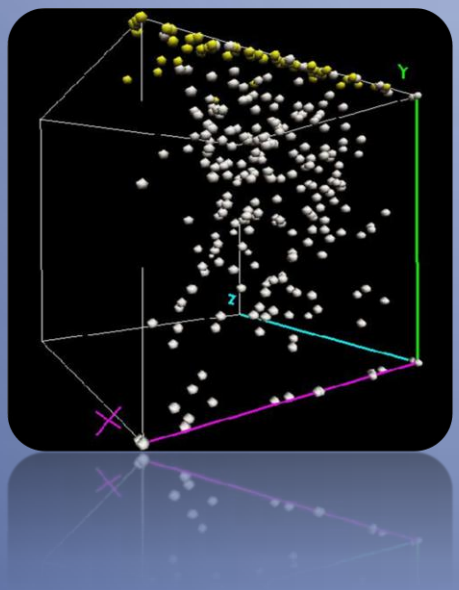

ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 8 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 2 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.

Em seguida, você precisa conhecer o algoritmo Heapsort. Para isto, você pode seguir o **Tutorial AED 007**, desta série, e/ou ler o capítulo referente no livro-texto.

Se você estiver lendo este tutorial tenha certeza de ter seguido o Tutorial AED 007. Agora que você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

2 O ALGORITMO DE ORDENAÇÃO HEAPSORT

2.1 REFAZ

Refaz o heap.

```
void refaz(int esq, int dir, int A[]) {
   int i, j, x;

   i = esq;
   j = 2 * i;
   x = A[i];

   while(j <= dir) {
      if(j < dir)
         if(A[j] < A[j + 1]) j++;
      if(x >= A[j]) goto 999;
   A[i] = A[j];
   i = j;
   j = 2 * i;
}

999: A[i] = x;
```

2.2 CONSTROI

Constrói o heap.

```
void constroi(int n, int A[]) {
   int esq;

   esq = n / 2;
   while(esq > 0) {
   esq--;
   refaz(esq, n, A);
}
```

Usando o *heap* obtido pelo procedimento constrói, pega o elemento na posição 1 do vetor (raiz do *heap*) e troca com o item que está na posição *n* do vetor. Agora usando o procedimento refaz para reorganizar o heap para os itens A[1], A[2], ..., A[*n*-1]. Repete-se as duas operações sucessivamente até que reste apenas um item.

2.3 HEAPSORT

```
void heapsort (int n, int A[]) {
   int esq, dir;

   esq = n / 2;
   dir = n - 1;
   while(esq > 1) {
      esq--;
      refaz(esq, dir, A);
   }

   while(dir > 1) {
      x = A[1];
      A[1] = A[dir]; A[dir] = x; dir--;
      refaz(esq, dir, A);
   }
}
```

2.1 COMPLEXIDADE DO HEAPSORT

Pior caso

Analisando cada parte do algoritmo:

algoritmo refaz

- se a árvore binária com n nós possui altura k, k é o menor inteiro que satisfaz
  $n \leq 2^{k+1} - 1$ e $2^k \leq n \leq 2^{k+1} - 1$, logo $k = \log n$
- o procedimento atua entre os níveis o e (k-1) da subárvore, efetuando, em cada nível, no máximo duas comparações e uma troca. Portanto,
  $= C(n) = 2k \leq 2 \log n \Rightarrow C(n) = O(\log n)$
  $T(n) = k \leq \log n \Rightarrow M(n) = 3T(n) \leq 3 \log n \Rightarrow M(n) = O(\log n)$

algoritmo constrói

- o algoritmo executa o algoritmo refaz (n **div** 2) vezes, portanto,
  $C(n) \leq (n/2)\ 2 \log n = n \log n \Rightarrow O(n \log n)$
  $T(n) \leq (n/2) \log n \Rightarrow M(n) = 3T(n) \leq 3n/2 \log n = O(n \log n)$

algoritmo heapsort

- a ordenação da estrutura *heap* requer a execução do refaz (n-1) vezes, portanto,
  $C(n) \leq (n\text{-}1)\ 2 \log n \Rightarrow C(n) = O(n \log n)$
  $T(n) \leq (n\text{-}1) \log n \Rightarrow M(n) = 3T(n) \leq 3\ (n\text{-}1) \log n = O(n \log n)$
- portanto, no pior caso: $M(n) = O(n \log n)$

Caso Médio

Como nenhum algoritmo de ordenação pode ser inferior a $O(n \log n)$ e $C(n)$ e $M(n)$ são $O(n \log n)$, decorre que $C(n) = O(n \log n)$ e $M(n) = O(n \log n)$.

2.2 MELHORIAS E IMPLEMENTAÇÕES

2.2.1 MELHORIAS

Um algoritmo de ordenação cai na família **ordenação adaptativa**, se tira vantagem da ordenação existente em sua entrada. Ele se beneficia do pré-ordenamento na sequência de entrada - ou uma quantidade limitada de desordem para as diversas definições de medidas de desordem - e ordenação mais rapidamente. A ordenação adaptativa é normalmente realizada modificando-se os algoritmos de ordenação existentes.

2.2.1.1 HEAPSORT ADAPTATIVO

O Heapsort adaptativo é um algoritmo de ordenação que é semelhante ao Heapsort, mas usa um árvore de busca binária aleatória para a estrutura da entrada de acordo com uma ordem preexistente. A árvore de busca binária aleatória é usada para selecionar os candidatos que são colocados no heap, de modo que o heap não precisa se manter a par de todos os elementos. O Heapsort adaptativo é parte da família de algoritmos de ordenação adaptativos.

O primeiro Heapsort adaptativo foi o **Smoothsort** de *Dijkstra*.

2.2.1.1.1 SMOOTHSORT

Algoritmo de ordenação relativamente simples. É um algoritmo de ordenação por comparação.

Smoothsort (método) é um algoritmo de ordenação por comparação. É uma variação do Heapsort desenvolvido por *Edsger Dijkstra* em 1981. Como o Heapsort, o limite superior do Smoothsort é de $O(n \log n)$. A vantagem de Smoothsort é que ele se aproxima de tempo $O(n)$ se a entrada já tem algum grau de ordenação, enquanto a média do Heapsort é de $O(n \log n)$, independentemente do estado inicial em termos de ordenação.

Para uma análise completa sobre o algoritmo Smoothsort, leia o excelente artigo O Algoritmo de Ordenação Smoothsort Explicado, Cadernos do IME – Série Informática, volume 28, página 23.

3 Desenvolva

Tente criar algoritmos de ordenação *heapsort* genéricos (que ordenem ascendente ou descendente e por qualquer chave fornecida) para os problemas a seguir.

1. Ordenação de inteiros;
2. Ordenação de strings;
3. Ordenação de strings a partir de um dado caractere, por exemplo, a partir do 3º caractere;
4. Ordenação de Estruturas por um campo escolhido;
5. Ordenação de estruturas por uma chave composta escolhida;

4 Exercícios Propostos

1. Crie um programa que faça uma comparação entre todos os métodos de ordenação estudados em aula com relação a estabilidade (preservar ordem lexicográfica), ordem de complexidade levando em consideração comparações e movimentações para um vetor de 100 inteiros contendo os 100 primeiros números naturais em ordem decrescente.

2. Escreva um programa para ordenar os 200 primeiros números da lista de 100000 números inteiros fornecida no blog, primeiro usando o *pivot* como o primeiro elemento, depois usando o *pivot* como o elemento central, depois usando um *pivot* escolhido aleatoriamente e por fim, um *pivot* usando a mediana entre três o primeiro, o último e elemento central dos valores.

3. Escreva um programa que leia compromissos para uma agenda, anotando o assunto, a data do compromisso (no formato dd/mm/aaaa) e a hora do compromisso (no formato hh:mm) e ordene os compromissos em ordem crescente de data, hora e assunto.

4. Desenvolver os algoritmos “Inserção”, “seleção”, “Shellsort”, “Quicksort” e “Heapsort” em linguagem C. Usar um registro com três campos: Código (numérico), Nome (string), Endereço (string). A ordenação deve ser pelo código. Os outros dados podem ser preenchidos aleatoriamente.
   Executar os programas com listas contendo 100, 200 e 400 elementos na seguinte situação inicial:
   a) Lista inicial aleatória;
   b) Lista inicial ascendente;
   c) Lista inicial descendente.
   Os valores devem ser atribuídos no próprio programa fonte ou lidos uma única vez no início da execução. Calcular o tempo gasto em cada execução.
   Colocar um contador para calcular o número de comparações executadas. Ao final de cada execução imprimir o contador. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.
   No lugar do contador, colocar um *delay*. Calcular o tempo gasto. Ao final das execuções, fazer uma tabela e comparar os resultados encontrados.
   Fazer um documento texto analisando os dados encontrados. Deve apresentar as tabelas com os dados encontrados e os valores esperados, quando possível. O que se pode concluir observando os dados encontrados nos itens anteriores e a teoria apresentada? Apresente uma comparação com a complexidade apresentada.

5 Terminamos

Terminamos por aqui.

Corra para o próximo tutorial.