Algoritmos de Ordenação: *HeapSort*

ACH2002 - Introdução à Ciência da Computação II

Delano M. Beder

Escola de Artes, Ciências e Humanidades (EACH)
Universidade de São Paulo
dbeder@usp.br

10/2008

Material baseado em slides do professor Marcos Chaim

HeapSort

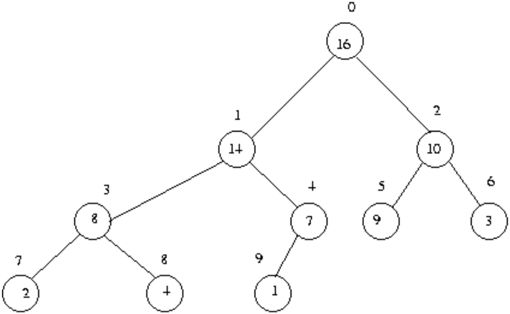
- *Heap* – Dicionário Merriam-Webster:
  1. Coleção de coisas jogadas uma em cima da outra – monte;
  2. Grande número ou grande quantidade – lote.
- Em computação, dois sentidos :
  1. Espaço de memória variável onde são criados objetos;
  2. Estrutura de dados para armazenar dados segundo uma regra particular
     - Próximo do sentido original, traduzido como *monte*.
- O algoritmo de ordenação *HeapSort* utiliza a estrutura de dados *heap* para ordenar um vetor.

- O projeto por indução que leva ao *HeapSort* é essencialmente o mesmo do *Selection Sort*: selecionamos e posicionamos o maior (ou menor) elemento do conjunto e então aplicamos a hipótese de indução para ordenar os elementos restantes.
- A diferença importante é que no *HeapSort* utilizamos a estrutura de dados *heap* para selecionar o maior (ou menor) elemento eficientemente.
- Um *heap* é um vetor que simula uma árvore binária completa, a menos, talvez, do último nível, com estrutura de *heap*.

Estrutura *Heap*

Definição

A estrutura de dados Heap (binário) é um vetor que pode ser visto como uma árvore binária quase completa



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

Estrutura *Heap*

- Cada nó da árvore corresponde a um elemento do vetor que armazena o valor no nó.

- A árvore está completamente preenchida em todos os níveis, exceto possivelmente no nível mais baixo, que é preenchido da esquerda para a direita até certo ponto.

- Um vetor V representa uma estrutura *heap* através de dois parâmetros:

  1. comprimento de V (`V.length`): tamanho total do vetor;
  2. comprimento do *heap* (`heapComp`): comprimento da parte do vetor que contém elementos da estrutura *heap*.

Estrutura *Heap*

Parâmetros da estrutura *heap*:

- Os elementos válidos para a estrutura do *heap* vão de `0` até `heapComp-1`.
- Portanto, `heapComp` $\leq$ `V.length`.
- Os elementos de `heapComp` até `V.length-1` não fazem parte da estrutura *heap* mesmo sendo valores válidos.

A raiz da árvore representada na estrutura *heap* é `V[0]`. Dado um elemento da estrutura *heap* de índice i:

- `pai(i)`: $\lfloor$ (i-1)/2 $\rfloor$
- `esquerda(i)`: $2 * i + 1$
- `direita(i)`: $2 * i + 2$

As operações para determinar o `pai(i)`, `esquerda(i)` e `direita(i)` podem ser implementadas muito rapidamente com operações sobre bits.

Estrutura *Heap*

Há dois tipos de *heap* binário: *heap* máximo e *heap* mínimo.

Propriedades de *heap* máximo

1. A[`pai(i)`] $\geq$ A[i].
   - Isto é, o valor de um nó é no máximo o valor de seu pai.
2. O maior elemento do *heap* está na raiz.
3. as subárvores de um nó possuem valores menores ou iguais ao do nó.

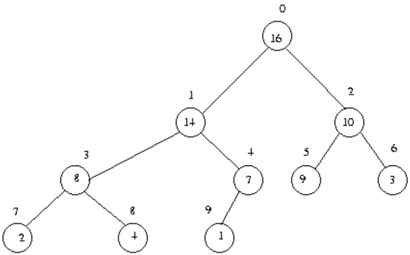
Propriedades de *heap* mínimo

1. A[`pai(i)`] $\leq$ A[i].
   - Isto é, o valor de um nó é maior ou igual o valor de seu pai.
2. O menor elemento do *heap* está na raiz
3. as subárvores de um nó possuem valores maiores ou iguais ao do nó.

Definição

A altura de um nó em um heap é o número de arestas no caminho descendente simples mais longo desde o nó até uma folha.

- Qual a altura do nó de número 1 na figura abaixo ?
- Qual o nó de maior altura?



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

Teorema

A altura h de uma heap com n nós é $\lfloor log_2 n \rfloor$.

- A altura é a mesma se o *heap* é completo (a última fileira está completa)
  - O número de nós n é dado por $n = 2^{h+1} - 1$
  - Ou seja, o número de nós n é tal que $2^h < n < 2^{h+1}$.
  - Aplicando o logaritmo de 2 na inequação, obtém-se $h < log_2 n < h + 1 \Rightarrow h = \lfloor log_2 n \rfloor$.
- Ou se o *heap* é o mais incompleto possível (a última fileira possui um único elemento).
  - O número de nós n é dado por $2^h - 1 + 1$, ou seja, $n = 2^h$
  - A altura $h = log_2 n \Rightarrow h = \lfloor log_2 n \rfloor$.

Manutenção da propriedade de *heap*

- Toda manipulação de um *heap* deve manter a sua propriedade de *heap* (máximo ou mínimo) inalterada.
- Suponha a seguinte situação:
  - Um vetor V;
  - Um índice i do vetor;
  - as árvores `esquerda(i)` e `direita(i)` são *heaps* máximos;
  - Devido a alguma alteração no *heap*, V[i] pode ser menor que seus filhos
    - Violação da propriedade de *heap* máximo.
- Como manter a propriedade de *heap* máximo nessa situação?

Manutenção da propriedade de *heap*

A idéia para manter a propriedade de *heap* máximo é fazer A[i] *afundar* no *heap* máximo.

```
void refazHeapMax(int V[], int i, int compHeap) {
  int esq, dir, maior, temp;
  esq = esquerda(i); dir = direita(i);
  if(esq < compHeap && V[esq] > V[i]) {
    maior = esq;
  } else {
    maior = i;
  }
  if(dir < compHeap && V[dir] > V[maior]) {
    maior = dir;
  }
  if(maior != i) {
    // trocar V[i] <==> V[maior]
    temp = V[i];
    V[i] = V[maior];
    V[maior] = temp;
    // Ajusta a posicao de maior, se incorreta.
    refazHeapMax(V, maior, compHeap);
  }
}
```

Manutenção da propriedade de *heap*

- O tempo de execução do método `refazHeapMax()`:
  1. $\Theta(1)$ para corrigir os relacionamentos entre os elementos `A[i]`, `A[esquerda(i)]` e `A[direita(i)]`.
  2. o tempo para executar `refazHeapMax()` em uma subárvore com raiz em um dos filhos do nó *i*.
- As subárvores de cada filho têm tamanho máximo $2n/3$
  - O pior caso ocorre quando a última linha da árvore está exatamente metade cheia.
  - O tempo de execução de `refazHeapMax()` pode então ser descrito pela recorrência:

$$T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log n)$$

- Como fazemos para transformar um vetor V de tamanho *n* em um *heap* ?

Construindo um *heap*

Observação:

- Se o vetor V fosse um *heap* máximo então todos os elementos V[$\lfloor$n/2$\rfloor$] ... V[n-1] seriam folhas. Por quê?
  - Imagine-se que V[$\lfloor$n/2$\rfloor$] ... V[n-1] não fossem folhas.
    - Isto significa que $\exists$ V[k] tal que $\lfloor$n/2$\rfloor$ $\leq$ pai(k) $\leq$ n-1.
    - Neste caso, k $>$ n, pois o pai(k) = $\lfloor$k/2$\rfloor$-1.
    - Porém, o comprimento do arranjo V é *n* (V[0]...V[n-1]).
    - Logo, k estaria fora do vetor e, portanto, não existe.

Partindo da idéia de que os elementos V[$\lfloor$n/2$\rfloor$] ... V[n-1] são *heaps* triviais (folhas):

- pode-se construir um *heap* chamando o método `refazMaxHeap()` para os elementos V[$\lfloor$n/2$\rfloor$-1], V[$\lfloor$n/2$\rfloor$-2] até V[0] do arranjo.
- é como se imaginássemos que metade do arranjo são *heaps* triviais aos quais foram adicionados os elementos V[$\lfloor$n/2$\rfloor$-1] até V[0], sucessivamente.

Construindo um *heap*

Implementando a idéia anterior:

```
void constroiHeapMax(int [] V) {
  int compHeap = V.length;
  for(int i = (V.length)/2 - 1; i >= 0 ; i--)
     refazHeapMax(V, i, compHeap);
}
```

Pode não parecer, mas o custo do método acima é O(n).

(Ver [2], página 109).

Exemplo

`int v[] = {16, 4, 10, 14, 7, 9, 3, 2, 8,1}` é um *heap* ?

O algoritmo *HeapSort* (ordenação por monte) utiliza os métodos `constroiHeapMax()` e `refazHeapMax()`:

HeapSort

```
void heapSort(int V[]) {
  int i, compHeap, temp;
  // Constrói o heap máximo do arranjo todo
  compHeap = V.length;
  constroiHeapMax(V);

  for(i = V.length-1; i > 0; --i) {
    // Troca V[0] <==> V[i]
    temp = V[0];
    V[0] = V[i];
    V[i] = temp;
    // Diminui o heap, pois V[i] está posicionado
    compHeap--;
    refazHeapMax(V, 0, compHeap);
  }
}
```

O algoritmo *HeapSort*

Comentários:

- O *HeapSort* começa com uma chamada ao método `constroiHeapMax()` para o vetor V todo (V[0]...V[n-1]).

- O maior elemento está em V[0] depois de chamado `constroiHeapMax()`.

- Então é feita a troca V[0] e V[n-1].

- Agora o subvetor V[0...n-2] precisa ser mantido, pois V[0] foi alterado com a troca realizada e a propriedade de *heap* máximo pode ter sido violada.

- O método `refazHeapMaximo()` é então chamado com diminuição do tamanho do *heap* decrementado — afinal V[n-1] já não faz parte, pois já contém o maior valor.

O algoritmo *HeapSort*

O custo do algoritmo *HeapSort* é $O(n \log n)$ porque:

1. `constroiHeapMax()` custa $O(n)$.

2. cada uma das (n-1) chamadas para o método `refazHeapMax()` custa $O(\log n)$.

3. Portanto, custo total do *HeapSort* é:

$$T(n) = O(n) + (n-1)O(\log n) = O(n \log n)$$

Resumo

- Estrutura *heap*:
  - `pai(i)`: $\lfloor$ (i-1)/2 $\rfloor$
  - `esquerda(i)`: 2i+1
  - `direita(i)`: 2i+2
- Propriedade *heap* máximo: A[`pai(i)`] $\geq$ A[i].
- Propriedade *heap* mínimo: A[`pai(i)`] $\leq$ A[i].
- `refazHeapMax()`: $O(\log n)$.
- `constroiHeapMax()`: $O(n)$.
- `HeapSort()`: $O(n \log n)$.

Exercícios

1. Onde em um *heap* máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?
2. A seqüência $<$23, 17, 14, 6, 13, 10, 1, 5, 7, 12$>$ é um *heap* máximo?
3. Usando a Figura 1 como modelo, ilustre a operação de `refazHeapMax(A,2)` sobre o vetor V = $<$27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0$>$.
4. Escreva um método em Java `refazHeapMin()` que refaz um *heap* mínimo.
5. Qual é o efeito de chamar `refazHeapMax(V,i)` quando V[i] é maior que seus filhos?
6. Qual é o efeito de chamar `refazHeapMax(V,i)` quando i $>$ compHeap/2?
7. Siga a operação do algoritmo *HeapSort* sobre o arranjo A = $<$5, 13,2, 25, 7, 17, 20, 8, 4$>$.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. *Algoritmos - Tradução da 2a. Edição Americana*. Editora Campus, 2002.

[2] Nívio Ziviani. *Projeto de Algoritmos com implementações em C e Pascal*. Editora Thomson, 2a. Edição, 2004.

Importante

Ao ler a referência [2] (páginas 103-111) atentar para o detalhe de que os vetores V de comprimento *n* variam de V[1] até V[n].
Os algoritmos apresentados nestes slides utilizam a notação de Java, ou seja, variação de V[0] até V[n-1].
Por isso, prestar atenção nas diferenças de notação ao ler a referência [2].