Heapsort

Universidade Federal do Ceará - Campus de Crateús

Roberto Cabral
rbcabral@crateus.ufc.br

03 de Maio de 2017

Estrutura de Dados

Introdução

- Algoritmo criado por John Williams.
- Complexidade O(n log n) no pior e médio caso.
- Utiliza a abordagem proposta pelo selectionSort.
- O selectionSort pesquisa entre os n elementos o que precede todos os outros $n - 1$ elementos.
- Para ordenar em ordem ascendente, o heapsort pões o maior elemento no final do array e o segundo maior antes dele, etc.
- O heapsort começa do final do array pesquisando os maiores elementos, enquanto o selection sorte começa do início do array pesquisando os menores.

Heapsort

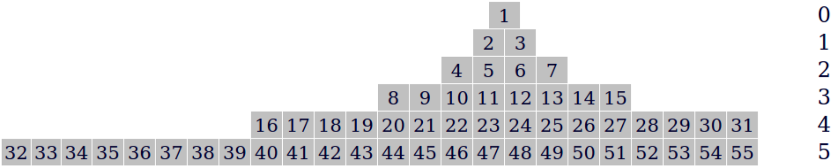
- Para ordenar, o heapsort usa um Heap.
- Heap é uma árvore binária com as seguintes propriedades:
    - O valor de cada nó não é menos que os valores armazenados em cada filho.
    - A árvore é perfeitamente balanceada e as folhas no último nível estão todas nas posições mais a esquerda.

Árvore binária armazenada em um vetor

- Suponha dado um vetor $A[1..n]$. Por enquanto, os valores armazenados no vetor não nos interessam; só interessam certas relações entre índices. Para todo índice i, diremos que:
  - $\lfloor \text{i}/2 \rfloor$ é o pai do índice i.
  - $2i$ é o filho esquerdo de i
  - $2i + 1$ é o filho direito de i.
- Isso deve ser entendido com cautela:
  - o índice 1 não tem pai;
  - um índice i só tem filho esquerdo se $2i \leq n$;
  - um índice i só tem filho direito se $2i + 1 \leq n$.

Intercalação de vetores ordenados

- Com essa história de pais e filhos, o vetor adquire uma estrutura de árvore binária quase completa e os elementos do vetor, identificados pelos índices 1 a n, passam a ser chamados nós.
- A figura abaixo sugere uma maneira de encarar o vetor. Cada caixa é um nó da árvore binária quase completa $A[1..55]$. (O número dentro de cada caixa é i e não $A[i]$.)



Árvore binária armazenada em um vetor

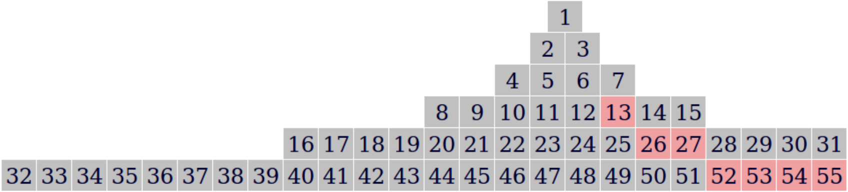
- Os números na coluna à direita indicam os níveis da árvore.
- Cada nível p, exceto talvez o último, tem exatamente $2p$ nós e esses nós são:
$$2^p, 2^p + 1, \ldots 2^{p+1} - 1.$$
- O último nível pode ter menos nós.
- O nó i pertence ao nível $\lfloor \lg i \rfloor$
- Portanto, o número total de níveis é $1 + \lfloor \lg n \rfloor$.
- Uma propriedade simples mas importante: o número de nós em qualquer nível p (exceto talvez o último) é $1 + S$, onde S é a soma do número de nós nos níveis $0, \ldots, p - 1$.

Intercalação de vetores ordenados

- O nó 1 é a raiz da árvore. Qualquer nó i é raiz da subárvore formada por i, seus filhos, seus netos, etc.
- Ou seja, a subárvore com raiz i é o vetor:

$$A[i, 2i, 2i+1, 4i, 4i+1, 4i+2, 4i+3, 8i, \ldots, 8i+7, \ldots].$$

- Exemplo: a figura abaixo destaca a subárvore cuja raiz é 13.



Heap

- O segredo do algoritmo Heapsort é uma estrutura de dados, conhecida como heap, que enxerga o vetor como uma árvores binária.
- Há dois sabores da estrutura: max-heap e min-heap, mas trataremos aqui apenas do primeiro, omitindo o prefixo max.
- Um heap (= monte) é um vetor $v[1..m]$ tal que $v[f/2] \leq v[f]$ para $f = 2, \ldots, m$
- De maneira mais informal, podemos dizer que um vetor $v[1..m]$ é um heap se o valor de todo pai é maior ou igual ao valor de qualquer de seus dois filhos, sendo $v[i]$ o valor de i.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
999	888	777	555	666	777	555	222	333	444	111	333	666	333

Construção de um heap

- Para entender o Heapsort, será preciso tratar, às vezes, de heaps com defeitos.
- Diremos que um vetor $v[1..m]$ é um heap exceto talvez pelo índice k se a desigualdade $v[f/2] \leq v[f]$ vale para todo f diferente de k.
- É fácil rearranjar um vetor $v[1..m]$ para que ele se torne um heap.
- A ideia é repetir o seguinte processo enquanto o valor de um filho for maior que o de seu pai:
    - troque os valores de pai e filho e suba um passo em direção à raiz.
- Mais precisamente, se $v[f/2] < v[f]$, faça troca $(v[f/2], v[f])$ e em seguida $f = f/2$. A operação de troca é definida assim:

```
#define troca (A, B)  int t = A; A = B; B = t;
```

Construção de um heap

```
static void constroiHeap (int m, int *v){
   int k;
   for (k = 1; k <= m; ++k) {
      // v[1..k] é um heap
      int f = k+1;
      while (f > 1 && v[f/2] < v[f]) {  // 5
         troca (v[f/2], v[f]);          // 6
         f /= 2;                        // 7
      }
   }
}
```

- Em cada iteração do bloco de linhas 6-7, o índice f pula de uma camada do vetor para a anterior.
- Portanto, esse bloco de linhas é repetido no máximo lg(k) vezes para cada k fixo.
- Segue daí que o número total de comparações entre elementos do vetor (todas acontecem na linha 5 do código) não passa de m lg (m).

A função peneira

- O coração de muitos algoritmos que manipulam heaps é uma função que, ao contrário de constroiHeap, desce em direção à base da árvore.
- Essa função, que chamaremos peneira, recebe um vetor qualquer $v[1..m]$ e faz $v[1]$ descer até sua posição correta, pulando de uma camada para a seguinte.
- Como isso é feito?
    - Se $v[1] \geqslant v[2]$ e $v[1] \geqslant v[3]$, não é preciso fazer nada.
    - Se $v[1] < v[2]$ e $v[2] \geqslant v[3]$, basta trocar $v[1]$ com $v[2]$ e depois fazer $v[2]$ descer para sua posição correta.
    - Os dois outros casos são semelhantes.

85	99	98	97	96	95	94	93	92	91	90	89	97	86

99	85	98	97	96	95	94	93	92	91	90	89	97	86

99	97	98	85	96	95	94	93	92	91	90	89	97	86

99	97	98	93	96	95	94	85	92	91	90	89	97	86

A função peneira

- Segue o código da função.
- Cada iteração começa com um índice p e escolhe o filho f de p que tem maior valor:

```c
static void peneira1 (int m, int *v) {
   int f = 2;
   while (f <= m) {
      if (f < m && v[f] < v[f+1])  ++f;
      // f é o filho mais valioso de f/2
      if (v[f/2] >= v[f]) break;
      troca (v[f/2], v[f]);
      f *= 2;
   }
}
```

A função peneira

- A seguinte versão é um pouco melhor, porque faz menos movimentações de elementos do vetor.
- Também faz menos divisões de f por 2.

```
static void peneira (int m, int *v)
{
   int p = 1, f = 2, t = v[1];
   while (f <= m) {
      if (f < m && v[f] < v[f+1])  ++f;
      if (t >= v[f]) break;
      v[p] = v[f];
      p = f, f = 2*p;
   }
   v[p] = t;
}
```

Desempenho

- A função `peneira` é muito rápida.
- Ela faz no máximo $\lg(m)$ iterações, uma vez que o vetor tem $1 + \lg(m)$ camadas.
- Cada iteração envolve duas comparações entre elementos do vetor e portanto o número total de comparações não passa de $2 \lg(m)$.
- O consumo de tempo é proporcional ao número de comparações e portanto proporcional a $\lg m$ no pior caso.

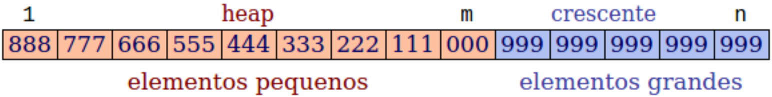
O algoritmo Heapsort

- Não é difícil reunir o que foi falado para obter um algoritmo que rearranja um vetor $v[1..n]$ em ordem crescente.
- O algoritmo tem duas fases:
  - a primeira transforma o vetor em heap
  - a segunda rearranja o heap em ordem crescente.

```
void heapsort (int n, int *v)
{
   int m;
   constroiHeap (n, v);
   for (m = n; m >= 2; --m) {
      troca (v[1], v[m]);
      peneira (m-1, v);
   }
```

O algoritmo Heapsort

- No início de cada iteração valem as seguintes propriedades (invariantes):
    - o vetor $v[1..m]$ é um heap,
    - $v[1..m] \leq v[m+1..n]$,
    - $v[m+1..n]$ está em ordem crescente e
    - $v[1..n]$ é uma permutação do vetor original.
- É claro que $v[1..n]$ estará em ordem crescente quando m for igual a 1.



Desempenho do Heapsort

- Quantas comparações entre elementos do vetor a função `heapsort` executa?
- A função auxiliar `constroiHeap` faz n lg(n) comparações no máximo.
- Em seguida, a função peneira é chamada cerca de n vezes e cada uma dessas chamadas faz 2 lg(n) comparações no máximo.
- Logo, o número total de comparações não passa de 3 n lg (n).
- Quanto ao consumo de tempo do `heapsort`, ele é proporcional ao número de comparações entre elementos do vetor, e portanto proporcional a n lg n no pior caso.

LAB

- Implemente a função heapSortCres que recebe como entrada um vetor de tamanho n e retorna esse vetor ordenado em ordem crescente.
- Implemente a função heapSortDec que recebe como entrada um vetor de tamanho n e retorna esse vetor ordenado em ordem decrescente.

Heapsort

Universidade Federal do Ceará - Campus de Crateús

Roberto Cabral
rbcabral@crateus.ufc.br

03 de Maio de 2017

Estrutura de Dados