Estruturas de Dados II

Heap Sort

Prof. Bruno Azevedo

Instituto Federal de São Paulo

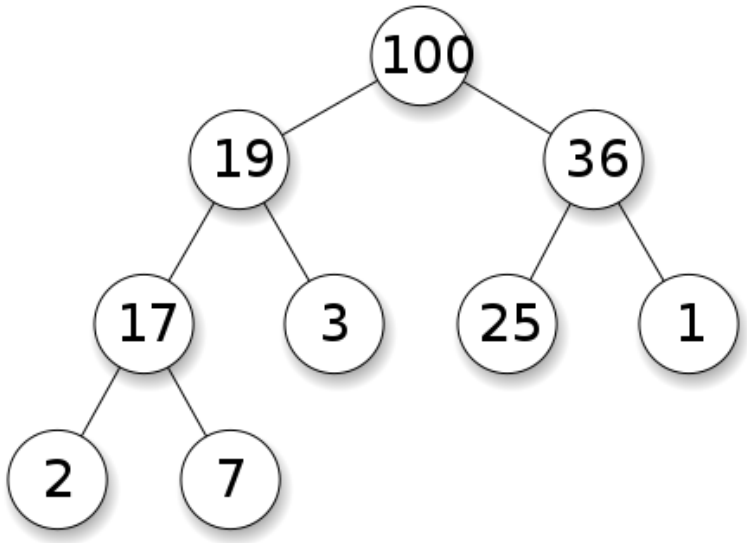
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Heap Sort

- Vamos conhecer mais um algoritmo para ordenação de elementos: o **Heap Sort**.
- Possui tempo de execução de $O(n \cdot \log_2 n)$.
- Apresenta mais uma técnica de design de algoritmos: o uso de uma estrutura de dados para gerenciar informações.
- Vocês já conhecem a estrutura, ensinada em ED1, o Heap.

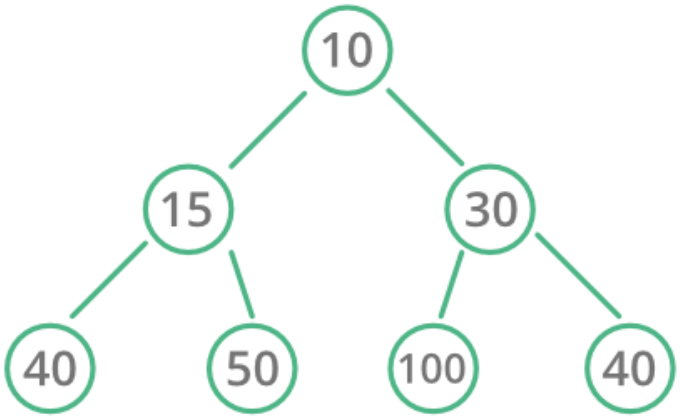
Heap

- O **Heap** é uma estrutura de dados baseada em árvore que satisfaz a propriedade de Heap.
- Existem diversos tipos de Heap, abaixo temos um Heap Binário, que assume a forma de uma árvore binária.
- Uma árvore binária é uma estrutura de dados em árvore na qual cada nó tem no máximo dois filhos.

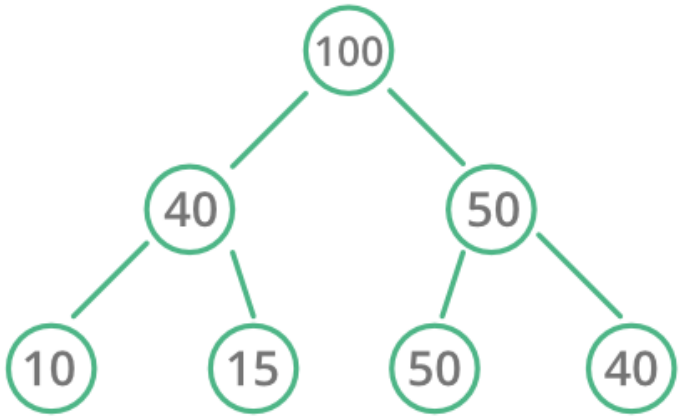


Heap

- Existem dois tipos de Heaps Binários: o Heap Máximo e o Heap Mínimo.
- Em um Heap Máximo, a chave (valor) de qualquer nó pai é maior ou igual às chaves dos nós filhos.
- Em um Heap Mínimo, a chave do pai é menor ou igual às chaves dos filhos.



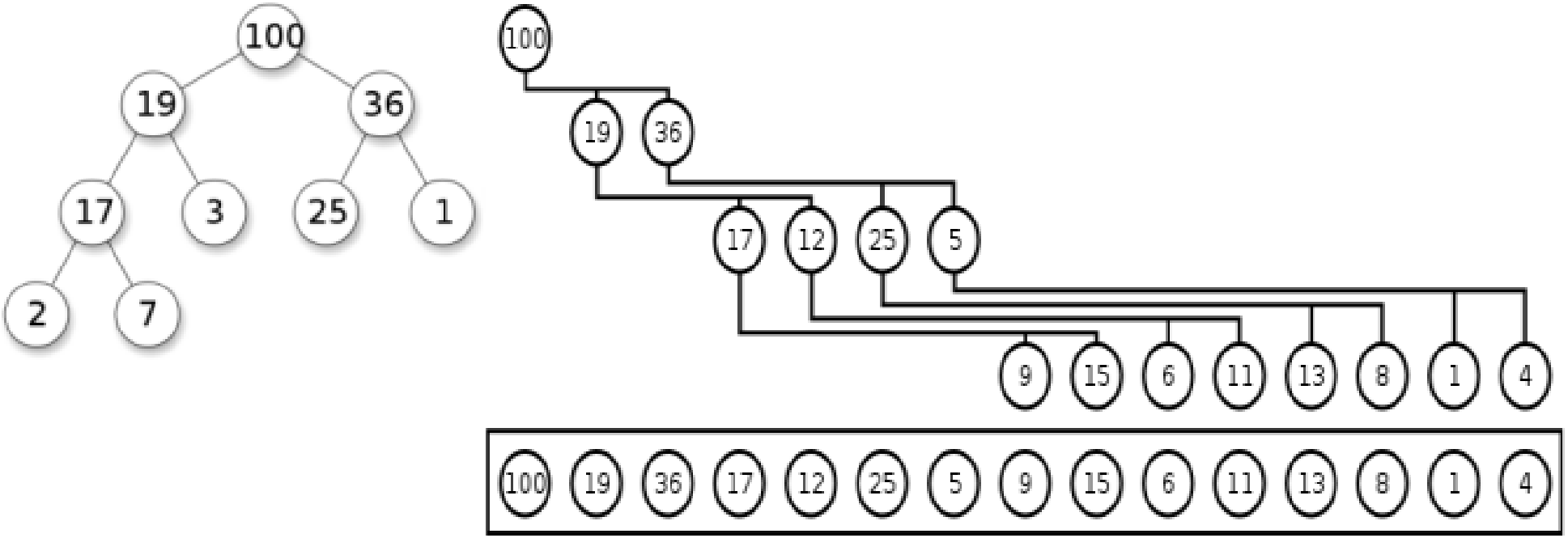
Heap Mínimo



Heap Máximo

Heap

- Abaixo podemos visualizar como um Heap Binário é armazenado em um vetor.

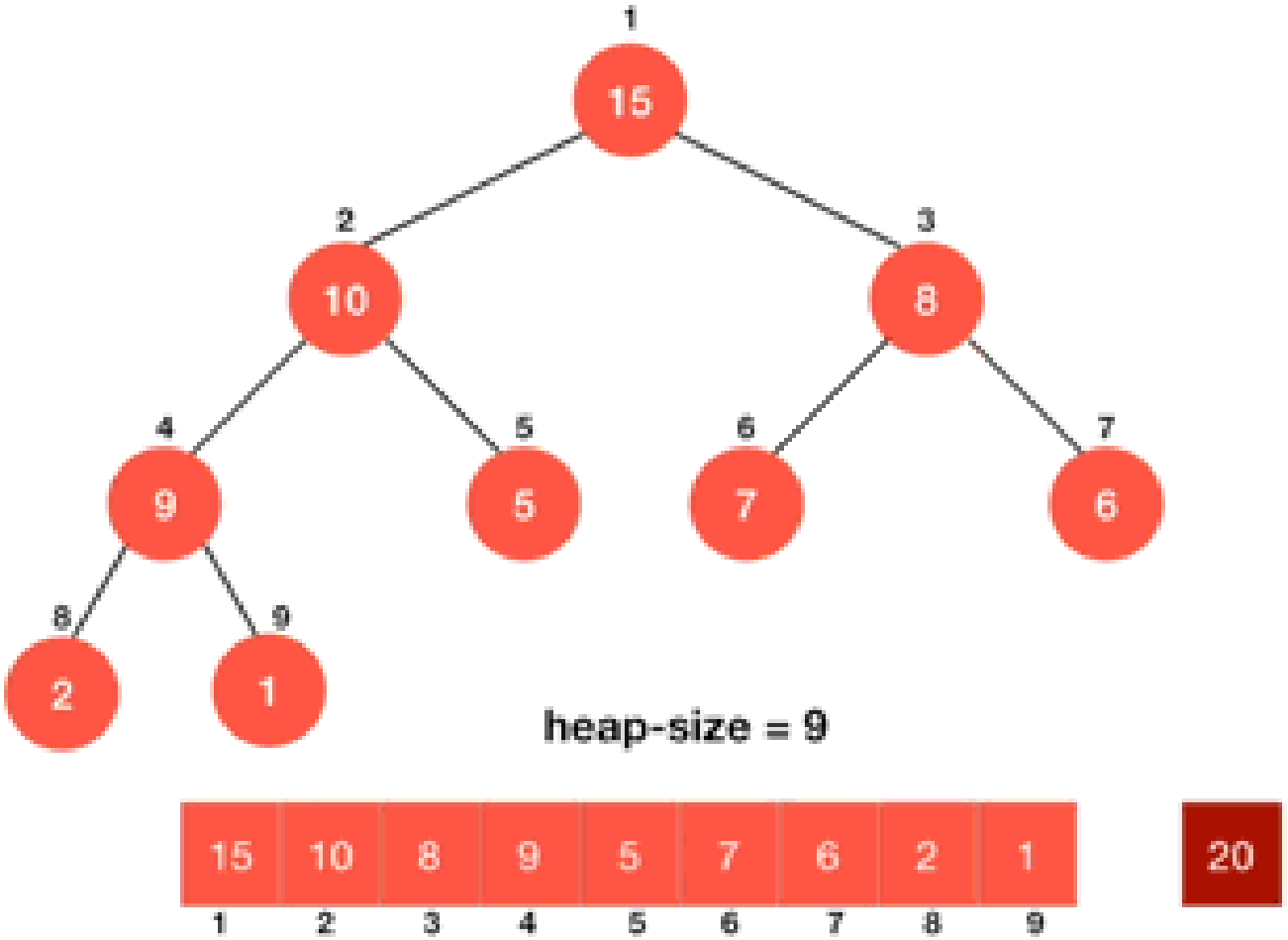


Funcionamento do Heap Sort

- A primeira etapa é transformar o vetor de entrada em um Heap máximo.
- Ou seja, reorganizar os elementos de forma que o maior elemento do vetor esteja na raiz e que todas as subárvores também satisfaçam a propriedade de Heap máximo.
- Em seguida, o elemento na raiz é trocado com o último elemento, colocando-a em sua posição correta.
- O tamanho do Heap é reduzido em um e o algoritmo Heapify é chamado na raiz para garantir a propriedade de Heap.
- Este processo de troca e reordenação é repetido até que todos os elementos estejam ordenados.

Funcionamento do Heap Sort

- Visualmente:



Análise do Tempo de Execução do Heap Sort

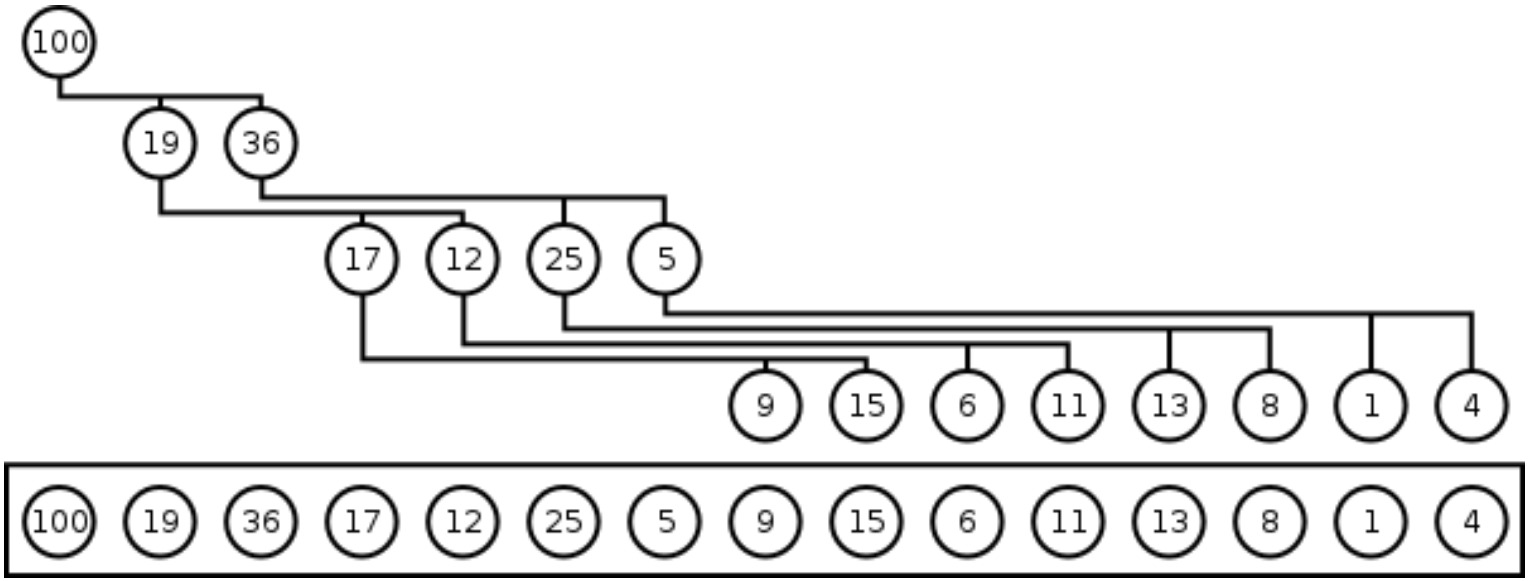
- O primeiro passo é a transformação do vetor em um Heap Máximo.
- Vamos ver como é feita essa transformação e analisar o tempo de execução assintoticamente.

Análise do Tempo de Execução do Heap Sort

- Vocês podem argumentar que uma solução simples seria utilizar a função de inserção de elementos em um Heap que vocês aprenderam em ED1.
- Ou seja, em vez de fazer uma construção *in-place*, podemos criar um novo vetor contendo o Heap, inserindo elemento por elemento do vetor original.
- Esta solução terá a complexidade de tempo de $O(n \cdot \log_2 n)$.
- Afinal, a ação de inserção possui complexidade de $O(\log_2 n)$, e precisaremos efetuar essa operação n vezes.
- Vamos aprender uma abordagem mais eficiente.

Construindo um Heap Máximo

- Mas antes, precisam entender algo sobre a **estrutura** do Heap representado em um vetor.



- Observem como as **folhas** estão todas localizadas a partir da posição $\left\lfloor \frac{n}{2} \right\rfloor$ do vetor (iniciando em zero).
- No exemplo, a partir do índice 7, já que temos 15 elementos.

Construindo um Heap Máximo

- O algoritmo `ConstroiHeapMaximo` transforma um vetor em um Heap Máximo.

```
void ConstroiHeapMaximo(int A[], int n) {
    for(int i = n/2 - 1; i >= 0; i--)
        Heapify(A, n, i);
}
```

- O algoritmo adota uma abordagem **bottom-up**, começando a construção do Heap a partir dos nós-pais mais baixos.
- Iniciamos a partir da posição `n/2 - 1` (o último nó pai) e subimos até a raiz.

Construindo um Heap Máximo

- O algoritmo Heapify é utilizado para garantir que a propriedade do Heap seja mantida.

```
void Heapify(int A[], int tamanho, int i) {
    int esquerda = 2 * i + 1;
    int direita = 2 * i + 2;
    int maior = i; \\ A raiz está em i.
    if(esquerda < tamanho && A[esquerda] > A[maior])
        maior = esquerda;
    if(direita < tamanho && A[direita] > A[maior])
        maior = direita;
    if(maior != i) {
        int temp = A[i];
        A[i] = A[maior];
        A[maior] = temp;
        Heapify(A, tamanho, maior);
    }
}
```

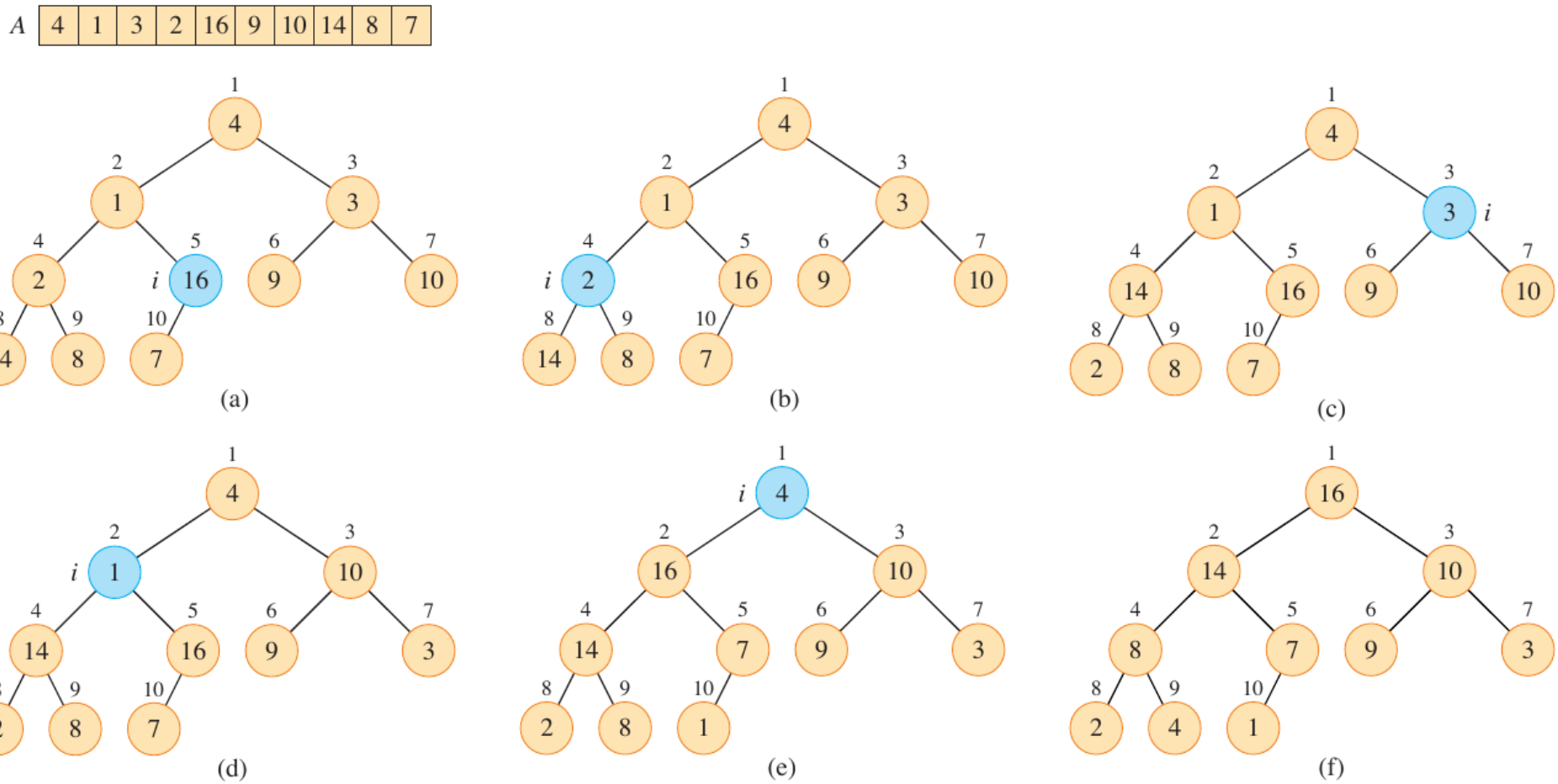
- A recursão permite que o algoritmo desça pela árvore, ajustando as posições dos nós até que todos satisfaçam a condição do heap.
- Ou seja, o algoritmo realiza trocas **para baixo**, garantindo a propriedade de Heap para a subárvore afetada.

Construindo um Heap Máximo

- Portanto, o algoritmo `ConstroiHeapMaximo` constrói o Heap de baixo para cima, iniciando nos nós-pais mais baixos.
- Mas o algoritmo `Heapify` desce recursivamente pela subárvore do nó em questão, ajustando as posições dos nós.

Construindo um Heap Máximo

- Vamos transformar o vetor A em um Heap Máximo.



- Observem que os passos (d) e (e) possuem várias chamadas do algoritmo Heapify até ajustar o Heap.

Análise do Tempo de Execução de ConstroiHeapMaximo

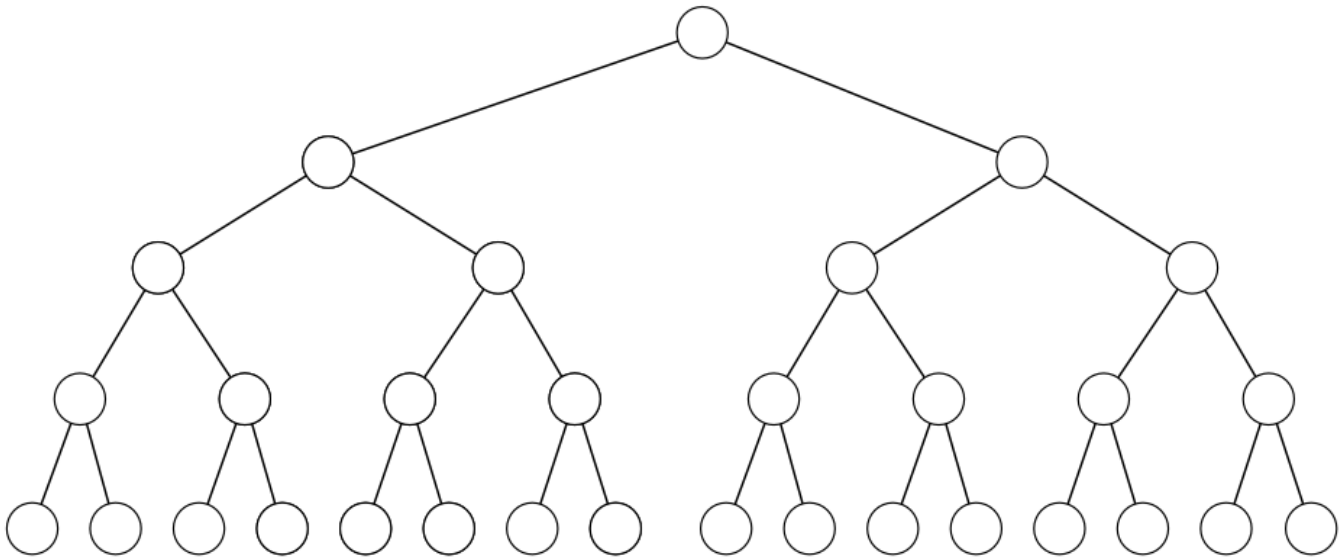
- Um limitante superior para o tempo de execução de ConstroiHeapMaximo é $O(n \cdot \log_2 n)$.
- Afinal, cada chamada para `Heapify` custa $O(\log_2 n)$ de tempo [1], e `ConstroiHeapMaximo` faz $O(n)$ dessas chamadas.
- Assim, o tempo de execução é $O(n \log_2 n)$.
- Esse limitante superior é correto, mas não é o mais preciso possível.

[1] Vimos isso em ED1, e logo revisitaremos essa análise.

Análise do Tempo de Execução de ConstroiHeapMaximo

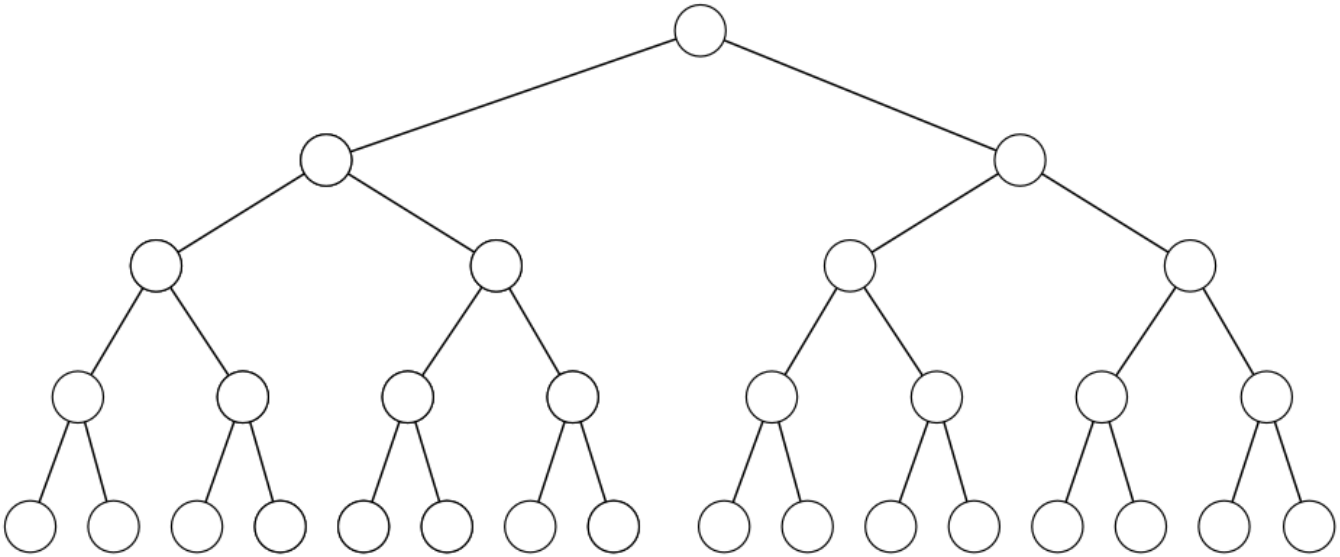
- A análise de complexidade computacional que nos obterá um limitante mais preciso exigirá algum conhecimento externo ao curso.
- Mas quero que, inicialmente, entendam a intuição de porquê não precisarmos de $O(n \log_2 n)$.

Análise do Tempo de Execução de ConstroiHeapMaximo



- Os nós na parte inferior do Heap (mais próximos das folhas) precisam de menos comparações, enquanto os nós na parte superior precisam de mais.
- Ou seja, a quantidade de trabalho necessária para manter a propriedade do Heap diminui conforme nos movemos para baixo na árvore.
- Aproximadamente metade dos nós não têm filhos e não precisam de Heapify.

Análise do Tempo de Execução de ConstroiHeapMaximo



- Para um nó de altura h, o tempo de `Heapify` é $O(h)$.
- Lembre que um Heap possui altura $O(\log_2 n)$, e teremos $\log_2 n$ comparações e trocas apenas se precisarmos percorrer toda a altura do Heap.

Análise do Tempo de Execução de ConstroiHeapMaximo

- Podemos então derivar uma expressão que represente o custo total do nosso algoritmo `ConstroiHeapMaximo`.

- n é o número total de nós.
- h é a altura do Heap.
- $\frac{n}{2^{h+1}}$ é o número de nós no nível h (por quê?).
- $c \cdot h$ representa o custo do Heapify em um nó no nível h (ou seja, $O(h)$).

$$\sum_{h=0}^{\log_2 n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot c \cdot h \tag{1}$$

Análise do Tempo de Execução de ConstroiHeapMaximo

$$\sum_{h=0}^{\log_2 n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot c \cdot h \tag{2}$$

- Vamos simplificar o termo removendo o teto e a soma no expoente do denominador. Podemos então afirmar que:

$$\leq \sum_{h=0}^{\log_2 n} \frac{n \cdot c \cdot h}{2^h} \tag{3}$$

- Vale notar que estamos buscando um limitante superior, portanto, é aceitável fazer aproximações que resultem em valores maiores ou iguais à soma original.

Análise do Tempo de Execução de ConstroiHeapMaximo

- Como c e n não dependem da variável h que está sendo somada, podemos removê-los do somatório.

$$= c \cdot n \sum_{h=0}^{\log_2 n} \frac{h}{2^h} \tag{4}$$

- Para facilitar o cálculo da série, vamos estender a soma até o infinito..

$$\leq c \cdot n \sum_{h=0}^{\infty} \frac{h}{2^h} \tag{5}$$

- A expressão geral para a soma de uma série geométrica infinita é: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$.
- Vamos reescrever o somatório de forma que ele se aproxime dessa expressão.

$$= c \cdot n \sum_{h=0}^{\infty} h \cdot (\frac{1}{2})^h \tag{6}$$

- Observem que essa expressão não é exatamente igual à fórmula da série geométrica simples, pois há o fator multiplicador h que precisamos considerar.

Análise do Tempo de Execução de ConstroiHeapMaximo

- Vamos derivar ambos os lados da expressão: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ para obter o seguinte resultado:

$$\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2} \tag{7}$$

- Multiplicando por x:

$$\sum_{k=0}^{\infty} k x^{k} = \frac{x}{(1-x)^2} \tag{8}$$

- Agora podemos aplicar a fórmula na série geométrica infinita obtida previamente.

Análise do Tempo de Execução de ConstroiHeapMaximo

- A série geométrica em questão:

$$c \cdot n \sum_{h=0}^{\infty} h \cdot (\frac{1}{2})^h \tag{9}$$

- Aplicando a fórmula obtida (x = 1/2).

$$\leq c \cdot n \cdot \frac{1/2}{(1-\frac{1}{2})^2} \tag{10}$$

- Obtemos:

$$\leq c \cdot n \cdot 2 = O(n). \tag{11}$$

Análise do Tempo de Execução do Heap Sort

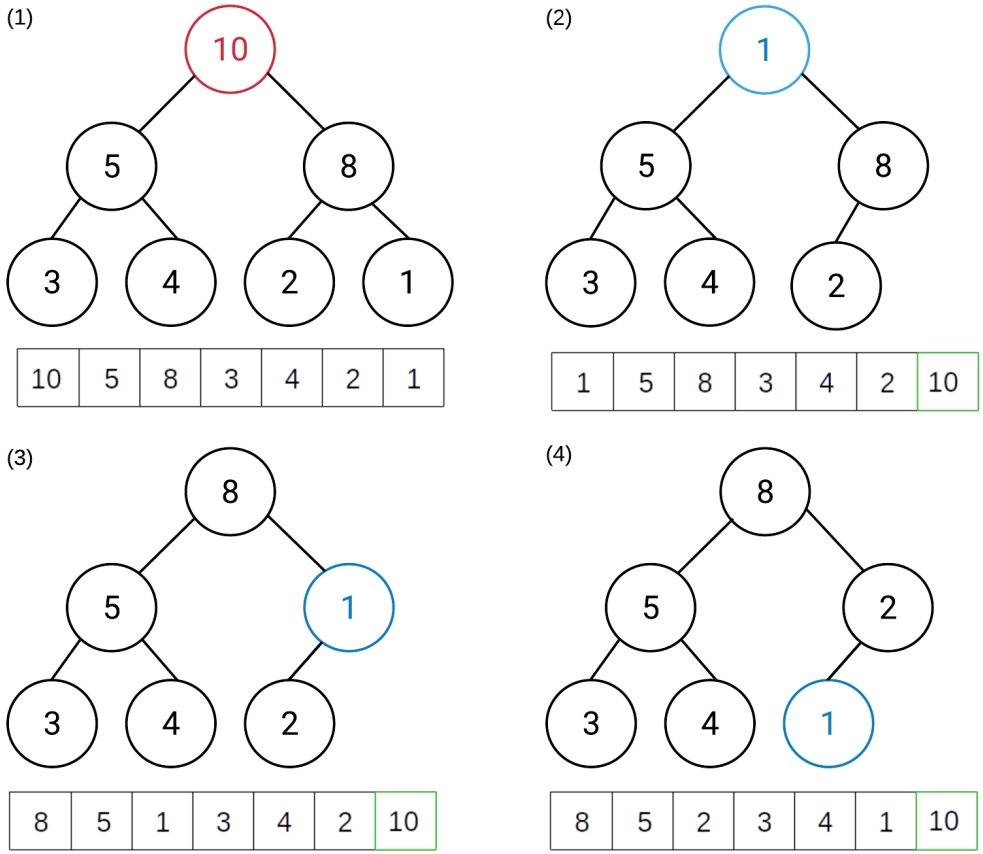
- A segunda etapa é a troca do elemento raiz com o último elemento do vetor e sua subsequente remoção do Heap.
- Estamos montando um vetor ordenado no fim do Heap.
- O Heap é reduzido em um elemento, ou seja, não considerará o elemento que foi colocado no vetor ordenado.
- Vamos acompanhar como é feita essa etapa e analisar o tempo de execução assintoticamente.

Troca do elemento de valor máximo com o último elemento

- A segunda etapa consiste dos seguintes passos:
  1. Trocar a raiz pelo último nó do Heap.
  2. O novo nó raiz pode violar a propriedade de Heap, portanto, chamamos o algoritmo `Heapify`. Ele compara a nova raiz com seus filhos e, se necessário, troca-a com o filho máximo, garantindo que o maior valor esteja na posição correta.
  3. Esse processo é repetido até a propriedade do Heap máximo seja restaurada, ou seja, até que o todos os nós sejam maiores ou iguais aos seus filhos.
     Notem que o Heap teve seu tamanho reduzido em um, portanto, a antiga raiz **não será considerada** nesse processo.
- Esta etapa será repetida até que o Heap tenha zero elementos.

Troca do elemento de valor máximo com o último elemento

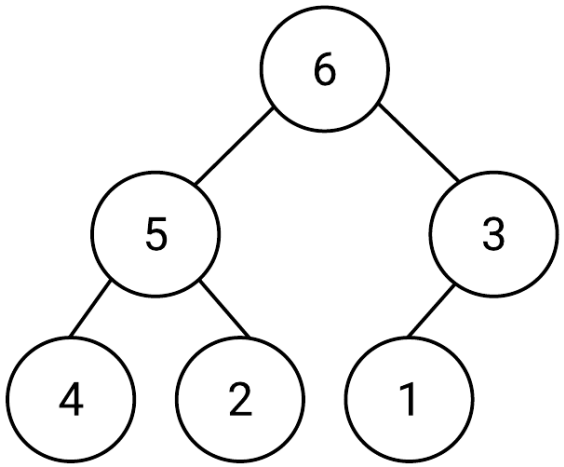
- Vamos acompanhar um passo da segunda etapa:



- Após a remoção da raiz do Heap, a propriedade de Heap é garantida pelo algoritmo recursivo Heapify.
- O Heapify efetuará no máximo h trocas de nós, onde h é a altura do Heap.
- A altura do Heap h é dada pela distância máxima (número de arestas)

Análise do Tempo de Execução do Heapify

- Portanto, para sabermos o número de comparações, precisamos saber a altura do Heap.
- Considere os níveis iniciando em zero na raiz. Em um nível i do Heap pode haver até 2^i nós.



- Notem que o último nível não necessariamente estará totalmente preenchido.

Análise do Tempo de Execução do Heapify

- Queremos descobrir, dado um número de n nós de um Heap, qual será o número máximo de níveis que ele pode possuir.
- Com esta informação, saberemos o número máximo de trocas, que será o número de níveis menos um.
- Ou seja, o número de **arestas** entre a raiz e a folha mais profunda do Heap.

Análise do Tempo de Execução do Heapify

- Sabemos que cada nível i terá até 2^i nós. Portanto, se temos k níveis em um Heap, teremos no máximo $2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$ nós no Heap.
- Isso é a soma de uma progressão geométrica (P.G.) finita. A soma dos termos de uma P.G. é dada por:

$$S = a_1 \left(\frac{r^n - 1}{r - 1} \right)$$

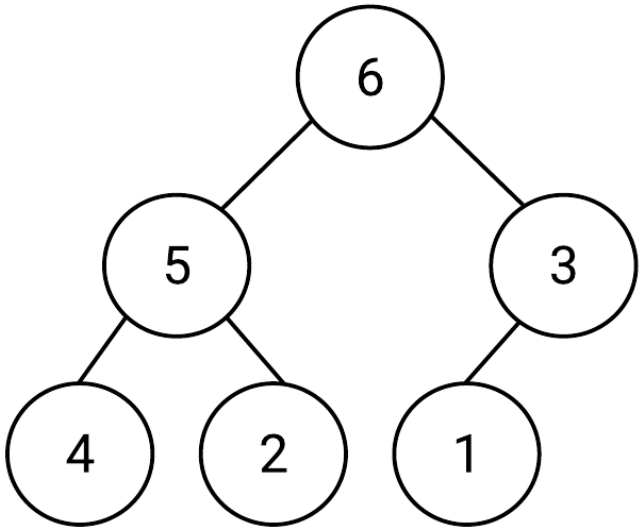
- Onde a_1 é o primeiro termo, r é a razão da P.G., e n é o número de termos, respectivamente.
- Substituindo na fórmula:

$$S = 2^0 \left(\frac{2^k - 1}{2 - 1} \right) = 2^k - 1$$

- Portanto, um Heap de k níveis possui até $2^k - 1$ nós.

Análise do Tempo de Execução do Heapify

- Se um Heap possui 3 níveis, ele terá até $2^3 - 1 = 7$ nós.
- Observem um Heap de 3 níveis.



Análise do Tempo de Execução do Heapify

- Em um Heap de n nós: $n = 2^k - 1$, onde k é o número de níveis. Mas precisamos obter o valor de k em função de n.
- Para este fim, vamos isolar k.

$$n = 2^k - 1$$

$$n + 1 = 2^k$$

- Vamos aplicar log em ambos os lados.

$$\log_2(n+1) = k$$

- Portanto, um heap com n nós possui $\log_2(n+1)$ níveis.
- Por exemplo, se um Heap possuir 7 nós, ele terá $\log_2 8 = 3$ níveis.

Análise do Tempo de Execução do Heapify

- Portanto, o número máximo de comparações será $(\log_2(n+1) - 1) \times C$, onde C é uma constante representando o número máximo de comparações efetuadas pelo `Heapify` em uma execução.
- Vamos utilizar a notação-O.
- Podemos descartar a constante aditiva (-1) e a constante multiplicativa (C).
- Mas temos algo que não conseguimos desmembrar facilmente: $\log_2(n+1)$
- Vamos pensar o que fazer aqui.

Análise do Tempo de Execução do Heapify

- Vamos expressar $\log(n+1)$ como uma multiplicação:

$$\log(n+1) = \log\left(n\left(1+\frac{1}{n}\right)\right) \tag{12}$$

- Aplicando a propriedade $\log(ab) = \log(a) + \log(b)$:

$$\log(n+1) = \log(n) + \log\left(1+\frac{1}{n}\right) \tag{13}$$

- Note que, à medida que n aumenta, o termo $\frac{1}{n}$ se torna cada vez menor, o que faz com que $\log_2\left(1+\frac{1}{n}\right)$ também diminua.
- Portanto, para n suficientemente grande, o termo $\log_2\left(1+\frac{1}{n}\right)$ tende a zero (PS: $\log_2(1) = 0$).
- Desse modo, para n grande, podemos aproximar $\log_2(n+1) \approx \log_2 n$, resultando em um tempo de execução de $O(\log_2 n)$ para o algoritmo Heapify.

Análise do Tempo de Execução do Heap Sort

- Vamos rever o funcionamento do algoritmo Heap Sort.
- A primeira etapa é transformar o vetor de entrada em um Heap máximo.
- Na segunda etapa, trocamos a raiz com o último elemento, o tamanho do Heap é reduzido em um e o algoritmo Heapify é chamado na raiz para garantir a propriedade de Heap.
- Este processo de troca e reordenação é repetido até que todos os elementos estejam ordenados.

Análise do Tempo de Execução do Heap Sort

- A construção do Heap a partir do vetor original possui tempo $O(n)$.
- O Algoritmo `Heapify` será chamado $n - 1$ vezes e o algoritmo `Heapify` executa em $O(\log_2 n)$, ou seja, a segunda etapa possui complexidade computacional de $O(n \cdot \log_2 n)$.
- Portanto, o tempo de execução do algoritmo Heap Sort é de: $O(n) + O(n \cdot \log_2 n) = O(n \cdot \log_2 n)$.

Exercícios

1. Aplique manualmente o algoritmo `ConstroiHeapMaximo` no vetor V = {5, 3, 17, 10, 84, 19, 6, 22, 9} e exiba todos os passos intermediários.
2. Aplique manualmente o Heap Sort no vetor V = {5, 13, 2, 25, 7, 17, 20, 8, 4} e exiba todos os passos intermediários.
3. Qual é o tempo de execução do Heap Sort em um vetor de comprimento n que já está ordenado em ordem crescente?
4. Implemente o Heap Sort em C/C++.
5. É viável a implementação do algoritmo Heap Sort utilizando um Heap Mínimo? Analise as possíveis vantagens e desvantagens dessa abordagem em relação à utilização de um Heap Máximo.