

# Ordenação

Prof. Paulo Henrique Pisani

abril/2022

# Tópicos

- Heap
- Heap sort

# Heap

# Heap (binário)

- É uma estrutura de dados que representa uma árvore binária completa, em que apenas o último nível pode não estar completo. As folhas também estão sempre nas posições mais à esquerda;
- O heap deve atender à **propriedade do heap**:
  - **Heap de máximo**: para toda chave  $v$ , o pai possui uma chave com valor igual ou maior;
  - **Heap de mínimo**: para toda chave  $v$ , o pai possui uma chave com valor igual ou menor;

# Heap (binário)

- É uma estrutura de dados que representa uma árvore binária completa, em que apenas o último nível pode não estar completo. As folhas também estão sempre nas posições mais à esquerda;
- O heap deve atender à **propriedade do heap**:

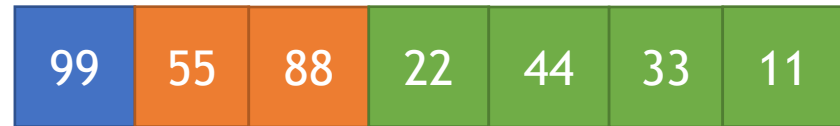
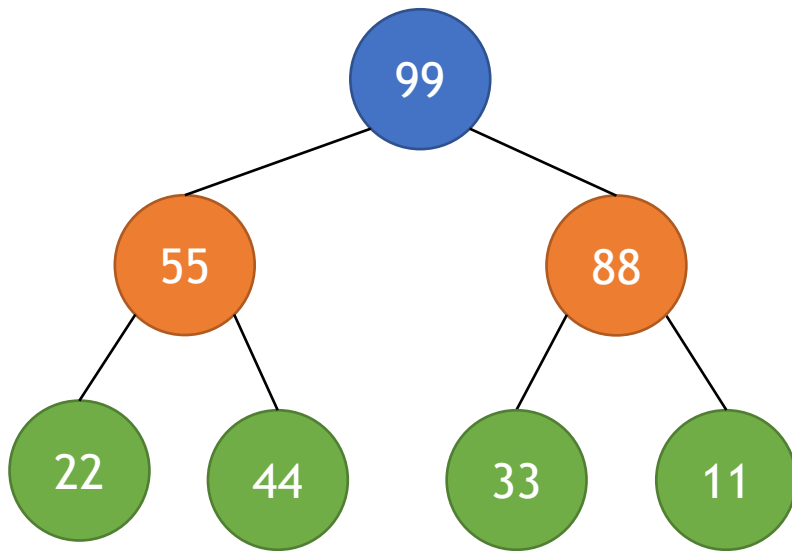


- **Heap de máximo**: para toda chave  $v$ , o pai possui uma chave com valor igual ou maior;
- **Heap de mínimo**: para toda chave  $v$ , o pai possui uma chave com valor igual ou menor;

Um heap de máximo será usado no algoritmo de ordenação heap sort.

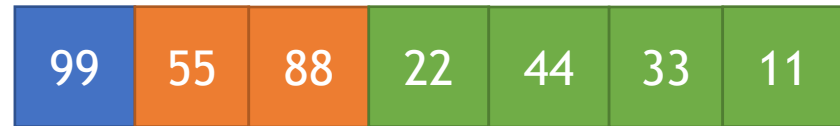
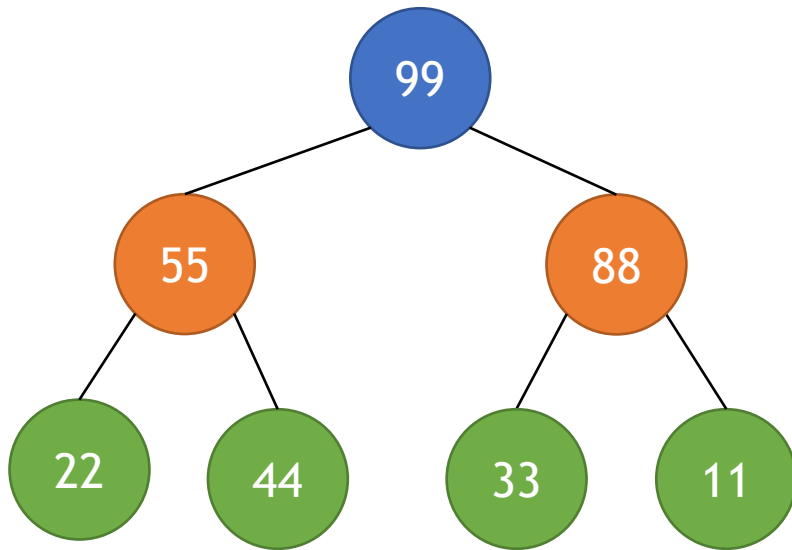
# Heap

- Armazenamento em vetor:



# Heap

- Armazenamento em vetor:



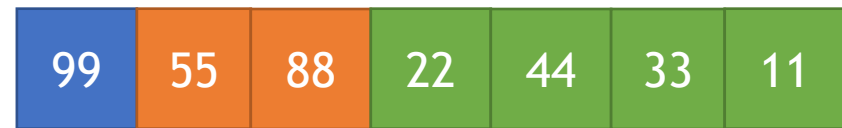
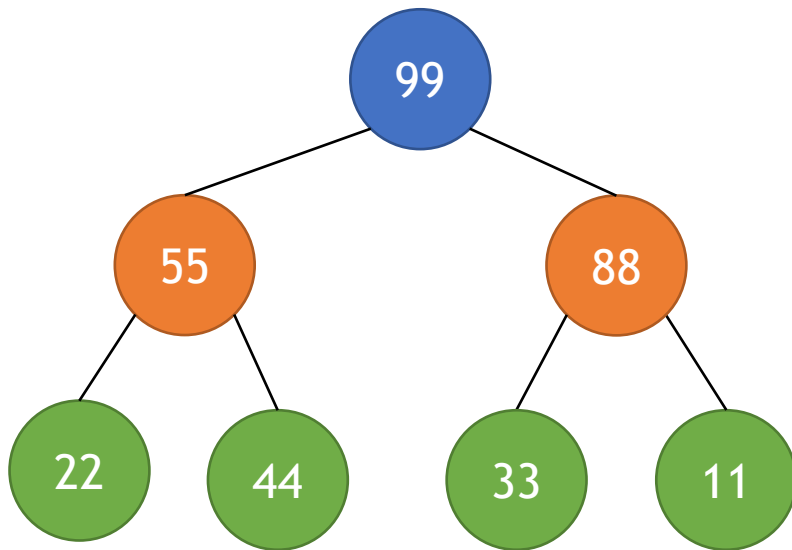
$$i_{esq} = 2.i + 1$$

$$i_{dir} = 2.i + 2$$

$$i_{pai} = \lfloor (i - 1) / 2 \rfloor$$

# Heap

- Armazenamento em vetor:



$$i_{esq} = 2.i + 1$$

$$i_{dir} = 2.i + 2$$

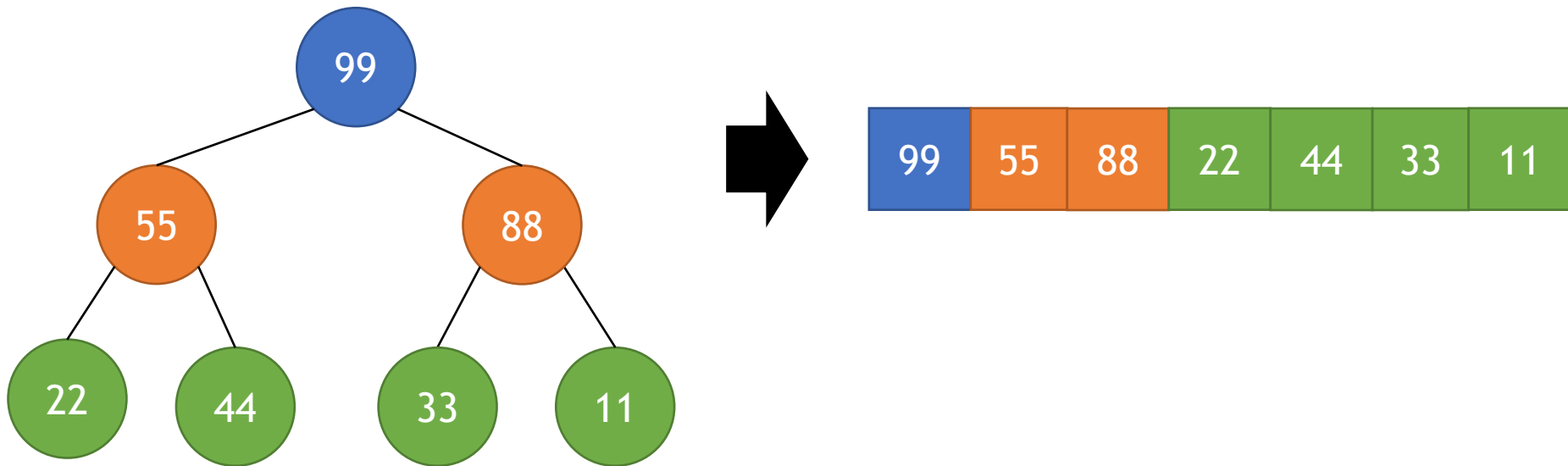
$$i_{pai} = \lfloor (i - 1) / 2 \rfloor$$

Nós folha ficam nos índices  $\lfloor n/2 \rfloor$  até  $n - 1$ , em que  $n$  é a quantidade de elementos no heap.



# Heap

- Armazenamento em vetor:

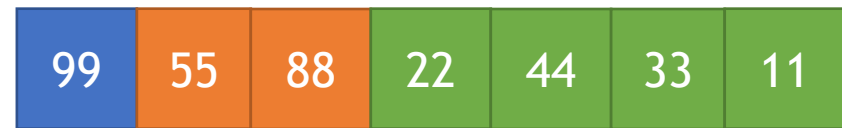
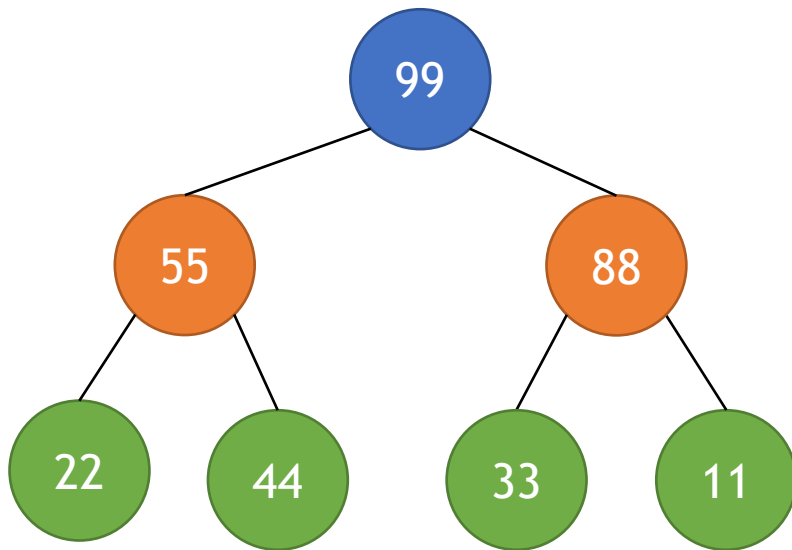


$h = O(\lg(n))$ , em que  $n$  é a quantidade de elementos no heap.

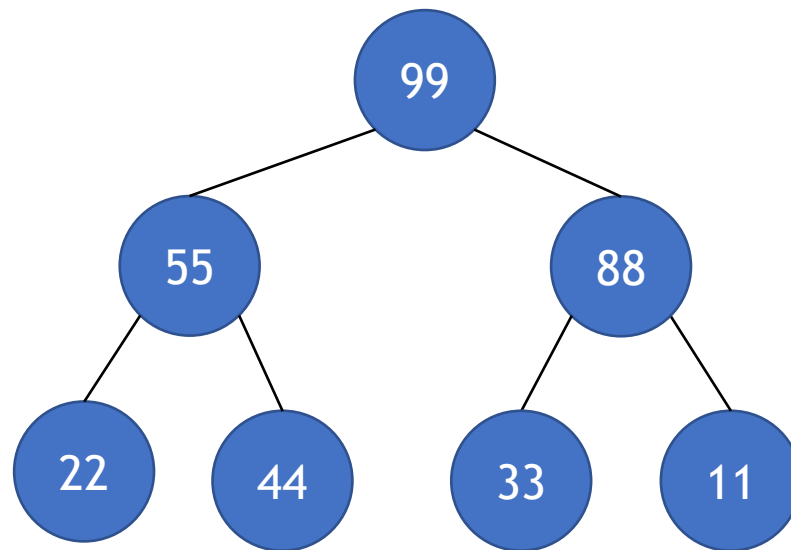
# Heap

Maior elemento fica na raiz,  
primeiro elemento do vetor.

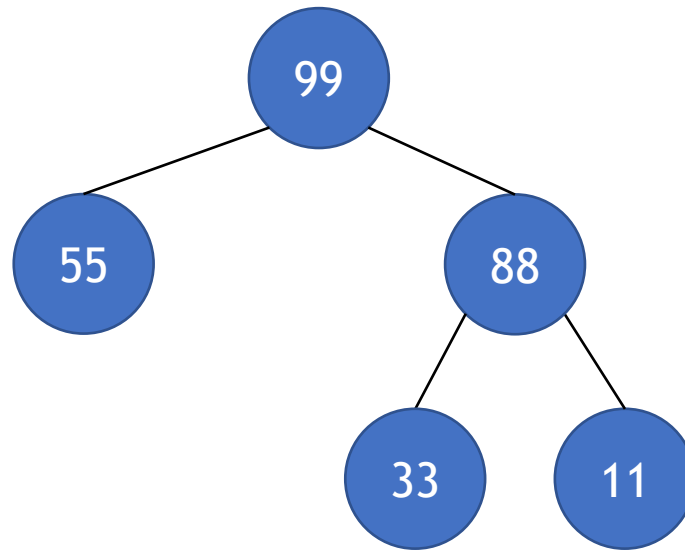
- Armazenamento em vetor:



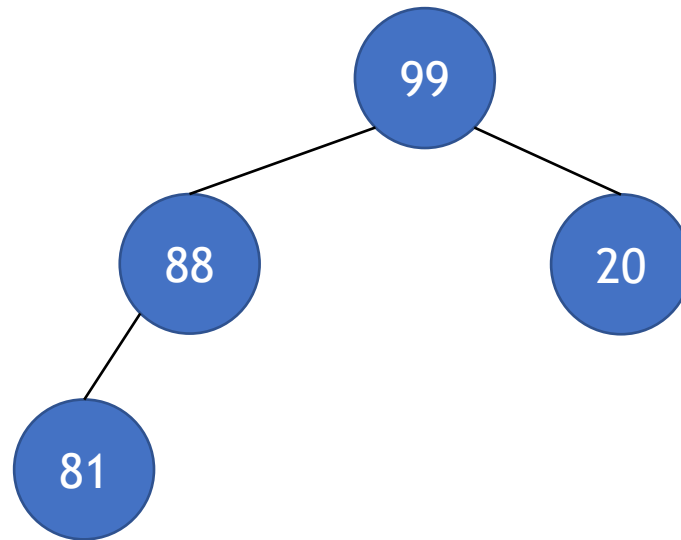
# É heap de máximo?



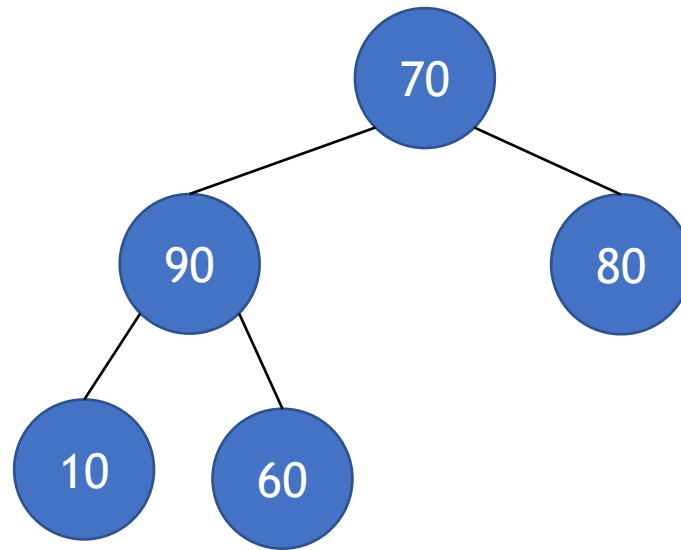
# É heap de máximo?



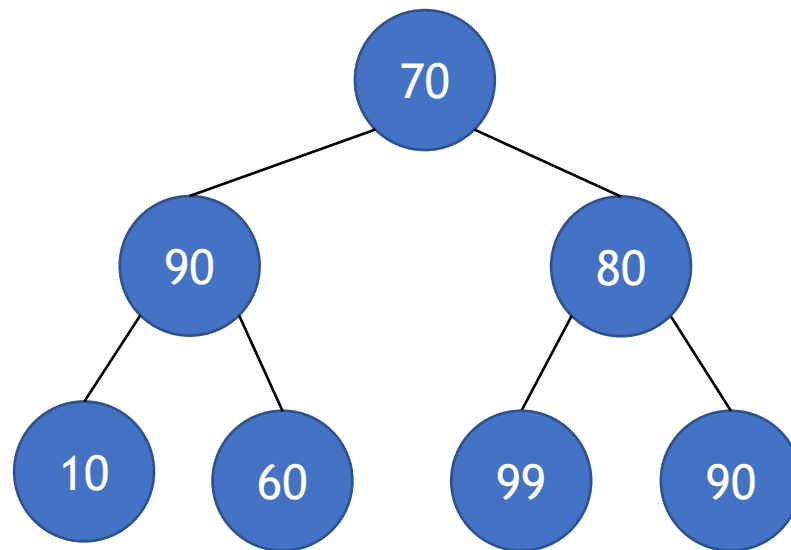
# É heap de máximo?



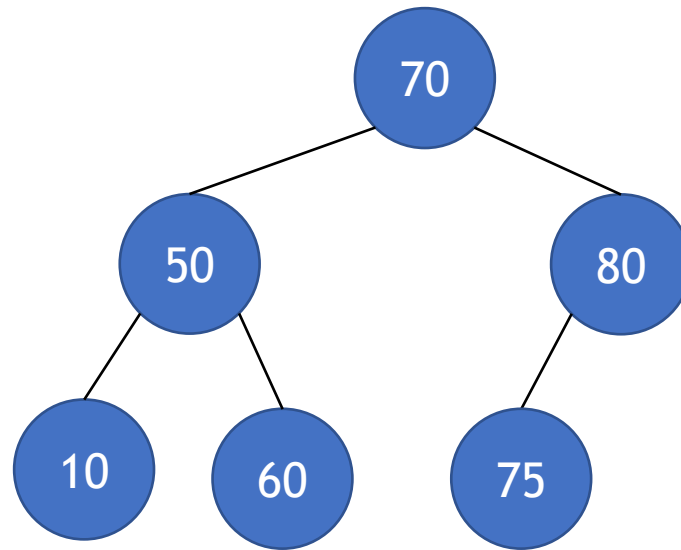
# É heap de máximo?



# É heap de máximo?



# É heap de máximo?





# Manutenção da propriedade de heap

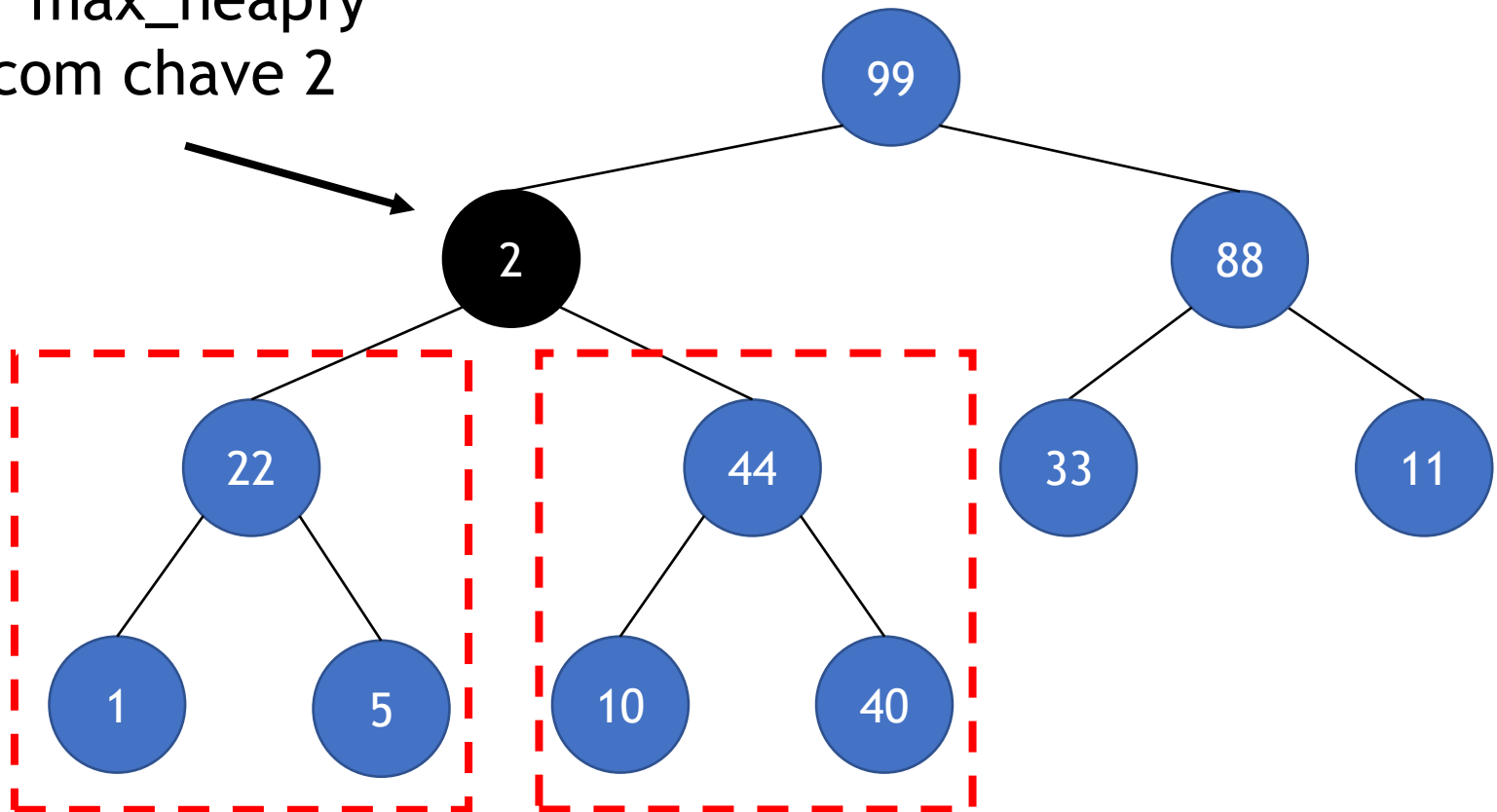
- Veremos algumas funções para utilizar um heap de máximo e manter a propriedade:
  - `max_heapfy`
  - `build_max_heap`
- Essas funções são suficientes para a implementação do heap sort;
- Uma aplicação comum de heaps é para implementação de fila de prioridade (nesse caso, funções adicionais são necessárias).

# Função: max\_heapfy

- Essa função é aplicada sobre um nó **x** e utilizada para manter a propriedade de heap de máximo quando:
  - As duas subárvores (esquerda e direita) são heaps de máximo;
  - Mas **x** pode violar a propriedade.
- Ao aplicar a função em **x**, a chave desse nó será deslocada na árvore até que a propriedade do heap seja reestabelecida.

# Função: max\_heapfy

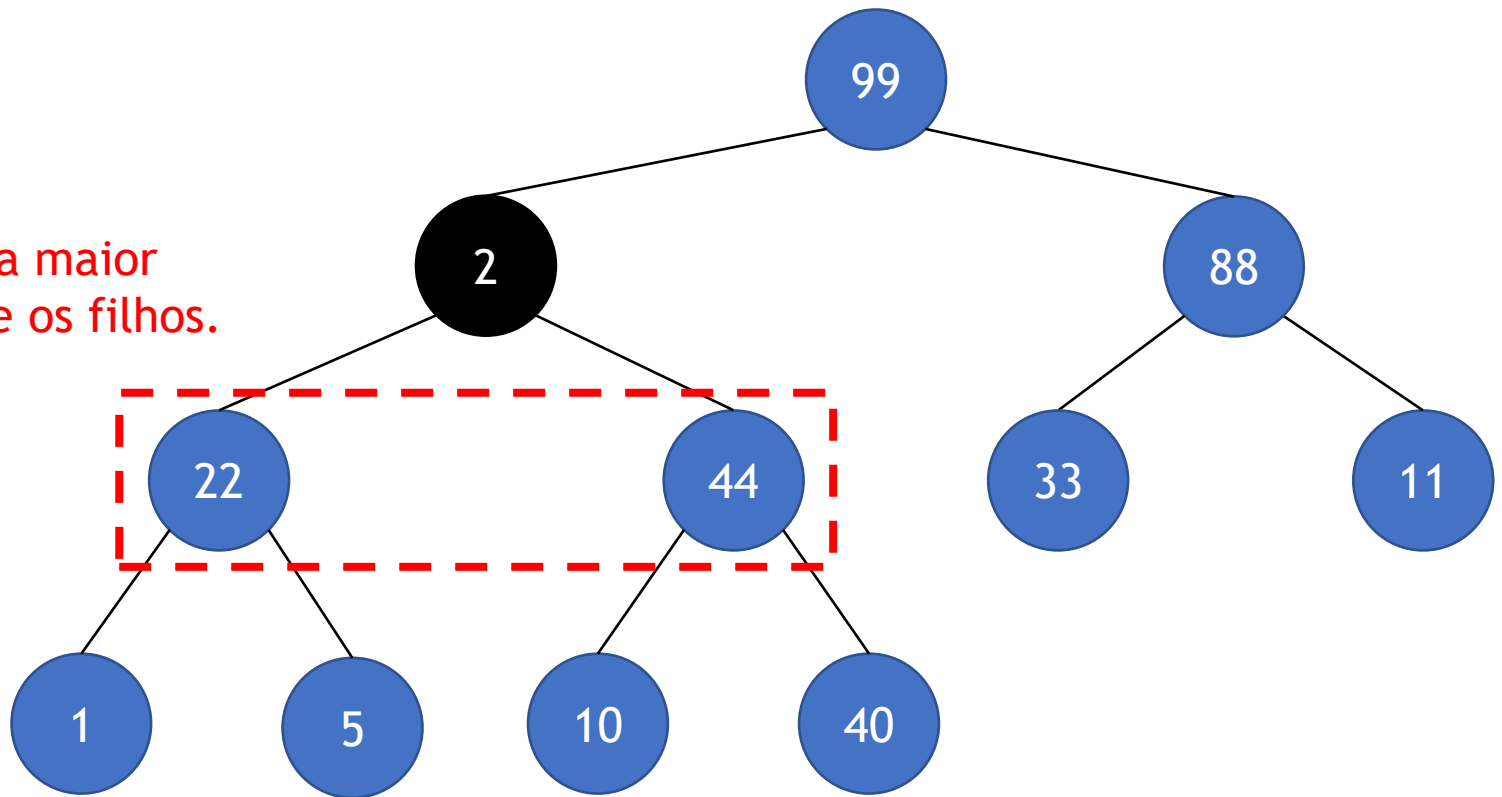
Aplicar max\_heapfy  
no nó com chave 2



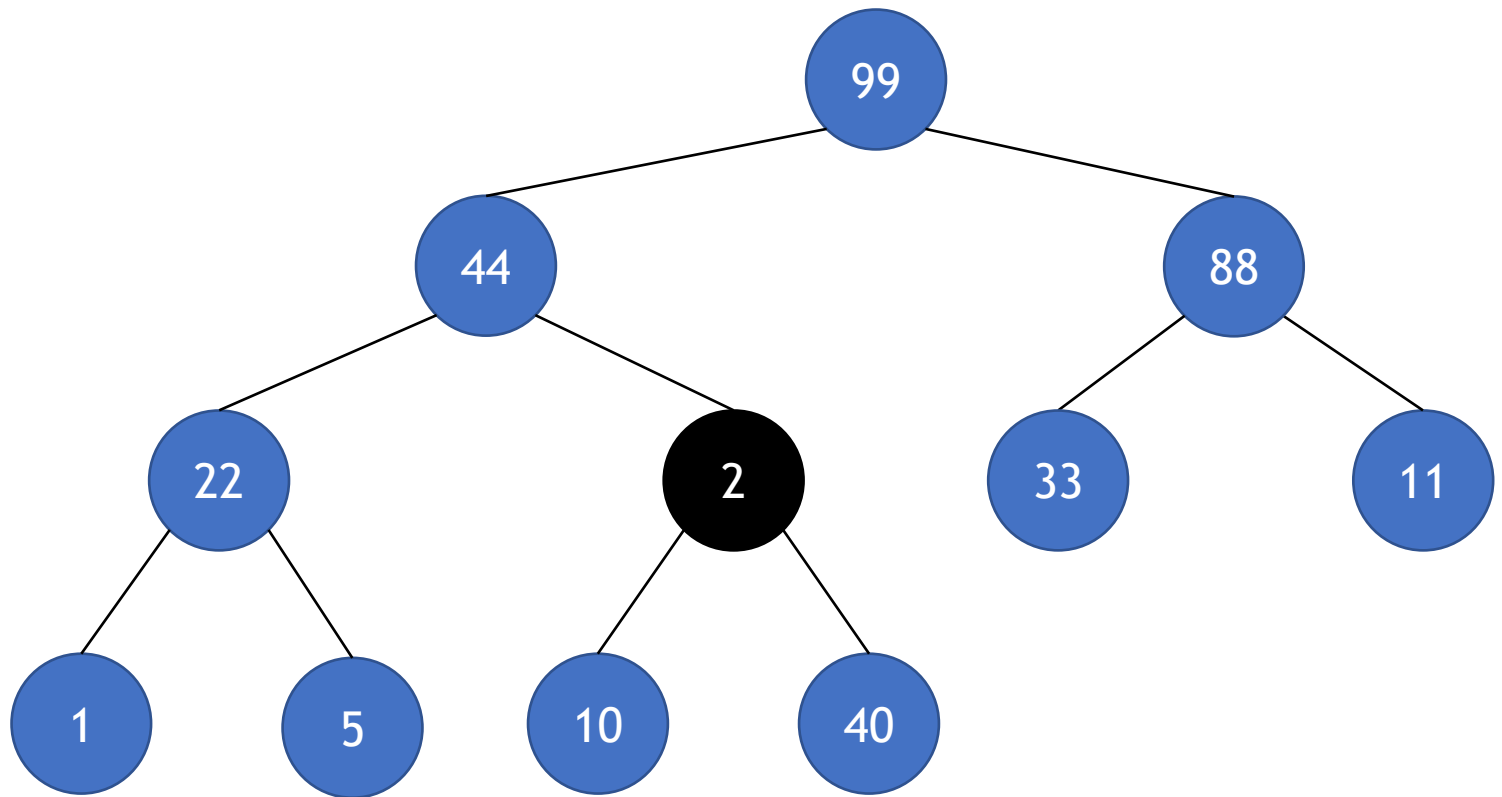
As duas subárvores (esquerda e direita) são heaps de máximo e, portanto, podemos aplicar max\_heapfy.

# Função: max\_heapfy

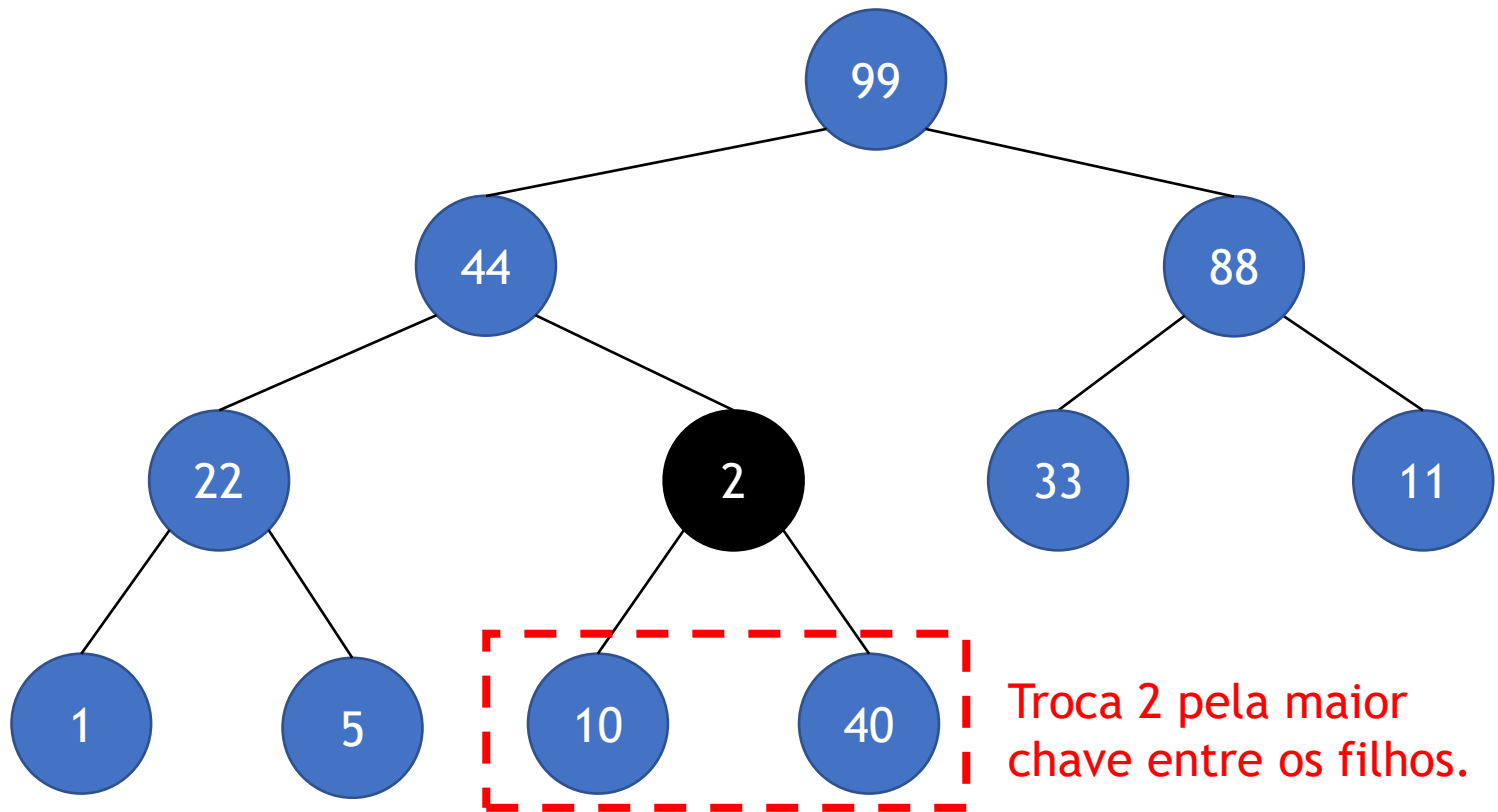
Troca 2 pela maior  
chave entre os filhos.



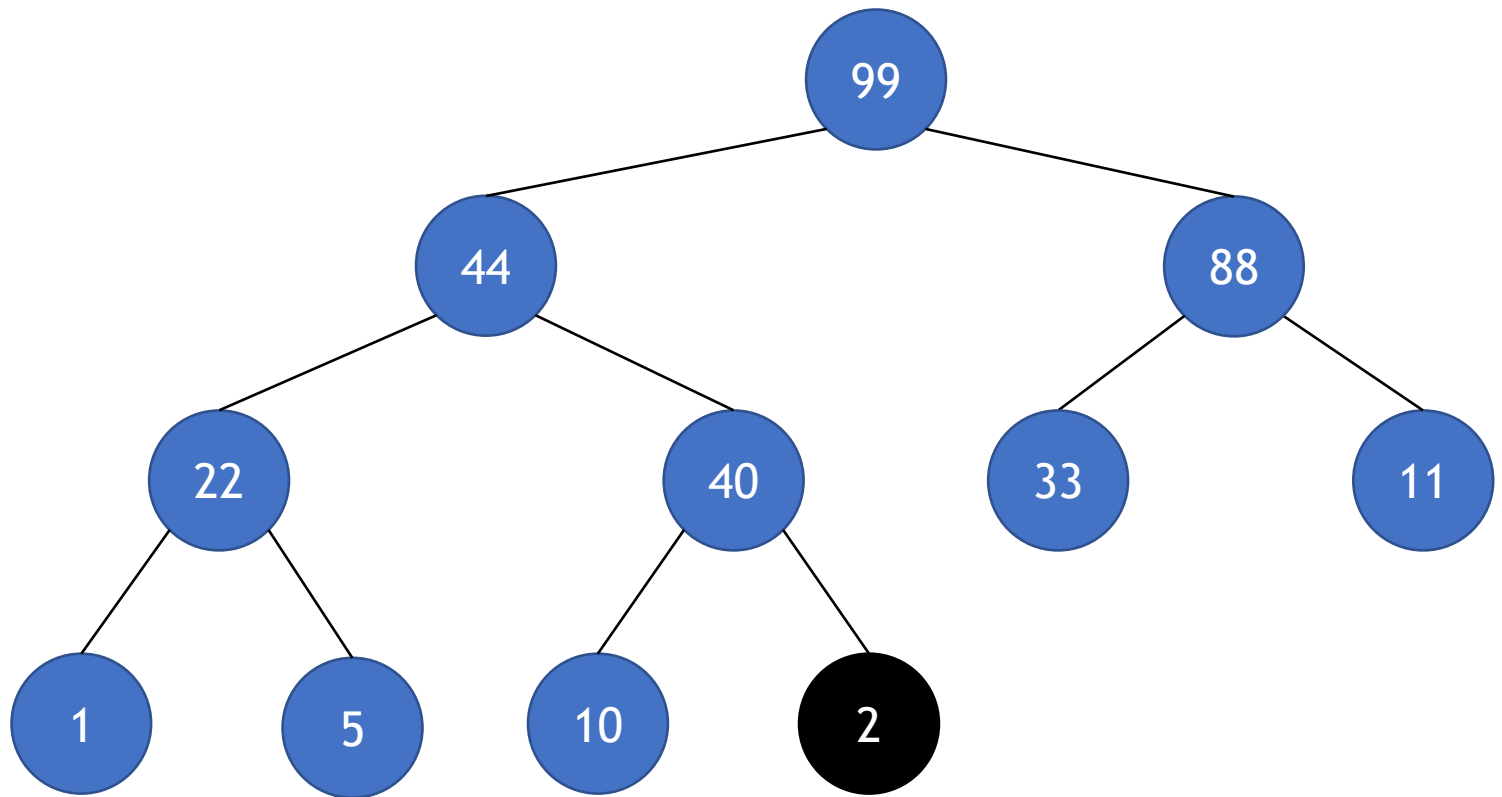
# Função: max\_heapfy



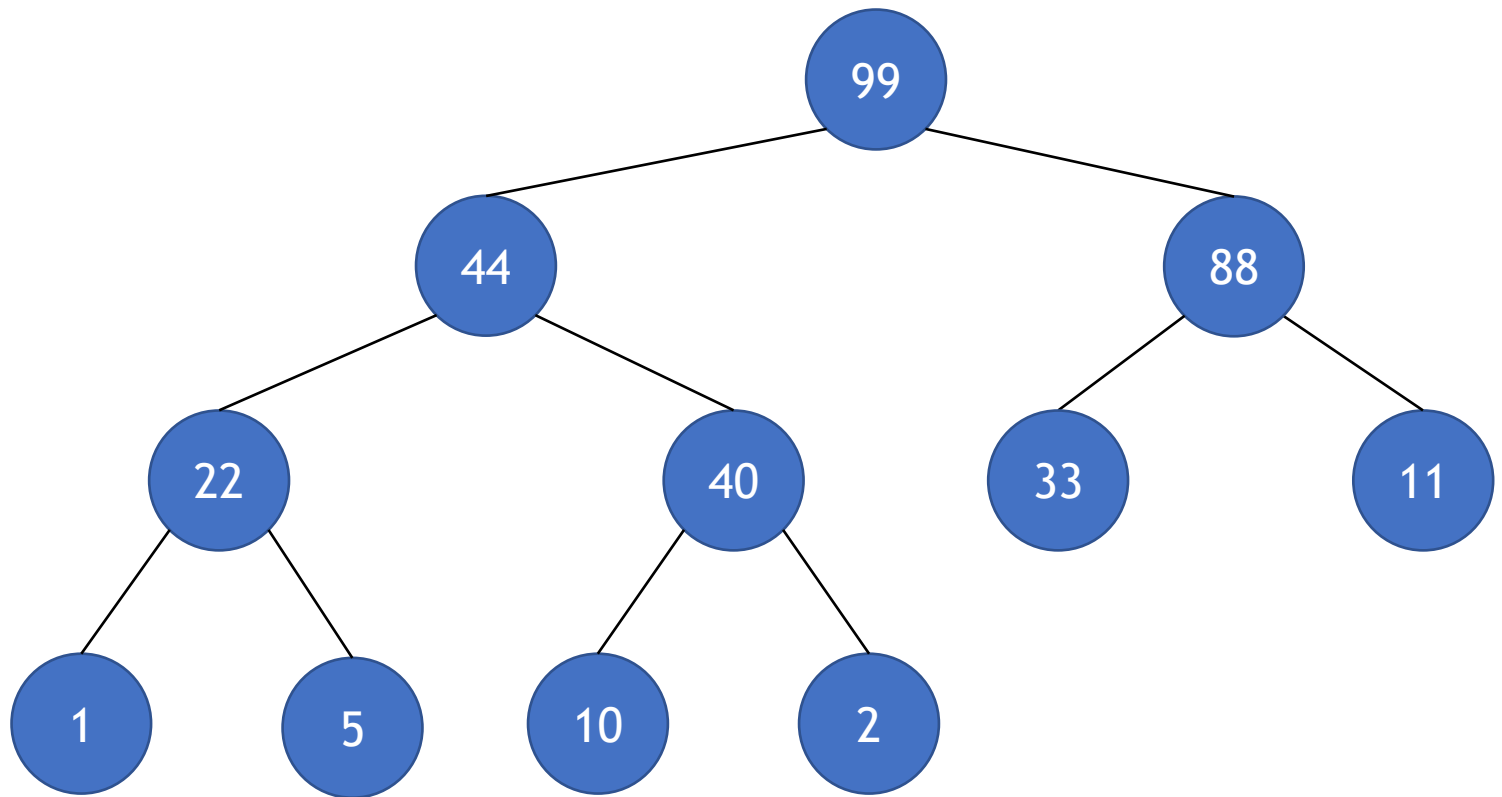
# Função: max\_heapfy



# Função: max\_heapfy



# Função: max\_heapfy



Procedimento encerrado.

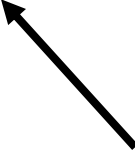


# Função: max\_heapfy

- Implementação em C:

Chamada:

```
max_heapfy(vetor, n, i);
```



*i* é o índice do elemento *x* (ponto onde o procedimento será iniciado).

# Função: max\_heapfy

```
void max_heapfy(int *v, int n, int i) {  
    int esq = 2*i + 1;  
    int dir = 2*i + 2;  
    int maior = i;  
    if (esq < n && v[esq] > v[maior])  
        maior = esq;  
    if (dir < n && v[dir] > v[maior])  
        maior = dir;  
    if (maior != i) {  
        int aux = v[i];  
        v[i] = v[maior];  
        v[maior] = aux;  
        max_heapfy(v, n, maior);  
    }  
}
```

Encontra maior chave entre o nó no índice  $i$  e seus dois filhos.

Se necessário, realiza a troca com o maior filho e depois chama a função recursivamente para o nó alterado.

Custo =  $O(\lg(n))$

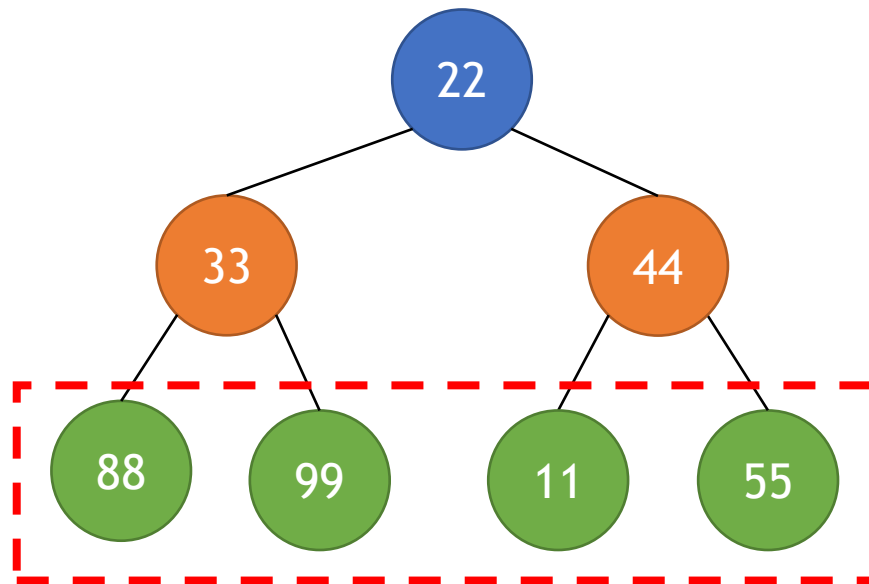
# Função: build\_max\_heap

- Essa função é aplicada sobre um vetor e o transforma em um heap de máximo;
- Para isso, percorre cada um dos nós não-folha e aplica max\_heapfy.

# Função: build\_max\_heap

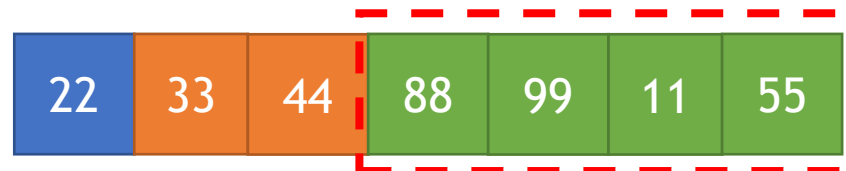
- Essa função é aplicada sobre um vetor e o transforma em um heap de máximo;
- Para isso, percorre cada um dos nós não-folha e aplica max\_heapfy.
  - Nós folha ficam nos índices  $\lfloor n/2 \rfloor$  até  $n - 1$ , em que  $n$  é a quantidade de elementos no heap.
  - Nós não-folha ficam nos índices menores que  $\lfloor n/2 \rfloor$

# Função: build\_max\_heap

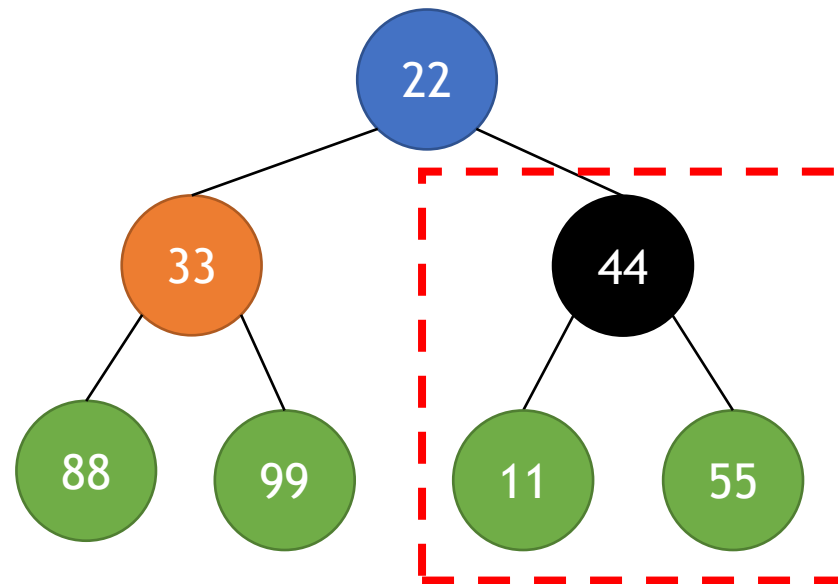


Os nós folha já são  
heaps de máximo.

Então, podemos  
aplicar max\_heapfy  
nos pais (33 e 44).



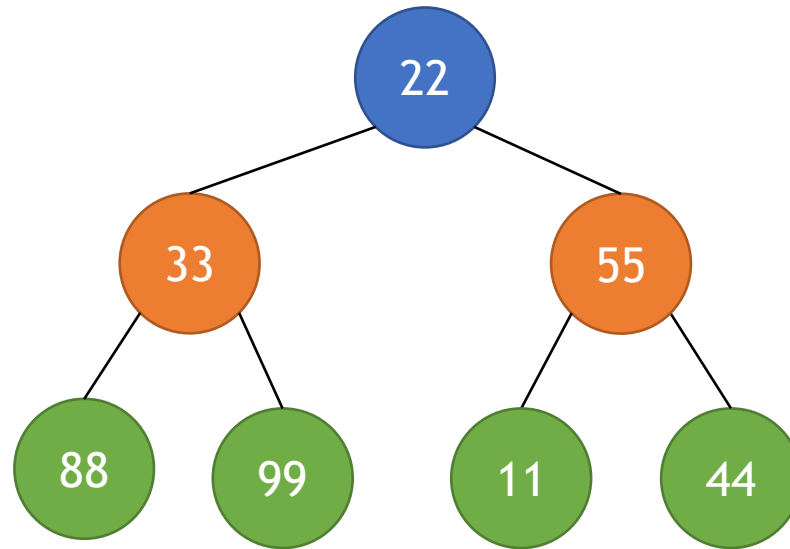
# Função: build\_max\_heap



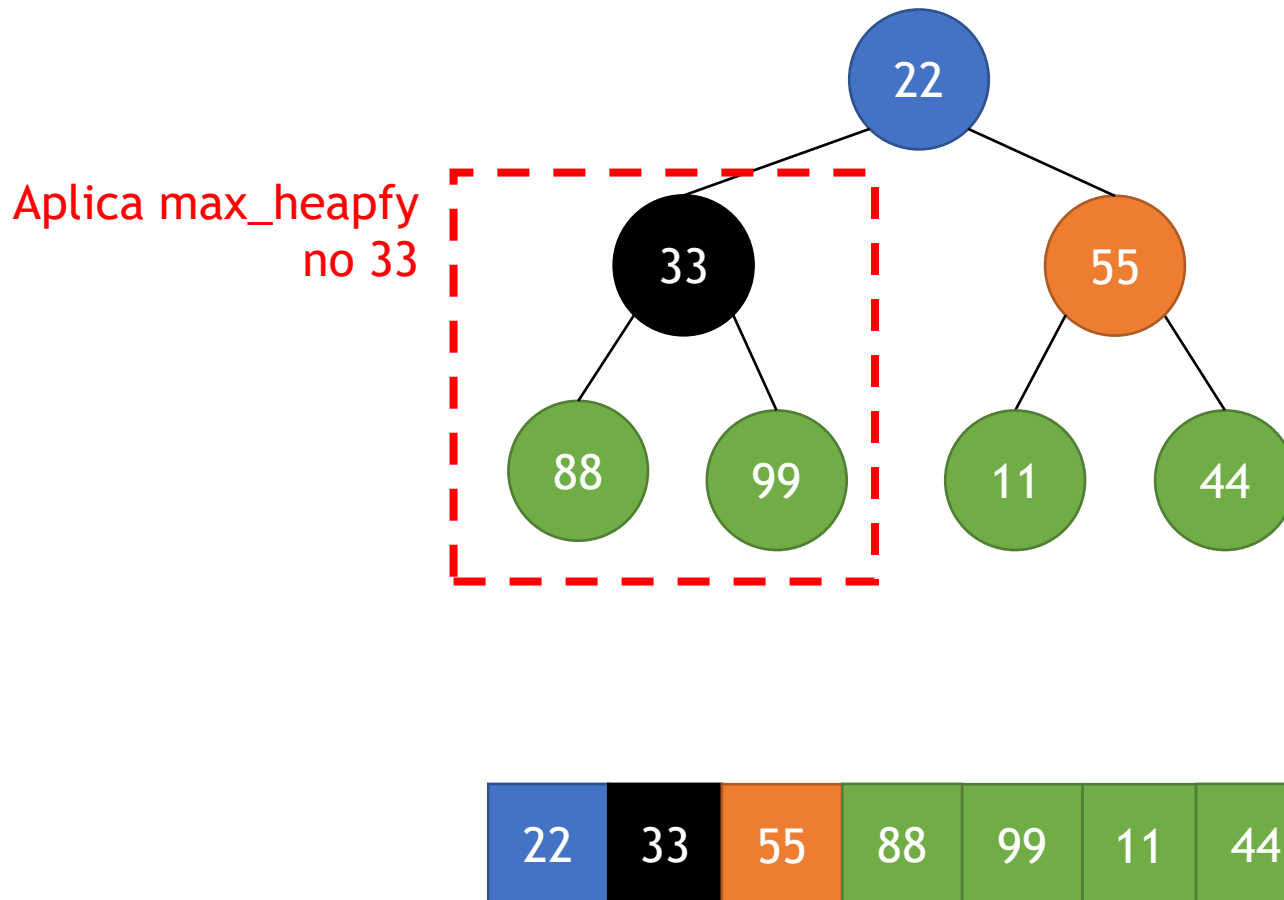
Aplica max\_heapfy  
no 44



# Função: build\_max\_heap

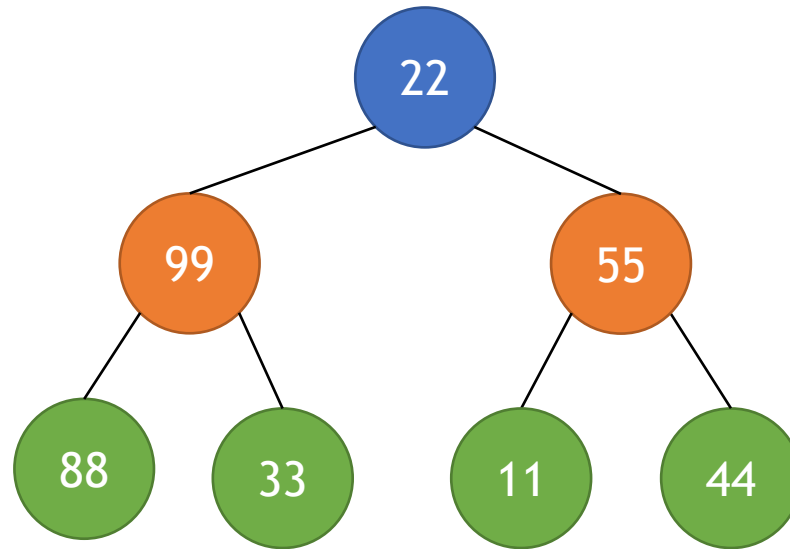


# Função: build\_max\_heap

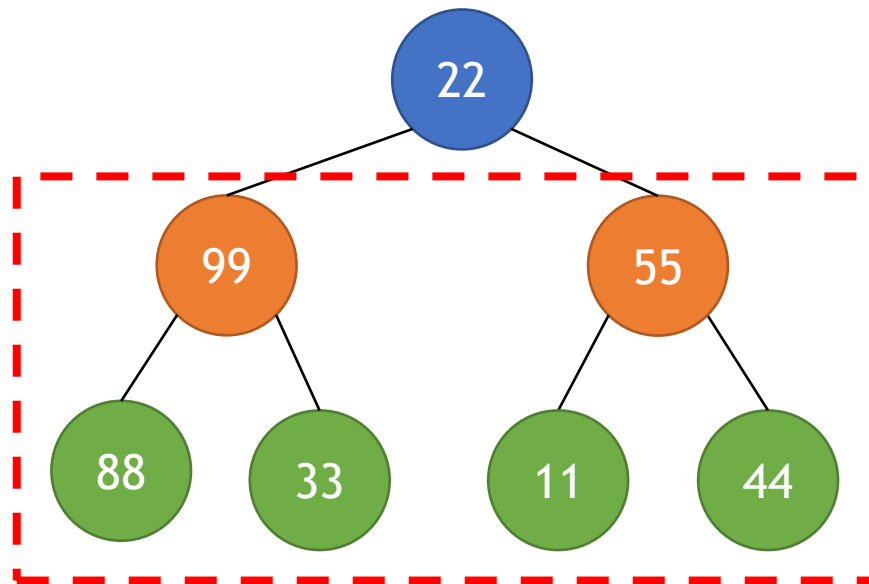




# Função: build\_max\_heap



# Função: build\_max\_heap

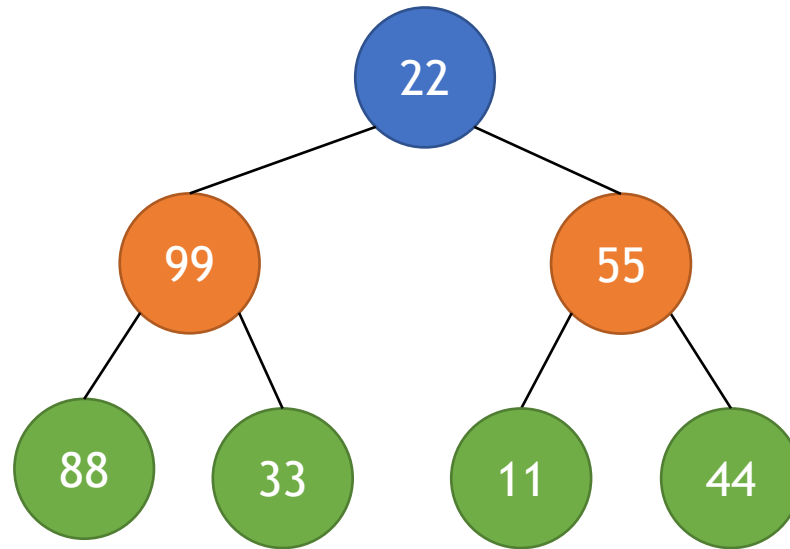


As duas subárvores de 22 são heaps de máximo.

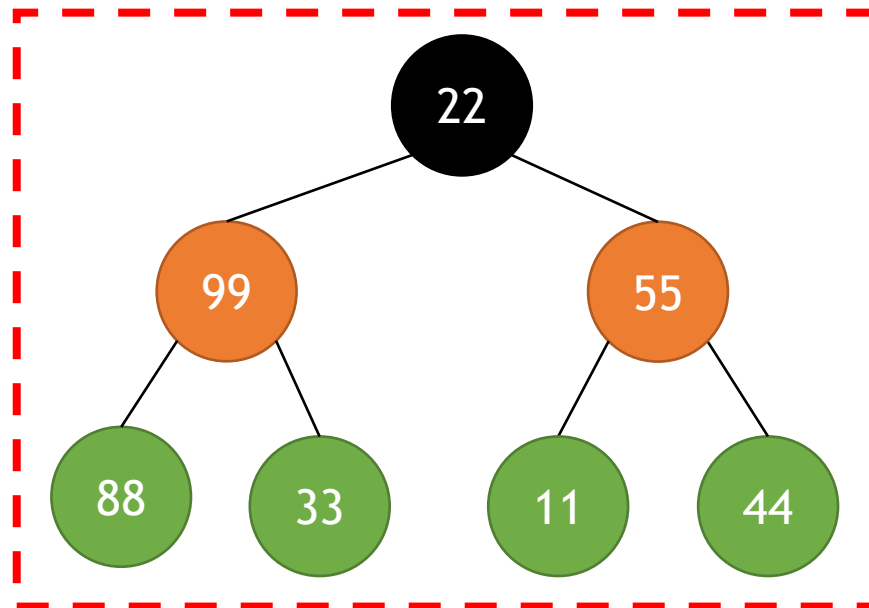
Então, podemos aplicar max\_heapfy no pai (22).



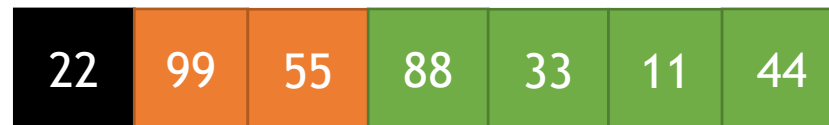
# Função: build\_max\_heap



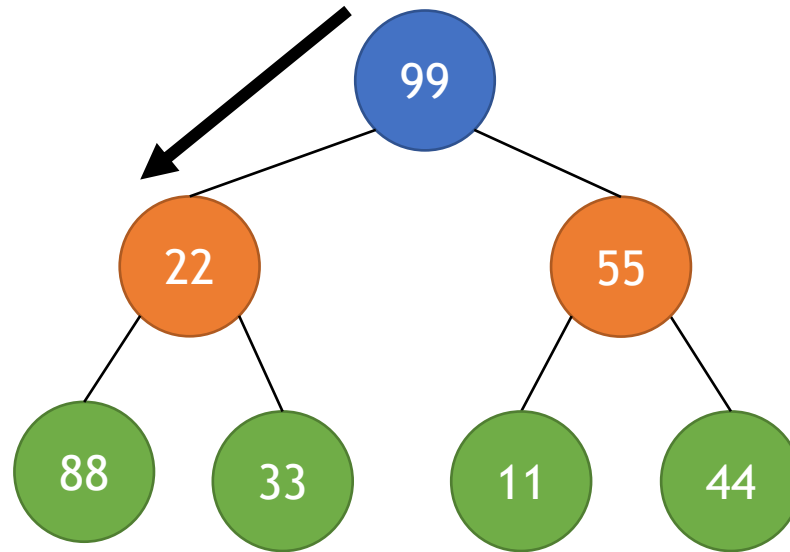
# Função: build\_max\_heap



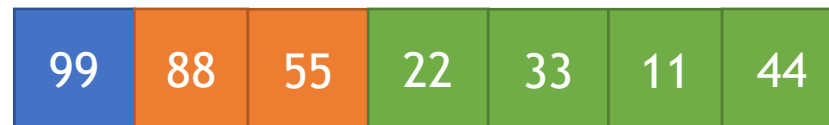
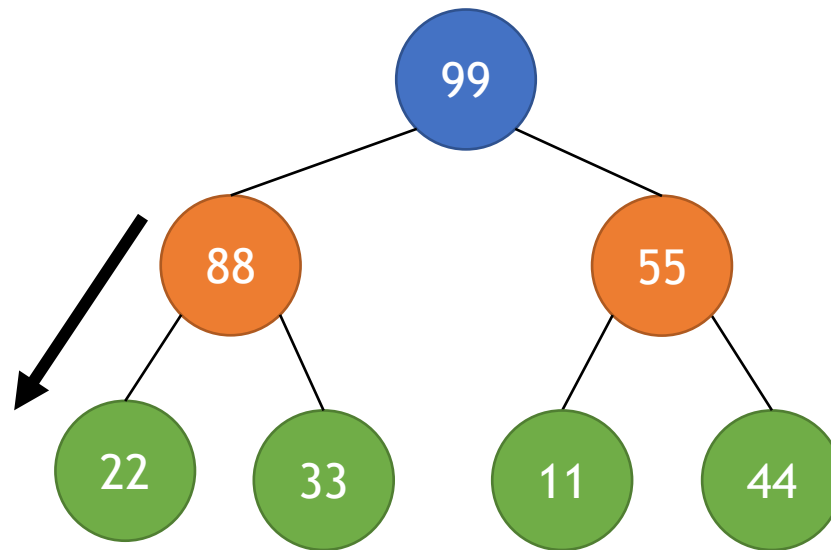
Aplica max\_heapfy  
no 22



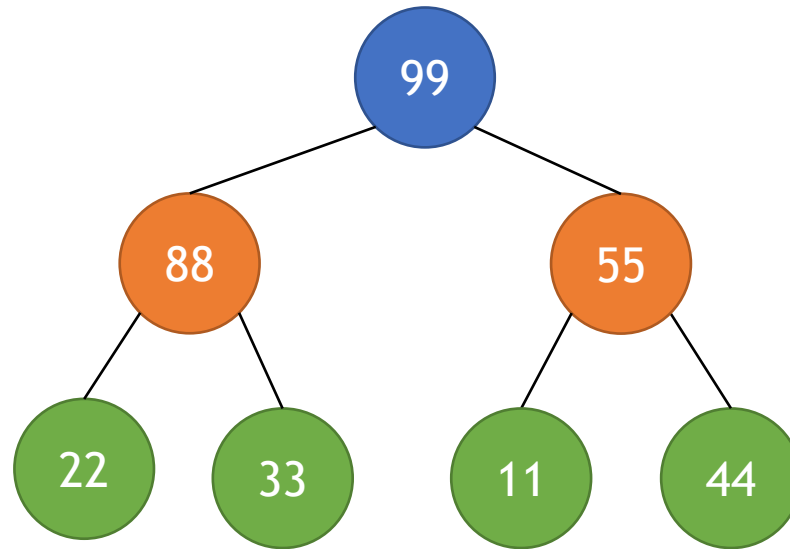
# Função: build\_max\_heap



# Função: build\_max\_heap



# Função: build\_max\_heap



# Função: build\_max\_heap

- Implementação em C:

Chamada:

```
build_max_heap(vetor, n);
```



# Função: build\_max\_heap

```
void build_max_heap(int *v, int n) {  
    int i;  
    for (i = n/2 - 1; i >= 0; i--)  
        max_heapfy(v, n, i);  
}
```

Mais detalhes na Seção 6.3 (páginas 115 e 116): CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 3ª edição. Rio de Janeiro, RJ: Elsevier, 2012.

Custo =  $O(n)$

# Heap sort

# Heap sort

- Idéia geral do algoritmo:
  - Transforma vetor em heap de máximo (**build\_max\_heap**);
  - Repete processo enquanto tiver pelo menos dois elementos para ordenar:
    - Troca o primeiro elemento com o último elemento do heap;
    - Diminui tamanho do heap;
    - Aplica **max\_heapfy** na raiz (primeiro elemento do vetor).

# Heap sort

Maior elemento do heap  
vai para final.

- Idéia geral do algoritmo:
  - Transforma vetor em heap de máximo (**build\_max\_heap**);
  - Repete processo enquanto tiver pelo menos dois elementos para ordenar:

- Troca o primeiro elemento com o último elemento do heap;

- Diminui tamanho do heap;
  - Aplica **max\_heapfy** na raiz (primeiro elemento do vetor).


# Heap sort

Último elemento já está posicionado, agora heap vai considerar apenas os elementos anteriores.

- Idéia geral do algoritmo:
  - Transforma vetor em heap de máximo (**build\_max\_heap**);
  - Repete processo enquanto tiver pelo menos dois elementos para ordenar:
    - Troca o primeiro elemento com o último elemento do heap;
    - **Diminui tamanho do heap;**
    - Aplica **max\_heapfy** na raiz (primeiro elemento do vetor).

# Heap sort

Manutenção da propriedade de heap

- Idéia geral do algoritmo:
    - Transforma vetor em heap de máximo (**build\_max\_heap**);
    - Repete processo enquanto tiver pelo menos dois elementos para ordenar:
      - Troca o primeiro elemento com o último elemento do heap;
      - Diminui tamanho do heap;
      - Aplica **max\_heapfy** na raiz (primeiro elemento do vetor).
- 

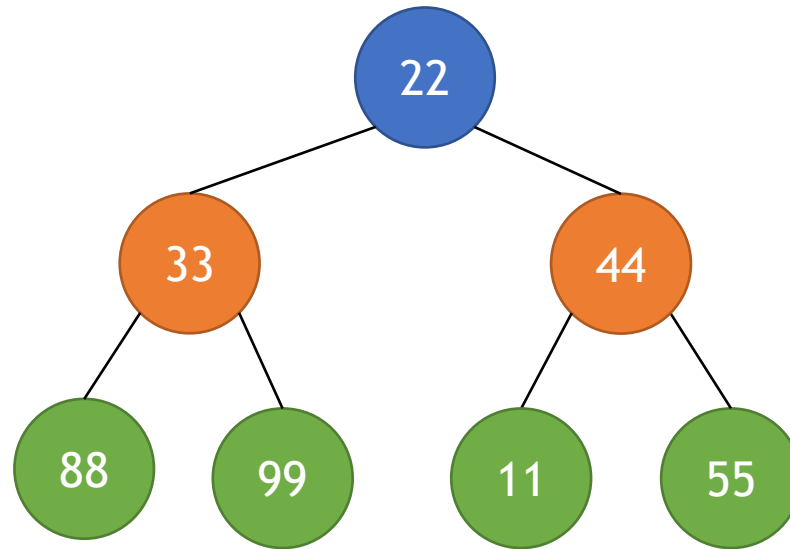
# Heap sort

- Idéia geral do algoritmo:
  - Transforma vetor em heap de máximo (**build\_max\_heap**);
  - Repete processo enquanto tiver pelo menos dois elementos para ordenar:
    - Troca o primeiro elemento com o último elemento do heap;
    - Diminui tamanho do heap;
    - Aplica **max\_heapfy** na raiz (primeiro elemento do vetor).

## Heapsort: `build_max_heap`;

Repete (enquanto há pelo menos dois elementos):

- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica `max_heapfy` na raiz.

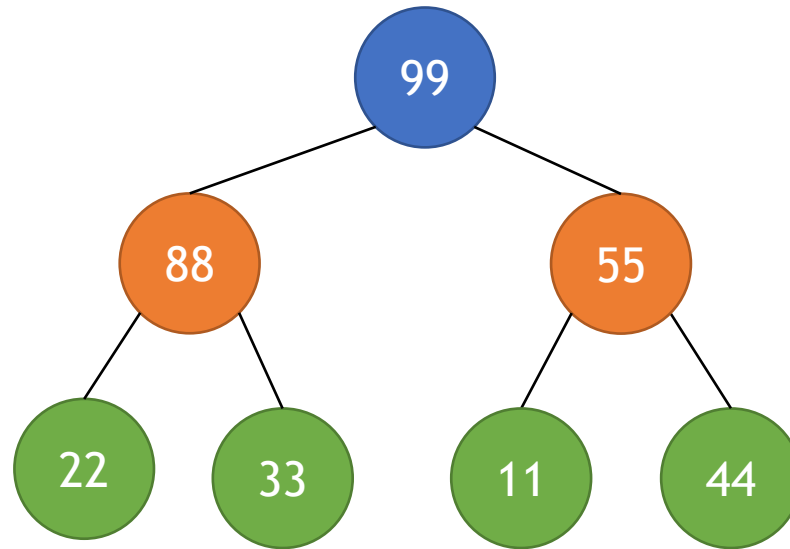




## Heapsort: `build_max_heap`;

Repete (enquanto há pelo menos dois elementos):

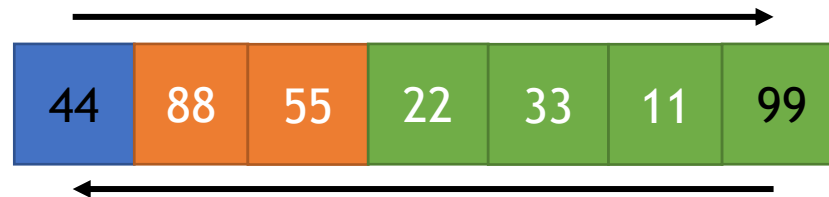
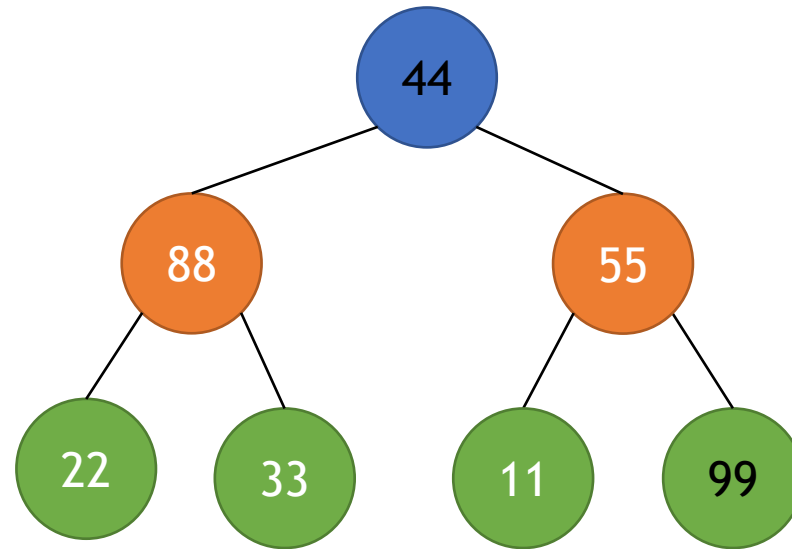
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica `max_heapfy` na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

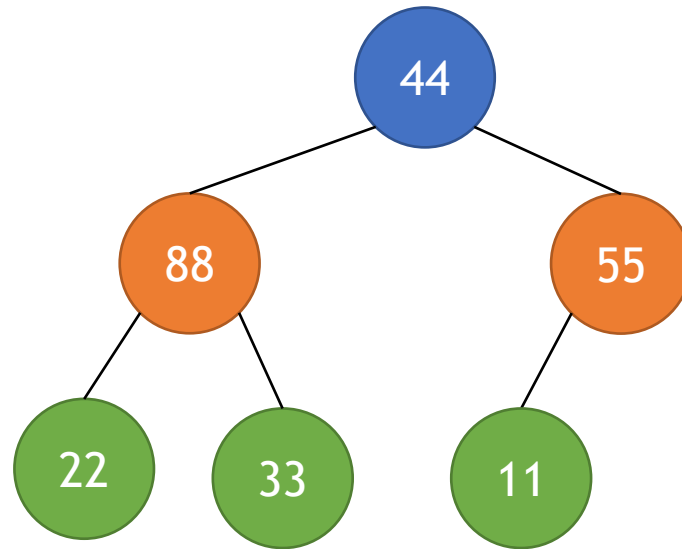
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

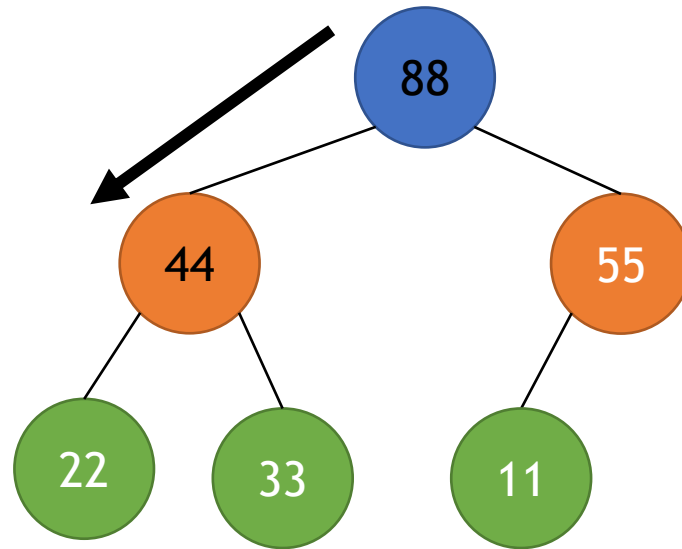
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

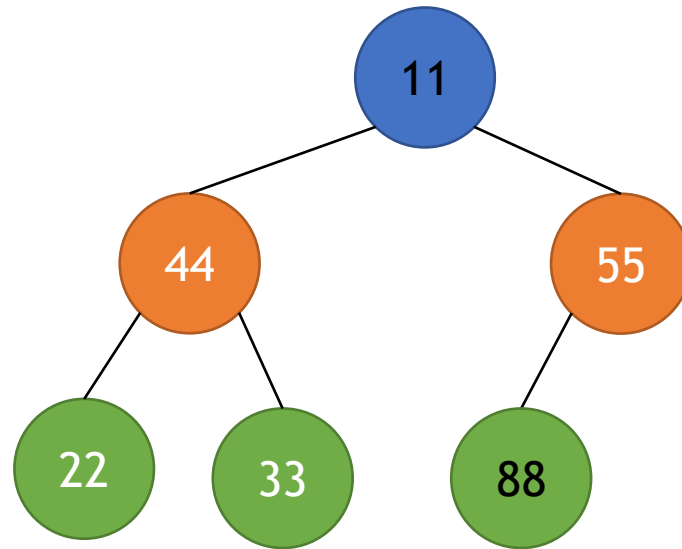
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

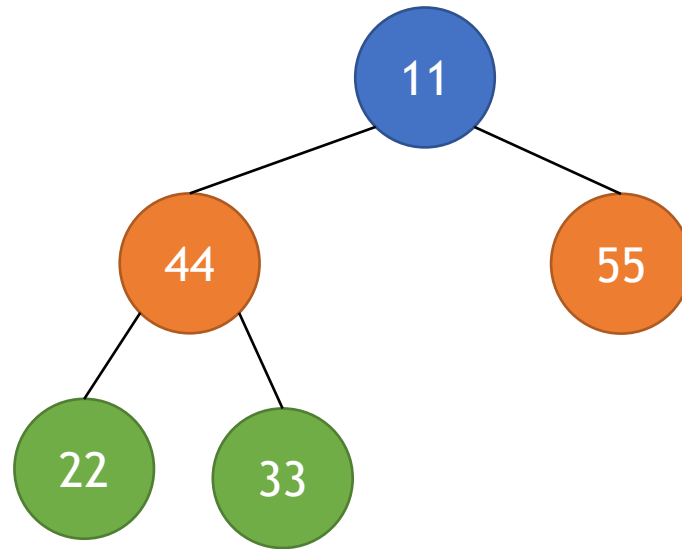
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

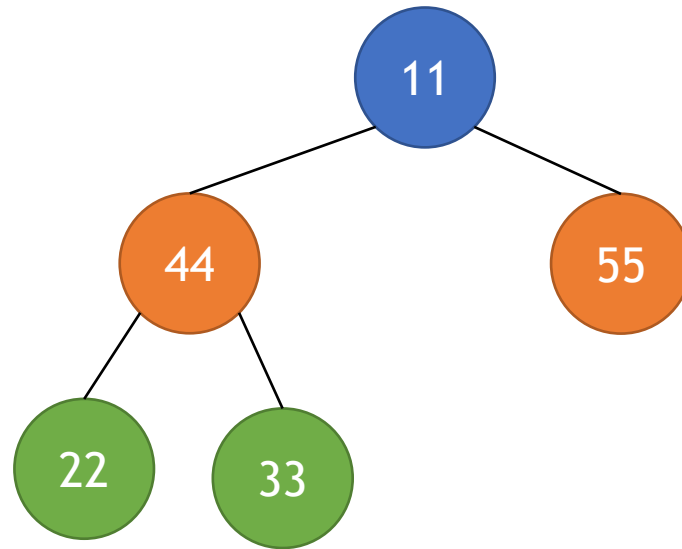
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

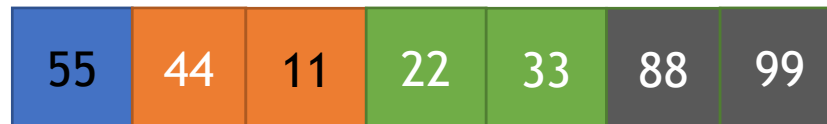
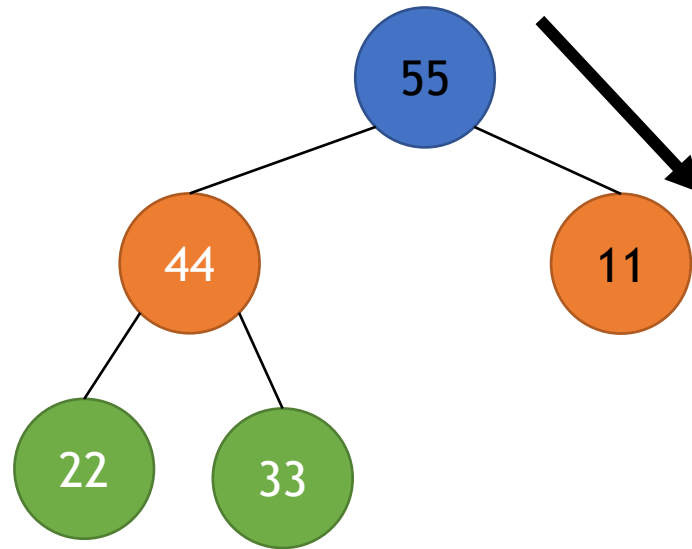
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**

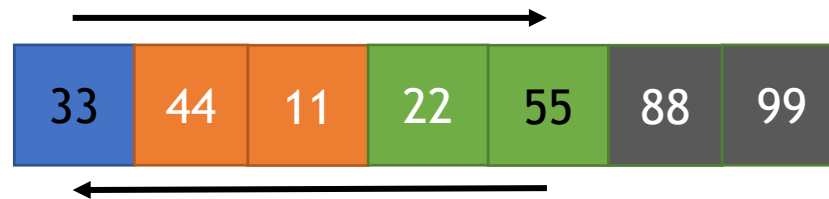
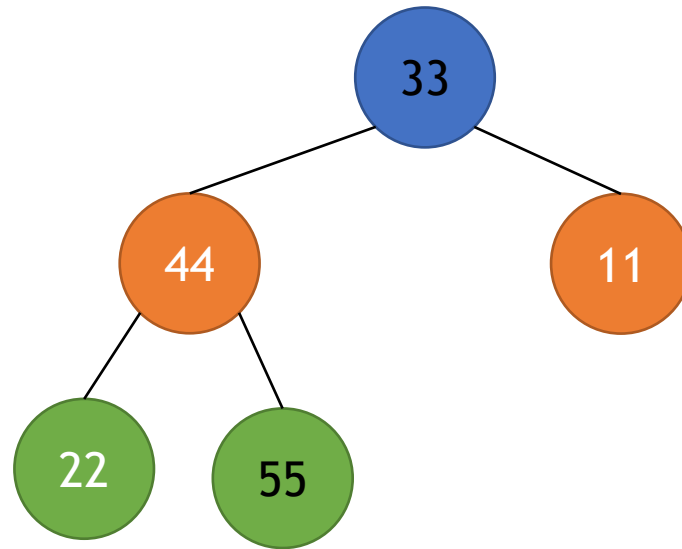




**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

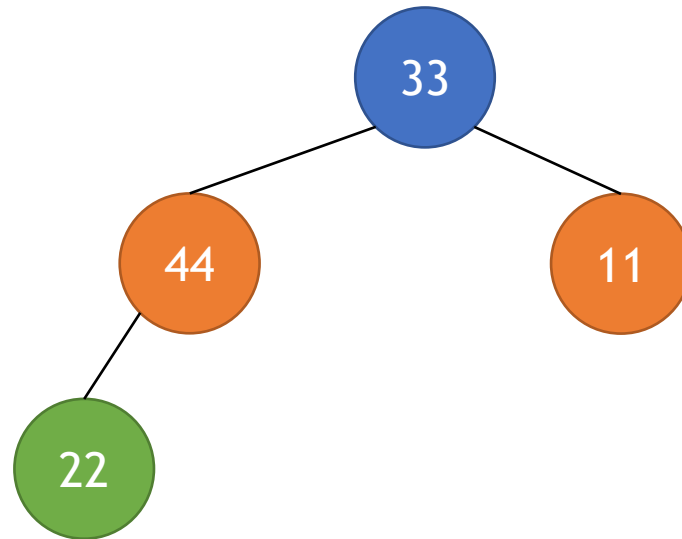
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

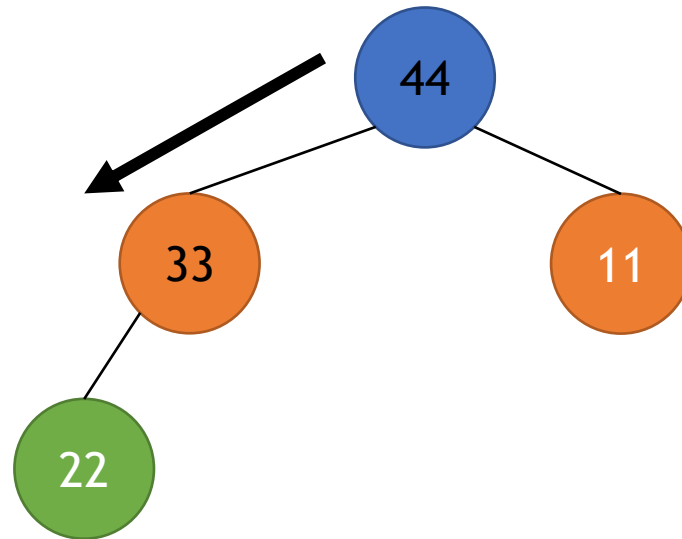
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

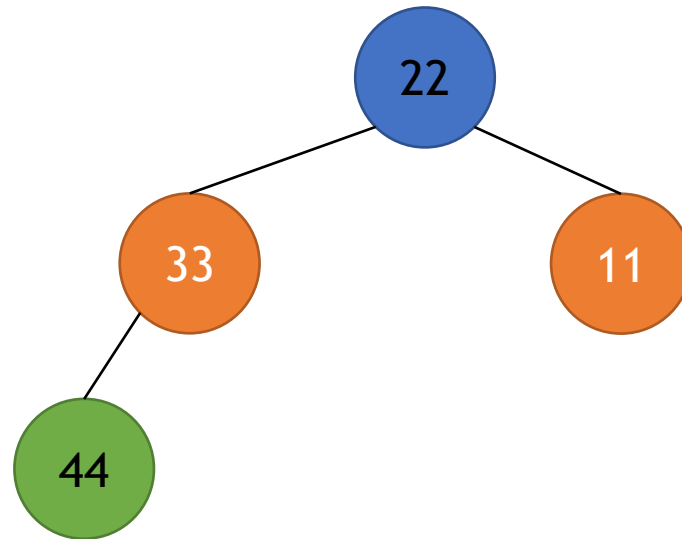
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

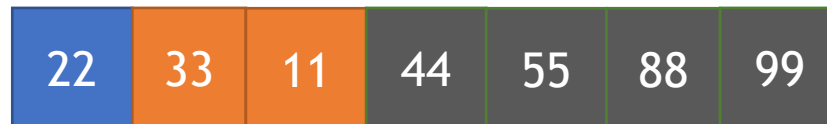
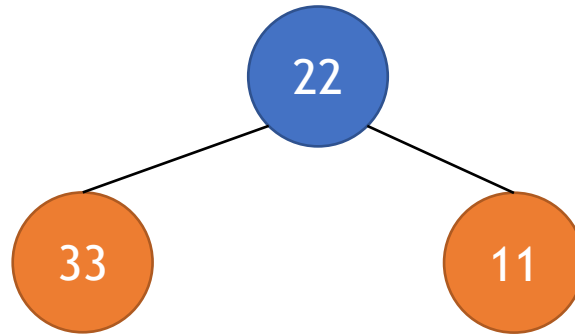
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

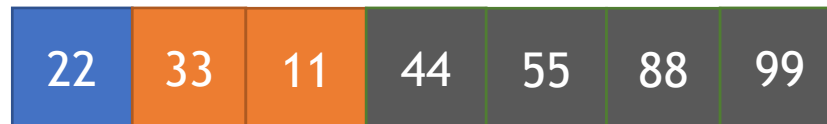
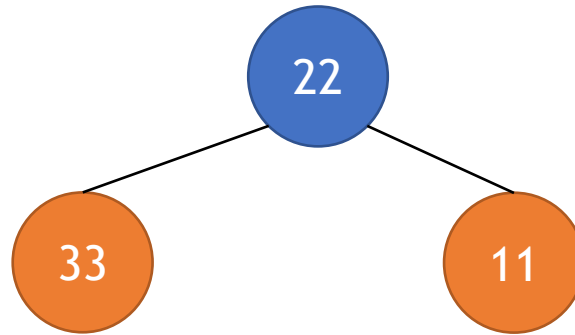
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

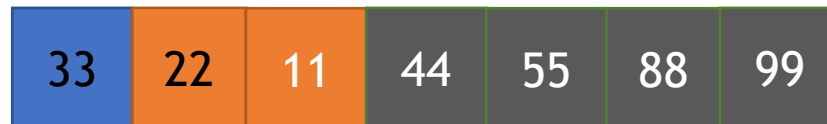
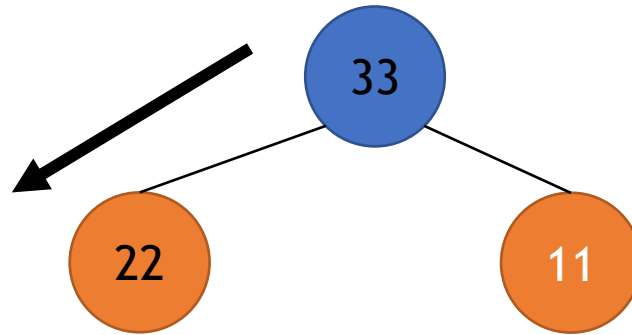
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

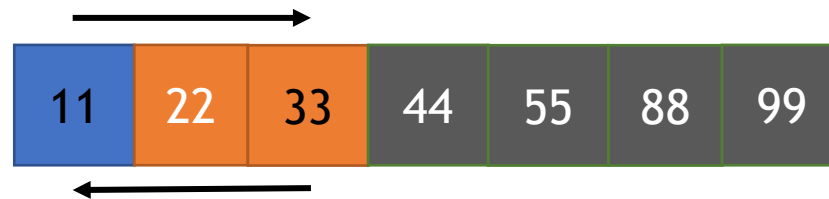
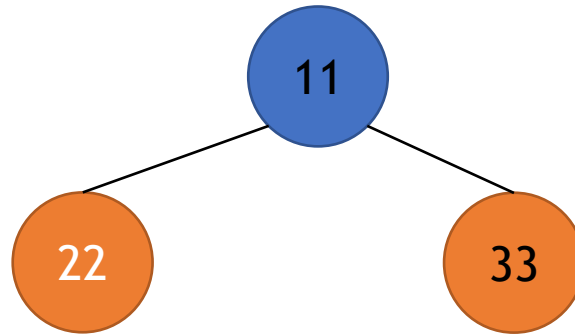
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.

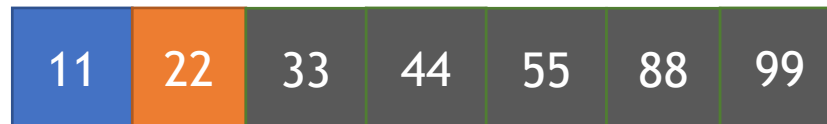
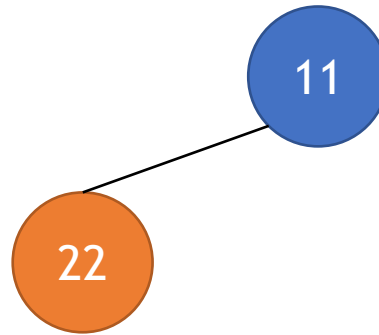




**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

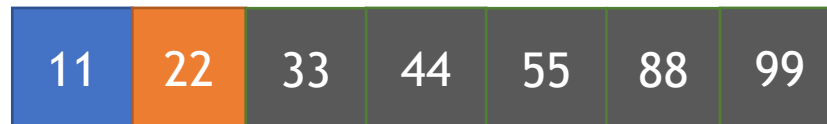
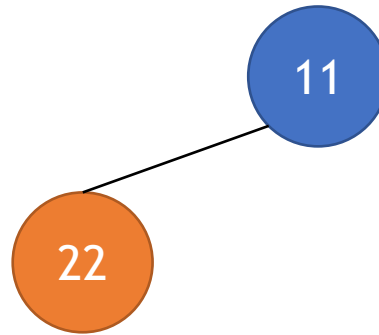
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapify na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

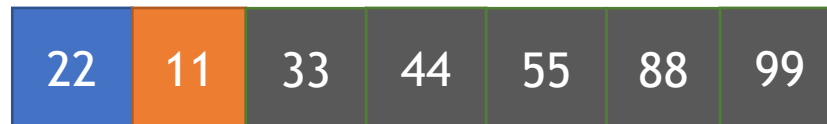
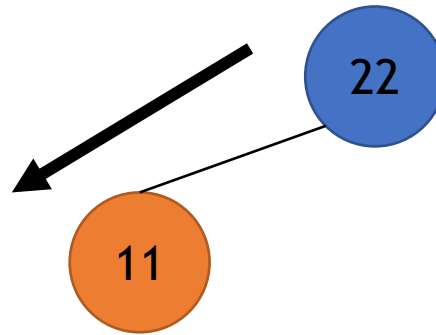
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

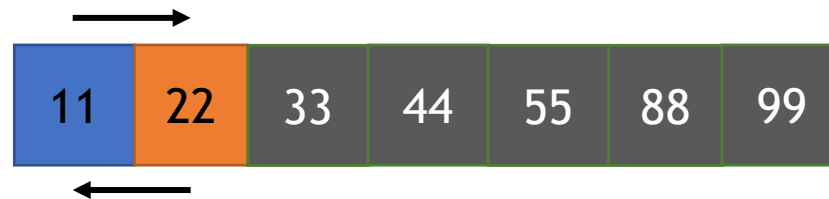
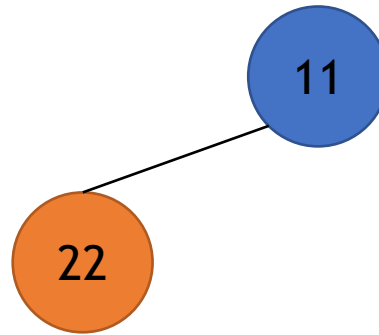
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- **Aplica max\_heapfy na raiz.**



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

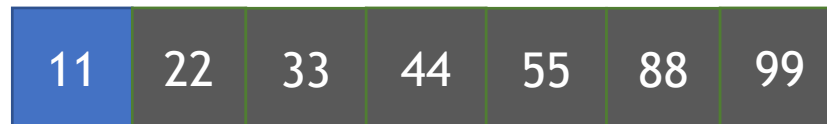
- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (**enquanto há pelo menos dois elementos**):

- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.



**Heapsort:** build\_max\_heap;

Repete (enquanto há pelo menos dois elementos):

- Troca raiz com o último elemento do heap e reduz tamanho do heap;
- Aplica max\_heapfy na raiz.

Ordenação finalizada.

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 11 | 22 | 33 | 44 | 55 | 88 | 99 |
|----|----|----|----|----|----|----|

# Heap sort

- Implementação em C:




Chamada:  
`heapsort(vetor, n);`

# Heap sort




```
void heapsort(int *v, int n) {  
    build_max_heap(v, n);  
    int i;  
    for (i = n-1; i > 0; i--) {  
        int aux = v[i];  
        v[i] = v[0];  
        v[0] = aux;  
        max_heapfy(v, i, 0);  
    }  
}
```



# Custo (de tempo) do heap sort

```
void heapsort(int *v, int n) {  
    build_max_heap(v, n);   $O(n)$   
    int i;  
    for (i = n-1; i > 0; i--) {  Executa n-1 vezes.  
        int aux = v[i];  
        v[i] = v[0];  
        v[0] = aux;  
        max_heapfy(v, i, 0);   $O(\lg(n))$   
    }  
}
```

# Custo (de tempo) do heap sort

```
void heapsort(int *v, int n) {  
    build_max_heap(v, n);   $O(n)$   
    int i;  
    for (i = n-1; i > 0; i--) {  Executa n-1 vezes.  
        int aux = v[i];  
        v[i] = v[0];  
        v[0] = aux;  
        max_heapfy(v, i, 0);   $O(\lg(n))$   
    }  
}
```

Custo (de tempo) do heap sort:  $O(n \cdot \lg(n))$

# Resumo dos algoritmos

|             | Selection sort | Insertion sort | Merge sort          | Quick sort          | Heap sort           |
|-------------|----------------|----------------|---------------------|---------------------|---------------------|
| Pior caso   | $O(n^2)$       | $O(n^2)$       | $O(n \cdot \lg(n))$ | $O(n^2)$            | $O(n \cdot \lg(n))$ |
| Caso médio  | $O(n^2)$       | $O(n^2)$       | $O(n \cdot \lg(n))$ | $O(n \cdot \lg(n))$ | $O(n \cdot \lg(n))$ |
| Melhor caso | $O(n^2)$       | $O(n)$         | $O(n \cdot \lg(n))$ | $O(n \cdot \lg(n))$ | $O(n \cdot \lg(n))$ |

(para lista com chaves diferentes)



# Referências

- Slides do Prof. Monael Pinheiro Riberio:
  - <https://sites.google.com/site/aed2019q1/>
- Slides do Prof. Fabrício Olivetti de França
  - <https://folivetti.github.io/courses/AEDI/>

# Bibliografia básica

- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de programação: a construção de algoritmos e estruturas de dados. 3ª edição. São Paulo, SP: Prentice Hall, 2005.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2ª edição. Rio de Janeiro, RJ: Campus, 2002.

# Bibliografia complementar

- AGUILAR, L. J. Programação em C++: algoritmos, estruturas de dados e objetos. São Paulo, SP: McGraw-Hill, 2008.
- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Cengage Learning, 2009.
- KNUTH D. E. The art of computer programming. Upper Saddle River, USA: Addison- Wesley, 2005.
- SEDGEWICK, R. Algorithms in C++: parts 1-4: fundamentals, data structures, sorting, searching. Reading, USA: Addison-Wesley, 1998.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3a edição. Rio de Janeiro, RJ: LTC, 1994.
- TEWNENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.