

UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE CIENCIAS EXATAS E NATURAIS

FACULDADE DE COMPUTAÇÃO

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho Final (Breves Conclusões)

Disciplina: Estrutura de Dados I

Professor: Pedro Alberto Bento Gomes

Estudante: Pedro Paulo Lisboa de Sousa/ 201711140038

Belém/PA

2018

**ESTRUTURA DE DADOS - TRABALHO FINAL**

**O QUE FOI PEDIDO?**

**DATAS**

* Apresentações: 10/07/2018
* Entrega do Relatório: 09/07/2018

**REQUISITOS NÃO FUNCIONAIS**

* Linguagem livre desde que não use bibliotecas de Listas, pilhas, filas, ordenação e arvores.
* Entrega de um relatório em PDF e apresentação do código

**REQUISITOS DO PROJETO**

* Leitura de CSV
* Implementar árvores AVL e RB (apenas inserção, busca e "ordem simétrica")
* Implementar algoritmos de ordenação: Seleção, QuickSort, MergeSort, HeapSort e algum método de ordenação em tempo linear
* Implementar Lista, Pilha e Fila (lista encadeada ou matriz)

**Leitura do CSV**

De acordo com o que foi pedido, foi feito um código em JAVA que lê uma base de dados previamente fornecida e localizada dentro da pasta do projeto.

Para leitura foi utilizado um BuffReader para ler a linha completa, em seguida a linha foi dividida em colunas a cada vírgula encontrada. Tais colunas foram adicionados a objetos jogador por meio de um loop utilizando Reflection API, relacionando cada coluna ao seu respectivo atributo.

OBS: Foram identificados 2 registros com valores nulos para o peso, na base de dados. Para contornar o problema, na hora da leitura, esses campos são preenchidos randomicamente.

**Ordenação dos objetos criados a partir do CSV**

Os objetos foram ordenados utilizando o mesmo método, BubbleSort (por ter simples implementação), para ordená-los por ID, Peso e Altura.

* ID: Forma mais rápida de ordenação, pois o ID não apresenta repetições, e a base de dados encontra-se ordenada por id. Caso seja modificado pelo bubble com boolean, o tempo é 0 ms já que a base está ordenada por id.
* Peso: Geralmente demora mais que a ordenação por ID, pois apresenta inúmeras repetições.
* Altura: Mais lenta que a ordenação por ID e geralmente mais rápida que a ordenação por peso.

|  |  |  |  |
| --- | --- | --- | --- |
| Tempo das Ordenações – bubble sem boolean (ms) | | | |
| Testes | ID | Altura | Peso |
| 1 | 15 | 25 | 26 |
| 2 | 15 | 25 | 26 |
| 3 | 15 | 26 | 26 |
| 4 | 15 | 25 | 25 |
| 5 | 17 | 24 | 25 |

**Implementar Algoritmos de Ordenação**

Resultado médio dos tempos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Teste de Tempo de execução (ms) | | | | | |
| Quantidade | SelectionSort | QuickSort | MergeSort | HeapSort | RadixSort |
| 10 | 0 | 0 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 5 | 1 | 2 | 0 | 0 |
| 10000 | 49 | 5 | 4 | 4 | 3 |
| 100000 | 4250 | 95 | 33 | 22 | 12 |
| 1000000 | 442183 | 9074 | 224 | 154 | 64 |

Pude observar que o RadixSort teve tempo menor em todos os testes, já o SelectionSort torna-se inviável para ordenar grandes quantidades de dados.

Em relação ao tempo de complexidade esperado obtém-se a seguinte tabela:

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmo | Complexidade | | |
|  | Melhor caso | Caso médio | Melhor Caso |
| SelectionSort | Ω(n2) | θ(n2) | O(n2) |
| QuickSort | Ω(n log(n)) | θ(n log(n)) | O(n2) |
| MergeSort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| HeapSort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| RadixSort | Ω(nk) | θ(nk) | O(nk) |

**SelectionSort**: Sempre compara um elemento com outros a cada interação, para encontra o menor elemento. Assim não possui um melhor caso mesmo que o vetor esteja ordenado ou em ordem inversa serão executados os dois loops.

Vantagens:

* Ele é um algoritmo simples de ser implementado em comparação aos demais.
* Não necessita de um vetor auxiliar (in-place).
* Por não usar um vetor auxiliar para realizar a ordenação, ele ocupa menos memória.
* Ele é uns dos mais velozes na ordenação de vetores de tamanhos pequenos.

Desvantagens

* Ele é um dos mais lentos para vetores de tamanhos grandes.
* Ele não é estável.
* Ele faz sempre n2 comparações, independente do vetor está ordenado ou não.

**QuickSort**: Usa estratégia de divisão e conquista. Rearranja chaves de modo que as chaves menores precedam as maiores. Em seguida ordena as duas sublistas de chaves menores e maiores até que a lista completa esteja ordenada.

Melhor caso: Ocorre quando ele produz duas listas de tamanho não maior que n/2 (uma com n/2 e outra com n/2 -1).

Pior caso: Quando o pivô divide a lista de forma desbalanceada. Isso ocorre quando o elemento pivô é o maior ou menor elemento da lista, ou seja, quando a lista está ordenada ou inversamente ordenada.

**MergeSort**: Usa estratégia de divisão e conquista. Possui alto consumo de memória sendo inviável em alguns casos.

Os passos do MergeSort são:

1. Calcula o ponto médio do sub-arranjo
2. Recursivamente resolve dois subproblemas, cada um de tamanho n/2
3. Unir os sub-arranjos em um único conjunto ordenado

Desvantagens:

Utiliza funções recursivas

Gasto extra de memória. O algoritmo cria uma cópia do vetor para cada nível da chamada recursiva, totalizando um uso adicional de memória igual a O(n log n)

**HeapSort**: Algoritmo de ordenação generalista, faz parte dos algoritmos de seleção. Possui desempenho em pior cenário praticamente igual ao cenário médio. Utiliza a estrutura de dados heap (arvore binaria especial) para ordenar os elementos à medida que insere na estrutura.

Passos da HeapSort:

* Construir MaxHeap a partir dos dados inseridos
* Neste ponto, o maior elemento é armazenado na raiz do heap. Substitua-o pelo último item do heap seguido pela redução de tamanho do heap por 1.
* Repetir os passos enquanto o tamanho da heap for maior que 1

**RadixSort**: Ordena um vetor A de n números inteiros com um número constante d de dígitos, através de ordenações parciais dígito a dígito. Ele surgiu no contexto de cartões perfurados

|  |  |  |  |
| --- | --- | --- | --- |
| Array Desordenado | Passo1 | Passo 2 | Ordenado |
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 657 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |

**Implementação de Lista**

Foi implementada lista encadeada simples utilizando matriz onde foram criados métodos de inserção (início ou fim) e remoção de elementos da lista, a classe lista usa a classe auxiliar noLista.java.

**Implementação de Fila**

Foi implementada em Java segundo a teoria de fila, na qual só pode inserir no final da fila e só se retira o elemento do início da fila (FIFO).

**Implementação de Pilha**

Conhecida como estrutura LIFO (último a entrar, primeiro a sair). Segundo isso, foi implementado em Java utilizando array de inteiros. Foram criados métodos de inserção e remoção, onde os valores são inseridos exclusivamente no final e retirados também do final da pilha.

**Implementação AVL e RubroNegra**

Foram implementadas as estruturas AVL e RedBlack, nas quais foram criados métodos de inserção, remoção e ainda estou trabalhando no método de busca.

Ainda assim pode-se notar que a RB é mais rápida que a AVL, com 20000 registros a inserção e ordenação da AVL demorou 2614 ms enquanto que a RB demorou 127ms.