Complexidade de Algoritmos
Lista de Exercícios 2

Vinícius Takeo Friedrich Kuwaki

7 de Fevereiro de 2020

1 BuildHeap

A execução do BuildHeap para o vetor determinado no enunciado, possui um laço de repetição que itera sob o vetor, aplicando a função **heapify()** para os nós (ou no caso, posições do vetor) 3,2,1 e 0. O passo a passo é apresentado nas Figuras 1,2,3,4,5,6 e 7.

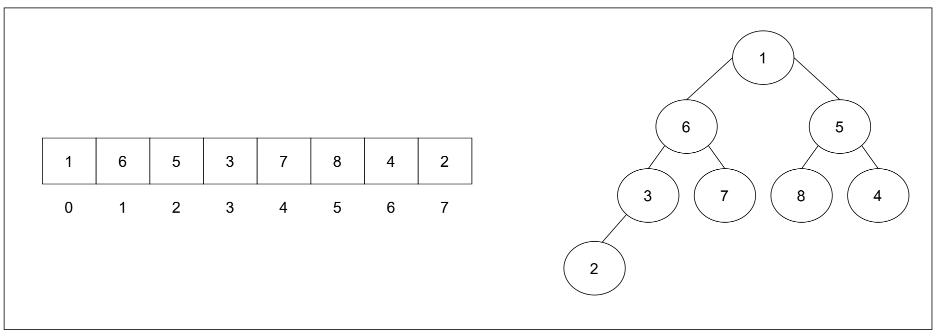

Figura 1: Conjunto de dados originais

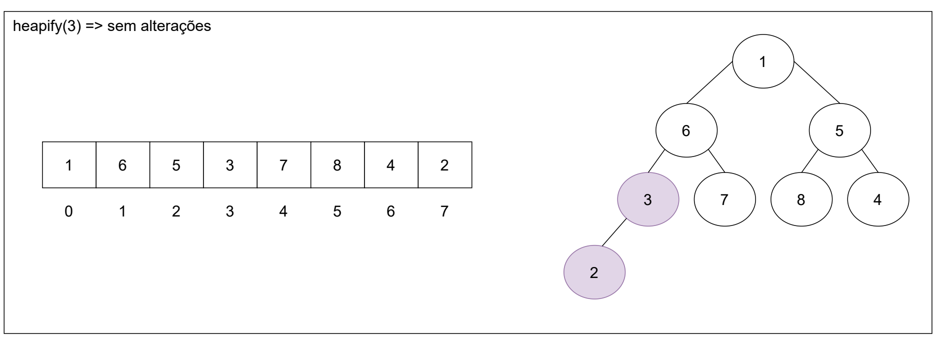

Figura 2: Aplicando o **heapify()** na posição 3, como seu filho, o elemento na posição 7 (2*3+1) é menor, a chamada recursiva do **heapify()** não ocorre, logo, essa chamada não gera alterações. Os elementos comparados estão destacados em roxo.

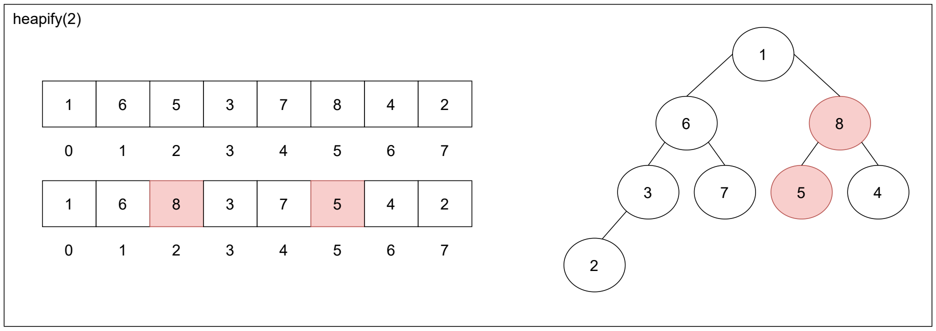

Figura 3: Voltando a função **buildHeap()**, agora, invoca-se a função **heapify()** para o elemento da posição 2, como seu filho a direita é maior, ocorre a troca entre os elementos destacados em vermelho. Como a sub-árvore agora encontra-se em heapMáximo, não há a necessidade de chamadas recursivas.

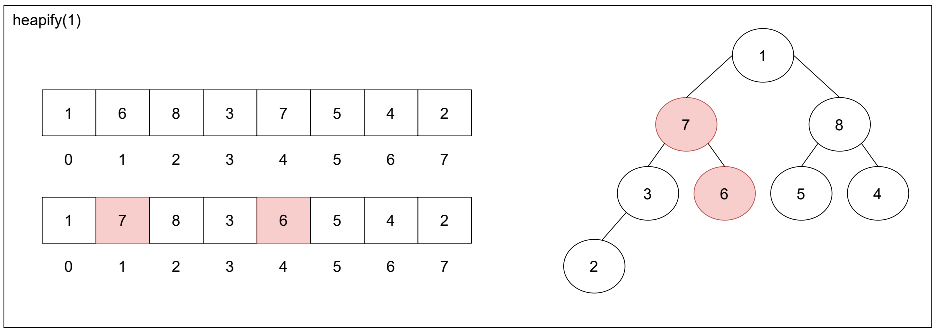

Figura 4: Novamente na função **buildHeap()**, o elemento a ser *heapificado* é o da posição 1. Como seu filho a direita é maior, ocorre o *swap* destacado na cor vermelha. Ao final, a sub-árvore encontra-se em heapMaximo.

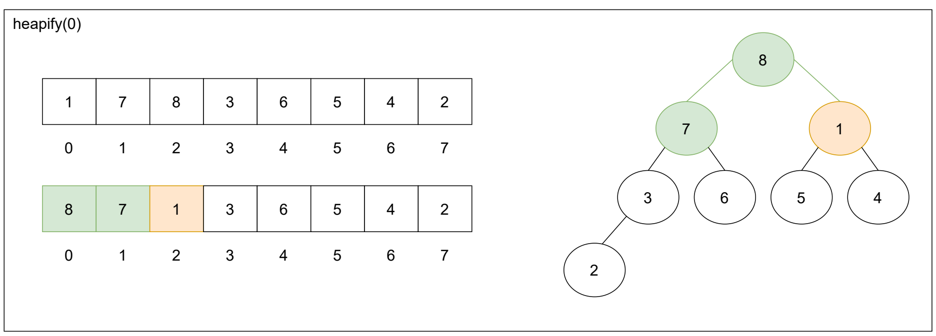

Figura 5: Ao retornar a função **buildHeap()**, aplica-se o **heapify()** na raiz, ou seja, no elemento 0. Ocorrendo uma troca tripla entre os elementos, o resultado está destacado em verde e amarelo. Como a sub-árvore em 1 não se encontra em heapMaximo, uma chamada recursiva é aplicada nesse nó, ajustando-o.

2 HeapSort: Complexidade de Tempo e Espaço

Para o algoritmo do HeapSort, precisamos analisar as funções **heapSort()**, **buildHeap()** e **heapify()** e suas recorrências. A função **buildHeap()** possui complexidade **O(n)**, enquanto que a função **heapify()** possui complexidade de tempo**O(log n)**. Ao analisar o algorito da função **heapSort()**, é possível observar

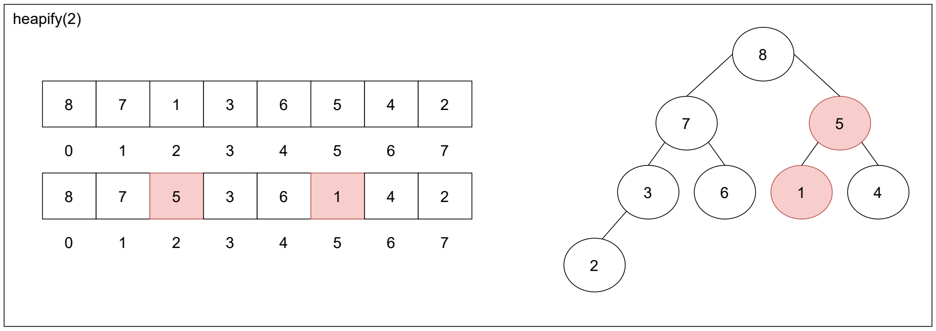

Figura 6: Ao aplicar o **heapify()** na posição 2, para atingir-se o heapMaximo, os nós realizam um *swap*, este destacado em vermelho.

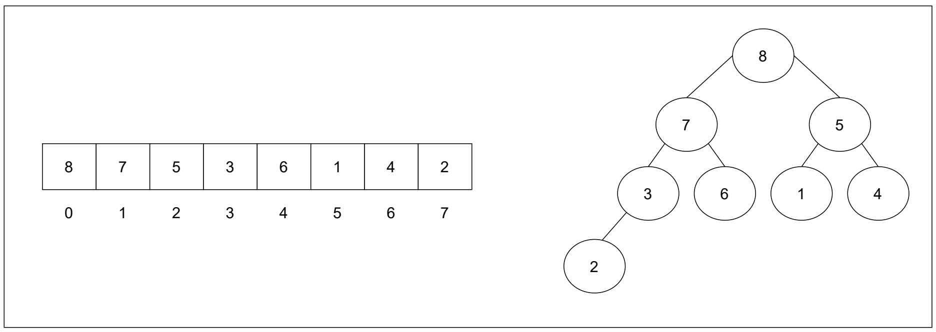

Figura 7: Por fim, esse é o resultado obtido.

que o laço de repetição realiza (n-1) chamadas da função **heapify()**, esta que possui complexidade de tempo **O(log n)**. Logo, a complexidade do laço é **O(n log n)**. Como ela domina assintoticamente a chamada do **buildHeap()**, que possui **O(n)** de complexidade de tempo, a função **heapSort()** possui complexidade de tempo igual a **O(n log n).** Na Figura 8 é possível ver o cálculo da complexidade.

```
void heapSort(int *a, int n)
{    O(n log n) > O(n) => O(n log n)
    int i, aux; O(1)
    buildHeap(a, n); O(n)
    for (i = n - 1; i > 0; i--)
    {
        O(1) aux = a[0]; a[0] = a[i]; a[i] = aux;
             heapify(a, i, 0); O(log n)
    }
}    (n-1)*O(log n) => (n-1)(log n) => O(n log n)
```

Figura 8: Cálculo da Complexidade de tempo do HeapSort.

2.1 Complexidade de espaço

Já no cálculo da complexidade de espaço, é necessário considerar a quantidade de memória máxima utilizada em certo ponto da execução. Para isso, na Figura 9, observaremos o laço *for*. Nele observa-se que o **heapify()**

possui complexidade **O(log n)**, entretanto, diferentemente da complexidade de tempo, o fator memória não é cumulativo como o tempo, logo, a memória utilizada em uma iteração é a mesma utilizada na seguinte. Isso implica que a complexidade de espaço do trecho *for* é **O(log n)**. Como **O(n)** (complexidade de espaço do buidHeap) domina assintoticamente **O(log n)**, tem-se que a complexidade de espaço é **O(n)**.

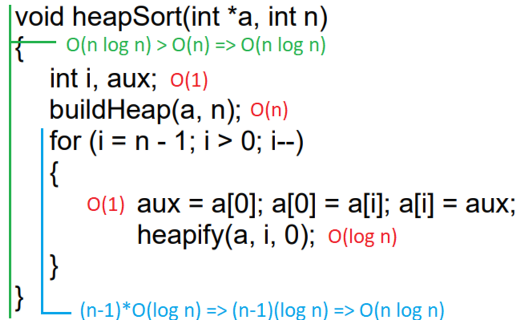

Figura 9: Cálculo da Complexidade de espaço do HeapSort.

2.2 Complexidade de Tempo para um conjunto contendo apenas elementos repetidos

Quando tem-se um conjunto de dados contendo apenas elementos iguais, por exemplo: $\{1, 1, 1, 1, ..., 1\}$, o cálculo da complexidade torna-se diferente. Para tal, é preciso analisar novamente a função **heapify()**. Na Figura 10 é possível ver o trecho contornado na cor verde. Ele quem determina a complexidade do algoritmo, visto que como todos os elementos de **a** são iguais, o primeiro laço *if* é ignorado, restando seu *else*. Nele, o valor da varíavel **maior** torna-se o valor de i. Como o todos os elementos são iguais, o *if* após tal *else* é ignorado, assim como a última verificação, em que **maior** é diferente de **i**. Tornando assim a complexidade do **heapify() O(1)**, já que chamadas recursivas não ocorrem.

Em seguida, podemos analisar a função **buildHeap()** (Figura 11). Como a complexidade de tempo do **heapify()** é **O(1)** e tal trecho repete-se $\frac{n-1}{2}$ vezes, eliminando as constantes, tem se que a complexidade de tempo do **buildHeap()** é O(n).

Por fim, tendo analisado as demais funções, é possível calcular a complexidade do **heapSort()** (Figura 12. A complexidade do **buildHeap()** já fora calculada (**O(n)**), logo, basta analisar o laço *for*. Nele é possível notar que a função **heapify()**, que possui complexidade **O(1)** se repete *n-1* vezes, fazendo com que a complexidade do laço *for* seja **O(n)** (ao eliminar as constantes). A complexidade da função **heapSort()**, ao somar as demais é **O(n) + O(n)**, isto é, **2*O(n)**. Eliminando-se as constantes novamente, chega-se em **O(n)**.

Logo, é possível afirmar que a complexidade do **heapSort()** quando todos os elementos são iguais, para a implementação do enunciado é **O(n)**.

```
void heapify (int *a, int n, int i)
{
    int e, d, maior, aux;  O(1)

    e = esquerda(i);  O(1)
    d = direita(i);  O(1)
    if (e < n && a[e] > a[i])  O(1)
            maior = e;
    else
            maior = i;  O(1)
    if (d < n && a[d] > a[maior])
            maior = d;
    if (maior != i)
    {
            aux  = a[i];
            a[i] = a[maior];
            a[maior] = aux;
            heapify(a, n, maior);
    }
}
```

Figura 10: Cálculo da Complexidade do Heapify quando todos os elementos são repetidos.

```
void buildHeap(int *a, int n)
{
    int i;  O(1)

    for (i = (n-1)/2; i >= 0; i--)
            heapify(a, n, i);  O(1)
}   O(1) * (n-1)/2 => O(n)
```

Figura 11: Cálculo da Complexidade do BuildHeap quando todos os elementos são repetidos.

```
void heapSort(int *a, int n)
{
    int i, aux;  O(1)
    buildHeap(a, n); O(n)
    for (i = n - 1; i > 0; i--)
    {
        O(1) aux = a[0]; a[0] = a[i]; a[i] = aux;
        O(1) heapify(a, i, 0);
    }
}   O(1) * (n-1) => O(n)
```

Figura 12: Cálculo da Complexidade do HeapSort quando todos os elementos são repetidos.