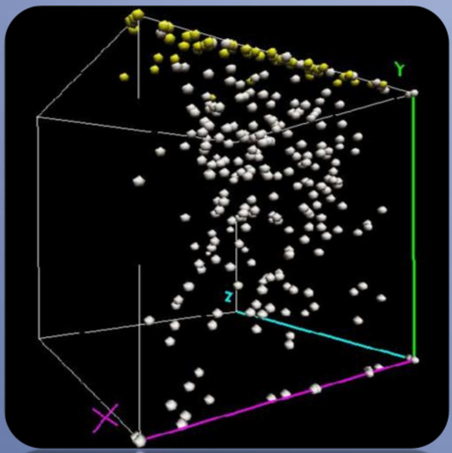


Algoritmos e Estruturas de Dados III

Tutorial 7 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 1 de 3 sobre o algoritmo de ordenação heap (monte) conhecido como Heapsort.

1 Introdução

Esta série de tutoriais sobre *Algoritmos e Estruturas de Dados III* foi escrita usando o **Microsoft Windows 7 Ultimate**, **Microsoft Office 2010**, **Bloodshed Dev-C++** versão 4.9.9.2 (pode ser baixado em http://www.bloodshed.net), referências na internet e notas de aula do professor quando estudante. Ela cobre desde os algoritmos de ordenação, passando pela pesquisa em memória primária e culminando com a pesquisa em memória secundária.

Nós entendemos que você já conhece o compilador Dev-C++. No caso de você ainda não o conhecer, dê uma olhada nos tutoriais Dev-C++ 001 a 017, começando pelo Tutorial Dev-C++ - 001 - Introdução.

Se não tem problemas com a linguagem C/C++ e o compilador Dev-C++, então o próximo passo é saber ler, criar e alterar arquivos em disco usando linguagem C/C++. Se ainda não sabe como fazê-lo, dê uma olhada nos tutoriais Dev-C++ 001 e 002, começando pelo Tutorial Dev-C++ 001 – Criação, Leitura e Alteração de Arquivos.

Se sabe todas as coisas anteriores, então a próxima etapa é conhecer os algoritmos mais básicos de ordenação. Em minhas notas de aula você encontra um material básico, porém detalhado e com algoritmos resolvidos, dos principais métodos de ordenação existentes.

Adotaremos o livro ***Projeto de Algoritmos com Implementação em Pascal e C***, Editora Cengage Learning, de Nivio Ziviani, como livro-texto da disciplina. Nele você encontrará os métodos de ordenação que iremos estudar.

Seu próximo passo será estudar os algoritmos de ordenação por **Inserção**, **Seleção**, **Shellsort** e **Quicksort**. Você pode usar os links anteriores (em inglês) ou fazer uso do livro-texto.

Se você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

2 O Algoritmo de Ordenação Heapsort

O algoritmo Heapsort é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção. Foi desenvolvido em 1964 por Robert W. Floyd e J.W.J. Williams.

Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio. Alguns algoritmos de ordenação rápidos têm desempenhos espetacularmente ruins no pior cenário, quer em tempo de execução, quer no uso da memória. O Heapsort trabalha no lugar e o tempo de execução em pior cenário para ordenar n elementos é de $O(n \log_2 n)$. Para valores de n, razoavelmente grande, o termo $\log_2 n$ é quase constante, de modo que o tempo de ordenação é quase linear com o número de itens a ordenar.

2.1 Uma Olhada no Funcionamento do Algoritmo

O Heapsort utiliza uma estrutura de dados chamada *heap*, para ordenar os elementos a medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da *heap*, na ordem desejada, lembrando-se sempre de manter a propriedade de *max-heap*.

A *heap* pode ser representada como uma árvore (uma árvore binária com propriedades especiais[1]) ou como um vetor. Para uma ordenação crescente, deve ser construído um *heap* máximo (o maior elemento fica na raiz, isto é, primeiro elemento do vetor). Para uma ordenação decrescente, deve ser construído um *heap* mínimo (o menor elemento fica na raiz).

2.1 A função peneira

O coração de qualquer algoritmo que manipule um *max-heap* é uma função que recebe um vetor arbitrário $v[1..m]$ e um índice p e faz $v[p]$ "descer" para sua posição "correta".

Como se faz isso? A ideia é óbvia. Se $v[p] \geq v[2p]$ e $v[p] \geq v[2p+1]$ então não é preciso fazer nada. Se $v[p] < v[2p]$ e $v[2p] \geq v[2p+1]$ então basta trocar $v[p]$ com $v[2p]$ e depois fazer $v[2p]$ "descer" para sua posição "correta". Não é difícil imaginar o que se deve fazer no terceiro caso.

Eis um exemplo com $p = 1$. Cada linha da tabela é uma "foto" do vetor no início de uma iteração.

[1] BAASE, Sara. ***Computer Algorithms: Introduction to Design and Analysis*** (em inglês). 2ª ed. Reading, Massachusetts: Addison-Wesley, 1988. 71 p. ISBN 0-201-06035-3

85	99	98	97	96	95	94	93	92	91	90	89	87	86

99	**85**	98	97	96	95	94	93	92	91	90	89	87	86

99	97	98	**85**	96	95	94	93	92	91	90	89	87	86

99	97	98	93	96	95	94	**85**	92	91	90	89	87	86

Eis uma função iterativa que faz o serviço. A variável f é sempre um filho de p; no início de cada iteração, f é ajustado de modo a ser o filho de maior valor de p.

```
// Recebe p em 1..m e rearranja o vetor v[1..m]
// de modo que o "subvetor" cuja raiz é p seja
// um max-heap.
// Supõe que os "subvetores" cujas raízes são
// filhos de p já são max-heaps.

void peneira (int p, int m, int v[]) {
   int f = 2 * p;
   int x;

   while(f <= m) {
      if((f < m) && (v[f] < v[f + 1])) ++f;
      // f é o filho "mais valioso" de p
      if (v[f / 2] >= v[f]) break;
      x = v[f / 2];
      v[f / 2] = v[f];
      v[f] = x;
      f *= 2;
   }
}
```

A seguinte implementação é um pouco melhor, porque faz menos trocas e executa a divisão $f/2$ uma só vez:

```
void peneira (int p, int m, int v[]) {
   int f = 2 * p;
   int x = v[p];

   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;
}
```

2.1.1 A função peneira: desempenho

A função peneira é muito rápida. O consumo de tempo é proporcional ao número de iterações, e esse número não passa de $\log_2 m$, pois o valor de f pelo menos dobra a cada iteração.

2.2 Exercícios

1. A seguinte alternativa para a função peneira funciona corretamente[2]?

```
void peneira (int p, int m, int v[]) {
   int x;
   int f;

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

2. Escreva uma versão recursiva da função peneira.

3. Por que a seguinte implementação de peneira não funciona?

```
void peneira (int p, int m, int v[]) {
   int x;
   int f = 2 * p;

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

[2] Um algoritmo é correto se cumpre o prometido, ou seja, se faz o que promete fazer.

2.3 Por que um heap é útil?

Por que um *max-heap* é uma estrutura de dados tão poderosa? Suponha que $v[1..m]$ é um *max-heap*; então

a. a pergunta "qual o maior elemento de vetor?" pode ser respondida instantaneamente: o maior elemento do vetor é $v[1]$;
b. se o valor de $v[1]$ for alterado, o *max-heap* pode ser restabelecido muito rapidamente: a operação peneira(1, m, v) não demora mais que $\log_2 m$ para fazer o serviço;
c. um vetor v[1..m] arbitrário pode ser transformado em um *max-heap* muito rapidamente: o comando

   ```
   for(p = m/2; p >= 1; --p) peneira(p, m, v);
   ```

   faz o serviço em tempo proporcional a m. (É fácil ver que o consumo de tempo é limitado por $(m \log_2 m)/2$, pois o tempo gasto em cada uma das $m/2$ iterações é limitado por $\log_2 m$. É um pouco mais difícil verificar que o tempo é, na verdade, limitado por m apenas.)

2.3.1 Exercícios

1. Mostre que o fragmento de programa abaixo faz no máximo m comparações entre elementos do vetor.

   ```
   for(p = m/2; p >= 1; --p) peneira(p, m, v);
   ```

2. O fragmento de programa abaixo transforma um vetor arbitrário $v[1..m]$ em *max-heap*?

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

3. Critique a seguinte ideia: para transformar um vetor arbitrário em *max-heap*, basta colocá-lo em ordem decrescente.

4. Escreva uma função **ff** que receba um vetor v e um índice k tais que $v[1..k - 1]$ é um *max-heap* e transforme $v[1..k]$ em *max-heap*. Sua função deve fazer no máximo $2 \log_2 k$ comparações entre elementos do vetor. Agora use **ff** para construir uma função que transforme qualquer vetor $v[1..m]$ em *max-heap*. Sua função deve fazer no máximo $2m \log_2 m$ comparações entre elementos do vetor.

2.4 O algoritmo heapsort

Não é difícil juntar tudo que dissemos acima para obter um algoritmo que coloque $v[1..n]$ em ordem crescente.

```
#include <>

void peneira (int p, int m, int v[]) {
   int f = 2 * p;
   int x = v[p];

   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;
}

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

void heapsort(int n, int v[]) {
   int p, m, x;

   // usa a função peneira para criar o heap
   printf("Criando heap...\n\n\n");
   for(p = n / 2; p >= 1; --p)
      peneira(p, n, v);

   printf("Heap criado\n");
   for(int i=0; i<n; i++) printf("%d ", v[i]);
   printf("\n\n");

   // usa a função peneira para ordenar o vetor
   printf("Ordenando vetor a partir do heap cria-
do...\n\n\n");
   for(m = n; m >= 2; --m) {
      x = v[1]
      v[1] = v[m];
      v[m] = x;
      peneira(1, m - 1, v);
   }
}
```

```
int main() {
    int v[5] = {5, 2, 7, 10, 4};
    int i;

    printf("Vetor original:\n");
    for(i=0; i<5; i++) printf("%d ", v[i]);
    printf("\n\n");

    heapsort(5, v);

    printf("Vetor ordenado:\n");
    for(i=0; i<5; i++) printf("%d ", v[i]);
    printf("\n\n");

    return 0;
}
```

O primeiro comando **for** transforma o vetor em um *max-heap* recorrendo cerca de $n/2$ vezes à função peneira. Feito isso, temos um processo iterativo controlado pelo segundo comando **for**. No início de cada iteração valem os seguintes invariantes:

- o vetor $v[1..n]$ é uma permutação do vetor original,
- o vetor $v[1..m]$ é um *max-heap*,
- $v[1..m] \leq v[m + 1..n]$ e
- o vetor $v[m + 1..n]$ está em ordem crescente.

É claro que $v[1..n]$ estará em ordem crescente quando m for igual a 1.

1				max-heap				m			crescente		n
888	777	666	555	444	333	222	111	000	999	999	999	999	999
				elementos pequenos							elementos grandes		

2.5 Uma Olhada na Implementação do Algoritmo

Listagem 1:

```
void heapsort(tipo a[], int n) {
    int i = n / 2;
    int pai, filho;
    tipo t;

    for(;;) {
        if(i > 0) {
            i--; t = a[i];
        }
        else {
            n--;
            if(n == 0) return;
            t = a[n]; a[n] = a[0];
        }
        pai = i;
        filho = i * 2 + 1;

        while(filho < n) {
            if((filho + 1 < n) && (a[filho + 1] > a[filho]))
                filho++;
            if(a[filho] > t) {
                a[pai] = a[filho];
                pai = filho;
                filho = pai * 2 + 1;
            }
            else break;
        }
        a[pai] = t;
    }
}
```

2.6 Um Exemplo Ilustrado

O segredo do funcionamento do algoritmo é uma estrutura de dados conhecida como **heap** (= monte). Um *max-heap* é um vetor $v[1..m]$ tal que:

$$v[f/2] \geq v[f]$$

para $f = 2, \ldots, m$. Aqui, como no resto desta página, vamos convencionar que as expressões que figuram como índices de um vetor são sempre *inteiras*. Uma expressão da forma "$f/2$" significa $\lfloor f/2 \rfloor$, ou seja, o **piso** de $f/2$, isto é, a parte inteira do quociente de f por 2.

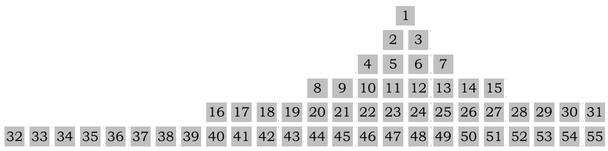
Assim, se $v[1..17]$ é um *max-heap* então, em particular, $v[5] \geq v[10]$ e $v[5] \geq v[11]$:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
999	999	999	999	999	999	999	999	999	999	999	999	999	999	999	999	999

Estranha essa definição de *max-heap*, não? Talvez a coisa fique mais clara se encararmos a sequência de índices $1..m$ como um árvore binária:

- o índice 1 é a *raiz* da árvore;
- o *pai* de um índice f é $f/2$ (é claro que 1 não tem pai);
- o *filho esquerdo* de um índice p é $2p$ e o *filho direito* é $2p + 1$ (é claro que o filho esquerdo só existe se $2p \leq m$ e o filho direito só existe se $2p + 1 \leq m$).

A figura abaixo procura desenhar um vetor $v[1..55]$ de modo que cada filho fique na "camada" imediatamente inferior à do pai. O vetor é definido por $v[i] = i$ e, portanto longe está de ser um *max-heap*. Observe que cada "camada", exceto a última, tem duas vezes mais elementos que a "camada" anterior. Com isso, o número de "camadas" de $v[1..m]$ é exatamente $1 + \lfloor \log_2 m \rfloor$.



Uma vez entendido o conceito de pai e filho, podemos dizer que um vetor é um *max-heap* se o valor de todo pai é maior ou igual que o valor de qualquer de seus dois filhos (onde o *valor* de um índice p é $v[p]$).

2.6.1 Exercícios de fixação

1. O vetor abaixo é um *max-heap*?
   161 41 101 141 71 91 31 21 81 17 16

2. Escreva uma função que decida se um vetor $v[1..m]$ é ou não um *max-heap*.

3. Escreva uma função que verifique se um vetor $v[1..m]$ é ou não um *max-heap* máximo.

4. Escreva uma função que verifique se um vetor $v[1..m]$ é ou não um *max-heap* mínimo.

2.7 Complexidade

Comparações no pior caso: $2n \log_2 n + O(n)$

Trocas no pior caso: $n \log_2 n + O(n)$

Melhor e pior caso: $O(n \log_2 n)$

3 Terminamos

Terminamos por aqui.

Corra para o próximo tutorial.