Aula 11:

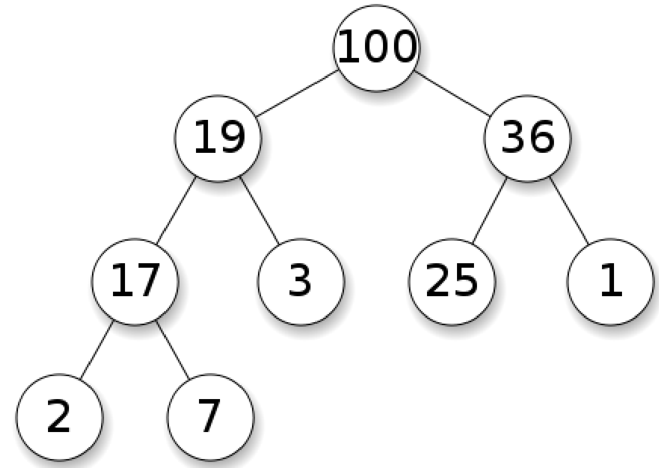
Heaps

PCO001 – Algoritmos e Estruturas de Dados

Prof. João Paulo Reus Rodrigues Leite
joaopaulo@unifei.edu.br

O que é uma **heap**?

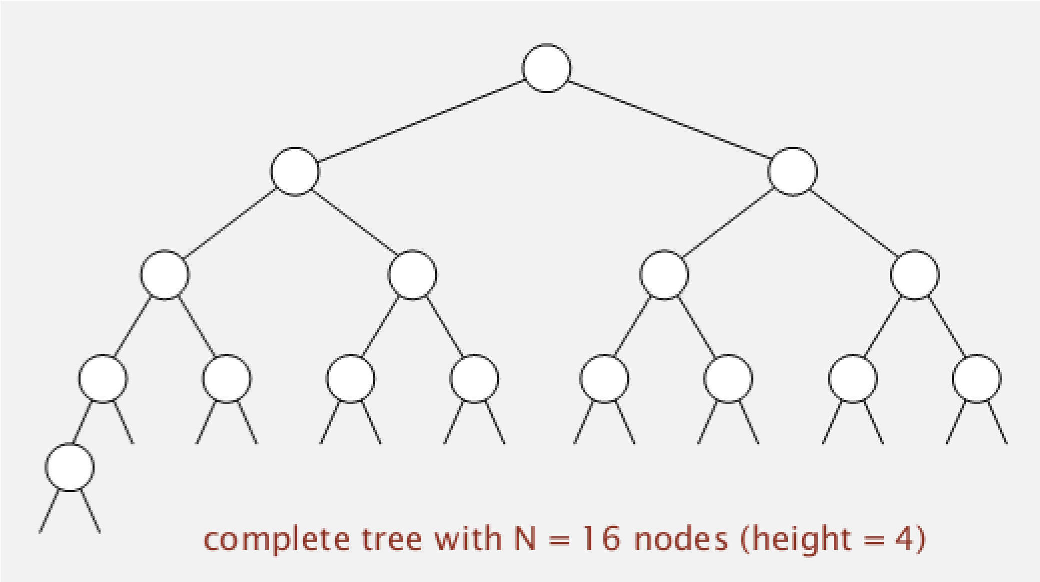
É uma estrutura de dados organizada como uma **árvore binária completa**, ou seja, uma árvore binária onde cada nível é preenchido da esquerda para a direita e deve estar cheio antes que o próximo nível seja iniciado.



Além disso, há uma restrição especial: <u>cada nó da heap deve conter um valor maior</u> (ou menor) do que todos os valores contidos por nós descendentes dele.

Árvore Binária Completa

Perfeitamente balanceada, exceto pelo último nível.



Altura de uma árvore binária completa com n nós é igual a [**log n**].

Sua altura apenas aumenta quando n chegar na próxima potência de 2.

Podem “faltar” nós no último nível da árvore, mas eles devem ser os nós mais a direita. A árvore é sempre preenchida de **cima pra baixo**, da **esquerda pra direita**.

Portanto, o que é uma **heap**?

Em outras palavras, é uma árvore binária completa com uma regra particular:

> Em qualquer nó, o **valor armazenado nele** deve ser ≥ aos valores armazenados em seus nós filhos.

Essa restrição garante que o **maior** estará sempre no **topo**, ou seja, na **raiz.**

- Nesse caso, é chamada de "heap de máximo".
- Em nossa implementação do heap sort, a **raiz da árvore sempre conterá o maior elemento**, mas poderia ser diferente (como em uma heap de mínimo).
- Heaps são, muitas vezes, implementadas como **filas de prioridade** (priority_queue, na STL de C++).

Onde uma **heap** pode ser utilizada?

- **Simulações** (Clientes ou carros em uma fila, colisão de partículas, etc.)
- **Compressão de dados** (Huffman)
- **Algoritmos em grafos** (Dijkstra, Prim)
- **Inteligência artificial** (A* *search*)
- **Estatística** (Mantém os maiores valores em uma sequência)
- **Sistemas Operacionais** (Balanceamento de carga, manipulação de interrupções)
- E muitos outros...

O(V) para inicializar
O(V) para criar a heap
O(V log V) para encontrar menores
O(E log V) para atualizar custos.

```
#include <vector>
#include <queue>
#include <utility>
#define INFINITY 1000000000
#define SIZE 5

using namespace std;

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
```

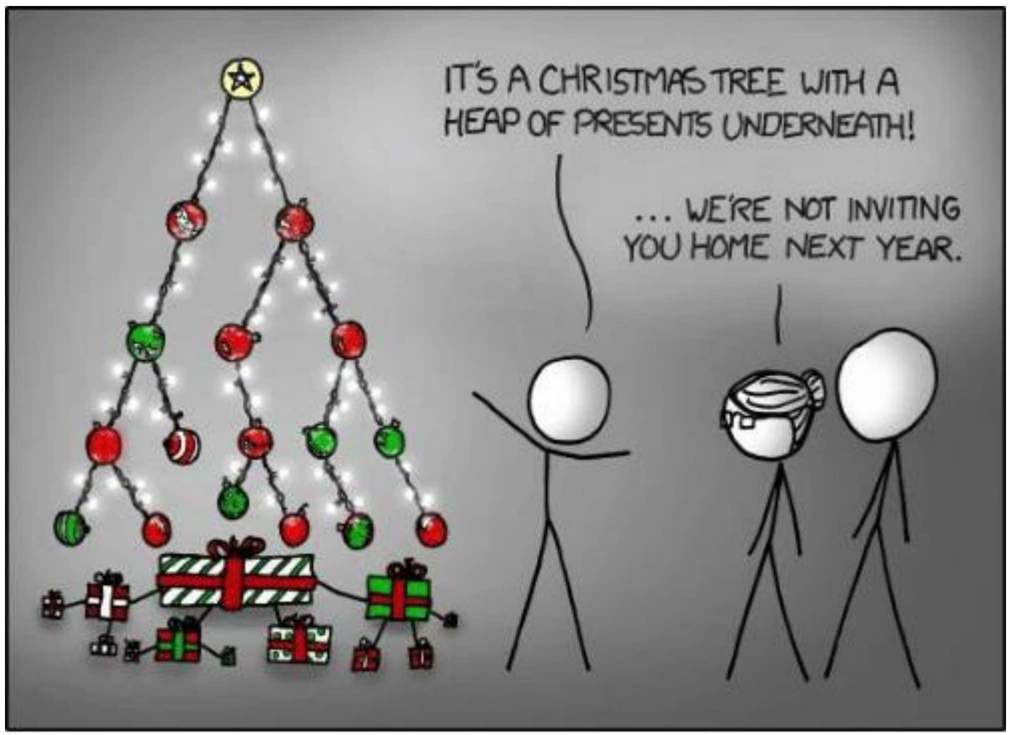
Implementação de Dijkstra utilizando a estrutura de heap provida pela STL, **priority_queue**, com complexidade O((E + V) log V)

```
vii adj[SIZE];
vi custo(SIZE, INFINITY);

void dijkstra(int s) {
  custo[s] = 0;
  priority_queue<ii, vii, greater<ii> > heap;
  heap.push(make_pair(0, s));

  while(!heap.empty()) {
    ii menor = heap.top(); heap.pop();
    int d = menor.first, u = menor.second;
    if(d > custo[u]) continue;
    for(int i = 0; i < adj[u].size(); i++) {
      ii v = adj[u][i];
      if(custo[u] + v.second < custo[v.first]) {
        custo[v.first] = custo[u] + v.second;
        heap.push(ii(custo[v.first], v.first));
      }
    }
  }
}
```

Mas como **representar uma heap**?

Precisaremos de ponteiros, *struct* Node, etc.?

Existe uma maneira mais fácil?



Estrutura da **Heap**

A <u>regularidade</u> de uma árvore binária completa facilita sua implementação utilizando apenas um **vetor**.

- Conseguimos prever, com facilidade e baixo custo computacional, as <u>posições de pais e filhos</u> na estrutura.

Os nós da árvore possuem uma ordem natural:

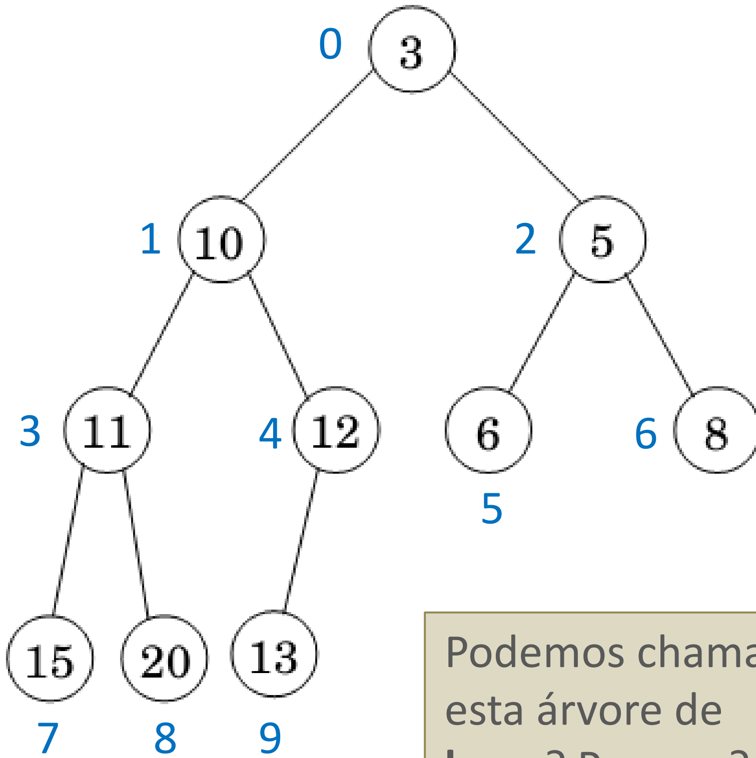
- Linha por linha, de cima para baixo, começando da raiz, e movendo-se da esquerda para a direita em cada linha.

Se houver *n* nós, essa ordem especifica suas posições 0, 1, 2, ..., n dentro do vetor.

- Mover-se para cima ou para baixo na árvore é facilmente simulado no vetor, usando o fato de que o nó j tem:
  pai na posição [(j-1)/2] e **filhos** em [2j+1] e [2j+2].

Exemplo (Árvore Binária Completa):

3	**10**	**5**	**11**	**12**	**6**	**8**	**15**	**20**	**13**
0	1	2	3	4	5	6	7	8	9



Tomemos o nó **j = 3**:

O seu **nó pai** é calculado por (j-1)/2 = (3-1)/2 = **1**

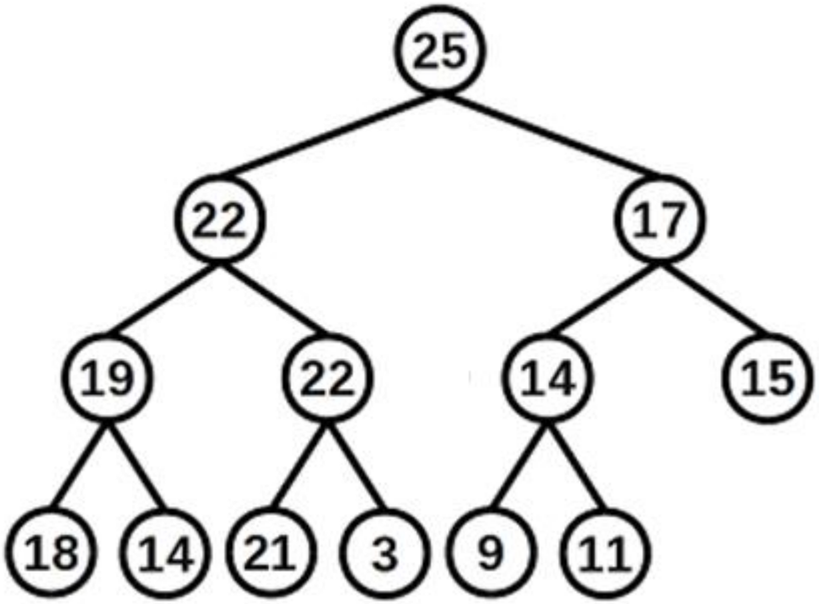
Seus **nós filhos** serão (2j+1) e (2j+2), ou seja, **7** e **8**.

Podemos chamar esta árvore de **heap**? Por que?

Exercício:

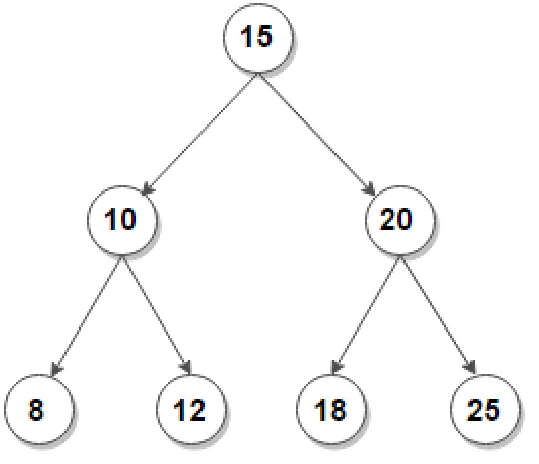
Dada heap abaixo, monte seu vetor correspondente e verifique se as fórmulas para encontrar pais e filhos estão corretas:

- Quem é o pai do nó com valor 3?
- Quem são os filhos da esquerda e direita do nó com valor 19?



Pai é [(j-1)/2]
Filhos são [2j+1] e [2j+2].

Mas como transformar uma árvore binária completa comum em uma ***heap***?



A função **create_heap** abaixo ajusta um dado vetor de maneira que seus elementos estejam organizados em uma *heap* de máximo, onde cada nó é maior que seus filhos, e a raiz da árvore (vet[0]) é o maior de todos os elementos.

- O processo ocorre de baixo para cima (*bottom-up*), começando nos **primeiros nós não-folha** (aqueles que tem filhos, a partir da posição *n/2* - 1) e prosseguindo em direção à raiz (na posição 0).
- Isso nos garante que, a cada passo, tenhamos duas *subheaps* de máximo já ajustadas abaixo do nó i, uma para cada filho.

```c
void create_heap(int *vet, int n) {
  // para cada nó que não é folha
  for(int i = n/2-1; i >= 0; i--)
    heapfy(vet, n, i); // ajusta heap de baixo pra cima
}
```

- Em cada iteração do laço for, o nó de índice *i* representa a raiz de uma *heap* de máximo, formado por um “galho” (ou sub-árvore) da árvore completa.
  – Os ajustes devem ocorrer apenas nos nós que possuem descendentes. Os nós folhas já estão automaticamente ajustados, pois formam uma (sub)*heap* de um único elemento.

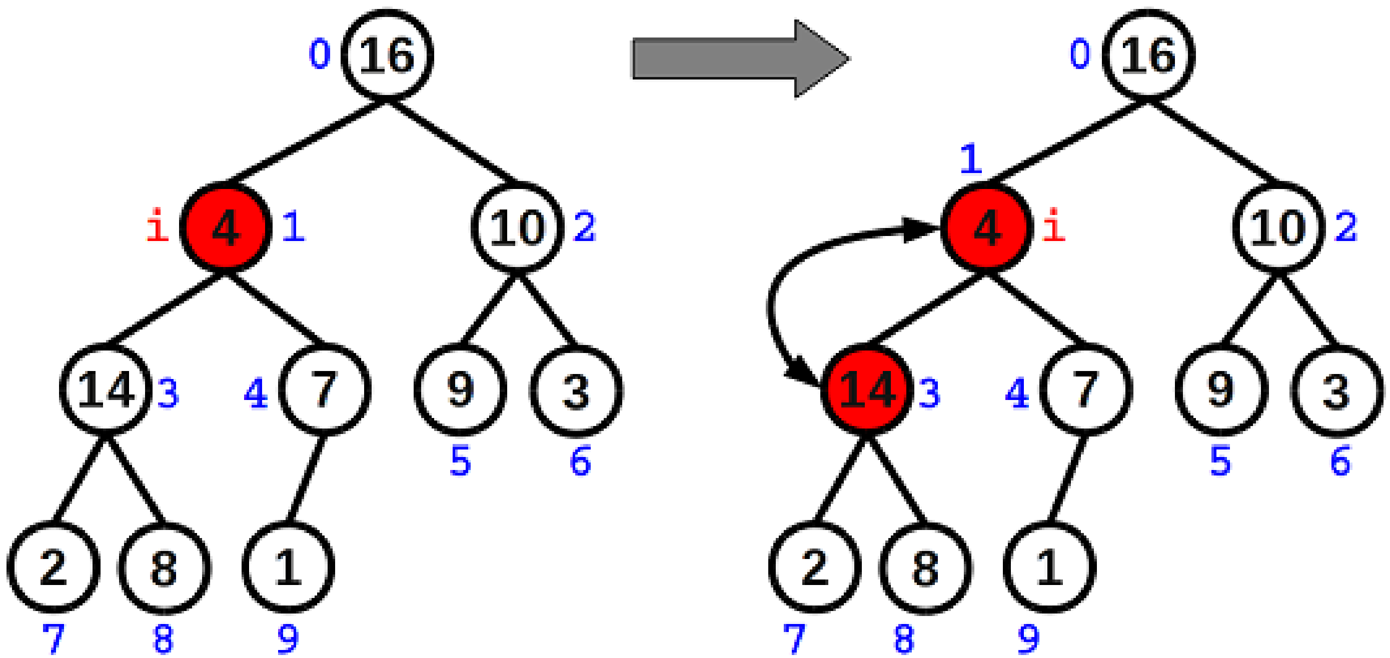
- Para manter a propriedade de heap de máximo, chamamos a função heapfy para cada um dos nós que não são folhas.

- Quando chamada em um determinado nó, a função *heapfy* assume que suas subárvores da esquerda e direita **já estão ajustadas e são heaps de máximo** (pois acabaram de passar pelo mesmo processo de “heapificação” (*heapfy)*, que é *bottom-up*).
  – Com isso, já sabemos que o valor em cada topo já é o maior em toda a sua sub-árvore. Não é preciso verificar novamente todos os outros nós abaixo dele.

A verificação que importa é com relação ao nó v[i], que pode conter um valor menor do que os de seus descendentes, violando a regra básica da *heap* de máximo.

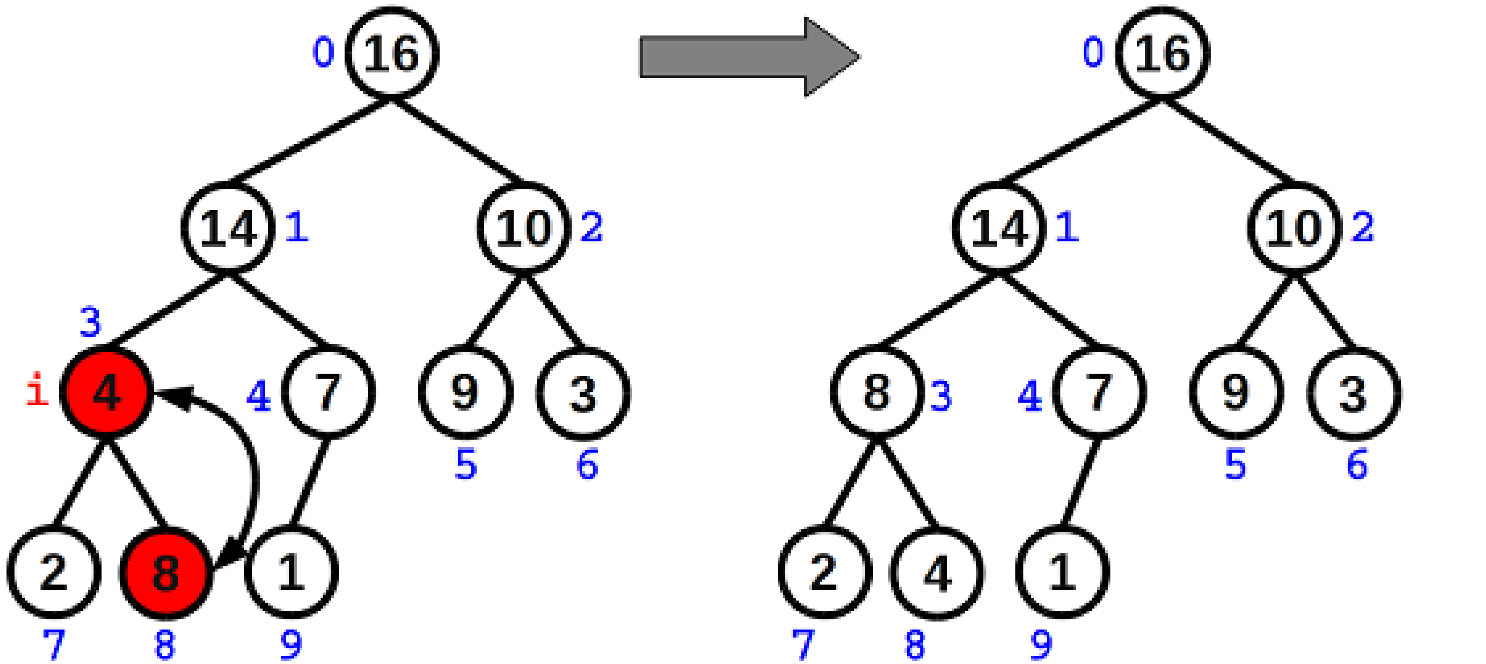
Quando uma violação à regra da heap de máximo for identificada, a função **heapfy** vai fazer com que esse elemento de menor valor seja "afundado" (*sink*) na *heap* de máximo até encontrar o seu local adequado.

- Ao final do processo recursivo, a subárvore com raiz em *i* passa também a obedecer à regra da heap de máximo.



16	4	10	14	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

16	14	10	4	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9



16	14	10	4	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

```c
void heapfy(int *vet, int n, int i) {
  int esq = 2*i+1; // filho do lado esquerdo
  int dir = 2*i+2; // filho do lado direito
  int maior, aux;

  // escolhe maior valor entre os filhos (esq e dir)
  maior = i;
  if((esq < n) && (vet[esq] > vet[maior]))
    maior = esq;
  if((dir < n) && (vet[dir] > vet[maior]))
    maior = dir;

  // se um dos filhos for maior que o pai, troca.
  if(maior != i) {
    aux = vet[i];
    vet[i] = vet[maior];
    vet[maior] = aux;

    // continua descida até local adequado
    heapfy(vet, n, maior);
  }
}
```

A variável “maior” ironicamente passa a ter o índice do elemento de menor valor (nó pai), que acaba de ser trocado de posição.

O **tempo de execução** da função *heapfy* em uma subárvore de tamanho *n* com raiz em um dado nó *i* é:

- $\Theta(1)$ para obter as relações entre os elementos v[i], v[esq] e v[dir];
- Mais o tempo para executar o mesmo **heapfy** em uma subárvore com raiz em um dos filhos de i (considerando que a chamada recursiva ocorra todas as vezes, no pior caso)
  - As subárvores de cada filho possuem tamanho máximo de 2*n/3. O pior caso ocorre quando a última linha da árvore está exatamente metade cheia.

Construindo a **Heap**

O tempo de execução da função *heapfy* pode, portanto, ser descrito pela relação de recorrência:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Através do **teorema mestre**, podemos concluir que, a partir do segundo caso, o tempo de execução é:

- $O(n^d \log n)$, se $d = \log_b a$
  - $O(n^0 \log n)$, se $0 = \log_{3/2} 1$
- $T(n) = O(\log n)$

Construindo a **Heap**

O tempo de execução da função createHeap é calculado, então, da seguinte maneira:

- Cada chamada de **heapfy** custa o tempo O(log n), e **create_heap** faz O(n) dessas chamadas.
- O tempo de execução é, portanto, **O(n log n)**.

Analisando-se mais profundamente a estrutura da heap e suas características (altura do nó na árvore varia!), é ainda possível derivar matematicamente um limite mais restrito para a criação da heap, e assumir que o tempo de createHeap é **O(n)**.
(Veja **Cormen** para maiores detalhes)

Heap Sort

Algoritmo de ordenação baseado no conceito de *heaps*, criado por J.W.J. Williams e Robert W. Floyd (1964).

Heap Sort é um algoritmo baseado em comparação e faz parte da família dos algoritmos de ordenação por seleção.

Seu tempo de execução é muito bom em conjuntos ordenados aleatoriamente e seu desempenho no pior caso é igual ao seu desempenho no caso médio: **O(n log n)**.

Utiliza o mesmo princípio do **Selection Sort**:

- **Seleciona** o maior elemento do vetor;
- **Troca o elemento** selecionado com o que está na última posição do vetor, colocando-o em sua posição definitiva;
- **Repete a operação** de seleção e troca no restante dos elementos, até que todos estejam em seus respectivos lugares.

Qual a diferença?

O Heap Sort utiliza um <u>método mais eficiente para a seleção</u> do maior elemento do vetor, através da criação de uma estrutura de dados: a **heap**.

- No Selection Sort que vimos na aula, selecionávamos sempre o maior (ou menor) elemento através de uma busca linear no vetor, em tempo O(n).

No método *Selection Sort*, são necessárias por volta de n-1 comparações para encontrar o menor valor entre os elementos restantes, não ordenados. É uma busca linear em um vetor que vai diminuindo de tamanho.

Para este novo algoritmo, essa condição somente é necessária na primeira varredura:

- Na primeira varredura montaremos a *heap* e, a partir daí (n vezes, uma para cada elemento a ser colocado em ordem), somente iremos realizar trocas e ajustar a *heap* para manter sua restrição de ordem.
- Lembre-se de que a altura da árvore será sempre [log n].
- Passaremos de $O(n^2)$ para $O(n \log n)$.

Existem três procedimentos básicos para o método **Heap Sort** que são:

1. A construção da heap a partir de um vetor não ordenado, que é realizada pela função **create_heap**;
2. A garantia da propriedade da heap de máximo (ou mínimo), que é realizada pela função **heapfy**;
3. A ordenação, que é realizada pela função **heapsort**, depois que a heap está completamente construída.

Já sabemos criar uma heap a partir de um vetor desordenado, em tempo O(nlogn) com a função create_heap, mas como fazemos para ordenar os números a partir da heap construída?

1. O método **heapsort** irá começar com uma chamada do método **create_heap** para o vetor completo v[0 ... n-1]
2. Como é uma heap de máximo, o maior valor estará na posição v[0].
3. Sabendo disso, realizamos a troca entre v[0] e v[n-1], colocando o primeiro valor em seu lugar definitivo: a última posição.
4. Agora, o subvetor v[0 ... n-2] precisa passar de novo pelo mesmo processamento, uma vez que a troca pode ter gerado uma violação na regra da heap de máximo.
5. A função **heapfy** é chamada para o elemento na posição 0, utilizando apenas os elementos restantes da *heap*, ou seja, decrementando-se o tamanho do vetor em 1, já que a última posição já contém o valor definitivo.
6. O processo se repete n-1 vezes, até que todos estejam em seus lugares definitivos.

```c
void heapsort(int *vet, int n) {
  int aux;
  create_heap(vet, n);
  for(int i = n-1; i > 0; i--) {
    // troca primeiro (maior) com ultimo
    aux = vet[0];
    vet[0] = vet[i];
    vet[i] = aux;

    n--; // diminui tamanho da heap (ultimo elemento já ordenado)
    heapfy(vet, n, 0); // ajusta heap (leva elemento do topo até seu lugar)
  }
}
```

O custo total do algoritmo é dado por:

- Uma chamada de create_heap() com custo **O(n logn)**;
- A troca de elementos possui custo **O(1)**;
- Cada uma das n-1 chamadas para o método heapfy custa **O(log n).**

Portanto, o custo total é:

- T(n) = O(n logn) + (n-1)O(log n)

T(n) = **O(n log n)**

A complexidade do heap sort é **O(n log n)** para **qualquer caso**.

A ordenação por heap é um algoritmo excelente, mas uma boa implementação de Quick ou Merge Sort normalmente o superam na prática.

Não obstante, a estrutura de dados *heap* propriamente dita possui muitas utilidades, sendo uma das mais populares é sua utilização como fila de prioridades eficiente. Neste tipo de estrutura, podemos:

- Obter o seu valor máximo em **$\Theta(1)$**;
- Extrair seu valor máximo em **$O(\log n)$** (ao remover, precisa recompor a heap);
- Inserir um novo elemento em, também, **$O(\log n)$**.

Conheça bem as **heaps**!

Hands-*On!*

Exercício:

Ordene o vetor abaixo, passo a passo, utilizando o **Heap Sort:**

V = {38, 27, 43, 3, 9, 82, 10}