Análise de Algoritmos – Tópico 11

Prof. Dr. Juliano Henrique Foleis

Introdução - Heaps e HeapSort

O tempo de execução do `Heapsort` é $O(n \lg n)$. No entanto o `Heapsort` é considerado mais eficiente que o `MergeSort` pois realiza a ordenação *in-place*, ou seja, somente uma quantidade constante de elementos são mantidos fora do vetor sendo ordenado. O `Heapsort` também introduz um conceito interessante: utilizar uma estrutura de dados especial para organizar informações sobre a ordenação: `heaps`.

Heaps

Uma heap é um vetor que pode ser interpretado como uma árvore binária quase completa. Cada nó da árvore corresponde a uma posição no vetor. Esta árvore é completa em todos os níveis, exceto o último, que é preenchido da esquerda para a direita. A raíz da árvore é $A[1]$. Dado um índice i de um nó, é possível computar os pais e os filhos de i:

$$\text{PAI}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$
$$\text{ESQUERDO}(i) = 2i$$
$$\text{DIREITO}(i) = 2i + 1$$

que podem ser computados com apenas uma instrução em computadores eletrônicos modernos.

Existem dois tipos de heaps binários: **heap-máximo** e **heap-mínimo**. Em ambos tipos os valores dos nós satisfazem uma **propriedade heap** que depende do tipo da heap. Em um **heap-máximo** a propriedade heap é:

"Para cada nó i, exceto a raíz, $A[\text{PAI}(i)] \geq A[i]$. Alternativamente, $\forall x(i \neq 1 \rightarrow A[\text{PAI}(i)] \geq A[i])$"

ou seja, o valor de um nó é, no máximo, o valor de seu pai. Desta forma, o maior elemento em um heap-máximo está na raíz e a subárvore de dado nó i contém elementos que não são maiores que o valor em i.

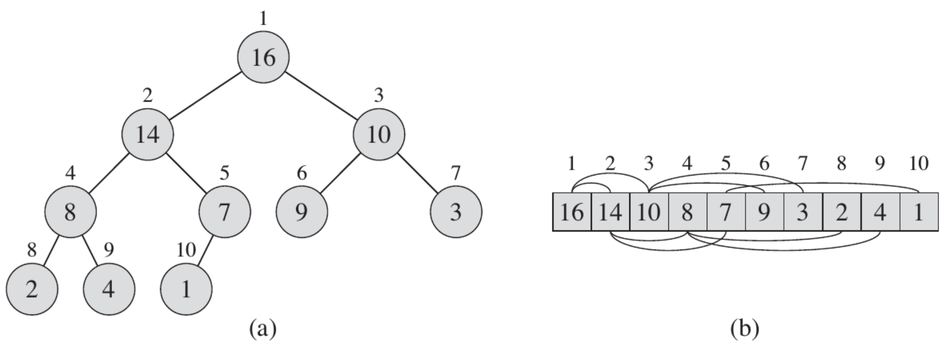

Figura 1: (a) Representação em Árvore de uma Heap-máxima. (b) Representação vetorial de uma Heap-máxima.

No algoritmo `Heapsort` são utilizados heap-maximos. Quando uma heap é interptretada como uma árvore, a altura de um nó é o número de arestas no caminho mais longo de um nó até uma folha. A altura da heap é a altura da raíz. Como uma heap de n elementos é função de uma árvore binária completa, sua altura é $\Theta(\lg n)$. Além disto, as operações binárias em uma árvore heap executam em tempo proporcial à altura da árvore, portanto executam em tempo $\Theta(\lg n)$. Estes são alguns fatos que garantem o bom desempenho do `Heapsort`, como será estudado mais adiante. As funções a seguir são utilizadas no `Heapsort`.

- `MAX-HEAPIFY`, executa em $\Theta(\lg n)$ e é responsável por manter a propriedade heap-máximo;

- `BUILD-MAX-HEAP`, executa em $\Theta(n)$, e converte um vetor desordenado em uma heap-máxima; e
- `HEAPSORT`, que executa em $\Theta(n \lg n)$ e ordena o vetor explorando a propriedade heap-máximo.

Mantendo a Propriedade Heap

A função `MAX-HEAPIFY` é responsável por manter a propriedade heap-maximo. As entradas são um vetor A, um índice i e o tamanho da heap th. A função assume que as árvores binárias em ESQUERDO(i) e DIREITO(i) são heaps-maximos, mas $A[i]$ pode ser menor que ESQUERDO(i) e DIREITO(i), o que fere a propriedade heap-máximo. th indica qual é o último elemento de A que faz parte da heap. `MAX-HEAPIFY` arruma a propriedade da árvore.

```
MAX-HEAPIFY(A, i, th)
l = ESQUERDO(i)
r = DIREITO(i)
IF l <= th AND A[l] > A[i]
   largest = l
ELSE largest = i
IF r <= th AND A[r] > A[largest]
   largest = r
IF largest <> i
   trocar A[i] e A[largest]
   MAX-HEAPIFY(A, largest, th)
```

`MAX-HEAPIFY` é um procedimento recursivo que funciona da seguinte maneira:

- O maior entre $A[1], A[\texttt{ESQUERDO}(i)]$ e $A[\texttt{DIREITO}(i)]$ é determinado.
- Se $A[i]$ já é o maior, então a propriedade heap-máximo já está OK. Caso contrário, um dos filhos é maior que $A[i]$. Então $A[i]$ é trocado com o maior dos filhos, arrumando a propriedade localmente.
- Após a troca, o nó que estava com o maior valor agora tem o valor que o nó i tinha. Desta forma, este nó pode violar a propriedade heap-maximo daquele índice em diante. A sub-árvore é arrumada recursivamente.

O tempo de execução de `MAX-HEAPIFY` em uma subárvore de altura n a partir do nó i é $\Theta(1)$ para arrumar a relação entre $A[1], A[\texttt{ESQUERDO}(i)]$ e $A[\texttt{DIREITO}(i)]$ mais o tempo necessário para executar `MAX-HEAPIFY` em uma subárvore de i (chamada recursiva). A altura das subárvores possuem no máximo $\frac{2n}{3}$ dos nós da árvore, e acontece quando o último nível está cheio até a metade. Portanto, podemos descrever o tempo de `MAX-HEAPIFY` pela recorrência

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

Pelo método mestre, esta recorrência é $T(n) = O(\lg n)$. Como a altura h de uma árvore heap com n elementos tem altura $\lg n$, o tempo de execução de `MAX-HEAPIFY` pode ser descrito por $O(h)$.

Construindo uma Heap

`BUILD-MAX-HEAP` pode ser utilizado para converter um vetor de $A[1 \dots n]$ em uma heap-máxima. Sabemos que os elementos no subvetor $A[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$ são folhas, que são heaps unitárias. Portanto, basta percorrer os demais nós da árvore e executar `MAX-HEAPIFY` em cada um deles em ordem inversa.

```
BUILD-MAX-HEAP(A, n)
1. FOR i = CHAO(n/2) DOWNTO 1 DO
2.    MAX-HEAPIFY(A, i, n)
```

Para provar que este algoritmo converte um vetor $A[1 \dots n]$ em uma heap-máxima, utilizamos a seguinte invariante de laço:

"No início de cada iteração do laço `FOR` das linhas 1–2, cada nó $i+1, i+2, \dots, n$ é raíz de uma heap-máxima".

INICIALIZAÇÃO Antes da primeira iteração, $i = \lfloor \frac{n}{2} \rfloor$. Cada nó $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \dots, n$ é folha, e portanto, a raíz de uma heap-máxima trivial.

MANUTENÇÃO Note que os índices dos filhos de i são maiores que i. Pela invariante de laço, estes filhos são ambos raízes de heaps máximos, que é a condição necessária para invocar `MAX-HEAPIFY` para fazer i tornar-se uma heap máxima. Além disto, `MAX-HEAPIFY` mantém a propriedade que os nós $i+1, i+2, \ldots, n$ são todos raízes de heaps-máximos. Ao decrementar i no laço `FOR`, a invariante de laço é reestabelecida para a próxima iteração.

TÉRMINO No término, $i = 0$. Pela invariante de laço, cada nó $1, 2, \ldots, n$ é a raíz de uma heap máxima. Portanto, após a última iteração, todos os elementos no subvetor $A[i+1, i+2, \ldots, n] = A[1, 2, \ldots, n]$, são raízes de heap-máximo, e inclui todos os elementos do vetor A. Por definição, um vetor é heap máxima se todos os elementos do vetor são raízes de heap-máximo. Como mostramos que isto é verdade ao final da execução de `BUILD-MAX-HEAP`, o vetor A é um heap-máximo.

Desempenho de BUILD-MAX-HEAP

O limite assintótico superior de `BUILD-MAX-HEAP` pode ser estimado da seguinte forma: cada chamada a `MAX-HEAPIFY` tem custo $O(\lg n)$ e `BUILD-MAX-HEAP` faz $O(n)$ chamadas delas. Desta forma, $T(n) = O(n) * O(\lg n) = O(n \lg n)$ descreve o limite assintótico superior. Contudo, embora correto, este limite não está ajustado o máximo possível.

Podemos encontrar o limite ajustado observando que o tempo gasto por `MAX-HEAPIFY` varia de acordo com a altura do nó na árvore, e que a altura da maioria dos nós é pequena (tende a uma constante). Esta análise mais refinada está embasada nas propriedades que uma heap com com n elementos possui altura $\lfloor \lg n \rfloor$, e no máximo, $\lceil \left(\frac{n}{2^{h+1}} \right) \rceil$ nós em uma altura h. Assim, o tempo necessário para `MAX-HEAPIFY` executar em um nó de altura h é $O(h)$, portanto o custo de `BUILD-MAX-HEAP` está limitada superiormente por:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

como

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

O limite assintótico superior de `BUILD-MAX-HEAP` pode ser limitado por

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n).$$

HEAPSORT

```
HEAPSORT(A, n)
1. BUILD-MAX-HEAP(A, n)
2. th = n
3. FOR i = n DOWNTO 2 DO
4.     troca A[1] e A[i]
5.     th = th - 1
6.     MAX-HEAPIFY(A,1,th)
```

O algoritmo `HEAPSORT` inicia com a construção de uma heap máxima a partir do vetor original. Como o elemento máximo do vetor está em $A[1]$, é possível trocá-lo com $A[n]$, sua posição correta. Ao descartar o último elemento da heap e mantendo uma variável de controle junto ao vetor heap indicando o final da heap, podemos verificar que os filhos da raíz mantém-se heaps máximos. No entanto, o novo elemento raíz pode violar a propriedade heap-maximo. Portanto, basta restaurar sua propriedade usando `MAX-HEAPIFY`, obtendo uma nova uma heap-máxima $A[1, \ldots, i-1]$. O algoritmo repete este processo até obter uma heap de tamanho 2.

Heapsort invoca `BUILD-MAX-HEAP` que executa em $\Theta(n)$ e que cada uma das $n-1$ chamadas a `MAX-HEAPIFY` executa em tempo $\Theta(\lg n)$. Assim, o tempo de execução de `HEAPSORT` é dado pela função

$$T(n) = \underbrace{(n-1)\Theta(\lg n)}_{\text{Linhas 3–6}} + \underbrace{\Theta(n)}_{\texttt{BUILD-MAX-HEAP}} = \Theta(n \lg n).$$

Bibliografia

[CRLS] CORMEN, T. H. et al. Algoritmos: Teoria e Prática. Elsevier, 2012. 3a Ed. Capítulo 6 (Ordenação por Heap), Seções 6.1–6.4.