Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

Iniciaremos esta aula com o seguinte exercício:

> Ordene os n elementos de um vetor $A[1..n]$ da seguinte forma: encontre o maior elemento de A e troque este elemento com o elemento $A[n]$. Em seguida, encontre o maior elemento de $A[1..n-1]$ e troque este elemento com $A[n-1]$, e assim por diante até que o vetor A esteja ordenado. Escreva o pseudocódigo do seu algoritmo e mostre que ele está correto.

Solução

Solução força-bruta:

Algorithm 1: selection-sort(A)

```
1 for i = n downto 2 do
2 |   max ← i;
3 |   for j = i − 1 downto 1 do
4 |   |   if A[j] > A[max] then
5 |   |   |   max ← j;
6 |   |   end
7 |   end
8 |   swap A[i] and A[max];
9 end
```

Análise assintótica: $T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} = \Theta(n^2)$.

A correção pode ser provada a partir da seguinte invariante:

> Antes de cada iteração do laço **for** indexado por i (linhas 1-8), o subvetor $A[i+1..n]$ está ordenado e contém os $(n-i)$ maiores elementos do vetor A.

1 O algoritmo *heapsort*

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 $O(n. \log n)$, como *mergesort*, e o processo de ordenação é feito *in place* (como em *insertion sort*). Portanto *heapsort* combina as vantagens de *insertion sort* e *mergesort*. A estrutura de dados utilizada por este algoritmo é conhecida como *heap*:

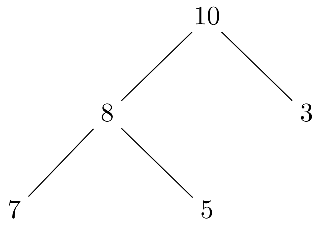
Definição 1.1. *Um* heap *(binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:*

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;*

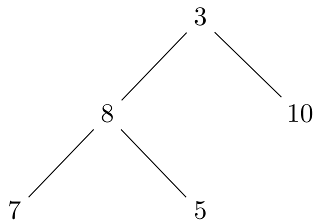
*flaviomoura@unb.br

3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

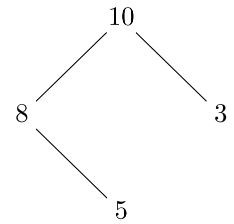
Os itens 1 e 2 da definição acima caracterizam a chamada de **propriedade do corpo do *heap***. O item 3 corresponde a **propriedade de *heap*** (de máximo). A propriedade 2 corresponde a dizer que uma enumeração dos nós de um *heap* deve começar de cima para baixo, *i.e.* a partir da raiz do *heap*, e da esquerda para a direita. 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 figura abaixo é um *heap*:



A figura abaixo não é um *heap* porque não satisfaz a propriedade 3:



A figura abaixo não é um *heap* porque não satisfaz a propriedade 2:



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 $A[1..A.\text{heap-size}]$ ($0 \leq A.\text{heap-size} \leq A.length$) 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 $2i$ (resp. $2i+1$), enquanto que o índice do nó correspondente ao pai do nó de índice i é igual a $\lfloor i/2 \rfloor$.

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 $A[\lfloor i/2 \rfloor] \geq A[i]$. Analogamente, podemos definir um *heap* de mínimo (ou *min-heap*) considerando que 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 em um *max-heap*. Este trabalho é feito pelo algoritmo a seguir:

Algorithm 2: Build-Max-Heap(A)

```
A.heap-size = A.length;
for i = ⌊A.length/2⌋ downto 1 do
    Max-Heapify(A,i);
end
```

Exercício 1.2. *Mostre que na representação vetorial de um* heap *com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.*

Exercício 1.3. *Faça a análise assintótica do algoritmo* Build-Max-Heap.

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*:

Algorithm 3: Max-Heapify(A,i)

```
l = 2i;
r = 2i + 1;
if l ≤ A.heap-size and A[l] > A[i] then
    largest = l;
end
else
    largest = i;
end
if r ≤ A.heap-size and A[r] > A[largest] then
    largest = r;
end
if largest ≠ i then
    exchange A[i] with A[largest];
    Max-Heapify(A,largest);
end
```

Exercício 1.4. *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.*

Considerando o fato estabelecido no exercício anterior, temos que 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 $O(\lg n)$.

Só prossiga com a leitura após resolver o exercício 1.3, e então compare a sua solução com o parágrafo a seguir.

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

1. A altura de um *heap* contendo n elementos 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: $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n. \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^h \rceil) = O(n)$, pois $\sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \leq \sum_{h=0}^{\infty} h/2^h$, que por sua vez converge. Assim, um *heap* pode ser construído em tempo linear.

Exercício 1.5. *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 $i+1$, $i+2$, ..., 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:

Algorithm 4: Heapsort(A)

```
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
```

A complexidade de Heapsort(A) no pior caso, se A é um vetor com $n > 0$ elementos, é $O(n) + \sum_{i=2}^{n} O(\lg n) = O(n) + O(\lg n. \sum_{i=2}^{n} 1) = O(n) + O((n-1). \lg n) = O(n. \lg n)$.

Exercício 1.6. *Mostre que a complexidade de tempo de Max-Heapify no pior caso é* $\Omega(\lg n)$*.*

Exercício 1.7. *Mostre que a complexidade de tempo de* Heapsort *no pior caso é* $\Omega(n \lg n)$*, e conclua que a complexidade de* Heapsort *é* $\Theta(n. \lg n)$*.*

Exercício 1.8. *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* $A[1..n]$*.*

2 Leitura complementar:

1. [2] (Capítulo 6)
2. [1] (Capítulo 6)
3. [3] (Capítulo 6, secão 6.4)

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.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[3] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.