FIAP ON

CÓDIGOS DE
ALTA PEFORMANCE

BUSCA

PATRICIA MAGNA

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

5 BUSCA

5.1 Introdução

Quando abrimos uma conta em um banco, sabemos que todos os nossos dados que fornecemos na abertura da conta, como: nome, CPF, RG, endereço e telefone, mais as informações que o banco gera para controle interno, como número da agência, número da conta, saldo inicial, entre outras, formarão um registro nosso no banco. Após a criação desse registro, ele será armazenado junto com todos os registros de todos os demais clientes desse banco.

Ao irmos a um caixa do banco e informarmos o número da nossa conta, sabemos que o nosso registro deve ser buscado nesse grande “arquivo”. Assim que for localizado o registro, poderemos ter acesso às informações que desejamos, como por exemplo, o saldo e depois podemos fazer alteração, efetuando um depósito.

Por meio desse exemplo, fica claro que o sistema de informática do banco muitas vezes será “avaliado” por nós como bom ou ruim, em função do tempo necessário para a recuperação do nosso registro na base de dados do banco. Dessa forma, quando vamos recuperar um registro dentro de um grande volume de registros, o tempo de recuperação é crítico.

Portanto, quando vamos desenvolver aplicações nas quais será necessário realizar, com muita frequência, operações de busca às informações em um grande volume de dados, precisamos sempre nos preocupar em utilizar um método que seja mais eficiente, ou seja, que requeira o menor tempo possível.

Neste momento, iniciaremos o nosso estudo sobre alguns métodos de busca de dados. Para este início, precisamos definir termos e situações que serão importantes na descrição e avaliação de cada método.

Um arquivo é formado por um conjunto de registros. Um registro é um conjunto de informações relacionadas entre si. Possíveis exemplos de registros seriam:

- Os dados de uma conta bancária.
- As informações de um aluno em uma faculdade.

A cada registro de uma coleção de registros associa-se uma chave, que é uma informação que distingue aquele registro dos demais. Se a chave somente ocorrer uma vez na coleção (por exemplo: o número da conta ou o número da matrícula do aluno), é chamada chave primária. No caso da possibilidade de múltiplas ocorrências de uma chave, essa é dita chave secundária.

As chaves podem ser internas ao registro, sendo constituídas por um ou mais campos de dados do próprio registro, ou externas. No segundo grupo, usar os índices de um vetor é o exemplo mais claro: o registro, em si, não tem um campo que defina sua posição no vetor, mas, dada essa posição, é possível encontrar esse registro diretamente, via deslocamento na memória, a partir do endereço do primeiro elemento.

Um algoritmo de pesquisa (ou algoritmo de busca) é um algoritmo que aceita uma chave como argumento e tenta descobrir um registro cuja chave seja coincidente com esse argumento, retornando o registro encontrado ou um ponteiro para ele. O algoritmo deve sinalizar a condição de inexistência de um registro com a chave recebida. Algoritmos de busca e inserção têm como propósito incluir registros quando essa condição de inexistência for verificada.

A forma de armazenamento da coleção de registros ou base de dados influencia na escolha dos algoritmos de busca, uma vez que esses dependem de fatores como ordenação, hierarquia, etc. A seguir, veremos algumas técnicas de pesquisa.

5.2 Busca Exaustiva Sequencial

A técnica de busca mais simples é a varredura sequencial exaustiva da coleção de registros.

Essa técnica tem diferentes implementações, conforme a estrutura de dados de armazenamento. Percorre-se o conjunto até o final, independentemente da descoberta de registro ou registros que satisfaçam à condição de busca.

O programa Código Fonte Programa do método de busca sequencial exaustiva escrito em JAVA apresenta como seria a implementação desse método de busca. Para simular a estrutura de dados na qual os registros estão armazenados, é criado

um vetor de elementos inteiros. É criado um vetor (*encontrados*) com as posições em que uma determinada chave escolhida pelo usuário (*chaveproc*) é encontrada.

```
import java.util.*;

public class Busca_Exaustiva_Sequencial{
      public static void main(String[] args)  {

            Scanner entrada = new Scanner(System.in);

      /*simula estrutura de dados com registros*/
            int basededados[] = new int[100];
            System.out.println("Digite numeros inteiros POSITIVOS e -1 para
SAIR: ");
            int item = entrada.nextInt();
            for(int i = 0; item >= 0; i++) {
                  basededados[i] = item;
                  item = entrada.nextInt();
                  }
      /*vetor para armazenar posicao onde registros foram encontrados*/
            int encontrados[] = new int[100];

            System.out.println("Digite chave procurada: ");
            int chaveproc = entrada.nextInt();
            int cont = 0;
            for(int i = 0; i < basededados.length; i++) {
                  if (basededados[i] == chaveproc) {
                        encontrados[cont] = i;
                        cont++;
                        }
                  }

            System.out.println("Posições onde chave foram encontradas");
            for(int i = 0; i < cont; i++)
                  System.out.println(encontrados[i]);

            entrada.close();
        }
}
```

Código Fonte 5.1 – Programa do método de busca sequencial exaustiva escrito em JAVA
Fonte: Elaborado pelo autor (2019)

Apesar de ser um método relativamente ineficiente, a varredura sequencial exaustiva é um método simples de se implementar e não necessita de hierarquização ou ordenação prévias, podendo ser aplicada a dados em memória RAM e em dispositivos auxiliares (discos, *pen drives*, etc.). É extremamente segura e gera resultados corretos mesmo quando os registros estão aleatoriamente distribuídos ou

quando existem chaves duplicadas e o método precisa retornar um conjunto de registros.

As demais técnicas de busca tentam melhorar a eficiência do processo por meio da redução do conjunto de dados verificado, com base em algum conhecimento sobre os dados e seu armazenamento, portanto, são vinculadas a pressupostos que, se falhos, invalidam o resultado da pesquisa.

5.3 Busca Sequencial

A busca sequencial tem como ideia central do algoritmo que o arquivo seja varrido sequencialmente, ou seja, cada registro deve ser verificado até que seja encontrada a chave pesquisada ou que seja atingido o final do arquivo. Observe que, nesse caso, a busca é concluída no encontro da primeira ocorrência da chave.

Esse é o método mais genérico entre todos propostos. Pode-se percorrer o conjunto até o final, quando todos os registros com chaves secundárias duplicadas serão recuperados, ou percorre-se até a descoberta do primeiro registro que satisfaça à condição de busca, recuperando apenas o primeiro registro que contenha a chave.

```
import java.util.*;

public class Busca_Sequencial{
      public static void main(String[] args)  {

            Scanner entrada = new Scanner(System.in);
            int i;

/*simula a criação de uma estrutura de dados com registros*/
            int basededados[] = new int[100];
            System.out.println("Digite números inteiros POSITIVOS ou -1
para SAIR: ");
            int item = entrada.nextInt();
            for(i = 0; item >= 0; i++) {
                  basededados[i] = item;
                  item = entrada.nextInt();
                  }

            System.out.println("Digite chave procurada: ");
            int chaveproc = entrada.nextInt();

/*Compara cada registro da base de dados até encontrar a chave procurada*/
            i = 0;
           while (i < basededados.length && (basededados[i] != chaveproc)
                 i++;
```

```
            int posicao = -1;
            if (i < basededados.length)
                  posicao = i;

            if (posicao >=0)
                  System.out.println(" chave encontrada na posição " +
posicao );
            else
                  System.out.println(" chave não foi encontrada " );

            entrada.close();
      }
}
```

Código Fonte 5.2 – Trecho de código para busca sequencial escrito em JAVA
Fonte: Elaborado pelo autor (2019)

É claro que a busca sequencial tem o seu desempenho melhorado quando a chave procurada encontra-se nas primeiras posições do arquivo. Assim, em situações específicas, ou seja, quando os dados mais frequentemente requisitados estiverem no início do arquivo, a eficiência média do algoritmo será bastante melhorada.

Existem algoritmos que periodicamente movem para o início do arquivo os registros que possuem maior probabilidade de serem procurados e, dessa forma, podem reduzir substancialmente o tempo de busca em uma base de dados, quando a pesquisa for conduzida por esse método de busca. Existem duas possibilidades para essa otimização. A cada vez que o registro é consultado, deve-se:

- Mover para a frente: ou para a primeira posição, assim, caso seja requisitada novamente a busca, o registro será encontrado com apenas uma comparação.

- Mover para a posição anterior: a busca ficará cada vez mais rápida conforme um registro é mais frequentemente pesquisado.

A probabilidade de que um registro seja consultado mais frequentemente do que outros registros, depende muito das características da aplicação. Por exemplo, em um arquivo de motoristas que foram multados em uma determinada rodovia, é melhor deixar os registros das pessoas que já foram multadas mais do que uma vez nas primeiras posições do vetor, pois terão uma chance maior de, mais uma vez, serem multadas.

5.4 Busca Sequencial Indexada

A busca sequencial indexada utiliza a técnica de busca sequencial para um subconjunto ordenado, sendo que todo o arquivo deve ser previamente ordenado. Para separar um arquivo em subconjuntos utiliza-se de uma tabela auxiliar, denominada **tabela de índice**.

Para gerar a tabela de índice, deve-se decidir, previamente, a quantidade de registros que formarão os subconjuntos (*Tamanho_Subconj*). Realizada essa decisão, então, deve ser armazenado, na coluna *k_índice,* o valor da chave usada para ordenação do arquivo e, na coluna *p_indice,* o índice do vetor original em que se inicia o próximo subconjunto. A Figura Modelo de genérico de uma tabela de índice apresenta um esquema genérico da geração da tabela índice, sendo que o valor inicial de i será o número de registros que formam um subconjunto.

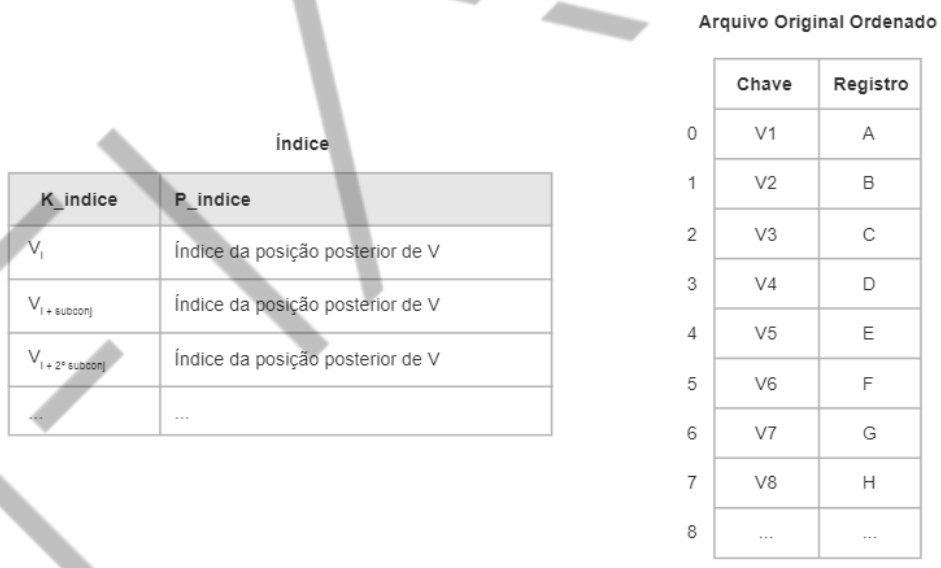

Figura 5.1 – Modelo de genérico de uma tabela de índice
Fonte: Elaborado pelo autor (2019)

Depois de a tabela de índice ter sido gerada, a operação de busca a percorre enquanto a chave de busca for menor do que o valor presente na tabela de índice. Ao fim do processo, o algoritmo pode determinar, na base de dados, um grupo (subconjunto) de registros a ser verificado para a efetiva busca do registro procurado. Esse grupo tem a propriedade de conter o registro procurado se ele existir. Deve ser

garantida a integridade do agrupamento para que não existam registros impropriamente posicionados.

Para entender melhor tanto a geração da tabela de índice quanto a forma de busca, suponha o seguinte exemplo.

Supondo o arquivo original já ordenado, apresentado a seguir, e que o número de elementos que compõem um subconjunto seja igual a quatro, a tabela de índice, então, será como a apresentada na Figura Exemplo de geração de uma tabela de índice. Observe que a tabela índice tem, na primeira linha, o valor da chave armazenada na posição 4 do arquivo original ordenado, já na segunda a chave da posição 8 do arquivo (já que o subconjunto é formado de quatro registros).

Quando se inicia a busca por uma determinada chave, por exemplo, 95, a busca inicia pela tabela índice. Pela primeira comparação verifica-se que 95 é maior que 38, ou seja, não está presente no primeiro subconjunto, a busca continua e como a próxima chave armazenada na tabela índice é 140, determina-se que 95, se existir no arquivo, estará contido no subconjunto formado por quatro registros posicionados entre os índices 4 e 7. Após ter sido determinado o subconjunto em que a chave deve estar, é utilizada uma busca sequencial no arquivo original, assim, verifica-se a não existência da chave.

Índice

K_indice	P_indice
38	4
140	8
…	…

Arquivo Original Ordenado

Chave	Registro
8	
14	
26	
38	
72	
106	
121	
140	
...	...

Figura 5.2 – Exemplo de geração de uma tabela de índice
Fonte: Elaborado pelo autor (2019)

O programa a seguir representa uma forma de implementação do método de busca sequencial indexada. Para poder ampliar o tamanho do vetor, é utilizado no programa o pacote **Random** para realizar a geração de números aleatórios. Além disso, como o vetor deve estar ordenado, um trecho é usado para efetuar a ordenação dos elementos no vetor (Não se preocupe com a forma como isso é feito, o estudo de métodos de ordenação contempla uma explicação do método utilizado).

A criação da tabela de índice ocorre pelos vetores k_indice e p_indice que armazenam o valor do primeiro elemento do subconjunto e a posição que esse elemento ocupa no vetor, respectivamente.

Apenas depois da criação desses vetores, o processo de localização da chave procurada é iniciado pela tabela de índice e, depois, dentro do subconjunto.

```
import java.util.Scanner;
import java.util.Random;

public class Busca_Binaria {
      public static void main(String[] args)  {

            Scanner entrada = new Scanner(System.in);
            Random gerador = new Random();
            int N = 100;
            int i;

      /*cria a estrutura de dados (vetor) com N elementos*/
            int basededados[] = new int[N];
            System.out.println("Criando vetor com 100 elementos: ");
            for(i = 0; i <N; i++)
                  basededados[i] = (int)gerador.nextInt()/1000;


            System.out.println("Ordenando o vetor criado...");
            boolean troca= true;
            for (i=0;i<N-1 && troca==true; i++){
                  int aux;
                  troca= false;
                  for (int j=0;j<N-i-1;j++){
                        if (basededados[j]>basededados[j+1]){
                              aux=basededados[j];
                              basededados[j]=basededados[j+1];
                              basededados[j+1]=aux;
                              troca=true;
                              }
                  }
            }
```

```
            for(i = 0; i <N; i++)
                  System.out.println(i +"\t"+basededados[i]);

/*gera tabela de índice especificando o tamanho do sub-conjunto*/
            int n,j;
            System.out.println("Digite tamanho de cada sub-conjunto ");
            int Tamanho_Subconj = entrada.nextInt();

            j=Tamanho_Subconj;
/*j é o valor do índice armazenado em k_indice sendo, Tamanho_Subconj é o
numero de registros que cada subconjunto tem*/

            n= N/Tamanho_Subconj-1;
/*n guarda o número de linhas na tabela índice*/

            int k_indice[] = new int[100];
//guarda o valor da chave do início do conjunto

           int p_indice[] = new int[100];
//guarda o índice do início do conjunto

            for (i=0;i<=n;i++) {
                  k_indice[i]=basededados[j];
                  p_indice[i]=j;
                  j+=Tamanho_Subconj;
                  }

            System.out.println("Digite chave procurada: ");
            int chaveproc = entrada.nextInt();

/*Procura na tabela de índice o conjunto onde pode estar a chave procurada*/
            System.out.println("Tabela de Índices: " );
            System.out.println("\t p \t k");
            for (i=0;i<=n;i++)
                  System.out.println("\t"+p_indice[i]+ "\t" + k_indice[i]);


            System.out.println("Digite chave procurada: ");
            int chaveproc = entrada.nextInt();

      /*Procura na tabela de índice o conjunto onde pode estar a chave
procurada*/
            int low,hi;
            for (i=0; i<=n && k_indice[i] < chaveproc; i++);
            if (k_indice[i] != chaveproc) {
                  if (i==0)
                        low=0;
                  else
                        low=p_indice[i-1];
                  if (i==n)
                        hi=qtd-1;
                  else
                        hi=p_indice[i];
```

```
            /*Procura na base de dados no intervalo do conjunto
            onde pode estar a chave procurada*/

                  for (j=low; j<hi && basededados[j] != chaveproc; j++) ;}
                  if (j< hi)
                        System.out.println(" chave encontrada na posição "
+ j);
                  else
                        System.out.println(" chave não foi encontrada " );
                  }
            else {
                  System.out.println(" chave encontrada na posição " +
p_indice[i]);
            }
      entrada.close();
      }

}
```

Código Fonte 5.3 – Código para busca sequencial indexada escrito em JAVA
Fonte: Elaborado pelo autor (2019)

Para grandes arquivos, pode-se melhorar o método de busca sequencial indexada criando outra tabela auxiliar, chamada de índice secundário que serve de índice do índice primário. A criação dessa segunda tabela faz com que a inserção e a remoção de registros sejam muito mais complexas, por isso seu uso deve ser feito apenas em casos extremos.

5.5 Busca Binária

Em arquivos nos quais os registros estejam previamente **ordenados**, pode ser utilizado o método de busca binária, que consiste em comparar a chave com o elemento central do conjunto. Se forem iguais, o registro foi encontrado; caso contrário, o processo será repetido na metade inferior ou na metade posterior, conforme o resultado da comparação, até que o registro seja encontrado ou se conclua pela sua inexistência.

A seguir, é apresentado um programa com uma versão não recursiva do algoritmo de busca binária. Novamente, é gerado um vetor com 100 elementos com valores aleatórios e, em seguida, o vetor é ordenado.

```java
import java.util.Scanner;
import java.util.Random;

public class Busca_Binaria {
	public static void main(String[] args)  {

		Scanner entrada = new Scanner(System.in);
		Random gerador = new Random();
		int N = 100;
		int i;

	/*cria a estrutura de dados (vetor) com N elementos*/
		int basededados[] = new int[N];
		System.out.println("Criando vetor com 100 elementos: ");
		for(i = 0; i <N; i++)
			basededados[i] = (int)gerador.nextInt()/1000;


		System.out.println("Ordenando o vetor criado...");
		boolean troca= true;
		for (i=0;i<N-1 && troca==true; i++){
			int aux;
			troca= false;
			for (int j=0;j<N-i-1;j++){
				if (basededados[j]>basededados[j+1]){
					aux=basededados[j];
					basededados[j]=basededados[j+1];
					basededados[j+1]=aux;
					troca=true;
					}
			}
		}

		for(i = 0; i <N; i++)
			System.out.println(i +"\t"+basededados[i]);

		System.out.println("Digite chave procurada: ");
		int chaveproc = entrada.nextInt();

		/*Pesquisa na estrutura de dados o valor solicitado*/
		System.out.println("Procurando o chave solicitada...");
		int i_baixo = 0;
		int i_medio = 0;
		int i_alto = N-1;
		int achou = 0;
		int posicao = -1;
		while( achou != 1 && i_baixo <= i_alto) {
			i_medio = (i_baixo + i_alto)/2;
			if (chaveproc== basededados[i_medio]) {
				posicao = i_medio;
				achou = 1;
				}
```

```
                else {
                        if (chaveproc < basededados[i_medio])
                                i_alto = i_medio - 1;
                        else
                                i_baixo = i_medio + 1;
                        }
                }
        if (posicao == -1)
                System.out.println("Chave não encontrada");
        else
                System.out.println("Chave encontrada posicao "+posicao );
        entrada.close();
        }
```

Código Fonte 5.4 – Código para busca binária escrito em JAVA
Fonte: Elaborado pelo autor (2019)

Esse método tem a desvantagem de apenas funcionar para o caso de armazenamento em vetor.

5.6 Busca Usando Árvore de Binária de Busca (ABB)

Lembrando que uma árvore de busca binária tem como regra de formação que seus nós estejam organizados de forma que todos os descendentes à esquerda de um nó devam ter valores (chaves) MENORES do que os dele e todos os descendentes à sua direita devam possuir chaves MAIORES OU IGUAIS.

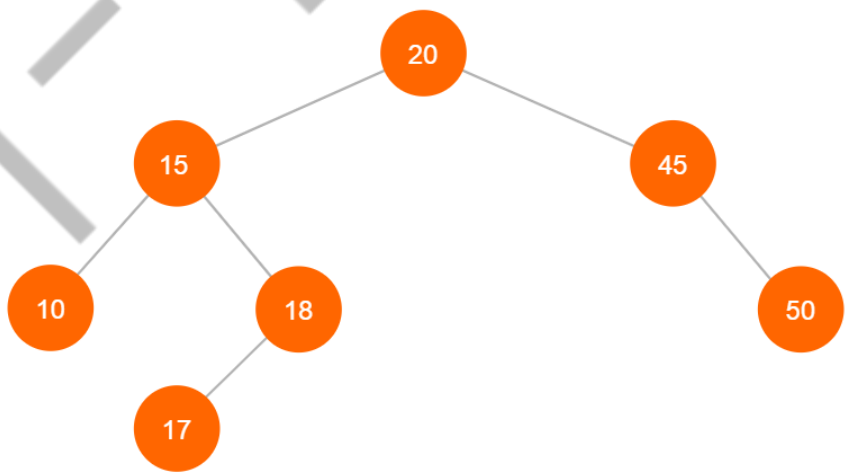

Figura 5.3 – Exemplo de uma Árvore de Busca Binária (ABB)
Fonte: Elaborado pelo autor (2019)

Para realizar a busca de uma determinada chave pode ser usado o seguinte método de busca apresentado no trecho de programa do Código Fonte Código para busca usando árvore de binária de busca (ABB) escrito em JAVA. Para uso desse método, deve-se utilizar o programa desenvolvido para implementar árvores de busca binária (ABB) que armazena valores inteiros.

```
private static ARVORE busca (ARVORE p, int info) {
      // procura elemento na árvore retornando ponteiro para
      // elemento caso o encontre ou null caso contrário
      if (p != null) {
            if (p.dado == info)
                  return p;
            else{
                   if(info > p.dado)
                         p = busca(p.dir, info);
                   else
                         p = busca(p.esq, info);
                    }
            }
      return p;
      }
```

Código Fonte 5.5 – Código para busca usando árvore de binária de busca (ABB) escrito em JAVA
Fonte: Elaborado pelo autor (2019)

5.7 Considerações finais

As pesquisas virtuais estão em nosso dia a dia, então, determinar uma boa organização dos dados pode auxiliar em uma busca mais eficiente. Essa é uma decisão importante que você precisa tomar durante um projeto, já que uma operação de busca é um dos mais comuns entre os problemas computacionais.

5.8 Exercícios Propostos

Utilize o vetor apresentado para as três questões seguintes.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-5	-3	4	10	27	32	45	52	58	66	79	80	97	100	115

1. Aplique o método Busca Binária, contando quantas iterações são necessárias para procurar as chaves 45 e a 97.

RESPOSTA

```
Chave procurada:
45
i_baixo     i_alto     i_medio
0     14          7
i_baixo     i_alto     i_medio
0     6           3
i_baixo     i_alto     i_medio
4     6           5
i_baixo     i_alto     i_medio
6     6           6
 chave encontrada na posição:6   comparações: 4


Chave procurada:
97
i_baixo     i_alto     i_medio
 0          14         7
 8          14         11
 12         14         13
 12         12         12
 chave encontrada na posição 12  comparações 4
```

2. Aplique o método Busca Indexada com tamanho de subconjunto 4, contando quantas iterações são necessárias para procurar as mesmas chaves da questão anterior.

RESPOSTA

Tabela de Indices:

```
    p      k
    4     27
    8     58
    12    97

Chave procurada: 45
comparações na tabela de índices: 2
chave encontrada na posição 6  comparações: 3


Chave procurada: 97
```

```
comparações na tabela de índices: 3
chave encontrada na posição 12      comparações:4
```

3. Analise qual o ganho em usar os métodos dos exercícios anteriores, se comparado com o número de comparações necessárias para buscar as mesmas chaves usando o método Busca sequencial.

RESPOSTA

Se for usado o método de busca sequencial, a chave 45 seria encontrada com 7 comparações e,para a chave 97 seriam necessárias 13 comparações.

chave	Busca sequencial	Busca binária	Busca indexada
45	7	4	3
97	13	4	4

REFERÊNCIAS

ASCÊNCIO, A. F. G.; ARAUJO, G. S. **Estruturas de Dados**: Algoritmos, Análise de Complexidade e Implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010.

DEITEL, P. J.; Deitel, H. M. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

PEREIRA, S. L. **Estruturas de Dados Fundamentais**: Conceitos e Aplicações. São Paulo: Érica, 1996.

TENEMBAUM, A. M. et al. **Estruturas de Dados usando C.** São Paulo: Makron Books Ltda, 1995.

GLOSSÁRIO

Termo	Explicação.
Termo	Explicação.