UFOP – UNIVERSIDADE FEDERAL DE OURO PRETO
ICEB – INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DECOM – DEPARTAMENTO DE COMPUTAÇÃO

JÚLIA EDUARDA MIRANDA DE SOUSA
JOÃO VITOR GONÇALVES SILVA
MYLLENE FERREIRA SILVA

TRABALHO PRÁTICO 1 – AVALIAÇÃO EMPÍRICA
PROJETO E ANÁLISE DE ALGORITMOS

RELATÓRIO DO TRABALHO

OURO PRETO
2021

JÚLIA EDUARDA MIRANDA DE SOUSA

JOÃO VITOR GONÇALVES SILVA

MYLLENE FERREIRA SILVA

TRABALHO PRÁTICO 1 – AVALIAÇÃO EMPÍRICA

PROJETO E ANÁLISE DE ALGORITMOS

RELATÓRIO DO TRABALHO

Relatório do primeiro trabalho prático da disciplina de Projeto e Análise de Algoritmos, que consiste em uma avaliação empírica de algoritmos de ordenação.

OURO PRETO

2021

OBJETIVO

Este trabalho teve como objetivo desenvolver uma análise empírica dos algoritmos MergeSort, InsertionSort e RadixSort, algoritmos de ordenação amplamente conhecidos e estudados.

INTRODUÇÃO

A ordenação pode ser definida como o processo de rearranjar um conjunto de elementos em uma ordem ascendente ou descendente. Esses elementos devem apresentar valores de um domínio em comum, de modo que é possível estabelecer uma relação entre eles e, consequentemente, ordená-los.

A ordenação pode ser classificada como interna ou externa. Os algoritmos que adotam a estratégia de **ordenação interna** têm os elementos a serem ordenados e seus dados sempre contidos em memória. Já os algoritmos baseados na estratégia de **ordenação externa** manipulam elementos contidos em arquivos de texto, binários e outros dispositivos de armazenamento, ou seja, na memória externa.

Os algoritmos baseados em ordenação interna utilizam de comparações ou de contagem para definir uma ordem entre os elementos, o MergeSort, InsertionSort e RadixSort são algoritmos de ordenação interna. O MergeSort e InsertionSort utilizam a estratégia de comparações e o RadixSort utiliza a estratégia de contagem para ordenar os elementos.

No desenvolvimento do trabalho os algoritmos foram testados de modo quedado um vetor com n valores inteiros como entrada, é retornado um vetor contendo esses valores, de modo que o valor de cada posição i é menor ou igual ao da posição $i+1$, com $1 \leq i < n$.

ANÁLISE DE COMPLEXIDADE

Abaixo estão descritas a análise e as complexidades encontradas para cada algoritmo testado no desenvolvimento do trabalho.

MergeSort – *O(n log n)*

O MergeSort adota a estratégia de Divisão e Conquista, de modo que para facilitar a resolução do problema, a entrada é quebrada em partes menores. Dessa maneira, essas pequenas partições são tratadas de forma separada e à medida que os resultados parciais são obtidos, são combinados para gerar o resultado final.

A maioria dos algoritmos que adotam essa estratégia são recursivos, de modo que dividem o problema em subproblemas de tamanho [*n/b*], sendo *b* o tamanho dos subproblemas. Algumas vantagens apresentadas são o uso eficiente da memória cache, de modo que no processo de combinação dos resultados os dados necessários já estão calculados. Além disso permitem a paralelização, sendo uma estratégia eficiente para casos em que vários processadores estão disponíveis.

A Divisão e Conquista deve ser utilizada em casos que é possível dividir o problema em subproblemas, o processo de combinação de resultados é eficiente e as subinstâncias são mais ou menos do mesmo tamanho. O MergeSort divide o problema em sub problemas do mesmo tamanho, ou seja, de forma balanceada.

A execução desse algoritmo pode ser comparada a uma árvore binária, de modo que cada nó representa uma chamada recursiva, o nó raiz é a chamada principal e os nós folha são o caso base, contendo um ou dois elementos, de modo que sua ordenação é trivial. A complexidade do MergeSort é *O(n log n)*, a altura da árvore de execução é de tamanho *log n*, uma vez que a instância é sempre divida pela metade, e o custo local de cada nível da árvore é *n*, uma vez que em todos os níveis são feitas *n* comparações.

Desse modo, temos *log n* chamadas, cada uma com custo *n,* resultando em uma complexidade $O(n \log_2 n)$. Esse método é interessante para um conjunto grande de dados, uma vez que divide as instâncias em subinstâncias.

```
#include "mergesort.hpp"
// Ordena o vetor v [0..n -1]
void mergeSort (int* v , int n) {
	mergeSort_ordena (v , 0, n -1) ;
}
```

```
// Ordena o vetor v[ esq .. dir ]
void mergeSort_ordena (int *v, int esq, int dir) {
       if ( esq >= dir )
       return ;

       int meio = ( esq + dir ) / 2;
       mergeSort_ordena (v,esq,meio);
       mergeSort_ordena (v,meio+1,dir);
       mergeSort_intercala (v,esq,meio,dir);
}
// Intercala os vetores v[esq .. meio ] e v[ meio +1.. dir ]
void mergeSort_intercala (int* v , int esq , int meio , int dir) {
       int i , j , k ;
       int a_tam = meio - esq +1;
       int b_tam = dir - meio ;
       int* a = new int [a_tam];
       int* b = new int [b_tam];

       for (i=0; i<a_tam ; i++)
              a[i] = v[i+ esq];
              for (i=0; i<b_tam ; i++)
                     b[i] = v[i+meio+1];
                     for (i=0, j=0, k=esq; k<=dir ; k++) {
                            if (i==a_tam)
                                   v[k] = b[j++];
                            else if (j == b_tam)
                                   v[k] = a[i++];
                            else if (a[i] < b[j])
                                   v[k] = a[i++];
                            else
                                   v[k] = b[j++];
                     }
       delete a;
       delete b; }
```

InsertionSort – O(n ²)

O InsertionSort pode ser entendido como o algoritmo utilizado pelo jogador de cartas, as cartas são ordenadas uma por uma da esquerda da direita. A segunda carta é verificada, analisando se deve ficar antes ou na posição que está, depois a terceira carta é classificada, sendo deslocada até sua posição correta e assim por diante.

Consiste em trocar o menor elemento do conjunto pelo elemento que está no início da lista, depois o segundo menor elemento pelo que está na segunda posição e assim sucessivamente, até os dois últimos elementos do conjunto. A complexidade do algoritmo é dada por:

$$(n-1) + (n-2) + \dots + 2 + 1 = [n(n-1)]/2 = n^2$$

Desse modo, temos que o InsertionSort é $O(n^2)$. Esse método é vantajoso para um conjunto pequeno de dados, uma vez que sua complexidade é quadrática.

```
#include "insertion.hpp"

void insertionsort ( int* vector, int n) {
    int i,j;

    for (i = n -2; i >= 0; i --) {
        vector[n] = vector[i];
        j = i + 1;

        while (vector[n] > vector[j] ) {
            vector[j - 1] = vector[j];
            j ++;
        }

    vector[j - 1] = vector[n];
 }
}
```

RadixSort – O(n)

Os algoritmos de ordenação por comparação apresentam limite assintótico inferior *O(n lg n)* porém, exitem algoritmos de ordenação em tempo linear, desde que a entrada apresente características especiais, algumas restrições sejam respeitadas e que o algoritmo não seja baseado puramente em comparações.

O RadixSort pressupõe que as chaves de entrada possuem limite no valor e no tamanho, em sua quantidade de dígitos. É utilizado um segundo algoritmo estável para realizar a ordenação de cada dígito, podendo ser feita a partir do dígito mais ou menos significativo. Adota a estratégia de contagem do número de elementos com cada valor e não pela comparação com os demais elementos.

É criada uma tabela com *n* contadores, varrendo a coleção do início ao fim, incrementando o contador que corresponde à chave *i* cada vez que o valor for encontrado. Quando todos os elementos são varridos, sabe-se quantas posições serão necessárias são necessárias para cada valor, de modo que os elementos são transferidos para as posições corretas.

Cada passado sobre os *n* elementos custa *O(n + Base)*, sendo necessárias *d* passadas: *O(d*n + d*Base)*. Se *d* é constante e *Base = O(n)*, então a complexidade é *O(n)*. A aplicação do princípio de contagem para domínios muito grandes não é muito viável, se tivermos um inteiro de 32 bits, por exemplo, a tabela de contadores teria 2^{32} contadores.

```
#include "radix.hpp"
// C++ implementation of Radix Sort
// A utility function to get maximum value in arr[]
int getMax(int* vector, int n)
{
	int max = vector[0];
	for (int i=1; i<n; i++)
		if (vector[i] > max)
			max = vector[i];
	return max;
}
```

```
// Using counting sort to sort the elements in the basis of significant places
void countingSort(int* array, int size, int place) {
    const int max = 10;
    int output[size];
    int count[max];

    for (int i = 0; i < max; ++i)
        count[i] = 0;
    // Calculate count of elements
    for (int i = 0; i < size; i++)
        count[(array[i] / place) % 10]++;
    // Calculate cumulative count
    for (int i = 1; i < max; i++)
        count[i] += count[i - 1];
    // Place the elements in sorted order
    for (int i = size - 1; i >= 0; i--) {
        output[count[(array[i] / place) % 10] - 1] = array[i];
        count[(array[i] / place) % 10]--;
    }
    for (int i = 0; i < size; i++)
        array[i] = output[i];
}

// Main function to implement radix sort
void radixsort(int* array, int size) {
    // Get maximum element
    int max = getMax(array, size);
    // Apply counting sort to sort elements based on place value.
    for (int place = 1; max / place > 0; place *= 10)
        countingSort(array, size, place);
}
```

DESENVOLVIMENTO

A análise foi realizada de forma empírica, testando o tempo de execução de cada um para instâncias de tamanhos diferentes. Foi proposta a geração de 20 instâncias para cada algoritmo e a avaliação de sua complexidade de tempo. As instâncias são de tamanho n = 100, 1.000, 10.000, 100.000, 1.000.000, …, ou seja, utilizando potências de 10, até a potência 10^{20}.

Os algoritmos de ordenação tiveram suas chamadas inseridas no driver da aplicação, função *main( )*, sendo contado o tempo do início até o final da execução com a função *clock( )*, que conta os milissegundos que passam até do início ao final da execução do bloco de código que encontra entre a função.

```
#include <iostream>

#include <time.h>
#include <string>
#include <cstdlib>
#include "utils.hpp"
#include "mergesort.hpp"
#include "radix.hpp"
#include "insertion.hpp"
using namespace std;

// Driver Code
int main (int argc, char *argv[]) {
    int metodo, quantidade, situacao;
    metodo = atoi(argv[1]);
    quantidade = atoi(argv[2]);
    situacao = atoi(argv[3]);

    Experimentos* experimentos= new Experimentos;

    int *vector = new int[quantidade];

    gerador(vector, quantidade, situacao);
```

```
switch (metodo) {
    case 1: {
        cout<<"Testando o MergeSort..."<<endl;
        clock_t c2, c1; /* variÃ¡veis que contam ciclos do processador */
        c1=clock(); /* coloque aqui o cÃ³digo que vocÃª quer medir o
tempo de execuÃ§Ã£o */
        mergeSort(vector, quantidade);
        c2=clock();
          experimentos->tempo=(c2-c1)*1000/CLOCKS_PER_SEC;/* agora tempo
guarda o tempo de execuÃ§Ã£o em milisegundos */
        printVector(vector, quantidade);
    } break;

    case 2: {
        cout<<"Testando o InsertionSort..."<<endl;
        clock_t c2, c1; /* variÃ¡veis que contam ciclos do processador */
        c1=clock(); /* coloque aqui o cÃ³digo que vocÃª quer medir o
tempo de execuÃ§Ã£o */
        insertionsort(vector, quantidade);
        c2=clock();
         experimentos->tempo=(c2-c1)*1000/CLOCKS_PER_SEC;/* agora tempo
guarda o tempo de execuÃ§Ã£o em milisegundos */
        printVector(vector, quantidade);
    } break;

    case 3: {
        cout<<"Testando o RadixSort..."<<endl;
        clock_t c2, c1; /* variÃ¡veis que contam ciclos do processador */
        c1=clock(); /* coloque aqui o cÃ³digo que vocÃª quer medir o
tempo de execuÃ§Ã£o */
        radixsort(vector, quantidade);
        c2=clock();
         experimentos->tempo=(c2-c1)*1000/CLOCKS_PER_SEC;/* agora tempo
guarda o tempo de execuÃ§Ã£o em milisegundos */
        printVector(vector, quantidade);
    } break;
```

```
        default: {
            printVector(vector, quantidade);
        } break;
    }
    cout<<"----------------------------------------------";
    cout<<"\nTempo de execucao..........: "<<experimentos->tempo<<" milissegundos";
    cout<<"\n----------------------------------------------"<<endl;

    delete [] vector;

    return 0; }
```

RESULTADOS

Não foi possível realizar os testes para instâncias de tamanhos muito grandes, a partir de 10^{10}, uma vez que a instância torna-se muito grande e a complexidade dos algoritmos também, tornando os testes inviáveis.

Com o Merge Sort, de complexidade *O(n log n)* foi possível realizar testes com instâncias maiores, uma vez que a complexidade é logarítmica, de modo que é adequado para instâncias maiores.

Merge Sort	
10^1	0 milissegundos
10^2	0 milissegundos
10^3	0 milissegundos
10^4	0 milissegundos
10^5	31 milissegundos
10^6	312 milissegundos
10^7	3578 milissegundos
10^8	41406 milissegundos
10^9	484062 milissegundos

Tabela 1: Resultados obtidos para o tempo de execução do Merge Sort

Com o Radix Sort foi possível realizar testes para instâncias menores, mesmo que o algoritmo tenha complexidade linear não é adequado para conjuntos de dados muito grandes, uma vez que utiliza a estratégia de contagem, além disso, a entrada tem que apresentar uma configuração específica para que tenha execução rápida.

Radix Sort	
10^1	0 milissegundos
10^2	0 milissegundos
10^3	0 milissegundos
10^4	1 milissegundos
10^5	8 milissegundos

Tabela 2: Resultados obtidos para o tempo de execução do Radix Sort

Com o Insertion Sort também foi possível realizar testes apenas para instâncias menores, uma vez que sua complexidade é $O(n^2)$. O custo de se classificar cada item é muito alto, uma vez que cada um é classificado fazendo comparações com todos os outros itens do conjunto. Desse modo, esse algoritmo não é adequado para instâncias muito grandes, sendo recomendado para conjuntos menores de dados, em casos que o conjunto está parcialmente ordenado ou quando se deseja inserir alguns dados em um conjunto.

Insertion Sort	
10^1	0 milissegundos
10^2	0 milissegundos
10^3	0 milissegundos
10^4	133 milissegundos
10^5	7169 milissegundos
10^6	1336152 milissegundos

Tabela 3: Resultados obtidos para o tempo de execução do Insertion Sort

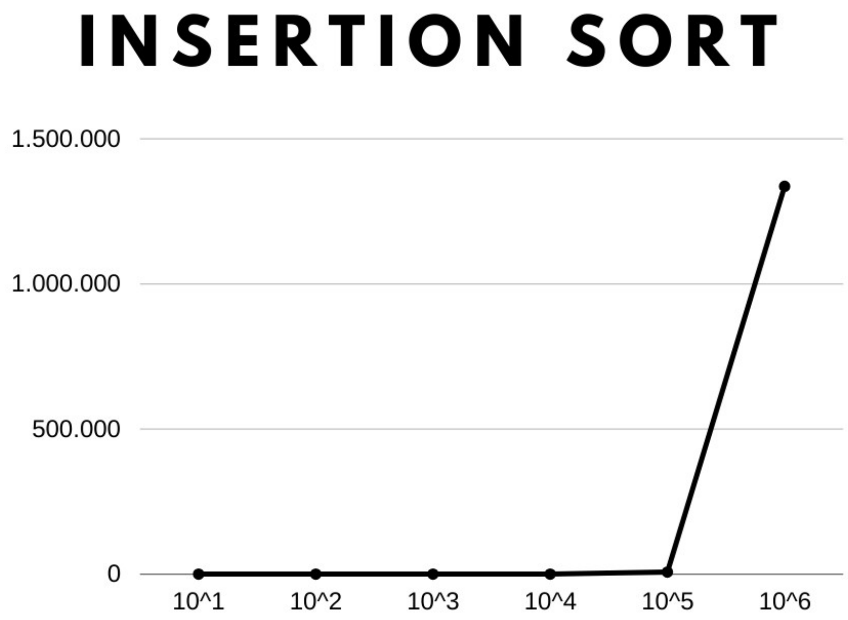

Imagem 1: Resultados obtidos para o tempo de execução do Insertion Sort

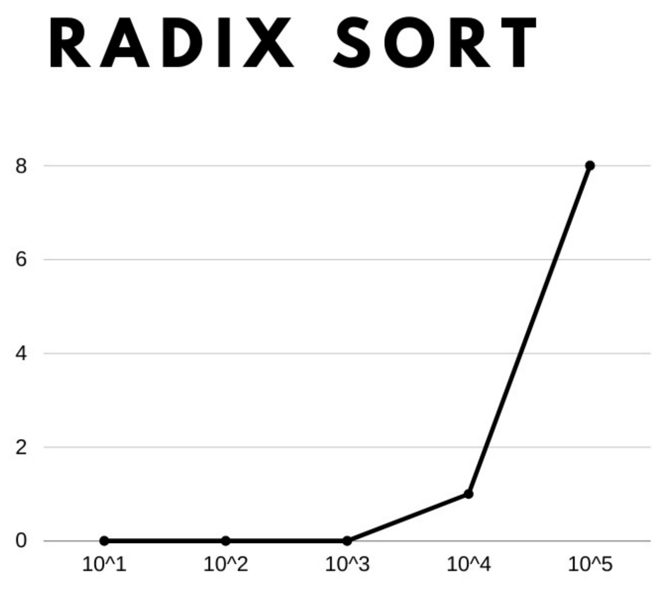

Imagem 2: Resultados obtidos para o tempo de execução do Radix Sort

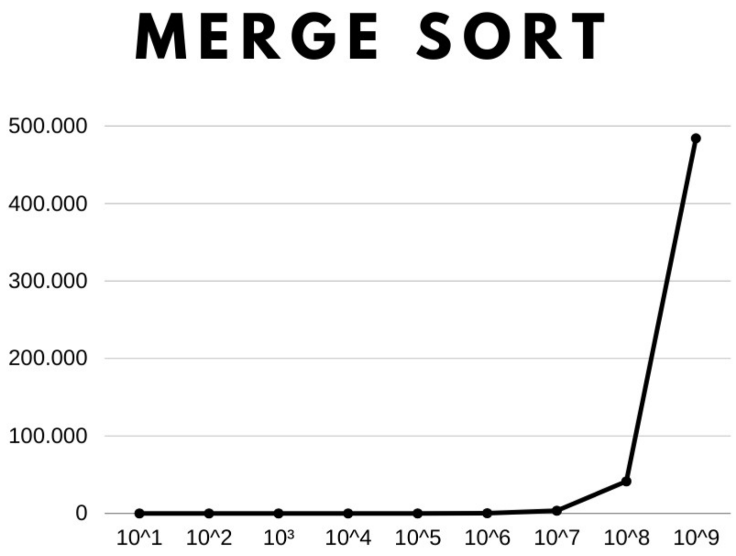

Imagem 3: Resultados obtidos para o tempo de execução do Merge Sort

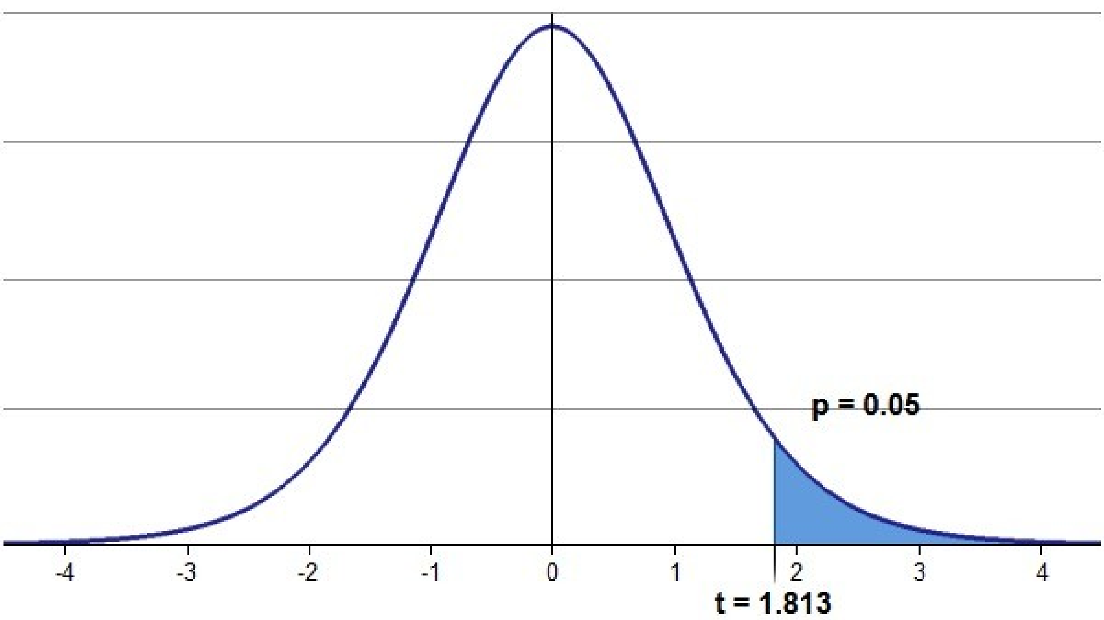

Imagem 4: Distribuição normal para um valor p = 0.05

DISTRIBUIÇÃO T

Abaixo estão descritos os cálculos da distribuição T para cada algoritmo.

Merge Sort

Números totais: 9

Soma dos números: 529389.0

Valor médio: 58821.0

Desvio padrão da população (σ): 150887.81215

$T = 58821.0 + 1.833 * [150887.81215 / \sqrt{9}] = 2.9 * 10^{9}$

Insertion Sort

Números totais: 6

Soma dos números: 1343454.0

Valor médio: 223909.0

Desvio padrão da população (σ): 497417.017814

$T = 223909.0 + 1.833 * [497417.017814 / \sqrt{6}] = 4.54 * 10^{10}$

Radix Sort

Números totais: 5

Soma dos números: 9.0

Valor médio: 1.8

Desvio padrão da população (σ): 3.12409987036

$T = 1.8 + 1.833 * [3.12409987036 / \sqrt{5}] = 5.07$

CONCLUSÃO

O objetivo deste trabalho foi demonstrar uma avaliação empírica de três algoritmos de ordenação o Merge Sort, Insertion Sort e Radix Sort. Esses algoritmos são amplamente conhecidos e estudados, havendo diversos trabalhos científicos desenvolvidos sobre elea. Cada um desses algoritmos apresenta suas estratégias próprias de implementação, sua complexidade e os cenários para qual sua aplicação é vantajosa.

O Merge Sort é um algoritmo de complexidade *O(n log n),* que utiliza as estratégias de comparação e divisão e conquista para ordenar os itens, sendo adequado para instâncias maiores.

O Insertion Sort é um dos algoritmos iniciais de ordenação, sendo de complexidade $O(n^2)$, que também adota a estratégia de comparação para ordenar seus itens. Funciona tal qual um jogador que insere cartas a sua mão, desse modo é recomendável para situações em que o conjunto de dados está quase ordenado ou quando se deseja adicionar poucos itens a um conjunto.

O Radix Sort, diferente da maioria dos algoritmos de ordenação que tem seu limite de complexidade inferior de *O(n log n),* adota a estratégia de contagem para realizar a ordenação. É um algoritmo extremamente rápido, de complexidade *O(n)*, porém, é adequado para casos muito específicos de entrada e para pequenos conjuntos de dados.

Desse modo, temos algoritmos que adotam diferentes estratégias de implementação e complexidades diferentes para diversos casos, de modo que é necessária uma ampla análise do contexto de aplicação, assim como o funcionamento e complexidade dos algoritmos para se utilizar aquele mais adequado.

REFERENCIAL BIBLIOGRÁFICO

LAUREANO, Marcos. “Estrutura de Dados com Algoritmos e C”, 2008.

RICARTE, Ivan Luiz Marques. “Estruturas de dados”, 2008. UNICAMP. São Paulo – SP

JÚNIOR, Edwar Salliba. “Estruturas de Dados – Notas de Aula”, 2007. Faculdade de Tecnologia INED. Belo Horizonte – MG

Material didático fornecido pela professora Amanda Sávio na disciplina de Estruturas de Dados I.

Material didático fornecido pelo professor Guilherme Tavares na disciplina de Estrutura de Dados II.