Capítulo 6

Heapsort

Considere novamente o problema de ordenar os $n \geq 1$ elementos de um vetor A. O algoritmo `selection_sort` faz esse trabalho selecionando, a cada iteração, o menor elemento do vetor A e o colocando na sua posição final:

```
def selection_sort(arr):
     n = len(arr)
     for i in range(n):
          min_idx = i
          for j in range(i + 1, n):
               if arr[j] < arr[min_idx]:
                    min_idx = j
          arr[i], arr[min_idx] = arr[min_idx], arr[i]
     return arr
```

Observe que a mesma estratégia de ordenação pode selecionar o maior elemento do vetor. Qual é a complexidade deste algoritmo? A operação básica é a comparação realizada dentro do `for` interno, então:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n - i - 1) = \sum_{i=0}^{n-1} i = \frac{(n-1).n}{2} = \Theta(n^2).$$

Já estudamos soluções mais eficientes para ordenar vetores, mas agora vejamos o que ocorre se utilizamos a ideia de `selection_sort` sobre outra estrutura de dados.

6.1 Heaps

A estrutura de dados que estudaremos nesta seção é conhecida como *heap*:

Definição 11. *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:*

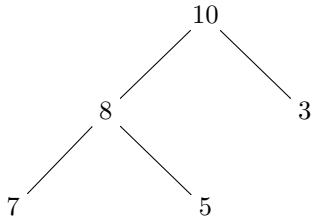
1. *T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;*
2. *Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível.*

Adicionalmente, se chave de cada nó é maior (resp. menor) ou igual do que a chave dos seus filhos então temos um *heap de máximo* (resp. *heap de mínimo*).

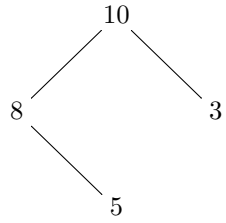
Segundo a propriedade 2, uma enumeração dos nós de um *heap* deve começar de cima para baixo, *i.e.* a partir da raiz do *heap*, e da esquerda para a direita.

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.

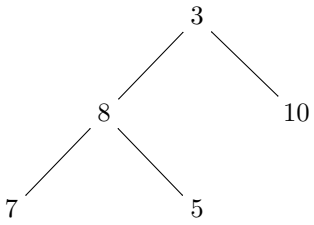
A figura abaixo é um *heap* de máximo:



A figura abaixo não é um *heap* porque não satisfaz a propriedade 1:



A figura abaixo não é um *heap* de máximo::



6.1.1 Propriedades

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. Obsereve que em um vetor (ou em uma lista), 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.

Implementação em 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 $\lfloor i/2 \rfloor$. Em um *heap* de máximo (ou *max-heap*), todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \geq A[i]$. Analogamente, em um *heap* de mínimo (ou *min-heap*), todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \leq A[i]$.

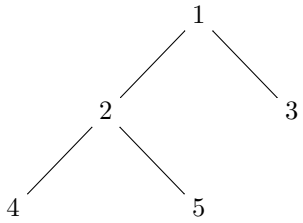
Desta forma, o maior (resp. menor) elemento de um *max-heap* (resp. *min-heap*) é armazenado na raiz, e a subárvore com raiz em um determinado nó contém apenas valores que são menores ou iguais

(resp. que são maiores ou iguais) ao valor deste nó.

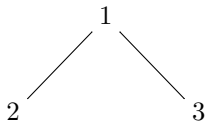
Exercício 65. *Qual é o número mínimo e o número máximo de elementos em um* heap *de altura h?*

Exercício 66. *Mostre que um* heap *com n elementos tem altura $\lfloor \lg n \rfloor$.*

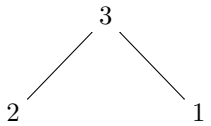
Qualquer vetor $A[1..n]$ pode ser visto como um *heap* com raiz $A[1]$ e os filhos do elemento $A[i]$ ($1 \leq i \leq n$), caso existam, estão na posição $2i$ e $2i+1$. Por exemplo, o vetor $[1, 2, 3, 4, 5]$ corresponde ao *heap*:



O trabalho que faremos a seguir, consiste em transformar um vetor qualquer em um *heap* de máximo. Observe que se o vetor for unitário não há nada a fazer. O trabalho também é simples se o vetor tiver apenas dois elementos: basta garantir que o maior deles está na primeira posição do vetor. A coisa fica mais interessante quando temos um vetor com pelo menos três elementos. Considere, por exemplo, o vetor $[1, 2, 3]$ que corresponde ao *heap*



Para transformá-lo em um *heap* de máximo basta trocarmos o maior dos filhos com a posição raiz:



Para generalizarmos esta ideia para um *heap* com n elementos, considere um *heap* com n elementos onde o elemento raiz é o único elemento que quebra a propriedade de *heap* de máximo. Neste caso, observe que a troca do elemento raiz pode quebra a propriedade de *heap* de máximo para o *subheap* com raiz na posição 2 ou 3, e portanto precisamos recursivamente fazer a reconstrução do *heap* de máximo:

```
1  l ← 2i;
2  r ← 2i + 1;
3  if l ≤ n and A[l] > A[i] then
4  |  largest ← l;
5  end
6  else
7  |  largest ← i;
8  end
9  if r ≤ n and A[r] > A[largest] then
10 |  largest ← r;
11 end
12 if largest ≠ i then
13 |  exchange A[i] with A[largest];
14 |  Max-Heapify(A,largest);
15 end
```

Algoritmo 12: Max-Heapify($A[1..n]$,i)

Qual a complexidade de Max-Heapify? Para respondermos esta pergunta precisamos antes resolver o seguinte exercício:

Exercício 67. *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

$$T(n) \leq T(2n/3) + \Theta(1) \tag{6.1}$$

que, pelo Teorema Mestre, tem solução $O(\lg n)$.

Agora podemos transformar um vetor qualquer em um *heap* de máximo aplicando o algoritmo Max-Heapify a cada nó que não seja folha. O exercício a seguir nos informa os nós que são folhas em um *heap* com n elementos:

Exercício 68. *Mostre que na representação vetorial de um* heap *com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.*

Assim, para construirmos um *heap* de máximo basta aplicarmos o algoritmo Max-Heapify do último nó que não é folha até a raiz:

```
1  for i = ⌊n/2⌋ downto 1 do
2  |  Max-Heapify(A,i);
3  end
```

Algoritmo 13: Build-Max-Heap($A[1..n]$)

Qual a complexidade de Build-Max-Heap? Considerando um *heap* com n elementos, temos $\sum_{i=1}^{\lfloor n/2 \rfloor} O(\lg n) = O(\lg n. \sum_{i=1}^{\lfloor n/2 \rfloor} 1) = O((n/2).\lg n) = O(n.\lg n)$.

A cota $O(n.\lg n)$, 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.

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 $i+1$, $i+2 \ldots$ n é a raiz de um *max-heap*.

6.2 O algoritmo *Heapsort*

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 14: Heapsort(A)

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

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 $A[1..n]$.

1. Leituras recomendadas

   (a) Capítulo 6 do livro do Cormen (Introduction to Algorithms);

   (b) Capítulo 6 do livro do Levitin (Introduction to the Design and Analysis of Algorithms).