Universidade Federal da Paraíba

Disciplina: Estrutura de Dados e Complexidade de Algoritmos

Professor: Gilberto Farias

João Pessoa, 28 de Março de 2019

Aluno: Vinícius Matheus Veríssimo da Silva – 20191000933

Heapsort

1 - Introdução

Desenvolvido em 1964 por Robert W. Floyd e J.W.J Williams, o Heapsort é uma técnica de ordenação por comparação baseado na estrutura de dados *Binary Heap* (ou Heap Binária, em tradução livre). É similar ao Selectionsort, onde o primeiramente busca-se o maior elemento de um vetor e o coloca-se no final dele, esse processo é repetido para todos os outros elementos restantes [1].

A *Binary Heap* é essencialmente uma Árvore Binária onde os itens são armazenados numa ordem especial de modo que o nó pai é maior (ou menor) que os seus dois filhos [1]. Quando o nó pai possui um valor maior que os filhos, diz-se que a *Binary Heap* está no formato de *max heap* (heap máxima), caso o nó pai possua um valor menor que os filhos, então diz-se que está no formato de *min heap* (heap mínima).

Apesar de se basear na representação de uma Árvore Binária, é possível realizar a *Binary Heap* sobre uma estrutura de lista. Nesse caso, seja uma lista de tamanho `n` e um elemento dessa lista de índice `i`, tal que `i < n/2`, este será um nó pai de modo que os seus dois filhos serão os elementos de índices `2 * i + 1` e `2 * i + 2`.

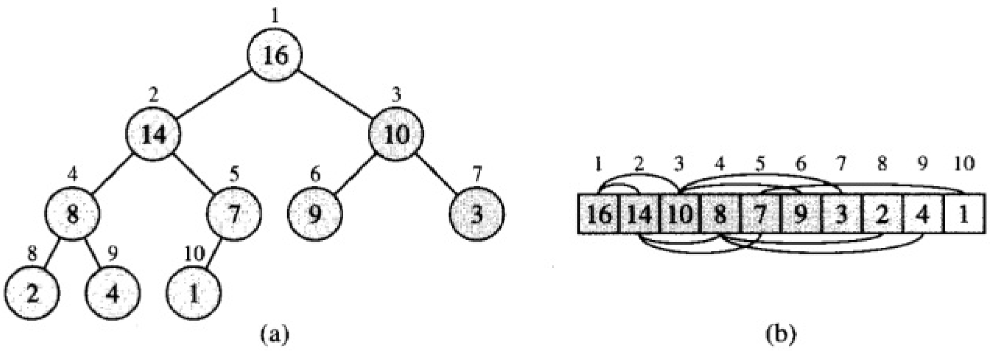

Figura 1: a) árvore em heap máximo; b) vetor que originou a árvore. Fonte: CORMEN et al. [2]

2 - Algoritmo

O algoritmo do Heapsort pode ser dividido em 3 etapas:

1. A partir de um vetor `v` desordenado de `n` elementos, garantir que o vetor está no estado de heap máximo.

2. Pegar o primeiro elemento do vetor, `v[0]`, que é o maior valor do vetor e raiz da Árvore Binária, e trocá-lo com o elemento da na última posição válida do vetor.

3. Repetir e processo `2` para os `n - 1` elementos restantes.

2.2 - Funções

O Heapsort possui 3 funções principais: **MAX-HEAPIFY (MH)**, **BUILD-MAX-HEAP (BMH)** e **HEAPSORT (HS)**. Estas que serão mais exploradas a seguir.

2.2.1 - MAX-HEAPIFY

A função **MAX-HEAPIFY** é responsável por manter o estado de *max heap*, ou seja, verifica se o pai de índice `i` é maior que os seus filhos de índices `LEFT(i)` (índice do filho esquerda) e `RIGHT(i)` (índice do filho da direita), caso negativo, troca-se o nó pai com o filho que possui o maior valor, ocorrendo a troca, o processo é aplicado novamente na nova posição do nó pai para garantir que a árvore atingirá o estado de heap máximo. O procedimento é repetido até que não haja mais nenhuma troca, ou seja, a árvore estará em estado de *max heap*. Vale ressaltar que o **MH** considera que as sub-árvores de raízes `LEFT(i)` e `RIGHT(i)` já estão em estado de heap máximo

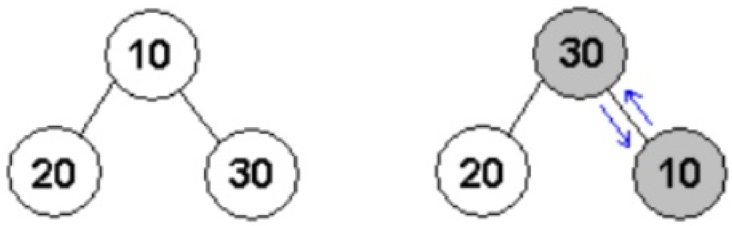

Figura 2: Antes e depois da aplicação do MAX-HEAPIFY em uma árvore. Fone: Santanché [4]

```
MAX-HEAPIFY (V, i, last_position)
 left <- 2*i + 1
 right <- 2*i + 2
 largest = i

 se left <= last_position e V[left] > V[i] então
  largest <- left

 se right <= last_position e V[right] > V[largest] então
  largest <- right

 se largest != i então

  trocar(A[i], A[largest])
  MAX-HEAPIFY(A, largest, last_position)
```

Complexidade

Podemos verificar que a complexidade do **MAX-HEAPIFY** depende da altura `h` da árvore [3]. No pior dos casos o nó raiz possuirá o menor valor da árvore e serão necessárias `h` trocas para que o nó esteja no lugar correto, ou seja, é um processo de complexidade *O(h)*. Como trata-se de uma árvore binária, a altura máxima da mesma é igual a `lg n`, onde `n` é quantidade de nós da árvore. Logo, a complexidade de todo o processo é de *O(log n)*.

2.2.2 - BUILD-MAX-HEAP

A função **BUILD-MAX-HEAP** é responsável pela primeira aplicação do processo de **MAX-HEAPIFY** no vetor de entrada. Logo, sua função é deixar toda árvore em estado de *max heap*. Ele é apenas aplicado na primeira metade do vetor, já que a segunda metade é apenas composta por nós folhas.

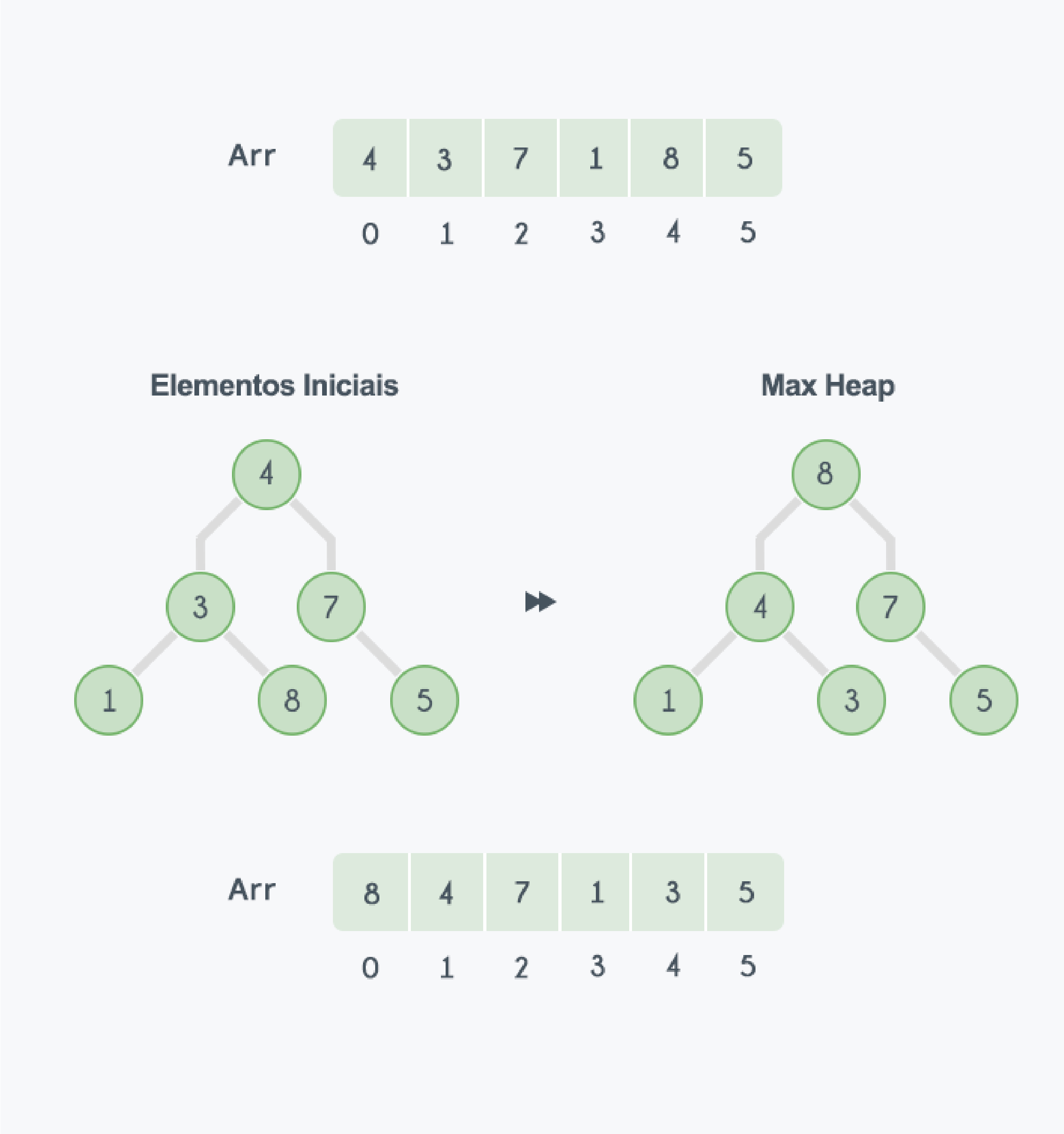

Figura 3: Antes e depois da aplicação do BUILD-MAX-HEAP em um vetor. Fonte: Hackearth [5]

```
BUILD-MAX-HEAP (V)
 n <- tamanho(V)
 para i <- round(n/2) até 0 faça
  MAX-HEAPIFY(V, i, n)
 fimpara
```

Complexidade

Como a ideia do **BUILD-MAX-HEAP** é executar o processo de **MAX-HEAPIFY** na primeira metade do vetor, é fácil de se pensar que com o **MH** tendo complexidade *O(lg n)* sendo executado `n` vezes, logo, o **BMH** teria complexidade igual a *O(n lg n)*, apesar de ter uma lógica correta, essa resposta está errada.

Sendo uma heap essencialmente uma árvore binária, algumas propriedades desse tipo de estrutura podem ser atribuídas a ela, como:

- a altura de uma árvore binária ser no máximo `lg n`, onde `n` é a quantidade total de nós da árvore;
- outra propriedade diz respeito à quantidade máxima de nós em cada nível da árvore, que pode ser calculada pela equação $\frac{n}{2^{h+1}}$, onde `n` é quantidade total de nós e `h` altura na qual está sendo

buscada a quantidade máxima de nós.

Considerando que o **BMH** é executado em todos nós de cada altura, sua complexidade se modifica com a altura `h` do nó, sendo ela no mínimo 0 e no máximo `lg n`. Logo a complexidade da **BMH** é dada pelo somatório das complexidades em cada altura da árvore, sendo assim:

$$\sum_{h=0}^{lg\,n} \frac{n}{2^{h+1}} \cdot O(h) = O\left(n \sum_{h=0}^{lg\,n} \frac{h}{2^{h+1}}\right)$$

A complexidade do **BMH** é dada por uma série, onde `n` é quantidade total de nós e `h` é a altura do nó da árvore. Simplificando essa série, temos:

$$n \sum_{h=0}^{lg\,n} \frac{h}{2^{h+1}} = n \sum_{h=0}^{lg\,n} \frac{h}{2^{h} \cdot 2} = \frac{n}{2} \sum_{h=0}^{lg\,n} \frac{h}{2^{h}}$$

Fazendo com que a altura máxima seja infinita, temos:

$$\frac{n}{2} \sum_{h=0}^{lg\,n} \frac{h}{2^{h}} \leq \frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^{h}}$$

A série infinita é uma série harmônica que tem como resultado o valor `2`, logo:

$$\frac{n}{2} \sum_{h=0}^{lg\,n} \frac{h}{2^{h}} \leq \frac{n}{2} \cdot 2 \Rightarrow \frac{n}{2} \sum_{h=0}^{lg\,n} \frac{h}{2^{h}} \leq n$$

Assim, podemos concluir que a complexidade correta para o **BMH** é *O(n)*.

HEAPSORT

A função **HEAPSORT** é responsável por aplicar os processos descritos no início da seção **2**. Primeiro é executado o processo de **BUILD-MAX-HEAP** no vetor de entrada, para que ele fique em estado de *max heap*. Depois troca-se o elemento da primeiro posição com o elemento da última posição válida, logo após isso, é aplicado novamente o processo de **MAX-HEAPIFY**, mas apenas na primeira posição do vetor, para garantir que ele estará novamente em estado de *max heap*. Nesse momento desconsidera-se a última posição válida, pois contém o último valor retirado. Repete-se os procedimentos de troca e execução do **MH** até que última posição válida seja a segunda posição do vetor.

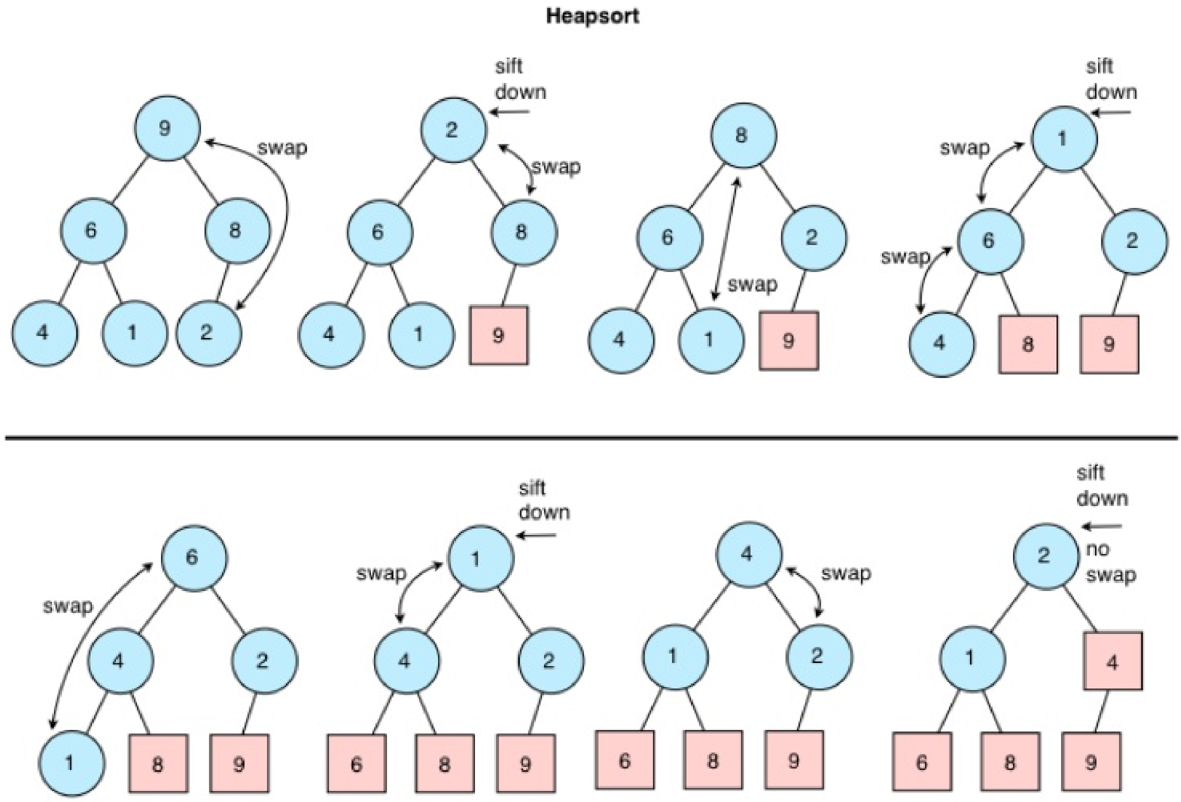

Figura 4: Exemplo da execução do Heapsort. Fonte: Le [6]

```
HEAPSORT (V)
 BUILD-MAX-HEAP (V)
 para i <- tamanho(V) até 1 faça
  trocar(V[0], V[i])
  MAX-HEAPIFY(V, 0, i - 1)
 fimpara
```

Complexidade

O cálculo da complexidade do **HEAPSORT** é bem simples, dadas as explicações das complexidades do **BUILD-MAX-HEAP** e **MAX-HEAPIFY**. O **BMH** é executado uma vez e tem complexidade igual a *O(n)*. No no início do *loop* são executadas `n - 1` trocas, onde `n` é a quantidade de elementos do vetor. No *loop* também é aplicado o **MH**, que tem complexidade igual a *O(lg n)*, é executado `n - 1` vezes, sempre no primeiro elemento do vetor. Com isso podemos determinar que a complexidade do **HEAPSORT** é igual a *O(n lg n)*.

Referências

[1] GeeksForGeeks. HeapSort. Disponível em: https://www.geeksforgeeks.org/heap-sort/ . Acessado em: 26/02/2019

[2] CORMEN, Thomas H. et al. Algoritmos: teoria e prática 2. Campus. 2002

[3] GeeksForGeeks. Time Complexity of building a heap. Disponível em: https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/ . Acessado em: 20/03/2019

[4] SANTANCHÉ, André. Capítulo 6 - Heapsort. Disponível em:

http://www.ic.unicamp.br/~meidanis/courses/mo417/2003s1/aulas/2003-03-12.html . Acessado em: 20/03/2019

[5] Hackearth. Heapsort Tutorials & Notes. Disponível em: https://www.hackerearth.com/pt-br/practice/algorithms/sorting/heap-sort/tutorial/ . Acessado em: 23/03/2019

[6] LE, James. Heapsort, Mergesort and Conver Hull. Disponível em: https://jameskle.com/writes/sorting-algorithms . Acessando em: 27/03/2019