Heapsort

Considere o problema da ordenação discutido em outro capítulo. O problema consiste em permutar os elementos de um vetor `v[1..n]` para colocá-los em ordem crescente. Em outras palavras, rearranjar os elementos do vetor de tal modo que tenhamos $v[1] \le \ldots \le v[n]$.

O outro capítulo analisou alguns algoritmos simples para o problema. O presente capítulo examina o algoritmo Heapsort, descoberto por J.W.J. Williams em 1964. Diferentemente dos algoritmos mais simples, o Heapsort é linearítmico, mesmo no pior caso.

Suporemos que os índices do vetor são `1..n` e não `0..n-1` (como é usual em C) pois essa convenção torna o código um pouco mais simples.

Sumário:

Vetores e árvores binárias

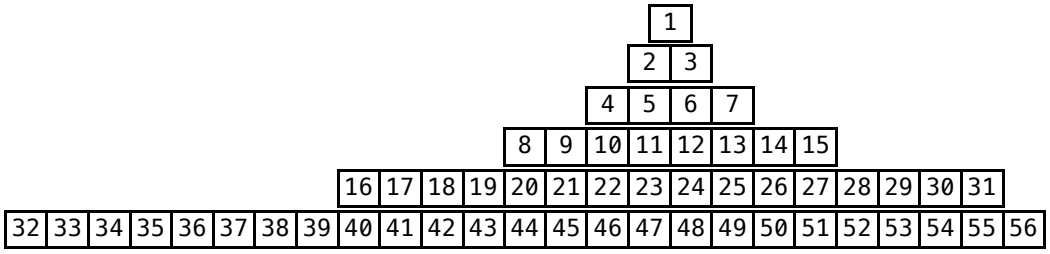
Antes de começar a discutir o Heapsort, precisamos aprender a enxergar a árvore binária que está escondida em qualquer vetor. O conjunto de índices de qualquer vetor `v[1..m]` pode ser encarado como uma árvore binária da seguinte maneira:

- o índice `1` é a *raiz* da árvore;

- o *pai* de qualquer índice `f` é `f/2` (é claro que `1` não tem pai);
- o *filho esquerdo* de um índice `p` é `2p` (esse filho só existe se `2p ≤ m`);
- o *filho direito* de `p` é `2p+1` (esse filho só existe se `2p+1 ≤ m`).

1	2	3	4	5	6	7	8	9	10	11	12	13	14
999	999	999	999	999	999	999	999	999	999	999	999	999	999

Para tornar a árvore binária mais evidente, podemos desenhar o vetor em *camadas*, de tal modo que cada filho fique na camada seguinte à do pai. A figura abaixo é um tal desenho do vetor `v[1..56]`; os números nas caixas são os índices `i` e não os valores `v[i]`. Observe que cada camada, exceto talvez a última, tem duas vezes mais elementos que a camada anterior. Com isso, o número de camadas de um vetor `v[1..m]` é exatamente `1 + lg(m)`, sendo `lg(m)` o piso de log `m`.



Exercícios 1

1. Seja `m` um número da forma 2^k-1. Mostre que mais da metade dos elementos de qualquer vetor `v[1..m]` está na última camada.

Heap

O segredo do algoritmo Heapsort é uma estrutura de dados, conhecida como heap, que enxerga o vetor como uma árvore binária. Há dois sabores da estrutura: max-heap e min-heap; trataremos aqui apenas do primeiro, omitindo o prefixo "max-".

Um heap (= monte) é um vetor em que o valor de todo pai é maior ou igual ao valor de cada um de seus dois filhos. Mais exatamente, um vetor `v[1..m]` é um *heap* se

```
v[f/2] ≥ v[f]
```

para `f` = `2, . . . , m`. Aqui, como no resto deste capítulo, vamos convencionar que as expressões que figuram como índices de um vetor são sempre calculadas em aritmética inteira. Assim, o valor da expressão `f/2` é `⌊f/2⌋`, ou seja, o piso de `f/2`.

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

Ocasionalmente, é preciso tratar 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] ≥ v[f]` vale para todo `f` diferente de `k`.

Exercícios 2

1. O vetor 161 41 101 141 71 91 31 21 81 17 16 é um heap?
2. Mostre que todo vetor decrescente é um heap. Mostre que a recíproca não é verdadeira.
3. ★ Escreva uma função que decida se um vetor `v[1..m]` é ou não um heap.
4. ★ Mostre que `v[1..m]` é um heap se e somente se cada índice `p` tem as seguintes propriedades: (a) `v[p] ≥ v[2p]` se `2p ≤ m` e (b) `v[p] ≥ v[2p+1]` se `2p+1 ≤ m`.
5. Suponha que `v[1..m]` é um heap e `p` é um índice menor que `m/2`. É verdade que `v[2p] ≥ v[2p+1]`? É verdade que `v[2p] ≤ v[2p+1]`?
6. Suponha que `v[1..m]` é um heap e sejam `i < j` dois índices tais que `v[i] < v[j]`. Se `v[i]` e `v[j]` forem intercambiados, `v[1..m]` continuará sendo um heap? Repita o exercício sob a hipótese `v[i] > v[j]`.
7. ★ Suponha que `v[1..m]` é um heap e que os elementos do vetor são diferentes entre si. É claro que o maior elemento do vetor está no índice `1`. Onde pode estar o segundo maior elemento? Onde pode estar o terceiro maior elemento? É verdade que o terceiro maior elemento é filho do segundo maior elemento?

Construção de um heap

É fácil rearranjar os elementos um vetor de inteiros `v[1..m]` para que ele se torne um heap. A idea 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 exatamente, enquanto `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;}
```

Eis o código completo:

```
// Rearranja um vetor v[1..m] de modo a
// transformá-lo em heap.

static void
constroiHeap (int m, int v[])
{
   for (int 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
      }
   }
}
```

(A palavra-chave `static` está aí apenas para indicar que `constroiHeap` é uma função auxiliar que não deve ser invocada diretamente pelo usuário final do Heapsort.)

No início de cada iteração controlada pelo `for`, o vetor `v[1..k]` é um heap. No curso da iteração, `v[k+1]` "sobe" em direção à raiz até encontrar seu lugar correto, sendo assim incorporado ao heap.

Em cada iteração do bloco de linhas `6–7`, o índice `f` pula de uma camada do vetor para a camada 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 (linha `5` do código) entre elementos do vetor não passa de

$$m \lg(m) .$$

(Como veremos adiante, é possível fazer o serviço com apenas `m` comparações.)

Exercícios 3

1. IMPORTANTE. Critique a seguinte ideia: para transformar um vetor em heap, basta rearranjá-lo em ordem decrescente (usando o algoritmo Mergesort ou o algoritmo Quicksort, por exemplo).
2. Prove que a função `constroiHeap` está correta. Comece por estabelecer os invariantes do processo iterativo nas linhas `5` a `7`.
3. Discuta a seguinte versão da função `constroiHeap`:

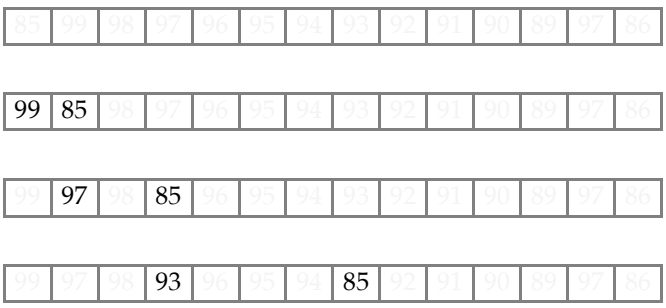
```
for (int k = 1; k < m; ++k) {
   int f = k+1, x = v[k+1];
   while (f > 1 && v[f/2] < x) {
      v[f] = v[f/2];
      f /= 2; }
   v[f] = x; }
```

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]` ≥ `v[2]` e `v[1]` ≥ `v[3]`, não é preciso fazer nada. Se `v[1]` < `v[2]` e `v[2]` ≥ `v[3]`, basta trocar `v[1]` com `v[2]` e depois fazer `v[2]` "descer" para sua posição correta. Nos dois outros casos, fala algo análogo. (Veja um rascunho do algoritmo em pseudocódigo.) No exemplo a seguir, cada linha da figura retrata o estado do vetor no início de uma iteração:



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

```
static void
peneira (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 será aplicada a vetores que são heaps exceto talvez por um ou dois índices. A função pode, portanto, ser documentada assim:

```
// Recebe um vetor v[1..m] que é um heap
// exceto talvez pelos índices 2 e 3 e
// rearranja o vetor de modo a
// transformá-lo em heap.
```

A seguinte versão é um pouco melhor, porque faz menos movimentações de elementos do vetor (e menos divisões de `f` por 2):

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

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 $\log m$ no pior caso.

Exercícios 4

1. Por que a seguinte implementação de `peneira` não funciona?

```
int p = 1, f = 2;
while (f <= m) {
   if (v[p] < v[f]) {
```

```
            troca (v[p], v[f]);
            p = f;
            f = 2*p; }
         else {
            if (f < m && v[p] < v[f+1]) {
               troca (v[p], v[f+1]);
               p = f+1;
               f = 2*p; }
            else break; } }
```

2. O seguinte código alternativo da função `peneira` está correto?

```
for (int f = 2; f <= m; f *= 2) {
   if (f < m && v[f] < v[f+1]) ++f;
   int p = f/2;
   if (v[p] >= v[f]) break;
   troca (v[p], v[f]); }
```

3. Discuta a seguinte variante do código de `peneira`:

```
int x = v[1], f = 2;
while (f <= m) {
   if (f < m && v[f] < v[f+1]) ++f;
   if (x >= v[f]) break;
   v[f/2] = v[f];
   f *= 2; }
v[f/2] = x;
```

4. Versão recursiva. Escreva uma versão recursiva da função `peneira`.

5. ★ Peneira generalizada. Suponha que um vetor `v[1..m]` é um heap exceto talvez pelos índices `2p` e `2p+1`. Escreva uma função `peneira_2` que receba `v[1..m]` e `p` e transforme o vetor em heap.

6. ★ Construção rápida de um heap. Mostre que a seguinte função tem o mesmo efeito que `constroiHeap` acima, ou seja, transforma qualquer vetor `v[1..m]` em heap:

```
void constroiHeap_2 (int m, int v[]) {
   for (int p = m/2; p >= 1; --p)
      peneira_2 (p, m, v);
}
```

   (Veja `peneira_2` no exercício anterior.) Mostre que `constroiHeap_2` faz no máximo `(m lg(m))/2` comparações entre elementos do vetor. Refine sua análise para mostrar que, na verdade, a função faz no máximo `m` comparações.

7. O seguinte fragmento de código transforma qualquer vetor `v[1..m]` em heap?

```
for (int p = 1; p <= m/2; ++p)
   peneira_2 (p, m, v);
```

8. Fila de prioridades. Uma *fila de prioridades* (= *priority queue*) é um conjunto de objetos (números, por exemplo) sujeito a duas operações: (1) remoção de um elemento de valor máximo e (2) inserção de um novo elemento. Se o conjunto for mantido num heap, essas duas operações podem ser realizadas de maneira muito rápida. Implemente as duas operações de modo que cada uma consuma tempo proporcional a $\log n$ no pior caso, sendo n o número de objetos no conjunto.

O algoritmo Heapsort

Não é difícil reunir o que dissemos acima 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 e a segunda retira

elementos do heap em ordem decrescente. (Comece fazendo um rascunho do algoritmo.)

```
// Rearranja os elementos do vetor v[1..n]
// de modo que fiquem em ordem crescente.

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

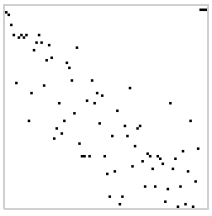
No início de cada iteração do `for` valem as seguintes propriedades (invariantes):

- `v[1..m]` é um heap,
- `v[1..m]` ≤ `v[m+1..n]` ,
- `v[m+1..n]` está em ordem crescente e
- `v[1..n]` é uma permutação do vetor original.

A expressão "`v[1..m]` ≤ `v[m+1..n]`" é uma abreviatura de "cada elemento de `v[1..m]` é menor ou igual a cada elemento de `v[m+1..n]`".

Segue daí que `v[1..n]` estará em ordem crescente quando `m` for igual a `1`. Portanto, o algoritmo está correto.

Animações do Heapsort



A animação à direita (copiada de Simon Waldherr / Golang Sorting Visualization) mostra a aplicação do Heapsort a um vetor `v[0..79]` de números positivos. Cada elemento `v[i]` do vetor é representado pelo ponto de coordenadas (`i`,`v[i]`). (Por algum motivo, a animação não executa as duas últimas iterações.)

Veja outras animações de algoritmos de ordenação:

- Heapsort, na Wikipedia.
- Heapsort animation, de David Galles (University of San Francisco).
- Animação de 15 algoritmos de ordenação, de Timo Bingmann, no YouTube.
- Sorting Algorithms Animations da Toptal.

Exercícios 5

1. Use a função `heapsort` para ordenar o vetor 16 15 14 13 12 11 10 9 8 7 6 5 4 . Mostre o estado do vetor no início de cada iteração.

2. Descreva o estado do vetor no início de uma iteração genérica do algoritmo Heapsort.

3. TESTE DE CORREÇÃO. Escreva um programa que teste, experimentalmente, a correção de sua implementação do algoritmo Heapsort. (Veja exercício análogo para Insertionsort.)

4. A função `heapsort` produz um rearranjo estável do vetor, ou seja, preserva a ordem relativa de elementos de mesmo valor?

5. Escreva uma função com protótipo `heap_sort (int *v, int n)` que rearranje um vetor `v[0..n-1]` (note os índices) em ordem crescente. (Basta invocar `heapsort` da maneira correta.)

6. MIN-HEAP. Escreva uma função que rearranje um vetor `v[1..n]` em ordem *decrescente*. Sugestão: Adapte a definição de heap e reescreva a função `peneira`.

7. Imite um heap por meio de um conjunto de células interligadas com ponteiros. Cada célula terá quatro campos: um `valor` e três ponteiros, um apontando o pai da célula, outro apontando o filho direito, e outro apontando o filho esquerdo. Escreva uma versão apropriada da função `peneira`. (Veja o capítulo *Árvores binárias*.)

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 $n \log n$ no pior caso. (Entretanto, a constante de proporcionalidade é maior que a do Mergesort e a do Quicksort.)

Exercícios 6

1. TESTE DE DESEMPENHO. Escreva um programa para cronometrar sua implementação do algoritmo Heapsort. (Veja exercício análogo para o Mergesort.)

Veja o capítulo 14 do "Programming Pearls" de Jon Bentley.

Veja Heapsort no Cprogramming.com.

Atualizado em 2019-02-01
https://www.ime.usp.br/~pf/algoritmos/

DCC-IME-USP