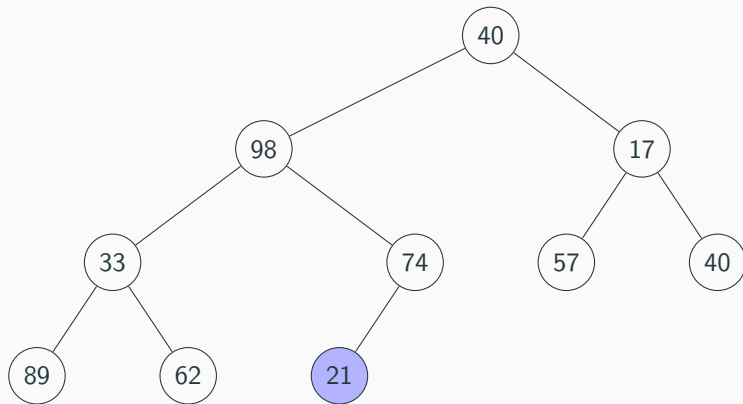


## Exemplo de execução da rotina heapify()



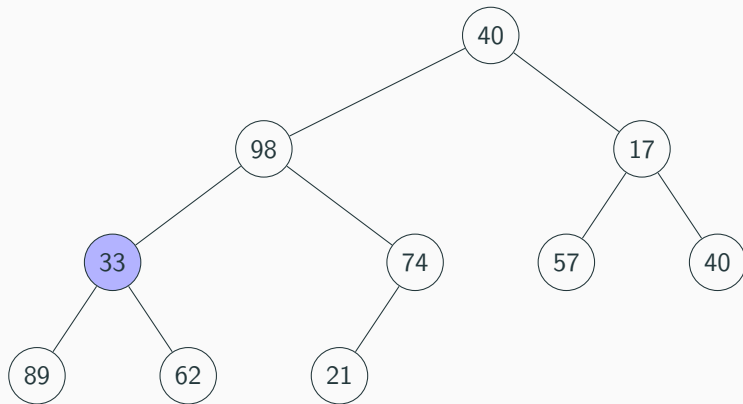
1	2	3	4	5	6	7	8	9	10
40	98	17	33	21	57	40	89	62	74

## Exemplo de execução da rotina heapify()



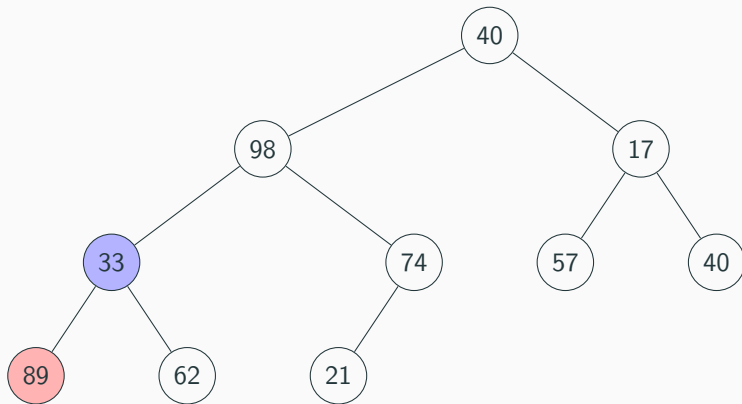
1	2	3	4	5	6	7	8	9	10
40	98	17	33	74	57	40	89	62	21

## Exemplo de execução da rotina heapify()



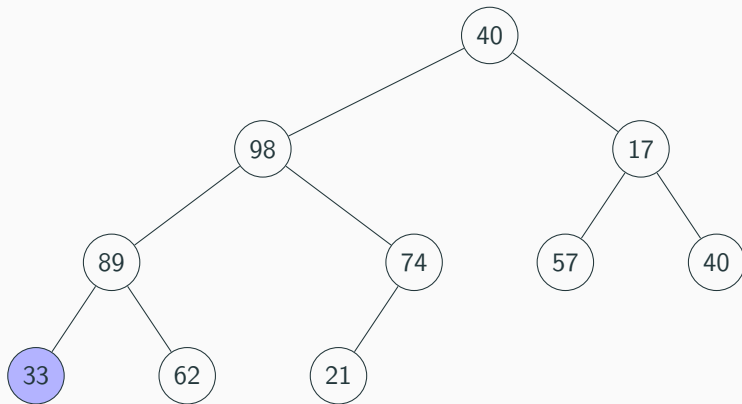
1	2	3	4	5	6	7	8	9	10
40	98	17	33	74	57	40	89	62	21

## Exemplo de execução da rotina heapify()



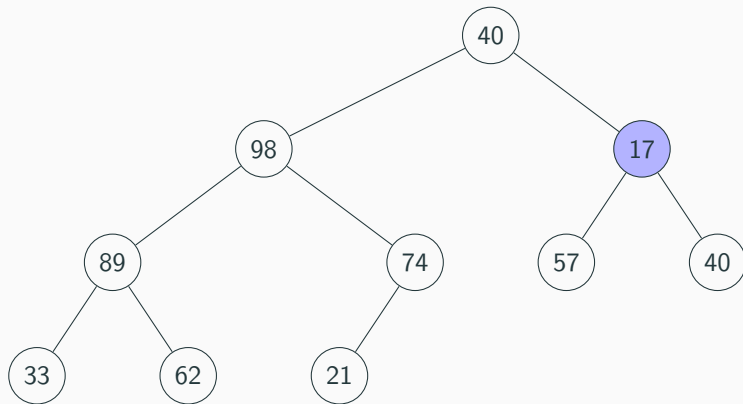
1	2	3	4	5	6	7	8	9	10
40	98	17	33	74	57	40	89	62	21

## Exemplo de execução da rotina heapify()



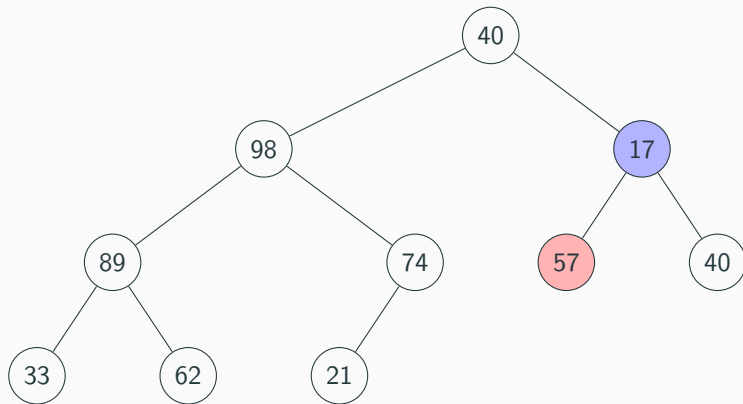
1	2	3	4	5	6	7	8	9	10
40	98	17	89	74	57	40	33	62	21

## Exemplo de execução da rotina heapify()



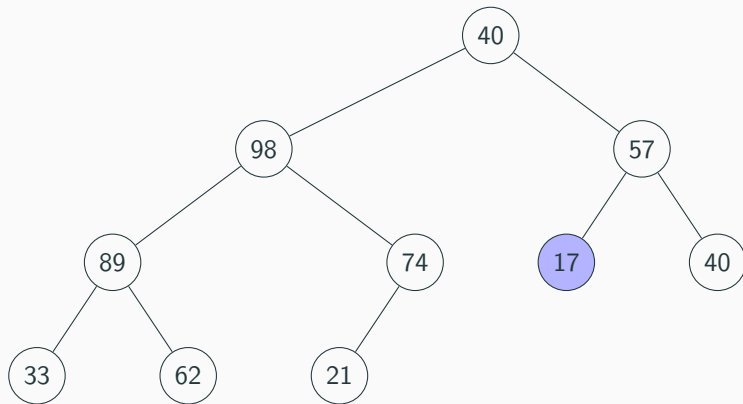
1	2	3	4	5	6	7	8	9	10
40	98	17	89	74	57	40	33	62	21

## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
40	98	17	89	74	57	40	33	62	21

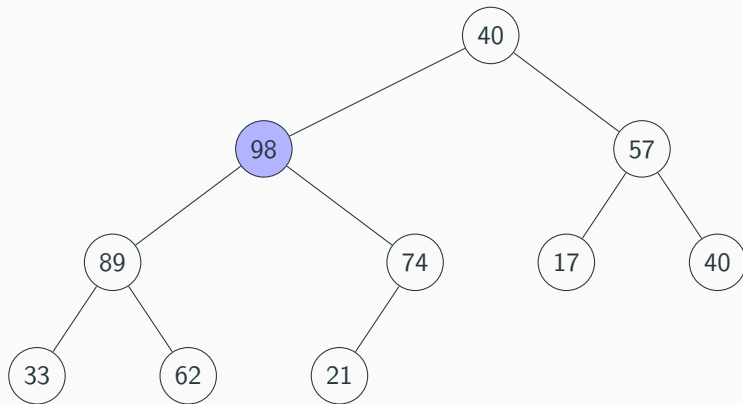
## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
40	98	57	89	74	17	40	33	62	21

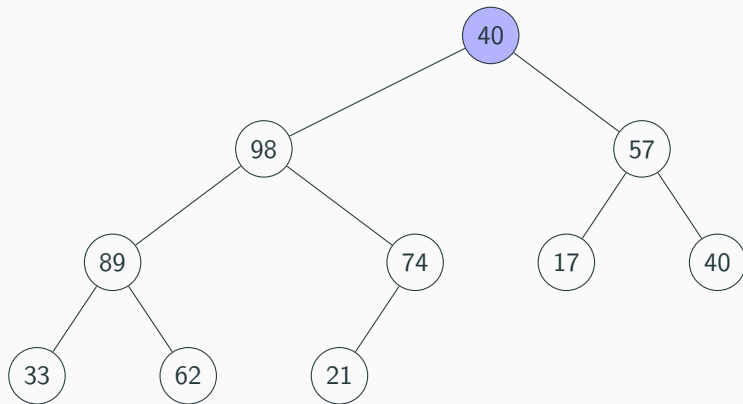


## Exemplo de execução da rotina heapify()



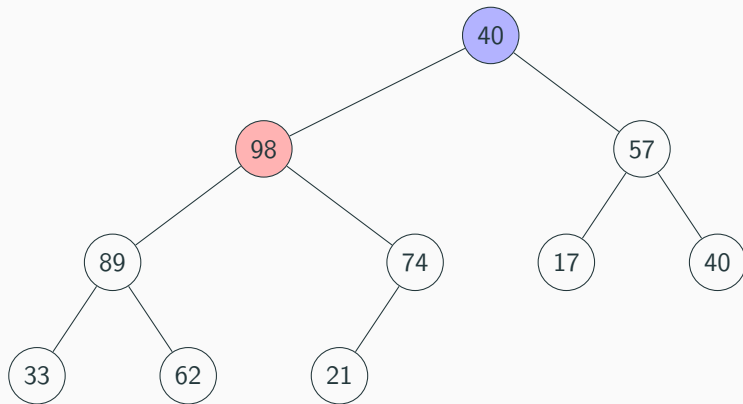
1	2	3	4	5	6	7	8	9	10
40	98	57	89	74	17	40	33	62	21

## Exemplo de execução da rotina heapify()



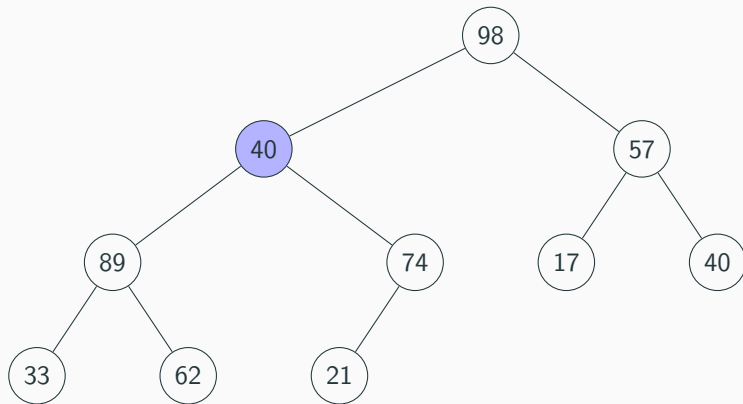
1	2	3	4	5	6	7	8	9	10
40	98	57	89	74	17	40	33	62	21

## Exemplo de execução da rotina heapify()



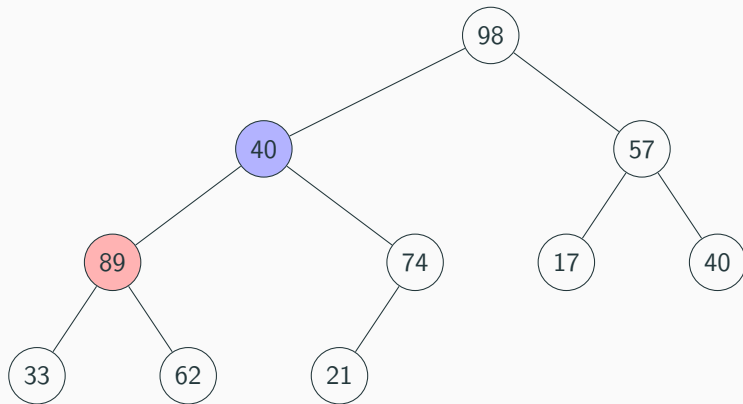
1	2	3	4	5	6	7	8	9	10
40	98	57	89	74	17	40	33	62	21

## Exemplo de execução da rotina heapify()



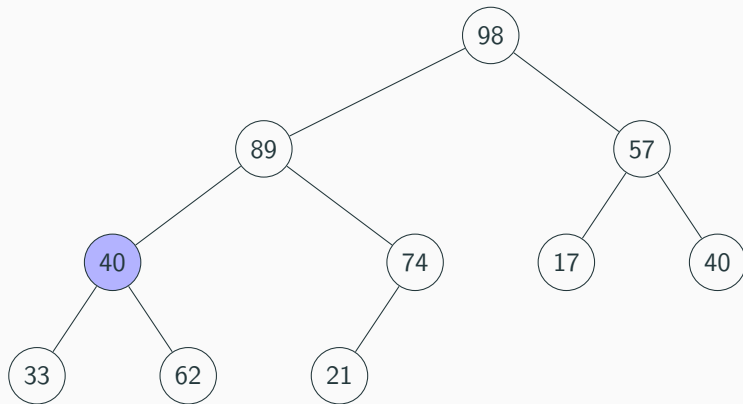
1	2	3	4	5	6	7	8	9	10
98	40	57	89	74	17	40	33	62	21

## Exemplo de execução da rotina heapify()



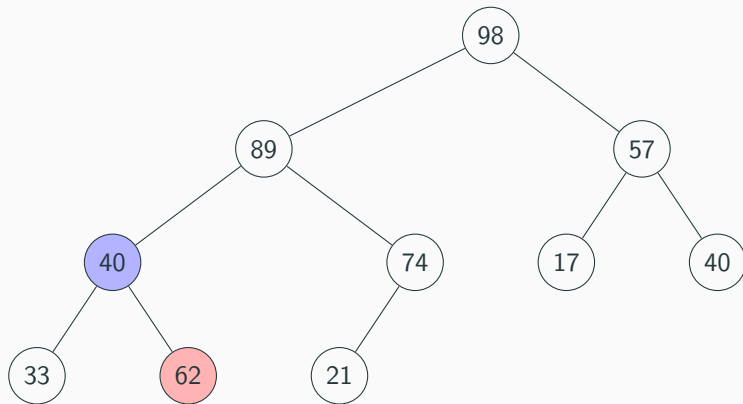
1	2	3	4	5	6	7	8	9	10
98	40	57	89	74	17	40	33	62	21

## Exemplo de execução da rotina heapify()



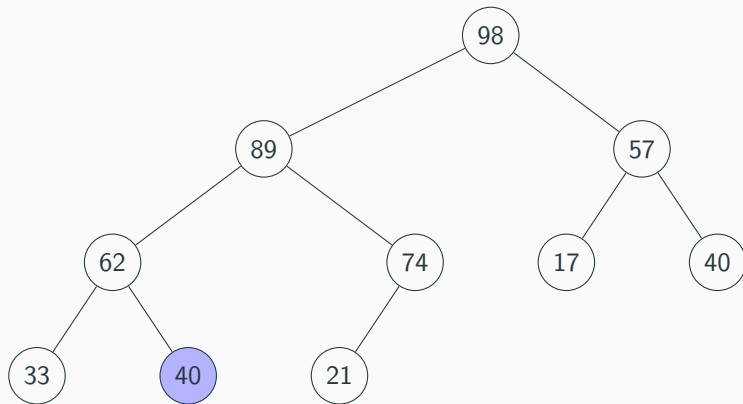
1	2	3	4	5	6	7	8	9	10
98	89	57	40	74	17	40	33	62	21

## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
98	89	57	40	74	17	40	33	62	21

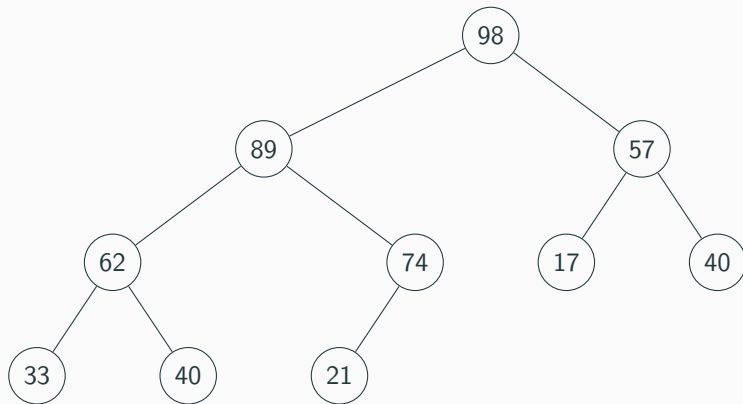
## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
98	89	57	62	74	17	40	33	40	21



## Exemplo de execução da rotina heapify()



1	2	3	4	5	6	7	8	9	10
98	89	57	62	74	17	40	33	40	21

## Implementação da rotina heapify()

```
79  static Heap heapify(const std::vector<T>& xs)
80  {
81      Heap<T> h;
82
83      h.xs.push_back(0);
84      h.xs.insert(h.xs.begin() + 1, xs.begin(), xs.end());
85
86      h.N = xs.size();
87
88      for (int i = h.N; i >= 1; --i)
89      {
90          auto p = i;
91          auto n = h.left(p) > h.N ? 0 : h.left(p);
92
93          while (n)
94          {
95              auto r = h.right(p) > h.N ? 0 : h.right(p);
96
97              if (r and h.xs[r] > h.xs[n])
98                  n = r;
99  }
```

## Implementação da rotina heapify()

```
100         if (h.xs[p] < h.xs[n])
101         {
102             std::swap(h.xs[p], h.xs[n]);
103             p = n;
104             n = h.left(p) > h.N ? 0 : h.left(p);
105         } else
106             n = 0;
107     }
108 }
109
110 return h;
111 }
112
```

## Complexidade da rotina `heapify()`

- Seja  $h = \lfloor \log N \rfloor$  a altura da *heap* binária
- O número de nós no nível  $h - i$  é tal que

$$h - i \leq \frac{2^{\lfloor \log N \rfloor}}{2^{h-i}} \leq \frac{n}{2^{h-i}}$$

- O número de trocas máximas para cada nó é  $O(h)$
- Assim, o total  $T$  de trocas feitas será dado por

$$\begin{aligned} T &= \sum_{h-i=0}^{\lfloor \log N \rfloor} \frac{n}{2^{h-i}} O(h-i) = \sum_{k=0}^{\lfloor \log N \rfloor} \frac{n}{2^k} O(k) \\ &= O\left(n \sum_{k=0}^{\lfloor \log N \rfloor} \frac{k}{2^k}\right) \leq O\left(n \sum_{k=0}^{\infty} \frac{k}{2^k}\right) = O(n), \end{aligned}$$

pois a série infinita  $a_k = \sum k/2^k$  converge

1. **ROUGHGARDEN**, Tim. *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures*, LLC, 2018.
2. Visualgo. [Binary Heap](#), acesso em 22/04/2019.
3. Wikipedia. [Building a heap](#), acesso em 22/04/2019.