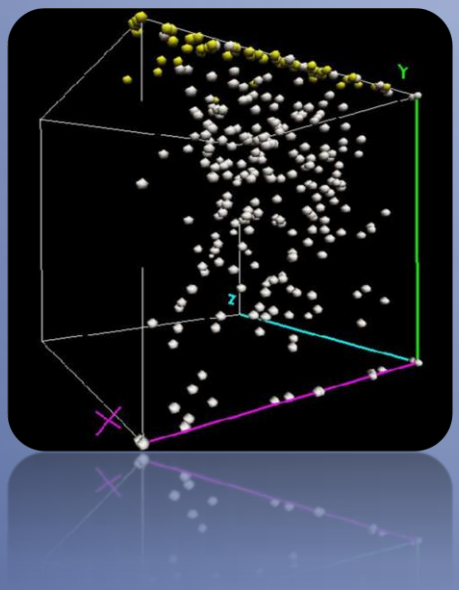



# ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 9 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 3 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 **Heapsort**. 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 os Tutoriais AED 007 e 008. Agora que você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

# 2 Situações Especiais

## 2.1 Ordenação por Índices

**Suponha-se que os objetos a ordenar são strings (vetores de caracteres):**

- funções que operam sobre os dados precisam ter em conta a questão da alocação de memória para strings
- quem deve ser responsável pela gestão desta memória?
- e se os objetos são "grandes"? Comparar e mover os objetos pode ser dispendioso!

**Imagine que cada objeto é o nome completo de um aluno (ou que é toda a sua informação: nome, endereço, telefone etc.):**

- mesmo que haja uma boa abstração para operar sobre os objetos ainda há a questão do custo operacional.

**Por que movê-los?**

- porque não alterar apenas a referência para a sua posição relativa?

**Solução 1:**

- dados numa tabela $data[0], \ldots, data[N-1]$.
- usar uma segunda tabela, $a[.]$, apenas de índices, inicializado de forma que $a[i] = i$, $i = 0, \ldots, N-1$.
  O objetivo é rearranjar a tabela de índices de forma que $a[0]$ aponte para o objeto com a primeira menor chave, $a[1]$ aponte para o objeto com a segunda menor chave, e assim por diante.
- objetos são apenas acessados para comparação

Rotinas de ordenação apenas acessam os dados através de funções de interface. Assim, apenas estas têm de ser reescritas.

Suponha os seguintes dados:

$data$ = [“rui”, “carlos”, “luis”]

Usamos uma tabela de índices:

$a = [0, 1, 2]$

- 1º passo: comparar $data[a[1]]$ com $data[a[0]]$:
  “carlos” < “rui”
  pelo que há troca de $a[1]$ com $a[0]$:
  $a = [1, 0, 2]$

- 2º passo: comparar *data*[*a*[2]] com *data*[*a*[1]]:
  "rui" < "luis"
  pelo que há troca de *a*[2] com *a*[1]:
  *a* = [1, 2, 0]
- 3º passo: comparar *data*[*a*[1]] com *data*[*a*[0]]:
  "carlos" < "luis"
  pelo que não há troca

Os valores ordenados são, portanto,

*data*[*a*[0]], *data*[*a*[1]] e *data*[*a*[2]],

ou seja,

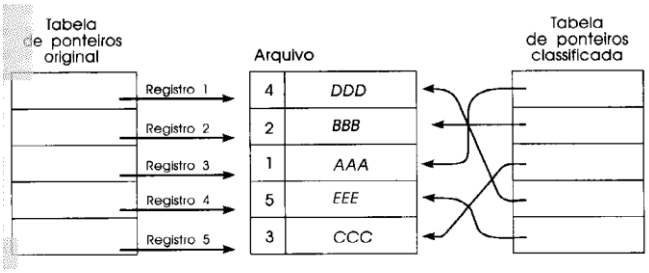
"carlos" < "luis" < "rui"

(de forma camuflada usamos uma "espécie" de Insertion sort).

## 2.2 Ordenação por Ponteiros

Outra solução é a tabela de índices conter de fato ponteiros para os dados. Sua forma é mais geral, pois os ponteiros podem apontar para qualquer lado.

Itens não precisam ser membros de uma tabela, nem de ter todos o mesmo tamanho.




Depois da ordenação, acesso sequencial à tabela de ponteiros devolve os elementos ordenados.

## 2.3 Ordenação por Ponteiros ou Índices

Não-intrusiva em relação aos dados, pois pode ser efetuada se os dados forem apenas de leitura. É possível efetuar ordenação em chaves múltiplas, por exemplo, listagens de alunos, com nome, número e nota. Evita o custo de mover/trocar os itens, que pode ser alto se estes itens representarem grandes quantidades de informação. É mais eficiente em problemas com dados grandes e chaves pequenas.

### 2.3.1 E se for Preciso Retornar os Dados Ordenados?

- ordenar por índice/ponteiro
- fazer permutações in-situ (como?[1])

# 3 Exercícios Resolvidos

1. Considere a seguinte sequência de entrada:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 26 | 34 | 9 | 0 | 4 | 89 | 6 | 15 | 27 | 44 |

   É solicitada a realização de uma classificação em ordem crescente sobre a sequência dada usando o algoritmo de ordenação Heapsort. Mostre como cada passo é executado.

2. Os exercícios de ordenação apresentados até agora solicitam o desenvolvimento de uma ordenação que pode ser classificada como destrutiva, porque a tabela original é destruída e substituída pela tabela ordenada. Uma boa alternativa é criar uma tabela auxiliar cujos índices representam a posição dos elementos na tabela a ser classificada. Faça um programa em C/C++ que use a tabela auxiliar e realize a classificação Heapsort.

3. Um vetor contém os elementos exibidos a seguir. Mostre o conteúdo do vetor depois de ter sido executada a função **constroi** do método Heapsort.

| 24 | 4 | 8 | 14 | 90 | 8 | 67 | 27 | 45 | 19 | 91 | 99 | 58 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

[1] A série iniciada com o **Tutorial AED 010** e finalizada com o **Tutorial AED 016**, faz exatamente isso, trabalhando com ordenação externa.

# 4 Exercícios Propostos

1. Implemente um algoritmo de ordenação Heapsort para ordenar a lista de 10000 inteiros, fornecida no meu blog, gravando a lista ordenada em um arquivo de saída.

2. Crie um algoritmo Heapsort para ordenar o pequeno banco de dados abaixo, usando a chave {departamento, nome, idade}. Você pode conferir a ordenação usando uma planilha eletrônica. **Dica**: *a função compare, do Tutorial AED 005, tópico 2.2.2 usa múltiplas chaves de comparação; dê uma olhada para se inspirar.*

| **Idade** | **Nome** | **Salario** | **Departamento** |
|---|---|---|---|
| 40 | Joao | 100.43 | Matriz |
| 42 | Maria | 200.32 | Filial |
| 35 | Amalia | 50.54 | Matriz |
| 30 | Joao | 150.73 | Filial |
| 32 | Mario | 250.22 | Matriz |
| 25 | Amauri | 60.14 | Matriz |

# 5 Terminamos

Terminamos por aqui.

Corra para o próximo tutorial.