

Projeto e Análise de Algoritmos

Flávio L. C. de Moura

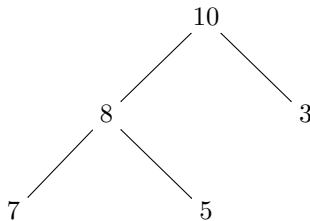
09 de maio de 2023

Estudaremos um novo algoritmo de ordenação baseado em comparação de chaves, mas bem diferente dos algoritmos (*insertion sort* e *mergesort*) vistos anteriormente. Este novo algoritmo, conhecido como *heapsort*, possui tempo de execução $O(n \cdot \log n)$, como *mergesort*, e o processo de ordenação é feito *in place* (como em *insertion sort*) (veja [1]). Portanto *heapsort* combina as vantagens de *insertion sort* e *mergesort*. A estrutura de dados utilizada por este algoritmo é conhecida como *heap*:

Definição 1. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

1. T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;
2. Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;
3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

Os itens 1 e 2 da definição acima caracterizam a chamada de **propriedade do corpo do heap**. O item 3 corresponde a **propriedade de heap**. Assim, em um *heap* o nó mais à direita pode ter apenas um filho à esquerda, mas não pode ter somente um filho à direita. Todos os outros nós internos possuem dois filhos.



A grande vantagem da estrutura de *heap* é que ela permite a implementação das operações de inserção de um novo elemento (ou uma nova chave), e extração do maior elemento (maior chave) em tempo logarítmico. Note que em um vetor (ou em uma lista) contendo n elementos, a inserção pode ser feita em tempo constante, mas a extração do maior elemento vai exigir, no pior caso, uma busca em todo o vetor (ou lista), o que tem custo linear. A estrutura de *heap* suporta simultaneamente as duas operações, e como veremos, de forma assintoticamente mais eficiente do que em listas ou vetores.

Um *heap* binário pode ser implementado como um subvetor de um vetor A , onde somente os elementos em $A[1..A.\text{heap-size}]$ ($0 \leq A.\text{heap-size} \leq A.\text{length}$) são elementos válidos do *heap*. A raiz do *heap* é $A[1]$, e dado o índice i de um nó, o índice do filho à esquerda (resp. direita) é dado por $2i$ (resp. $2i + 1$). Por fim, o índice do nó correspondente ao pai do nó de índice i é dado por $\lfloor i/2 \rfloor$.

Alguns autores chamam a estrutura definida acima de *heap* de máximo (ou *max-heap*), isto é, um *heap* onde todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \geq A[i]$. Nesta linha, com um ajuste na propriedade de *heap* podemos definir um *heap* de mínimo (ou *min-heap*):

Em um *min-heap* todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \leq A[i]$.

Desta forma, o maior (resp. menor) elemento de um *max-heap* (resp. *min-heap*) é armazenado na raiz, e a subárvore com raiz em um determinado nó contém apenas valores que são menores ou iguais (resp. que são maiores ou iguais) ao valor deste nó. O algoritmo *heapsort* utiliza *max-heaps*, e portanto o primeiro passo do algoritmo será transformar o vetor A de entrada em um *max-heap*. Este trabalho é feito pelo algoritmo Build-Max-Heap a seguir:

```

1  $A.heap-size \leftarrow A.length$ ;
2 for  $i = \lfloor A.length/2 \rfloor$  downto 1 do
3   | Max-Heapify( $A, i$ );
4 end

```

Algoritmo 1: Build-Max-Heap(A)

Exercício 2. *Mostre que na representação vetorial de um heap com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$.*

O algoritmo Build-Max-Heap constrói o *max-heap* de baixo para cima a partir do primeiro vértice que não é uma folha, e o algoritmo Max-Heapify(A, i) reconstrói um *max-heap* a partir de uma árvore cuja raiz $A[i]$ seja o único elemento que precise ser reposicionado, ou seja, as subárvores com raiz $A[2i]$ e $A[2i + 1]$ já são *max-heaps*:

```

1  $l = 2i$ ;
2  $r = 2i + 1$ ;
3 if  $l \leq A.heap-size$  and  $A[l] > A[i]$  then
4   |  $largest = l$ ;
5 end
6 else
7   |  $largest = i$ ;
8 end
9 if  $r \leq A.heap-size$  and  $A[r] > A[largest]$  then
10  |  $largest = r$ ;
11 end
12 if  $largest \neq i$  then
13  | exchange  $A[i]$  with  $A[largest]$ ;
14  | Max-Heapify( $A, largest$ );
15 end

```

Algoritmo 2: Max-Heapify(A, i)

Observe que cada uma das subárvores com raiz nas posições $2i$ e $2i + 1$ têm, no máximo, $2n/3$ elementos:

Exercício 3. *Mostre que, em um heap com n elementos e raiz $A[i]$, cada uma das subárvores com raiz em $2i$ e $2i + 1$ têm, no máximo, $2n/3$ elementos.*

Portanto o tempo de execução de Max-Heapify é dado pela recorrência

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

que, pelo teorema mestre, tem solução $O(\lg n)$.

Qual o tempo de execução do procedimento Build-Max-Heap(A)? Temos a seguinte cota superior, considerando um *heap* com n elementos: $\sum_{i=1}^{n/2} O(\lg n) = O(\lg n \cdot \sum_{i=1}^{n/2} 1) = O(\lg n \cdot (n/2)) = O(n \cdot \lg n)$. No entanto, esta cota, apesar de correta, não é a mais precisa que podemos encontrar. De fato, se observarmos que:

1. A altura do n -ésimo elemento de um *heap* é igual a $\lfloor \lg n \rfloor$.
2. Um *heap* com n elementos possui, no máximo, $\lceil n/2^{h+1} \rceil$ nós com altura h .

Então, observando que Max-Heapify tem complexidade $O(h)$ quando executado em um nó de altura h , concluímos que o tempo de execução de Build-Max-Heap(A), assumindo que A possui n elementos, tem a seguinte cota superior: $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^{h+1} \rceil) = O(n)$, pois $\sum_{h=0}^{\lfloor \lg n \rfloor} h/2^{h+1} \leq \sum_{h=0}^{\infty} h/2^{h+1}$, que por sua vez converge. Assim, um *heap* pode ser construído em tempo linear.

Exercício 4. Prove a seguinte invariante de laço, e conclua que o algoritmo Build-Max-Heap é correto:

No início de cada iteração do laço **for** (linhas 2-4), cada nó nas posições $i + 1, i + 2, \dots, n$ é a raiz de um max-heap.

O algoritmo *heapsort* recebe como argumento um vetor A qualquer contendo $n > 0$ elementos, e inicialmente o transforma em um *max-heap*. Neste momento, sabemos que a raiz do *heap* contém o maior elemento do vetor A , que pode então ser movido para sua posição correta. Em seguida, decrementamos o tamanho do *heap* em uma unidade, e repetimos o processo:

```

1 Build-Max-Heap(A);
2 for  $i = A.length$  downto 2 do
3   exchange  $A[1]$  with  $A[i]$ ;
4    $A.heap-size = A.heap-size - 1$ ;
5   Max-Heapify( $A, 1$ );
6 end
```

Algoritmo 3: Heapsort(A)

A complexidade de Heapsort(A) no pior caso, se A é um vetor com $n > 0$ elementos, é $O(n) + \sum_{i=2}^n O(\lg n) =$

$$O(n) + O(\lg n \cdot \sum_{i=2}^n 1) = O(n) + O((n-1) \cdot \lg n) = O(n \cdot \lg n).$$

Exercício 5. Mostre que a complexidade de tempo de Max-Heapify no pior caso é $\Omega(\lg n)$.

Exercício 6. Mostre que a complexidade de tempo de Heapsort no pior caso é $\Omega(n \lg n)$, e conclua que a complexidade de Heapsort é $\Theta(n \cdot \lg n)$.

Exercício 7. Prove a correção do algoritmo Heapsort utilizando a seguinte invariante:

No início de cada iteração do laço **for** (linhas 2-6), o subvetor $A[1..i]$ é um max-heap que contém os i menores elementos do vetor $A[1..n]$, e o subvetor $A[i+1..n]$ está ordenado e contém os $n - i$ maiores elementos do vetor $A[1..n]$.

Referências

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.