

Estruturas de Dados e Algoritmos

Gustavo Lermen

COLEÇÃO
EAD
Editora Unisinos

ESTRUTURAS DE DADOS E ALGORITMOS

GUSTAVO LERMEN

EDITORIA UNISINOS

2010

Agradecimentos

Gostaria de agradecer a todos os alunos que por mim já passaram nestes anos em que ministro esta disciplina – seja pelas contribuições em sala de aula, seja pelas conversas de corredor. Agradeço também à professora Patrícia Jaques, do PIPCA da UNISINOS, pelo material fornecido. Muito deste material serviu como base para a produção deste livro.

APRESENTAÇÃO

No contexto de um curso superior em informática, uma disciplina de estruturas de dados e algoritmos pode ser vista como uma das mais importantes por diferentes motivos. Inicialmente, porque esta serve como base para diferentes disciplinas que serão estudadas no decorrer do curso. Outro forte motivo é que um profissional com um sólido conhecimento neste tópico está seguramente mais apto a adaptar-se às constantes mudanças que ocorrem neste campo, onde a tecnologia atual se torna obsoleta rapidamente.

Com isso em mente, o objetivo deste livro é servir como bibliografia básica para as disciplinas de Programação II. Esta é uma disciplina em que serão estudadas diferentes estruturas de dados juntamente com os algoritmos utilizados para manipulá-las. Neste sentido, vale ressaltar que, embora usemos a linguagem de programação Java, não é o foco desta disciplina estudar os conceitos específicos da linguagem Java. Com exceção de alguns conteúdos específicos, a maioria dos conceitos abordados neste livro é independente da linguagem de programação utilizada.

Este livro está organizado da seguinte maneira:

O Capítulo 01 apresenta uma breve introdução à conversão de tipos na linguagem Java. No Capítulo 02, alguns conceitos vistos na disciplina de Programação I serão revistos de modo a introduzir o conceito de interface. Estruturas de dados fundamentais com alocação estática são abordadas no Capítulo 03. O Capítulo 04 aborda as mesmas estruturas vistas no Capítulo 03, utilizando alocação dinâmica. O Capítulo 05 introduz o conceito de recursão e como este pode ser utilizado em conjunto com as estruturas até então estudadas. No Capítulo 06 é introduzido o conceito de árvores e no Capítulo 07 a estrutura de árvore binária é apresentada. O Capítulo 08 apresenta a estrutura de árvore binária de pesquisa. Árvores binárias de pesquisa balanceadas do tipo AVL são apresentadas no Capítulo 09. Árvores genéricas são apresentadas no Capítulo 10. O Capítulo 11 é destinado a métodos de pesquisa e no Capítulo 12 são apresentados alguns métodos de classificação de dados.

SUMÁRIO

CAPÍTULO 1 – CONVERSÃO DE TIPOS

Casting

Atribuição

String

Boxing/unboxing

CAPÍTULO 2 – ORIENTAÇÃO A OBJETOS

Tipos abstratos de dados

Exemplos de TADs

Conceitos fundamentais

Classe

Objeto

Classe x Objeto

Método

Herança

Abstração

Encapsulamento

Polimorfismo

Classes abstratas

Interfaces

Definindo uma *interface*

Implementando uma *interface*

Utilizando *interfaces*

CAPÍTULO 3 – LISTAS LINEARES COM ALOCAÇÃO SEQUENCIAL

Listas lineares gerais

Pilha

Fila

CAPÍTULO 4 – LISTAS LINEARES COM ALOCAÇÃO DINÂMICA

Classes autorreferenciais

- Gerenciamento de memória
- Lista simplesmente encadeada
- Pilha
- Fila
- Lista duplamente encadeada

CAPÍTULO 5 – RECURSÃO

- Pilha de chamadas

CAPÍTULO 6 – ÁRVORES

- Definição e terminologia

CAPÍTULO 7 – ÁRVORES BINÁRIAS

- Aplicação de árvores binárias

CAPÍTULO 8 – ÁRVORES BINÁRIAS DE PESQUISA

- Construção de uma árvore binária de pesquisa

- Classe BST (Binary Search Tree)

- Busca

- Inserção

- Caminhamento

- Caminhamento em ordem

- Caminhamento em pré-ordem

- Caminhamento em pós-ordem

- Remoção

CAPÍTULO 9 – ÁRVORES AVL

- Rotação simples à direita

- Rotação simples à esquerda

- Rotação dupla à direita

- Rotação dupla à esquerda

- Inserção em uma árvore AVL

- Remoção em uma árvore AVL

CAPÍTULO 10 – ÁRVORES GENÉRICAS

CAPÍTULO 11 – MÉTODOS DE PESQUISA

Pesquisa sequencial

Pesquisa binária

CAPÍTULO 12 – CLASSIFICAÇÃO DE DADOS

Inserção direta

Método bolha (*bubble sort*)

Seleção direta

Quicksort

CAPÍTULO 13 – APÊNDICE A – EXCEÇÕES

Tratando exceções

Hierarquia de exceções

Lançando uma exceção

Criando uma classe de exceção

Bloco *finally*

Método *printStackTrace*

REFERÊNCIAS BIBLIOGRÁFICAS

CAPÍTULO 1

CONVERSÃO DE TIPOS

Toda expressão escrita em Java possui um tipo associado que pode ser determinado a partir da estrutura da expressão e também dos tipos de seus componentes: literais, variáveis, métodos etc.

Existem situações em que o tipo da expressão pode não ser apropriado para um determinado contexto. Neste caso duas situações podem ocorrer: erro de compilação ou uma conversão implícita de modo a adaptar a expressão ao contexto.

A linguagem Java possui diferentes contextos de conversão, e em cada um deles apenas certas conversões específicas são permitidas. Os tipos de conversões permitidos podem ser classificados em diferentes grupos que serão abordados nas próximas subseções:

Casting

Ocorre quando o tipo a ser convertido é explicitado através do operador de *cast*.

O exemplo abaixo demonstra a utilização do operador de *casting* na conversão de uma variável do tipo *float* para uma variável do tipo inteiro. Inicialmente, duas variáveis, uma do tipo float e outra do tipo int, são declaradas (linhas 3 e 4). A variável *f* é inicializada com o valor 12.5 juntamente com sua declaração.

```
1  public class TypeConversion1 {  
2      public static void main(String[] args) {  
3          float f = 12.5f;  
4          int I;  
5          //i = f; Erro de compilação. Operador de cast é necessário.  
6  
7          i = (int) f;  
8  
9          //Valor de I e de f é implicitamente convertido para String.  
10         System.out.println("Valor de f: " + f);  
11         System.out.println("Valor de i: " + i);  
12     }  
13 }
```

Exemplo 1 – Conversão utilizando operador de *casting*.

O código da linha cinco está comentado intencionalmente, pois este gera um erro de compilação. Ao tentar atribuir o valor de uma variável do tipo *float* para uma variável do tipo *int* existe perda de precisão. Por isso o erro de compilação.

Para realizar esta conversão, é necessário utilizar o operador de *casting* (linha sete). Este operador explica ao compilador a intenção de converter a variável do tipo *float* para o tipo *int* mesmo que com isso ocorra uma perda de precisão.

Ao executar o exemplo 1, a seguinte saída é exibida:

```
Valor de f: 12.5  
Valor de t : 12
```

Saída do exemplo 1.

A conversão entre tipos primitivos pode ocorrer com ou sem a utilização do operador de *casting*. Sempre que a conversão acarreta uma perda de precisão o operador se faz necessário. Quando a conversão não implica perda de precisão (para tipos numéricos), não há necessidade da utilização do operador.

A tabela abaixo demonstra as possíveis conversões entre tipos primitivos. O símbolo C indica que o operador de *casting* se faz necessário para a conversão. O símbolo D indica que a conversão ocorre de maneira direta (sem a necessidade do operador de *casting*). O símbolo N indica que a conversão não é possível.

	int	long	float	double	char	byte	short	boolean
int	-	D	D*	D	C	C	C	N
long	C	-	D*	D*	C	C	C	N
float	C	C	-	D	C	C	C	N
double	C	C	C	-	C	C	C	N
char	D	D	D	D	-	C	C	N
byte	D	D	D	D	C	-	D	N
short	D	D	D	D	C	C	-	N
boolean	N	N	N	N	N	N	N	-

Tabela 1 – Conversão entre tipos primitivos.

O asterisco em algumas das células da tabela indica que pode ocorrer perda de precisão em alguns casos em que o tipo alvo suporta números maiores. Isso pode acontecer, por exemplo, com um número inteiro que utiliza todos os 32 bits a ele destinados. Quando este é convertido para um tipo *float*, que também utiliza 32 bits

para sua representação, mas desses 8 bits são reservados para o expoente, alguns dígitos menos significativos são perdidos na conversão.

Atribuição

Ocorre quando o tipo de uma expressão é convertido para o tipo da variável que recebe seu valor.

```
1  public class TypeConversion2 {  
2  
3      public static void main(String[] args) {  
4  
5          //Operandos do tipo inteiro  
6          int intOper1      = 2;  
7          int intOper2      = 3;  
8  
9          float floatOper1   = 3.5f;  
10         float floatOper2   = 2.6f;  
11  
12         float resultFloat = 0.0f;  
13         int resultInt     = 0;  
14  
15         //resultado armazenando em uma variável do tipo int  
16         resultInt        = intOper1 + intOper2;  
17  
18         System.out.println("Resultado da operação (int) : " + resultInt);  
19  
20         //resultado armazenado em uma variável do tipo float (conversão implícita)  
21         resultFloat       = intOper1 + intOper2;  
22  
23         //cast explícito é necessário  
24         resultInt        = (int)floatOper1 + intOper2;  
25  
26         System.out.println("Resultado da operação (int) : " + resultInt);  
27         System.out.println("Resultado da operação (float): " + resultFloat);  
28  
29         //promoção numérica - Operador intOper1 é implicitamente tratado como float  
30         resultFloat       = intOper1 + floatOper1;  
31  
32         System.out.println("Resultado da operação (float): "+ resultFloat);  
33     }  
34 }  
35 }
```

Exemplo 2 – Conversão por atribuição.

Neste exemplo pode-se verificar a conversão por atribuição em diferentes situações. Inicialmente, diferentes variáveis, dos tipos *int* e *float* são declaradas (linhas

6 a 13). Na linha 16 duas variáveis do tipo *int* são somadas e o resultado é atribuído a uma terceira variável também do tipo *int*. Neste caso nenhuma conversão ocorre, pois todas as variáveis possuem o mesmo tipo.

Na linha 20 duas variáveis do tipo *int* são somadas e o resultado é atribuído a uma variável do tipo *float*. Nesta situação ocorre uma conversão implícita. Mesmo que o resultado da adição seja do tipo *int*, como este é atribuído a uma variável do tipo *float*, o resultado é convertido automaticamente.

Na linha 24 uma variável do tipo *float* é somada com uma variável do tipo *int* e o resultado é armazenado em uma variável do tipo *int*. Para que não ocorra um erro de compilação, um *casting* explícito é realizado na variável *floatOper1*.

Na linha 30 é demonstrado um exemplo de promoção numérica: nesta atribuição a variável *intOper1* é automaticamente convertida (promovida) para o tipo *float*. Esta conversão é realizada de maneira implícita pela máquina virtual Java toda vez que um tipo com menor precisão (*int*) é utilizado em uma expressão com um tipo de maior precisão (*float* neste caso).

```
Resultado da operação (int) : 5
Resultado da operação (int) : 6
Resultado da operação (float) : 5.0
Resultado da operação (float) : 5.5
```

Saída do exemplo 2.

String

A conversão de tipos primitivos (e também referências) é bastante facilitada na linguagem Java, pois esta é feita de maneira implícita pela máquina virtual em diferentes situações.

Esse tipo de conversão pode ser visto no exemplo 2 toda vez que o conteúdo de uma variável é impresso na tela. Nesta situação, mesmo a variável sendo do tipo *int* ou *float*, seu conteúdo é automaticamente convertido para *String* antes de ser impresso na tela.

Existem diferentes maneiras de converter um tipo primitivo para o tipo *String*. Uma delas é através da utilização do método estático *valueOf* da classe *String*. Este método é sobrecarregado de modo a receber diferentes tipos primitivos como

argumento. Cada versão deste método retorna à representação na forma de String do tipo correspondente.

```
1  public class TypeConversion6 {  
2  
3      public static void main(String[] args) {  
4          String s = String.valueOf(true);           // true  
5          System.out.println(s);  
6  
7          s = String.valueOf('a');                 // a  
8          System.out.println(s);  
9  
10         s = String.valueOf((short)123);        // 123  
11         System.out.println(s);  
12  
13         s = String.valueOf(123);               // 123  
14         System.out.println(s);  
15  
16         s = String.valueOf(123L);              // 123  
17         System.out.println(s);  
18  
19         s = String.valueOf(1.23F);             // 1.23  
20         System.out.println(s);  
21  
22         s = String.valueOf(1.23D);             // 1.23  
23         System.out.println(s);  
24     }  
25 }
```

Exemplo 3 – Conversão de tipos primitivos para String utilizando o método *valueOf*.

Outra maneira de realizar a conversão de tipos primitivos para String é através da utilização das classes chamadas *wrappers*. Para cada um dos oito tipos primitivos, a API¹ da linguagem Java disponibiliza uma classe *wrapper* correspondente. Estas classes *wrapper* disponibilizam, entre outras funcionalidades, métodos que podem ser utilizados para converter tipos primitivos para String.

Tipo primitivo	Classe <i>wrapper</i>
byte	Byte
short	Short
Int	Integer
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

Tabela 2 – Tipos primitivos e suas classes *wrapper* correspondentes.

Os métodos disponibilizados pelas classes *wrapper* para a conversão de tipos primitivos para String são demonstrados no exemplo 4. Este exemplo mostra também métodos disponibilizados pelas classes *wrapper* para converter uma String para diferentes tipos primitivos.

```
1 public class TypeConversion5 {  
2  
3     public static void main(String[] args) {  
4  
5         String str = new String("10");  
6  
7         int i = Integer.parseInt(str);  
8         double d = Double.parseDouble(str);  
9  
10        System.out.println("String convertido para int: " + i);  
11        System.out.println("String convertido para double: " + d);  
12  
13        i = i + 1;  
14  
15        str = new Integer(i).toString();  
16  
17        System.out.println("Int convertido para String: " + str);  
18  
19        d = d * 2;  
20  
21        str = new Double(d).toString();  
22  
23        System.out.println("Double convertido para String: " + str);  
24  
25    }  
}
```

Exemplo 4 – Conversão de tipos primitivos para String utilizando classes *wrapper*.

Na linha 5 do exemplo 4 um objeto do tipo String é criado com o valor “10”. Na linha 7 é mostrado como este objeto do tipo String pode ser convertido para um tipo primitivo *int*. Para realizar esta conversão o método estático *parseInt* da classe Integer foi utilizado. Na linha oito o mesmo objeto String é utilizado para converter seu valor para o tipo *double*. Para esta conversão foi utilizado o método estático da classe Double.

Todas as classes *wrapper* disponibilizam métodos semelhantes para a conversão de Strings para os tipos primitivos correspondentes. Caso a conversão não possa ser realizada, devido a uma String literal mal formada por exemplo, uma exceção é gerada.

```
String convertido para int : 18.0  
Int convertido para String : 11  
Double convertido para String : 20.0
```

Saída do exemplo 4.

Ainda no exemplo 4, as linhas 15 e 21 mostram como um tipo primitivo pode ser convertido para o tipo `String`, utilizando para isso as classes *wrapper* correspondentes.

Boxing/unboxing

O conceito de *boxing* e *unboxing* foi introduzido na versão 1.5 da linguagem Java. Eles permitem que tipos primitivos possam ser utilizados no lugar de suas classes *wrapper* correspondentes e vice-versa.

Toda vez que um tipo primitivo é utilizado no lugar de uma classe *wrapper*, o ambiente de execução da linguagem Java executa o *unboxing*. Em outras palavras, o tipo primitivo armazenado na classe *wrapper* é automaticamente retornado.

Da mesma maneira, toda vez que uma classe *wrapper* é utilizada no lugar de um tipo primitivo, o ambiente de execução da linguagem Java executa o *boxing*. Em outras palavras, um objeto do tipo da classe *wrapper* correspondente é instanciado automaticamente pelo ambiente de execução. Este objeto armazena internamente o valor do tipo primitivo.

```
1  public class TypeConversion4 {
2
3      public static Float soma(Float f1, Float f2){
4          return f1 + f2;
5
6      }
7
8      public static void main(String[] args) {
9
10         Integer i = 5;
11         System.out.println("Valor de i: " + i);
12
13         Float obFloat1 = new Float(2.6f);
14         Float obFloat2 = new Float(2.5f);
15
16         float f = soma(obFloat1, obFloat2);
17         Float obFloat3 = soma(obFloat1, obFloat2);
18
19         System.out.println("Valor de f (tipo primitivo): " + f);
20         System.out.println("Valor de obFloat3 (wrapper): " + obFloat3);
21         System.out.println("Soma utilizando argumento do tipo float: " + soma(f,f));
22
23     }
}
```

Exemplo 5 – Utilização *boxing* e *unboxing*.

Neste exemplo, além do método main, a classe TypeConversion4 apresenta também o método soma. Este método recebe como parâmetro dois objetos do tipo Float (classe *wrapper*) e também retorna um objeto do tipo Float. A implementação deste método contém apenas uma linha onde a soma dos objetos é realizada da mesma maneira que uma soma envolvendo tipos primitivos.

No método main desta classe, na linha 10, um objeto do tipo Integer é declarado e inicializado com o valor 5. Esta linha é equivalente a uma declaração seguida de uma instanciação. Neste caso a instanciação não é necessária, pois o objeto do tipo Integer é instanciado automaticamente (*boxing*) pelo ambiente de execução.

Nas linhas 13 e 14 dois objetos do tipo Float são declarados e instanciados. Estes objetos poderiam ter sido instanciados da mesma maneira que o objeto do tipo Integer na linha 10.

Na linha 16 o conceito de *unboxing* é demonstrado. O método soma é chamado recebendo os dois objetos do tipo Float instanciados anteriormente e seu retorno é armazenado em uma variável do tipo primitivo Float.

Na linha seguinte o mesmo método é chamado novamente, só que desta vez seu retorno é armazenado em um objeto do tipo Float. Através do conceito de *boxing* e *unboxing* este método pode ser chamado de diferentes maneiras, não importando se os

parâmetros ou retorno são tipos primitivos ou objetos.

Na linha 21 o método soma é chamado recebendo como parâmetro dois tipos primitivos. Neste caso o retorno é convertido automaticamente para o tipo String, pois a chamada do método está sendo concatenada com uma String que será mostrada na tela.

```
Valor de i: 5  
Valor de f(tipo primitivo): 5.1  
Valor de obFloat3 (wrapper): 5.1  
Soma utilizando argumento do tipo Float: 10.2
```

Saída do exemplo 5.

A linguagem Java, por ser fortemente tipada, exige que qualquer identificador seja declarado com uma especificação de tipo. Isso é válido tanto para tipos primitivos quanto para tipos de referência. Muitas vezes, por diferentes razões, há a necessidade de um tipo ser convertido de modo a executar alguma tarefa específica, e para realizar esta conversão existem diferentes alternativas.

Este capítulo apresentou diferentes tipos de conversão de tipos que podem ser utilizados na linguagem Java, juntamente com exemplos para cada um deles. Mais informações sobre as técnicas de conversões apresentadas neste capítulo podem ser encontradas no livro *The Java Language Specification* disponível em <http://java.sun.com/docs/books/jls/>.

¹ Application Programming Interface.

CAPÍTULO 2

ORIENTAÇÃO A OBJETOS

O objetivo deste capítulo é revisar alguns conceitos fundamentais de programação orientada a objetos (POO) vistos durante a atividade de Programação I e também introduzir conceitos novos ainda não apresentados.

A programação orientada a objetos é um paradigma de programação que (como o nome já sugere) utiliza “objetos” e suas interações no projeto de programas de computador. Técnicas de programação orientada a objeto compreendem os seguintes conceitos que veremos no decorrer deste capítulo:

- ▶ Classe
- ▶ Objeto
- ▶ Método
- ▶ Herança
- ▶ Abstração
- ▶ Encapsulamento
- ▶ Polimorfismo
- ▶ Classes abstratas
- ▶ Interfaces

A programação orientada a objetos teve seu início na década de 1960. Naquela época, recursos de *software* e de *hardware* estavam se tornando cada vez mais complexos, muitas vezes fazendo com que a qualidade do *software* ficasse comprometida devido a esta complexidade. Foi nesta época que pesquisadores começaram a estudar maneiras de manter a qualidade de *software* neste cenário. Neste sentido, a POO foi proposta, em parte, para solucionar problemas comuns ao desenvolvimento de *software* através da reutilização de unidades lógicas de programação.

A POO pode ser vista como um conjunto de objetos que cooperam de modo a

resolver algum problema, diferentemente da programação estruturada, onde um programa nada mais é do que uma coleção de procedimentos ou funções. Na POO cada objeto é capaz de receber mensagens, processar dados e também enviar mensagens para outros objetos.

Cada objeto pode ser visto como uma “máquina” independente com um papel ou uma responsabilidade distinta. As ações aplicadas ao objeto estão fortemente relacionadas com os objetos em si. Veremos, no decorrer desta atividade, como isso se reflete nas estruturas de dados que estudaremos.

De modo a compreender a programação orientada a objetos, um conceito fundamental a ser tratado diz respeito aos tipos abstratos de dados. Este conceito será visto na próxima seção para então tratarmos dos conceitos básicos da programação orientada a objetos.

Tipos abstratos de dados

Um tipo abstrato de dado, ou ainda TAD, é um conjunto de dados e operações que são aplicadas a estes dados. Independente da linguagem de programação utilizada para sua implementação, as operações em um TAD têm dois papéis: descrever os dados para o resto do programa e também permitir que o programa altere os dados. Neste sentido, quem irá utilizar o TAD enxerga apenas a interface deste. Interface esta que é disponibilizada pelas operações que podem ser aplicadas aos dados.

Dentre os benefícios da utilização de TADs pode-se citar:

- ◆ Ocultação de detalhes de implementação
 - Ao utilizar um TAD você pode modificar detalhes de implementação sem afetar outras partes do programa.
- ◆ Interfaces mais informativas
 - Uma vez que um TAD expõe funções que atuam apenas nos dados pertencentes a ele, as interfaces tornam-se naturalmente mais informativas, uma vez que dados e operações estão fortemente relacionados.
- ◆ Capacidade de trabalhar com entidades do mundo real em vez de implementações de baixo nível.
 - TADs podem ser implementados de modo que as operações por eles expostas escondam detalhes específicos de implementação, tornando a

utilização do TAD mais facilitada. Um TAD Veiculo, por exemplo, pode expor as operações acelerar e frear dentre outras.

- ◆ Reutilização de código

Uma vez que o TAD esteja definido em termos de seus dados e operações, este pode ser utilizado em diferentes programas sem maior esforço.

Exemplos de TADs

Suponha que você esteja escrevendo um *software* que controla o sistema de resfriamento de um reator nuclear. Você pode definir o sistema de resfriamento como um tipo abstrato de dados definindo as seguintes operações para ele:

- ◆ Obter a temperatura
- ◆ Configurar a taxa de circulação
- ◆ Abrir válvula
- ◆ Fechar válvula

O ambiente específico para o qual este sistema de resfriamento for aplicado é quem irá determinar os detalhes de implementação do mesmo. O restante do programa pode lidar com o sistema de resfriamento apenas em termos das operações descritas acima, não tendo que se preocupar com detalhes específicos de implementação.

Assim como o sistema de resfriamento do reator, outras entidades podem ser representadas por TADs, como mostra a tabela a seguir.

Tipo abstrato de dado	Operações
Tanque de gasolina	<ul style="list-style-type: none">◆ Encher o tanque◆ Esvaziar o tanque◆ Obter a capacidade do tanque◆ Obter a quantidade de gasolina no tanque
Elevador	<ul style="list-style-type: none">◆ Subir um andar◆ Descer um andar◆ Ir para um andar específico

	<ul style="list-style-type: none"> ◆ Exibir andar corrente ◆ Retornar ao andar térreo
Arquivo	<ul style="list-style-type: none"> ◆ Abrir arquivo ◆ Fechar arquivo ◆ Escrever no arquivo ◆ Configurar posição corrente ◆ Fechar o arquivo ◆ Obter o tamanho do arquivo ◆ Obter o nome do arquivo

Tabela 3 – Exemplos de tipos abstratos de dados (TAD).

Em linguagens orientadas a objetos, como a linguagem Java, um TAD é representado através de uma classe. O conceito de TAD é fundamental para o bom aprendizado de programação orientada a objetos. Desenvolvedores sem um bom entendimento deste conceito tendem a criar classes que não passam de uma coleção de dados e rotinas fracamente relacionadas, ou, ainda pior, programas estruturados “mascarados” dentro de classes.

Conceitos fundamentais

A seguir serão apresentados alguns conceitos fundamentais da POO. Alguns destes conceitos serão acompanhados de exemplos de código. Inicialmente, serão apresentados alguns conceitos já vistos na atividade Programação I em um nível de revisão. Em seguida serão tratados com mais detalhes conceitos ainda não vistos. Estes conceitos encontram-se presentes em diferentes linguagens de programação além da linguagem Java.

Classe

Uma classe define as características abstratas de um objeto. Estas características podem ser atributos (campos ou propriedades) como também podem ser métodos que definem o “comportamento” do objeto. Uma classe ainda pode ser vista como uma

fábrica a partir de onde os objetos são criados. Exemplos de classes: pessoa, automóvel etc. Como dito anteriormente, em linguagens orientadas a objetos classes são utilizadas para criar tipos abstratos de dados

Abaixo é mostrado o código fonte da classe Veiculo. Esta classe possui apenas um atributo que representa a velocidade. Como este atributo é protegido (modificador de acesso protected), a classe disponibiliza um método que modifica e também um método que retorna o valor deste atributo. A classe Veiculo também apresenta o método to String que é herdado da classe Object.

```
1  public class Veiculo{  
2      protected int velocidade;  
3  
4      public void acelera(int incremento){  
5          velocidade = velocidade + incremento;  
6      }  
7  
8      public void freia(int decremento) {  
9          velocidade = velocidade - decremento;  
10     }  
11  
12     public String toString(){  
13         return new String("Velocidade: " + velocidade);  
14     }  
15 }
```

Exemplo 6 – Classe Veiculo.

Objeto

Um objeto é uma instância de uma classe. Um objeto é capaz de armazenar estados através de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos. Exemplos de objetos: automóvel, motocicleta, João, Maria etc.

Classe x Objeto

Um conceito-chave relativo à programação orientada a objetos é a diferenciação entre objetos e classes. Um objeto é qualquer entidade específica que existe em tempo de execução (programa rodando). Uma classe é uma entidade estática que pode ser visualizada através do código fonte do programa. Por exemplo, você pode declarar

uma classe Pessoa que possui os atributos nome e idade. Em tempo de execução podem existir diferentes objetos do tipo Pessoa, como, por exemplo, João, Maria e assim por diante.

Método

Definem as habilidades dos objetos. Um método em uma classe é apenas uma definição. A ação só ocorre quando o método é chamado durante a execução do programa. Normalmente uma classe possui diversos métodos. A classe Veiculo vista no exemplo anterior possui os métodos acelera e freia que modificam o valor do atributo velocidade.

Herança

É o mecanismo pelo qual uma classe (subclasse) pode estender outra classe (superclasse), aproveitando seus métodos e atributos. Uma subclasse, normalmente, além de reutilizar os métodos e atributos de sua superclasse, também define seus próprios métodos e atributos. Esta relação é normalmente chamada de *é um*. Um exemplo de herança: Mamífero é superclasse de Humano, logo um Humano é *um* Mamífero.

Na linguagem Java a herança ocorre na definição de uma classe através da palavra reservada `extends`. Por exemplo, se quisermos criar a classe Onibus como sendo derivada da classe Veiculo descrita acima, teremos o código mostrado no exemplo 7. Neste exemplo a classe Onibus herda todas as propriedades da classe Veiculo (atributos e métodos) e também adiciona dois novos atributos: `qtdPassageiros` que representa a quantidade de passageiros no ônibus e também a constante `capacidade` que denota a capacidade máxima de passageiros no ônibus. Além destes atributos, a classe Onibus também adiciona três novos métodos: `embarca`, `desembarca` e `toString`. O método `embarca` incrementa o número de passageiros no ônibus, verificando se este não excede a capacidade máxima. O método `desembarca` decrementa o número de passageiros no ônibus, verificando se este não fica negativo. Ambos os métodos (`embarca` e `desembarca`) retornam um valor do tipo booleano que indica se a operação foi realizada com sucesso ou não.

```
1  public class Onibus extends Veiculo {
2
3      protected int qtdPassageiros;
4      protected final int capacidade = 50;
5
6      public boolean embarca(int n){
7          if ((qtdPassageiros + n) <= capacidade){
8              qtdPassageiros = qtdPassageiros + n;
9              return true;
10         }
11         else{
12             return false;
13         }
14     }
15
16     public boolean desembarca(int n){
17         if ((qtdPassageiros - n) >= 0){
18             qtdPassageiros = qtdPassageiros - n;
19             return true;
20         }
21         else{
22             return false;
23         }
24     }
25
26     public String toString(){
27         return (super.toString() + " Passageiros: " + qtdPassageiros);
28     }
29 }
```

Exemplo 7 – Código fonte da classe Onibus que é derivada da classe Veiculo.

Em específico na linguagem Java, todas as classes são derivadas da classe Object, mesmo que isso não seja explicitado através da palavra reservada extends. Neste sentido, podemos dizer que qualquer classe na linguagem Java (seja ela uma classe da biblioteca ou uma classe definida pelo usuário) é um Object.

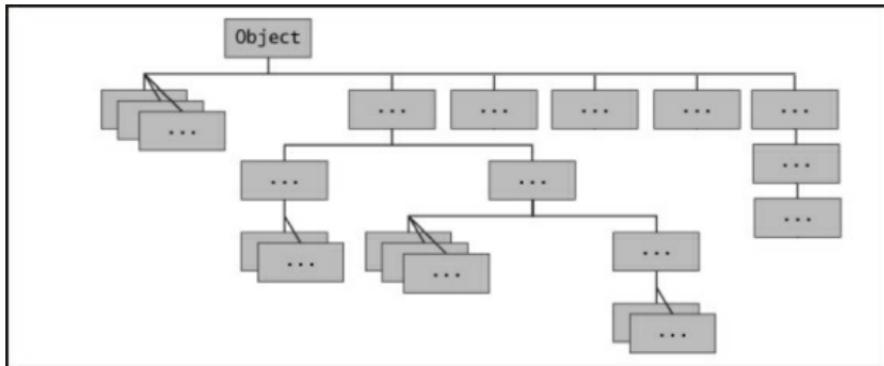


Figura 1 – Biblioteca de classes da linguagem Java.

Abstração

Em um contexto geral, a abstração pode ser vista como a habilidade de se concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em termos de orientação a objetos, uma classe é uma abstração de alguma entidade existente no domínio da aplicação sendo desenvolvida. Este é um conceito que não foi introduzido exclusivamente com a orientação a objetos, embora muito aplicado por quem a utiliza.

Como um exemplo de abstração, imagine sua casa. Quando você se refere à sua casa você está fazendo uma abstração, visto que você não está pensando na casa em termos de tijolos, madeira, vigas, aberturas etc. Embora você saiba que estes elementos estão presentes, eles não são importantes para quem é apenas um “usuário” ou “morador” da casa.

Abstração



Figura 2 – Abstração.

Encapsulamento

O conceito de encapsulamento consiste na separação dos aspectos internos e externos de um objeto. Este mecanismo é utilizado para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando para quem utiliza a classe apenas os métodos que alteram estes atributos. Exemplo: você não precisa conhecer os detalhes internos do motor de um automóvel para utilizá-lo. Estes são encapsulados e você interage apenas com uma interface mais amigável: chave de ignição, alavanca de marchas, pedais de acelerador e freio etc.

Outra característica do encapsulamento diz respeito à ocultação da informação. Como exemplo imagine um iceberg¹, onde apenas um décimo dele fica exposto na superfície da água, enquanto o restante (nove décimos) fica escondido, ou melhor, encapsulado abaixo da superfície.

Encapsulamento

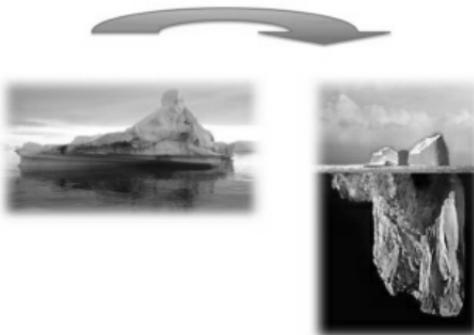


Figura 3 – Encapsulamento.

Polimorfismo

Princípio pelo qual duas classes derivadas de uma mesma superclasse podem invocar métodos que possuem a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada. Isso é obtido através da utilização de uma referência a um objeto do tipo da superclasse. A decisão sobre qual método deve ser executado é tomada em tempo de execução através de um mecanismo chamado ligação dinâmica.

Para haver polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação (assinatura), sendo utilizado para isso o mecanismo de redefinição (sobreposição) de métodos. Este mecanismo permite que um método definido em uma superclasse possa ser redefinido em uma subclasse.

Um erro comum de quem está aprendendo a linguagem Java é confundir os conceitos de sobrecarga e sobreposição. Para haver sobreposição de métodos, obrigatoriamente deve haver herança, pois os métodos devem possuir exatamente a mesma assinatura (nome e parâmetros). Sobrecarga, por outro lado, acontece apenas quando os métodos possuem o mesmo nome mas assinaturas diferentes. Neste caso não há herança, pois os métodos estão na mesma classe.

Classes abstratas

Uma classe abstrata é uma classe que não pode ser instanciada. Seu propósito é apenas servir de superclasse para as subclasses que implementarão seus métodos abstratos. Para uma classe ser abstrata, ela deve possuir pelo menos um método abstrato (embora isso não seja obrigatório). Um erro comum entre quem está aprendendo a linguagem Java é assumir que uma classe abstrata deve possuir apenas métodos abstratos. Esta afirmação deve ser vista da seguinte maneira: se a classe possuir um método abstrato, esta deve ser declarada como abstrata. Isto significa que, se uma classe possuir cinco métodos e se apenas um deles for um método abstrato, então a classe é considerada abstrata, não podendo ser instanciada.

Um método abstrato é um método que não possui implementação. Ele deve ser implementado em alguma subclasse da classe abstrata. Para ser considerado abstrato, um método deve conter a palavra reservada `abstract` logo após o modificador de acesso na sua assinatura.

Para ilustrar a utilização de classes abstratas, suponha que você tenha sido encarregado de projetar e implementar um sistema para manipulação de polígonos. Este sistema deve lidar com diferentes tipos de polígonos, tais como retângulos, quadrados, elipses e círculos. Todos os polígonos possuem propriedades em comum, mas também possuem comportamentos distintos para algumas operações, como, por exemplo, cálculo de área e cálculo de circunferência.

Este sistema pode ser projetado de maneira a ter uma classe abstrata que representa um polígono. Esta classe irá conter atributos e métodos comuns a todos os polígonos e também métodos abstratos que deverão ser implementados nas subclasses. Estes métodos irão implementar o comportamento específico para cada tipo de polígono.

```
1  public abstract class Poligono {  
2  
3      protected String nome;  
4      protected String cor;  
5  
6      public abstract void desenha();  
7      public abstract void move();  
8  
9      public String getName() {  
10          return nome;  
11      }  
12      public void setName(String nome) {  
13          this.nome = nome;  
14      }  
15      public String getCor() {  
16          return cor;  
17      }  
18      public void setCor(String cor) {  
19          this.cor = cor;  
20      }  
21  }
```

Exemplo 8 – Classe abstrata Poligono.

Neste exemplo todos os polígonos possuem um atributo que define o seu nome e um atributo que define sua cor. Além destes atributos, a classe Polígono apresenta também dois métodos abstratos: desenha e move. Estes métodos devem ser implementados nas subclasses, pois definirão comportamentos específicos para cada tipo de polígono.

A figura abaixo mostra um possível diagrama de classes com classes concretas como subclasses da classe Polígono. Neste diagrama existem três classes criadas como subclasses da classe Polígono: Círculo, Retângulo e Elipse. Todas estas classes herdam os atributos nome e cor da classe Polígono e ainda implementam suas próprias versões dos métodos desenha e move.

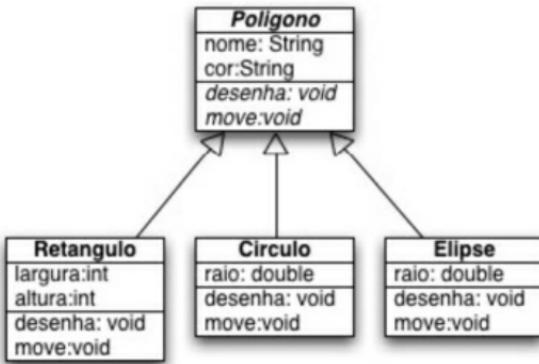


Figura 4 – Diagrama de classes do sistema de manipulação de polígonos.

Utilizando esta hierarquia de classes um vetor pode ser utilizado para armazenar diferentes tipos de polígonos.

```

1  public static void main(String args[]) {
2      Poligono poligonos[] = new Poligono[3];
3
4      poligonos[0] = new Retangulo();
5      poligonos[1] = new Circulo();
6      Poligonos[2] = new Elipse();
7
8      ...
9      for (int i = 0; i < poligonos.length; i++) {
10          poligonos[i].desenha();
11      }
12  }

```

Exemplo 9 – Utilização do sistema de manipulação de polígonos.

Neste exemplo, inicialmente um vetor que armazena objetos do tipo Polígono é instanciado com capacidade para três objetos. Cada uma das posições deste vetor recebe um tipo diferente de polígono (linhas 4 a 6). O código onde os atributos de cada polígono são configurados foi omitido por razões de simplicidade.

Nas linhas 9 a 11, uma iteração é realizada pelo vetor de polígonos e, para cada objeto armazenado no vetor, o método desenha é chamado. Na primeira iteração será chamado o método desenha da classe Retângulo. Na segunda iteração será chamado o método desenha da classe Círculo, e na terceira iteração será chamado o método

desenha da classe Elipse.

Interfaces

Como já visto anteriormente, objetos definem sua interação com o mundo externo através de seu comportamento, que, por sua vez, é definido pelos métodos que o objeto expõe. Estes métodos definem a *interface* do objeto com o mundo externo. Se tomarmos como exemplo um rádio a pilhas, sua *interface* pode ser definida pelo botão de ligar e desligar, pelo botão de sintonizar estações e pelo botão de volume. Podemos dizer que estes botões definem a *interface* do rádio para com o mundo externo.

Na linguagem Java, uma *interface* pode ser vista como uma classe abstrata que não contém implementação alguma, ou seja, todos os métodos são abstratos. Ainda, todos os atributos de uma *interface* são considerados constantes.

Na linguagem Java *interfaces* não podem ser instanciadas. Elas são somente implementadas por classes ou ainda estendidas por outras *interfaces*. Uma vez que uma classe implementa uma *interface*, esta deve prover uma implementação para todos os métodos definidos na *interface*. Neste sentido, uma *interface* pode ser vista como um contrato. Uma classe, ao implementar uma *interface*, está se comprometendo a implementar todos os métodos definidos na *interface*.

Interfaces têm outro papel bastante importante na linguagem Java. Como esta não suporta herança múltipla, ou seja, uma classe pode ser subclasse de apenas uma classe, *interfaces* são uma alternativa para esta característica. Uma classe pode implementar múltiplas *interfaces*, mesmo já sendo subclasse de outra classe.

Outro ponto importante a ser mencionado a respeito de *interfaces* é que estas podem ser utilizadas como tipos de referência, assim como classes, em qualquer programa Java. Como uma classe pode ser subclasse de apenas uma classe, mas pode implementar múltiplas *interfaces*, objetos instanciados a partir de uma classe que implementa uma ou mais *interfaces* possuem, além do tipo de sua própria classe, também o tipo de todas as *interfaces* que sua classe implementa.

Definindo uma interface

A definição de uma *interface* é semelhante a definição de uma classe, consistindo do modificador de acesso (*public*, *protected* ou *private*), a palavra reservada *interface*,

o nome da *interface* e, caso a *interface* possua *interfaces* pai, a palavra reservada *extends* seguida da lista de *interfaces* pai separadas por vírgula.

```
1 public interface ItemDeVenda {  
2  
3     double valorDoItem();  
4     double valorDoImposto();  
5  
6 }
```

Exemplo 10 – Definição da *interface* ItemDeVenda.

O corpo da *interface* deve possuir apenas declarações de métodos, sem implementação. Além de métodos uma *interface* pode definir também constantes. Neste sentido todos os métodos definidos em uma *interface* são automaticamente abstratos e todos os atributos definidos em uma *interface* são automaticamente constantes (*public*, *static* e *final*) mesmo que estes modificadores sejam omitidos.

A *interface* definida no exemplo 10 representa um item de venda. Esta *interface* define apenas dois métodos: *valorDoItem()*, que retorna um tipo *double*, e *valorDoImposto()* que retorna também um tipo *double*. Esta *interface* pode ser utilizada em um sistema de vendas *online* por exemplo. Como um item de venda pode representar produtos de diferentes categorias (eletrônicos, livros, produtos importados etc), tanto o cálculo do preço quanto o cálculo do imposto tendem a ser diferentes para as diferentes categorias. Desta maneira, tanto o cálculo do valor do item quanto o cálculo do valor do imposto devem ser implementados pelas classes que implementarem esta *interface*.

Implementando uma interface

Ainda utilizando o exemplo do sistema de vendas *online*, qualquer classe que implemente a *interface* ItemDeVenda deve, obrigatoriamente, implementar todos os métodos definidos por ela.

```
1  public class Eletronico implements ItemDeVenda {
2
3      protected double valor;
4      protected String tipo;
5
6      Public ProdutoEletronico(double valor, String tipo) {
7          this.valor = valor;
8          this.tipo = tipo;
9      }
10
11     Public double valorDoImposto() {
12         return valor * 0.10;
13     }
14
15     public double valorDoItem() {
16         return valor + valorDoImposto();
17     }
18 }
```

Exemplo 11 – Classe ProdutoEletronico que implementa a *interface* ItemDeVenda.

A classe Eletronico mostrada no exemplo 11 implementa a *interface* ItemDeVenda através da implementação dos métodos definidos por ela: valorDoImposto e valorDoItem. Além destes métodos a classe Eletronico ainda possui dois atributos: valor e tipo. O valor é o preço do produto sem imposto e o tipo é uma String que define o tipo de produto eletrônico sendo vendido.

A classe Livro mostrada no exemplo 12 também implementa a *interface* ItemDeVenda através da implementação de ambos os métodos definidos por ela. Além destes métodos a classe Livro ainda contém os atributos título, autor, ISBN e valor. O atributo título armazena o título do livro. O atributo autor armazena o autor do livro. O atributo ISBN armazena o ISBN do livro e o atributo valor armazena o valor de venda do livro.

```
1  public class Livro implements ItemDeVenda {
2
3      protected String titulo;
4      protected String autor;
5      protected String ISBN;
6
7      protected double valor;
8
9      public Livro(String titulo, String autor, String ISBN, double valor) {
10         this.titulo = titulo;
11         this.autor = autor;
12         ISBN = ISBN;
13         this.valor = valor;
14     }
15
16     public double valorDoImposto() {
17         return valor * 0.05;
18     }
19
20     public double valorDoItem() {
21         return (valor * 0.9) + valorDoImposto();
22     }
23
24 }
```

Exemplo 12 – Classe Livro que implementa a *interface* ItemDeVenda.

Através da utilização da *interface* ItemDeVenda, ambos os produtos comercializados pelo sistema possuem cálculos diferentes para o preço final e também para o imposto. Na classe Livro, alem do cálculo do imposto ser diferente, o sistema aplica um desconto de 10% no valor de venda do livro (linha 21).

Utilizando interfaces

Quando uma *interface* é definida, esta representa também um novo tipo. Ela pode ser utilizada em qualquer ponto do programa onde o tipo de uma classe seria utilizado. Quando uma variável de referência do tipo de uma *interface* é definida, qualquer objeto atribuído a esta referência deve ser de uma instância de uma classe que implemente esta *interface*.

No exemplo do sistema de vendas *online*, uma vez que se tenha definido uma referência do tipo ItemDeVenda, esta pode referenciar tanto objetos do tipo Elettronico quanto objetos do tipo Livro. Na verdade esta referência pode referenciar qualquer objeto que venha a implementar a *interface* ItemDeVenda.

A fim de demonstrar a utilização da *interface* ItemDeVenda no sistema de vendas

online, foi modelada a classe CestaDeCompras. Esta classe tem o objetivo de simular uma cesta de compras onde o usuário do sistema pode adicionar itens comprados. Visto que todos os itens da loja (neste caso apenas livros e produtos eletrônicos) implementam a interface ItemDeVenda, o usuário pode adicionar em sua cesta qualquer tipo de produto.

A classe CestaDeCompras contém três atributos: uma constante que define o número máximo de itens que ela armazena (linha 3), um vetor que armazena objetos que implementam a interface ItemDeVenda (linha 4) e um atributo do tipo *int* que armazena a quantidade de itens armazenados (linha 5). O construtor da classe (linhas 7 a 10) instancia o vetor *items* com o tamanho máximo definido pela constante *MAXITEMS* e inicializa o atributo *qtdItens* com zero.

```
1  public class CestaDeCompras {
2
3      public static final int MAXITENS = 10;
4      protected ItemDeVenda items[];
5      protected int qtdItens;
6
7      public CestaDeCompras() {
8          this.items = new ItemDeVenda[MAXITENS];
9          qtdItens = 0;
10     }
11
12     public void adicionaItem(ItemDeVenda item) {
13         if (qtdItens < MAXITENS){
14             items[qtdItens] = item;
15             qtdItens = qtdItens + 1;
16         }
17     }
18
19     public double calculaTotalDeImpostos(){
20         double totalImpostos = 0.0;
21         for (int i = 0; i < qtdItens; i++){
22             totalImpostos = totalImpostos + items[i].valorDoImposto();
23         }
24         return totalImpostos;
25     }
26
27     public double calculaTotalDaVenda(){
28         double totalVenda = 0.0;
29         for (int i = 0; i < qtdItens; i++){
30             totalVenda = totalVenda + items[i].valorDoItem();
31         }
32         return totalVenda;
33     }
34
35     public static void main(String[] args) {
36
37         Livro livro1 = new Livro("Java Como Programar", "Deitel",
38                             "123456789", 200.00);
39         Livro livro2 = new Livro("Estruturas de Dados e Algoritmos em Java",
40                             "Robert Lafore", "98765432", 150.00);
41         Eletronico eletr001 = new Eletronico(200.00 , "DVD Player");
42
43         CestaDeCompras cesta = new CestaDeCompras();
44         cesta.adicionaItem(livro1);
45         cesta.adicionaItem(livro2);
46         cesta.adicionaItem(eletr001);
47
48         System.out.println("Total de impostos da Venda: " +
49                           cesta.calculaTotalDeImpostos());
50         System.out.println("Valor total da venda (incluindo impostos): " +
51                           cesta.calculaTotalDaVenda());
52     }
53 }
```

Exemplo 13 – Implementação da classe CestaDeCompras.

O método adicionaItem (linhas 12 a 17) recebe como argumento um objeto que implementa a interface ItemDeVenda e o adiciona ao vetor de itens. Antes de adicionar o método realiza um teste para verificar se a quantidade de itens armazenadas no vetor é menor que a quantidade máxima (definida pela constante *MAXITEMS*). Caso negativo, o item é inserido no vetor de itens e o contador de itens armazenados é incrementado de acordo. Este método retorna *true* caso a inserção tenha ocorrido e *false* caso contrário.

O método calculaTotalDeImpostos (linhas 19 a 25) retorna um valor do tipo *double* que representa o total de impostos de todos os itens contidos no vetor de itens. Este método declara uma variável local do tipo *double* que é utilizada para acumular os impostos de todos os itens. A cada iteração do laço for (linhas 21 a 23) o método valorDoImposto() é chamado. Como o vetor de itens armazena qualquer objeto que implementa a interface ItemDeVenda, será em tempo de execução que o método correto será chamado, dependendo do tipo de objeto armazenado (Livro ou Eletronico). Ao final da iteração o método retorna o total de impostos acumulado na variável *totalImpostos*.

O método calculaTotalDaVenda funciona de maneira similar ao método calculaTotalDeImpostos. Uma variável local (*totalVenda*) é utilizada para acumular o valor total de cada item. A cada iteração do laço for (linhas 29 a 32) o método valorDoItem é chamado. Como no método calculaTotalDeImpostos, dependendo do objeto armazenado, o método correspondente será chamado. Ao final da execução o valor total estará armazenado na variável *totalVenda*, que é retornada pelo método.

O método *main* (linhas 35 a 52) instancia dois objetos do tipo Livro (linhas 37 a 40), um objeto do tipo Eletronico (linha 41) e um objeto do tipo CestaDeCompras (linha 42). Os três objetos que implementam a interface ItemDeVenda são então adicionados à cesta de compras (linhas 44 a 46). Em seguida (linhas 48 e 50), são chamados os métodos calculaTotalDeImpostos e calculaTotalDaVenda da classe CestaDeCompras. O retorno destes métodos é exibido na tela.

Embora este seja um exemplo bastante simples de um sistema de vendas *online*, ele demonstra como através da utilização de *interfaces* a implementação de detalhes específicos de cada item é facilitada. Caso mais um tipo de produto necessite ser vendido pelo sistema, basta que a classe que representa este produto implemente a interface ItemDeVenda. Neste caso nenhuma linha de código necessitará ser modificada na classe CestaDeCompras, pois ela está apta a lidar com qualquer objeto que implemente esta interface.

Neste capítulo foram revisados alguns conceitos de orientação a objetos já vistos

na disciplina de Programação I juntamente com o conceito de *interface*. Para demonstrar este exemplo foi utilizado um exemplo bastante simples de um sistema de vendas capaz de manipular diferentes tipos de produtos.

Uma *interface* pode ser vista como um protocolo de comunicação entre dois objetos. Uma vez que um objeto implementa uma *interface*, é garantido que este estará apto a responder as mensagens (métodos) implementadas por esta *interface*. Outro ponto a ser ressaltado é que uma *interface* pode ser utilizada como um tipo em qualquer ponto do código.

Os conceitos tratados neste capítulo são importantes, pois um bom entendimento dos mesmos é fundamental para o restante do livro. As estruturas de dados tratadas nos próximos capítulos serão implementadas através de classes, onde os conceitos vistos até estão estarão presentes.

1 <http://pt.wikipedia.org/wiki/Iceberg>.

CAPÍTULO 3

LISTAS LINEARES COM ALOCAÇÃO SEQUENCIAL

Neste capítulo é introduzido o conceito de estrutura de dados linear através do estudo da estrutura de dados lista linear com alocação sequencial de memória. Esta estrutura de dados é uma das mais simples e versáteis. Ela pode ser vista como um tipo abstrato de dados que representa uma série de itens. Em geral, uma lista linear permite a inserção e remoção de itens e também permite que todos os itens sejam visitados.

A lista linear permite representar um conjunto de dados de forma a preservar a ordem existente entre eles. Por definição, uma lista linear é um conjunto de n nós, onde $n \geq 0$ representada da seguinte maneira: $L[0], L[1], L[2], \dots, L[n]$, onde:

- Se $n > 0$, $L[0]$ é o primeiro elemento da lista
- Se $n > 0$, $L[n-1]$ é o último elemento da lista
- Para $1 < k \leq n$, o nó $L[k]$ é precedido pelo nó $L[k-1]$ e seguido por $L[k+1]$
- Se $n = 0$, então a lista está vazia

Em uma lista linear, normalmente a estrutura interna de cada nó é abstraída. Cada nó possui uma informação-chave, utilizada para identificar o nó na lista, e também informações satélite. A figura 5 representa uma lista linear em que a informação de cada item representa uma/um funcionário de uma empresa. Nesta lista a informação-chave é dada pelas letras de a até g e as informações satélite de cada nó são o CPF, nome, endereço e cargo de cada pessoa.

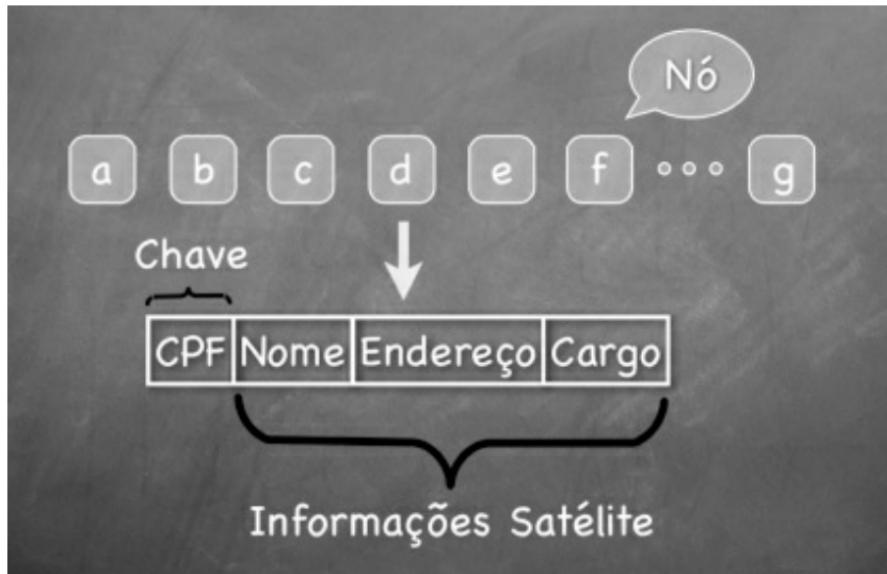


Figura 5 – Representação de uma lista linear geral.

Esta lista apresenta uma ordem específica, onde a pessoa *d* é precedida pela pessoa *c* e sucedida pela pessoa *e*.

Listas podem ser utilizadas em diferentes aplicações, como, por exemplo:

- ◆ Alunos de uma turma
- ◆ Cadastro de funcionários de uma empresa
- ◆ Pessoas esperando um ônibus
- ◆ Cartas de um baralho
- ◆ Dias da semana
- ◆ Vagões de um trem
- ◆ Capítulos de um livro

Listas lineares podem ser classificadas de diferentes maneiras. Uma delas diz respeito à ordem dos elementos da lista. Neste sentido uma lista pode ser ordenada, em que os elementos seguem uma ordem predefinida, como, por exemplo, ordem alfabética ou não ordenada, em que os elementos não seguem nenhuma ordem.

Listas lineares também podem ser classificadas de acordo com a política de

inserção e remoção de elementos da seguinte maneira:

- ▶ Listas lineares gerais
 - ▶ Não possuem restrição para inserção e remoção de elementos. Os elementos podem ser inseridos e removidos em qualquer posição da lista.
- ▶ Listas lineares particulares
 - ▶ Possuem restrição para inserção e remoção de elementos. Neste tipo de lista os elementos podem ser inseridos ou removidos apenas em posições específicas da lista. Neste livro serão vistos dois tipos de listas lineares particulares:
 - ◆ Pilhas: inserções e remoções são realizadas em apenas uma das extremidades da lista.
 - ◆ Filas: inserções são realizadas em uma extremidade da lista e remoções na outra extremidade da lista.

Listas lineares ainda podem ser classificadas quanto ao tipo de armazenamento:

- ▶ Listas lineares com alocação estática
 - ▶ Também chamada de lista sequencial. Neste tipo de lista os elementos são armazenados em posições contíguas de memória. Geralmente, este tipo de lista é representada por vetores.
- ▶ Listas lineares com alocação dinâmica
 - ▶ Também chamada de lista encadeada. Neste tipo de lista a memória é alocada sob demanda na medida em que novos elementos são inseridos. Os nós não ficam armazenados em posições contíguas, mas sim em posições aleatórias da memória.

Uma lista linear geral, quando vista como um tipo abstrato de dado, pode expor, entre outras, as seguintes operações:

- ▶ Verificação da condição de lista cheia
- ▶ Verificação da condição de lista vazia

- ▶ Inserção de um nó na i-ésima posição da lista
- ▶ Inserção de um nó no final da lista
- ▶ Exclusão do i-ésimo nó da lista
- ▶ Exibição do conteúdo da lista
- ▶ Localização de um determinado nó na lista

Nas próximas seções serão abordadas questões específicas de implementação referentes a listas lineares. Inicialmente, será estudada a lista linear com alocação estática, também chamada de lista sequencial.

Listas lineares gerais

Uma lista linear geral com alocação sequencial, também chamada de lista sequencial, é uma implementação de uma lista linear geral em que os elementos ficam armazenados em posições contíguas de memória. Este tipo de lista é normalmente implementado através da utilização de vetores, pois estes também armazenam seus elementos de maneira contígua na memória.

Para demonstrar o funcionamento de uma lista sequencial, suponha uma lista L de números inteiros com os seguintes elementos: 5, -4, 8, 0, 3, 2. Utilizando um vetor de inteiros de nome L e de tamanho dez, a lista será disposta nas primeiras posições do vetor da seguinte maneira:

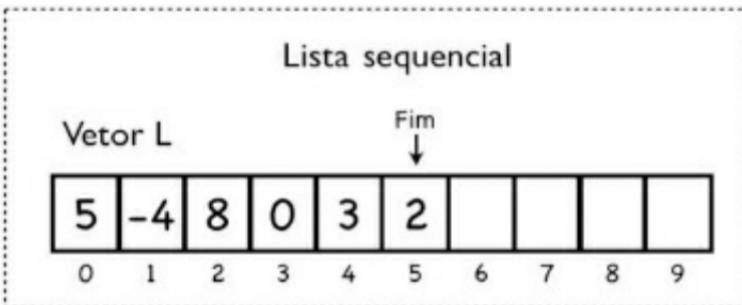


Figura 6 – Representação de uma lista sequencial.

Utilizando esta abordagem, o primeiro elemento da lista sempre estará na posição

zero do vetor que armazena os elementos. Uma variável inteira (*Fim*) é utilizada para controlar o final da lista (posição do último elemento da lista).

Ao inserir o elemento 3 na lista, esta ficará com a seguinte configuração:

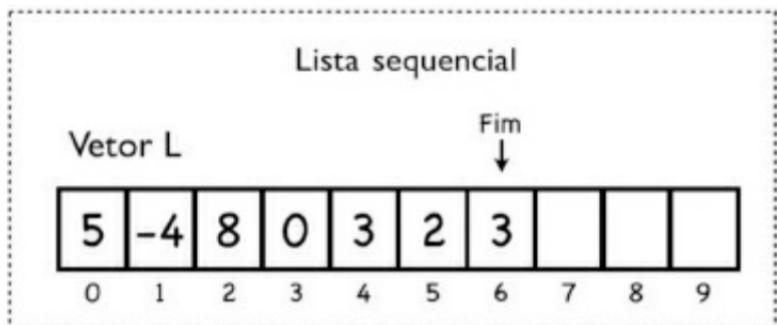


Figura 7 – Lista sequencial após a inserção do elemento 3.

O elemento 3 foi inserido na última posição da lista e a variável *Fim* foi atualizada de acordo. Neste momento o último elemento da lista se encontra na posição 6 do vetor L.

Ao retirar o elemento 8 da lista, esta ficará com a seguinte configuração:

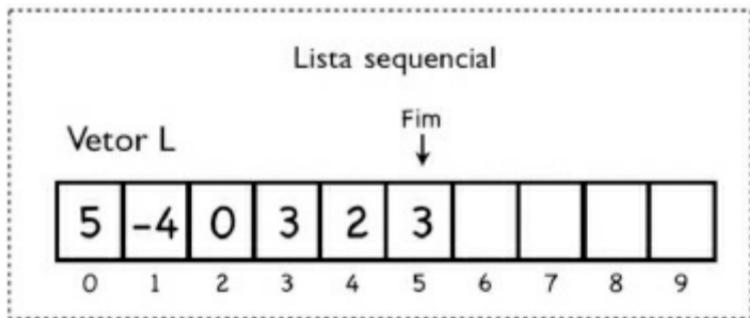


Figura 8 – Lista sequencial após a remoção do elemento 8.

Ao ser retirado o elemento 8, todos os elementos que estavam à sua direita no vetor foram deslocados uma posição para a esquerda para que os elementos fiquem dispostos sempre na extremidade esquerda do vetor.

Se o primeiro elemento da lista for retirado (elemento 5), está ficará com a seguinte configuração:

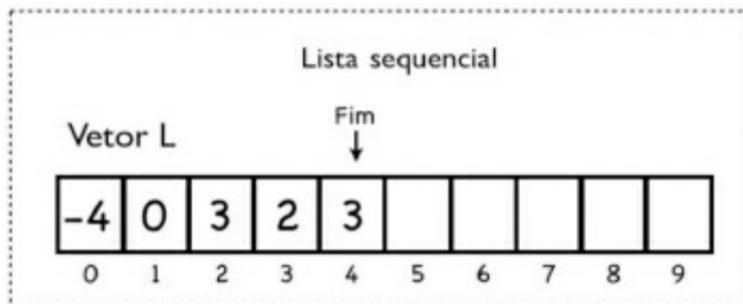


Figura 9 – Lista sequencial após a remoção do primeiro elemento.

Após esta operação todos os elementos da lista foram deslocados uma posição para a esquerda e a variável *Fim* foi atualizada de acordo.

A lista sequencial, além de permitir a remoção de elementos em qualquer posição da lista, permite também a inserção de elementos em qualquer posição da lista. Desta maneira, podemos inserir o elemento 5 na primeira posição da lista, deixando a lista com a seguinte configuração:

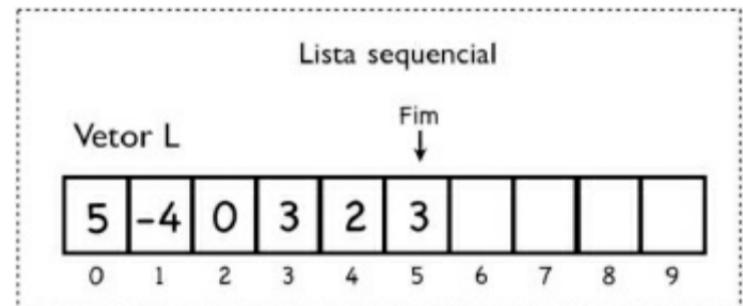


Figura 10 – Lista sequencial após a inserção do elemento 5 na primeira posição.

Ao realizar esta inserção todos os elementos da lista são deslocados uma posição para a direita para que o elemento 5 seja inserido na primeira posição.

Do ponto de vista de implementação, uma classe que representa uma lista sequencial deve possuir os métodos apresentados na tabela 4.

Método	Descrição
public boolean	

<code>isEmpty()</code>	Verifica se a lista está vazia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
<code>public boolean isFull()</code>	Verifica se a lista está cheia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
<code>public int getSize()</code>	Retorna o número de elementos armazenados na lista.
<code>public Object get(int index)</code>	Retorna uma referência ao objeto armazenado na posição.
<code>public boolean add(Object o)</code>	Insere o objeto referenciado por <i>o</i> na lista. Retorna <i>true</i> caso a inserção tenha ocorrido com sucesso, ou <i>false</i> caso contrário.
<code>public boolean remove(int pos)</code>	Remove o elemento da posição indicada pelo parâmetro <i>pos</i> . Retorna <i>true</i> caso a remoção tenha ocorrido com sucesso, ou <i>false</i> caso contrário.
<code>public void print()</code>	Exibe na tela todos os elementos contidos na lista.

Tabela 4 – Métodos da classe que implementa uma lista sequencial.

A implementação desta classe assume que a lista armazena objetos do tipo *Object*. Utilizando esta abordagem, esta lista pode armazenar objetos de qualquer classe que seja subclasse da classe *Object*, ou seja, qualquer classe da linguagem Java.

O código fonte da classe de lista sequencial é mostrado no exemplo abaixo.

A classe *SequentialList* mostrada no exemplo 14 define dois atributos: um vetor de objetos chamado *list*, utilizado para armazenar os elementos da lista (linha 3), e um inteiro chamado *last*, que é utilizado para armazenar a última posição da lista (linha 4). Ambos os atributos são protegidos, desta maneira a modificação deles ocorre somente através dos métodos públicos. Além destes dois atributos, ela também define os métodos mostrados na tabela 4.

O construtor da classe *SequentialList* (linhas 6 a 8) recebe como parâmetro um valor inteiro que é utilizado para instanciar o vetor de objetos. Este parâmetro é quem define o número máximo de elementos da lista.

O método *get* (linhas 11 a 14) recebe como parâmetro um índice e retorna o objeto armazenado neste índice do vetor, caso este exista, ou *null* caso contrário. Este método faz uma validação no índice recebido como parâmetro para verificar se este é um índice válido. Caso o índice seja inválido, menor que zero ou maior que *last*, o método retorna *null*.

O método *add* (linhas 17 a 22) insere um elemento no final da lista. Caso a lista não esteja cheia, o elemento é inserido e o método retorna *true*, indicando que a inserção ocorreu com sucesso. Caso a lista já esteja cheia, o método retorna *false*. Se a lista fosse ordenada, esta implementação teria que ser diferente, pois a inserção não seria sempre no final da lista. O método teria que, antes de inserir o elemento,

encontrar a posição correta do mesmo na lista.

```
1  public class SequentialList {
2
3      protected Object list[];
4      protected int last=-1;
5
6      public SequentialList (int size){
7          list = new Object[size];
8      }
9
10
11     public Object get (int index){
12         if (index<0 || index>last) return null;
13         else return list[index];
14     }
15
16
17     public boolean add (Object o) {
18         if (isFull()) return false;
19         last++;
20         list[last]=o;
21         return true;
22     }
23
24
25
26     public Boolean add (Object o, int pos) {
27         if (isFull()) return false;
28         for (int i=last+1; i>pos; i--) {
29             list[i]=list[i-1];
30         }
31         last++;
32         list[pos]=o;
33         return true;
34     }
35
36
37     public Object remove (int index) {
38         if (isEmpty()) return null;
39         else if (index<0 || index>last) {
40             System.out.println ("índice não existente!");
41             return null;
42         }
43         Object o = list[index];
44         int numberofElements = last - index ;
45         if (numberofElements > 0) {
46             System.arraycopy(list, index + 1, list, index, numberofElements);
47         }
48         list[last] = null;
49         last--;
50         return o;
51     }
52
53
54     public boolean isEmpty (){
55         if (last== -1) return true;
56         else return false;
57     }
58
59
60     public boolean isFull (){
61         if (last==list.length-1) return true;
62         else return false;
63     }
64
65
66     public int getSize() {
67         return last+1;
68     }
69
70
71     public void print(){
72         for (int i=0; i<last; i++){
73             System.out.println(list[i]);
74         }
75     }
76 }
```

Exemplo 14 – Código fonte da classe SequentialList.

O método *add* definido nas linhas 26 a 34 insere um elemento na lista na posição especificada pelo parâmetro *pos*. Caso a lista já esteja cheia, este retorna *false*. Caso contrário, todos os elementos situados à direita do posição especificada pelo parâmetro *pos* são deslocados uma posição para a direita a fim de dar lugar na lista ao elemento a ser inserido. Esta movimentação de elementos é realizada pelo laço *for* das linhas 28 a 30. O restante do código do método realiza a inserção do elemento na posição específica e incrementa o atributo *last*.

O método *remove* (linhas 37 a 51) remove da lista o objeto armazenado na posição especificada pelo parâmetro *pos*. Este método inicialmente testa se a lista está vazia (linha 38). Caso a lista esteja vazia, este retorna *null*. Caso contrário, uma validação é realizada de modo a determinar se o parâmetro *pos* representa um índice válido (linha 39), ou seja, se este índice está dentro do intervalo da lista. Caso o índice seja inválido, a mensagem “Índice Inexistente” é mostrada e um valor *null* é retornado. Caso o índice seja válido, uma referência ao objeto removido é armazenada na variável local *o* e todos os elementos que se encontram à direita do elemento removido são deslocados uma posição para a esquerda. Para isso este método utiliza o método *arrayCopy* definido pela API da linguagem Java. Por fim, o método seta a última posição da lista para *null*, decrementa o atributo *last* e retorna a referência ao objeto removido.

O método *isEmpty* definido nas linhas 54 a 57, retorna *true*, caso a lista esteja vazia, ou *false* caso contrário. A lista estará vazia sempre que o atributo *last* for igual a -1. Caso contrário ela terá pelo menos um elemento.

O método *isFull* definido nas linhas 60 a 63, retorna *true* caso a lista esteja cheia, ou *false* caso contrário. A lista estará cheia sempre que o atributo *last* for igual ao comprimento do vetor menos um (*length - 1*), pois este é o último índice válido do vetor.

O método *getSize* definido nas linhas 66 a 68 retorna o número de elementos armazenados na lista.

O método *print* definido nas linhas 71 a 75 imprime na tela todos os elementos contidos na lista. O laço *for* deste método inicia sua iteração na posição zero do vetor interno da classe e termina no índice definido por *last*, que é o último elemento da lista.

O exemplo abaixo demonstra como a classe *SequentialList* pode ser utilizada em um programa Java.

```

1  public class SequentialListTest {
2
3      public static void main(String[] args) {
4          SequentialList lista = new SequentialList(10);
5          lista.add(new Integer(1));
6          lista.add(new Integer(2));
7          lista.add(new Integer(3));
8          lista.add(new Integer(4));
9          lista.add(new Integer(6), 2);
10         lista.remove(1);
11         lista.print();
12     }
13 }
```

Exemplo 15 – Utilização da classe *SequentialList*.

Neste exemplo a classe *SequentialListTest* é apenas uma classe de teste em que, no método *main*, um objeto do tipo *SequentialList* é criado (linha 4) com capacidade máxima de dez elementos. O código das linhas 5 a 8 insere quatro objetos do tipo *Integer* na lista. O código na linha nove insere um objeto do tipo *Integer* na posição dois da lista, deslocando todos os elementos à direita da posição dois uma posição a direita da lista.

Ao final da execução do código do exemplo 15 o conteúdo do objeto *lista* será o seguinte:

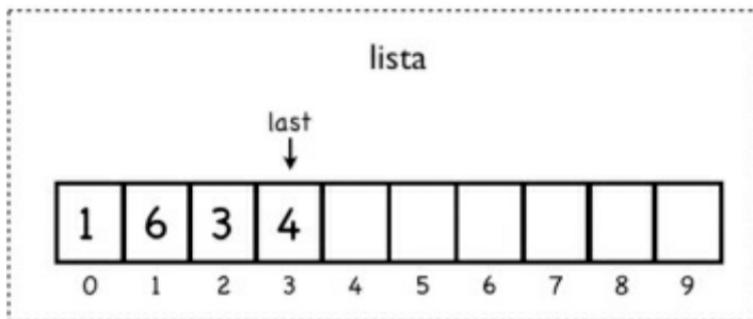


Figura 11 – Conteúdo da lista sequencial após a execução do código do exemplo 15.

Esta seção apresentou a classe *SequentialList*, que implementa uma lista linear geral com alocação estática. Esta classe utiliza internamente um vetor para armazenar seus elementos. Embora os elementos da lista sejam armazenados em um vetor, este não é acessado diretamente. O vetor interno é abstraido e seu conteúdo é modificado apenas pelos métodos disponibilizados pela classe *SequentialList*.

As próximas seções apresentarão listas lineares particulares também com alocação

estática. Estes são tipos específicos de listas que apresentam restrições quanto a inserção e remoção de elementos.

Pilha

Uma pilha pode ser vista como um tipo especial de lista onde os elementos são inseridos e removidos apenas em uma das extremidades. É dito que uma pilha implementa a política de inserção e remoção onde o último elemento inserido é sempre o primeiro a ser retirado. Esta política também é chamada de LIFO (do inglês *Last In First Out*).

O nome pilha é derivado da metáfora a uma pilha de pratos em uma cantina. Neste cenário as operações fundamentais consistem em inserir e retirar pratos da pilha. Cada vez que um prato é colocado na pilha (empilhado), este é colocado no topo. Cada vez que um prato é retirado da pilha, este é retirado sempre do topo.

A estrutura de dados pilha é utilizada em diferentes aplicações, como, por exemplo, o histórico de um navegador para internet. Cada vez que um endereço novo é acessado, este vai para o topo da pilha de endereços acessados. Cada vez que o botão “Voltar” é pressionado, o navegador retira da pilha o último endereço acessado e o endereço anterior se torna o endereço corrente.

Outra utilização desta estrutura é em editores de texto através da opção de reversão das operações realizadas (*undo* em inglês). Cada alteração feita no texto é mantida em uma pilha de modo a facilitar as reversões do documento a estados anteriores.

Embora a estrutura de dados pilha seja uma das mais simples, ela é utilizada em uma ampla gama de aplicações diferentes juntamente com estruturas de dados muito mais sofisticadas. O tipo abstrato de dados pilha define as seguintes operações:

- ◆ Inserção de um elemento na pilha
- ◆ Remoção de um elemento da pilha
- ◆ Obter o tamanho da pilha
- ◆ Verificar se a pilha está vazia
- ◆ Acessar o elemento no topo da pilha

Para demonstrar o funcionamento de uma pilha, suponha uma pilha P de números inteiros com os seguintes elementos: 3, 5 e 2. Utilizando um vetor de inteiros de nome

S e de tamanho 5, a pilha será disposta nas primeiras posições do vetor da seguinte maneira:

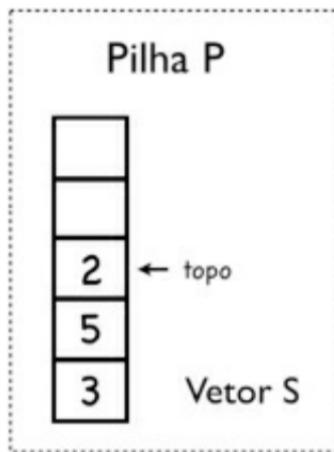


Figura 12 – Representação de uma pilha utilizando um vetor.

O primeiro elemento está na posição zero, o segundo na posição um e assim por diante. A variável inteira *topo* indica o topo da pilha.

Ao inserir o elemento 12 na pilha, esta ficará com a seguinte configuração:

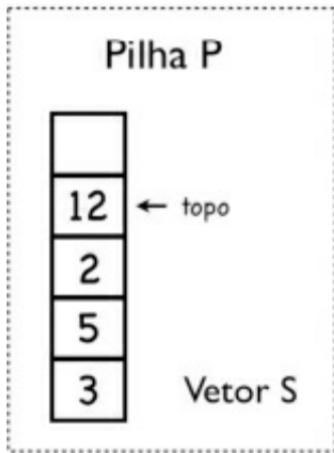


Figura 13 – Pilha após a inserção do elemento doze.

O elemento 12 torna-se o novo topo da pilha e a variável *topo* é atualizada de acordo. Ao inserir o elemento 7 na pilha, esta ficará com a seguinte configuração:

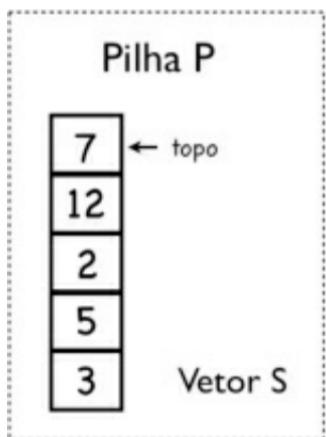


Figura 14 – Pilha após a inserção do elemento sete.

O elemento 7 torna-se o novo topo e a variável *topo* é atualizada de acordo. Neste momento a pilha encontra-se cheia, pois seu vetor interno, utilizado para armazenar seus elementos, não suporta mais a inserção de nenhum elemento. Ao retirarmos um elemento da pilha, esta ficará com a seguinte configuração:

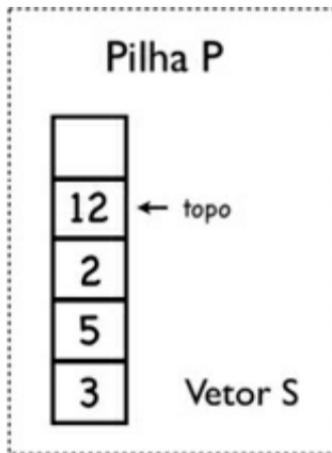


Figura 15 – Pilha após a remoção do elemento sete.

Toda inserção e remoção realizada em uma pilha sempre ocorre no topo. Neste caso o elemento retirado da pilha foi o elemento 12, que foi o último elemento a ser inserido.

Novamente, após a remoção de mais dois elementos da pilha, esta ficará com a seguinte configuração:

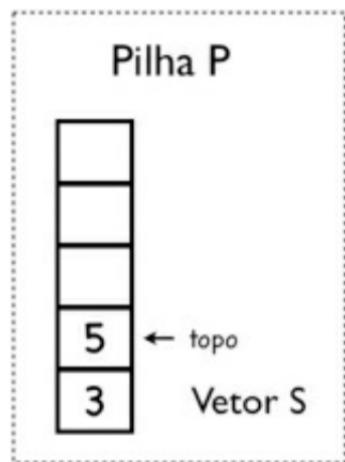


Figura 16 – Pilha após a remoção de dois elementos.

Após a remoção dos elementos 12 e 2, o elemento 5 passa a ser o novo topo.

Do ponto de vista de implementação, a classe que implementa uma pilha deve possuir os seguintes métodos:

Método	Descrição
public boolean isEmpty()	Verifica se a pilha está vazia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
public boolean isFull()	Verifica se a pilha está cheia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
public void push(Object o)	Insere um elemento no topo da pilha.
public Object pop()	Remove e retorna o objeto contido no topo da pilha.
public Object top()	Retorna (sem remover) uma referência ao objeto do topo da pilha.
public int size()	Retorna o tamanho da pilha

Tabela 5 – Métodos da classe que implementa uma pilha com alocação sequencial.

Assim como na classe de lista sequencial, esta implementação assume que a pilha

armazena objetos do tipo *Object*. O código fonte da classe que implementa a pilha com alocação estática é mostrado no exemplo abaixo.

```
1  public class Stack {
2
3      private Object s[];
4      private int top = -1;
5
6      public Stack(int size) {
7          s = new Object[size];
8      }
9
10     public int size() {
11         return top + 1;
12     }
13
14     public boolean isEmpty() {
15         if (top == -1)
16             return true;
17         return false;
18     }
19
20     public boolean isFull() {
21         if (top == s.length - 1)
22             return true;
23         return false;
24     }
25
26     public void push(Object obj) throws OverflowException {
27         if (!isFull()) {
28             s[++top] = obj;
29         } else
30             throw new OverflowException();
31     }
32
33     public Object pop() throws UnderflowException {
34         if (!isEmpty()) {
35             Object o = s[top];
36             s[top] = null;
37             top--;
38             return o;
39         } else
40             throw new UnderflowException();
41     }
42
43     public Object top() {
44         return s[top];
45     }
46 }
```

Exemplo 16 – Código fonte da classe *Stack*.

A classe *Stack* mostrada no exemplo 16 define dois atributos: um vetor de objetos

s utilizado para armazenar os elementos da pilha (linha 3) e uma variável inteira *top* utilizada para indicar o topo da pilha (linha 4). O atributo *top* é inicializado com o valor menos um, o que indica que a pilha está vazia.

O construtor da classe *Stack*, definido nas linhas 6 a 8 recebe como parâmetro um valor inteiro que é utilizado na instanciação do vetor *s*.

O método *size*, definido nas linhas 10 a 12 retorna o número de elementos armazenados na pilha. Para obter a quantidade de elementos o método utiliza o atributo *top* que armazena o índice do topo da pilha.

O método *isEmpty*, definido nas linhas 14 a 18, retorna *true* se a pilha está vazia, ou *false* caso contrário. A pilha estará vazia sempre que o atributo *top* for igual a menos um.

O método *isFull*, definido nas linhas 20 a 24 retorna *true* se a pilha está cheia, ou *false* caso contrário. A pilha estará cheia sempre que o atributo *top* for igual ao tamanho do vetor menos um.

Um elemento é adicionado na pilha com o método *push*¹, definido nas linhas 26 a 31. Este método recebe como parâmetro um objeto do tipo *Object*. Inicialmente, um teste é feito para determinar se a pilha já se encontra cheia. Caso a pilha já esteja cheia, uma exceção de *overflow* é lançada (linha 30). Caso a pilha não esteja cheia, o atributo *top* é incrementado e o objeto é inserido na pilha (linha 28).

Um elemento é removido da pilha com o método *pop*², definido nas linhas 33 a 41. Este método remove e retorna o objeto contido no topo da pilha. Antes de realizar a remoção um teste é realizado para verificar se a pilha está vazia (linha 34). Caso a pilha esteja vazia, uma exceção de *underflow* é lançada. Caso a pilha não esteja vazia, uma referência ao objeto do topo da pilha é armazenada na variável local *o*, o valor *null* é atribuído à posição do vetor em que se encontra o topo, o atributo *top* é decrementado e a referência ao elemento que estava no topo é retornada.

O método *top* definido nas linhas 43 a 45 apenas retorna uma referência ao elemento armazenado no topo da pilha sem removê-lo.

O exemplo abaixo demonstra como a classe *Stack* pode ser utilizada em um programa Java.

```
1  public class StackTest {
2      public static void main(String args[]) {
3          Stack s = new Stack(5);
4          try {
5              s.push("1");
6              s.push("2");
7              s.push("3");
8              s.push("4");
9              s.push("5");
10             s.pop();
11             s.pop();
12             s.push("6");
13         } catch (OverflowException e) {
14             System.out.println(e.toString());
15         }
16         catch (UnderflowException e) {
17             System.out.println(e.toString());
18         }
19     }
20 }
```

Exemplo 17 – Utilização da classe *Stack*.

O exemplo 17 mostra o código fonte da classe de teste *StackTest*. Esta classe possui apenas um método *main* onde um objeto do tipo *Stack* é criado com capacidade máxima de cinco elementos (linha 3). Dentro do bloco *try* (linhas 5 a 12), cinco objetos do tipo *String* são inseridos na pilha, deixando esta cheia. Após a inserção destes elementos, dois elementos são removidos (linhas 10 e 11) e novamente outro elemento é inserido na pilha (linha 12). No exemplo 17 não ocorre nenhuma exceção de *overflow* ou *underflow*, pois não há nenhuma tentativa de inserção com a pilha cheia ou remoção com a pilha vazia.

Após a execução deste código, a pilha *s* terá a seguinte configuração:

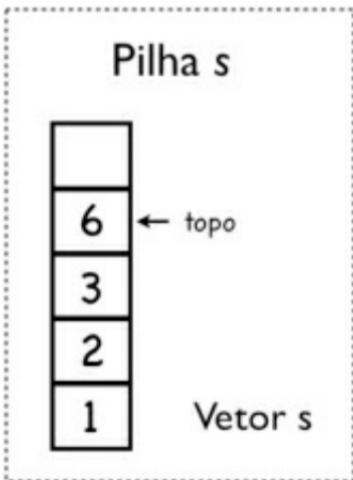


Figura 17 – Conteúdo da pilha após a execução do código do exemplo 17.

Esta seção apresentou a classe *Stack*, que implementa a estrutura de dados pilha com alocação estática. Assim como a classe *SequentialList* apresentada na seção anterior, a classe *Stack* também utiliza um vetor como estrutura interna de armazenamento. A manipulação dos dados do vetor não ocorre de maneira direta, mas sim através dos métodos *push* e *pop* que inserem e removem elementos da pilha respectivamente.

Fila

Uma fila pode ser vista como um tipo especial de lista linear geral em que os elementos são inseridos em uma extremidade e removidos da outra extremidade. É dito que uma fila implementa uma política de inserção e remoção onde o primeiro elemento a entrar é o primeiro elemento a sair. Esta política também é chamada de FIFO (do inglês *First In First Out*).

A estrutura de dados fila funciona de maneira semelhante a qualquer outro tipo de fila, como, por exemplo, uma fila de cinema, uma fila de supermercado, uma fila de banco etc. Uma pessoa entra sempre no final da fila e a pessoa que sai, sai sempre no início da fila.

Esta estrutura de dados é utilizada em diferentes tipos de aplicações. Uma

impressora, por exemplo, organiza as requisições de impressões em uma fila de impressões, de modo que a primeira requisição a chegar é sempre a primeira a ser atendida. Caso existam requisições em espera, uma nova requisição é inserida no final da fila.

Outra utilização desta estrutura ocorre em sistemas operacionais. Todos os processos que executam em um sistema operacional, como o Microsoft Windows, por exemplo, ficam em uma fila onde sempre o primeiro processo da fila é executado. Este processo executa por um certo período de tempo e é passado para o fim da fila para dar lugar a outro processo. Este mecanismo permite que os processos compartilhem o processador por instantes de tempo dando a impressão de que eles estão executando ao mesmo tempo, quando na verdade não estão.

Do ponto vista de um tipo abstrato de dados, uma fila define as seguintes operações:

- ◆ Inserção de um elemento na fila
- ◆ Remoção de um elemento da fila
- ◆ Obter o tamanho da fila
- ◆ Verificar se a fila está vazia
- ◆ Acessar o primeiro elemento da fila

Para demonstrar o funcionamento de uma fila, suponha uma fila F de números inteiros, inicialmente vazia, como mostra a figura 18. A estrutura de dados Fila utiliza duas variáveis, Início e Fim, para controlar o inicio e o fim da fila respectivamente.

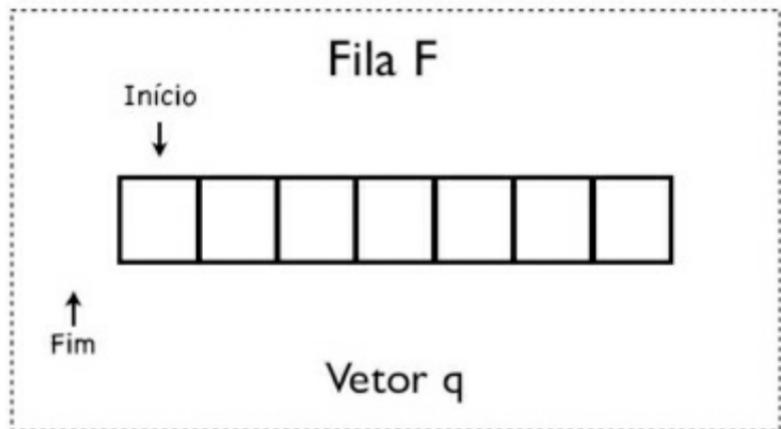


Figura 18 – Representação de uma fila utilizando vetor.

Na condição de fila vazia, a variável Inicio tem o valor zero e a variável Fim tem o valor menos um como indicado na figura 18.

Ao inserir o elemento 2 na fila F, esta ficará com a seguinte configuração:

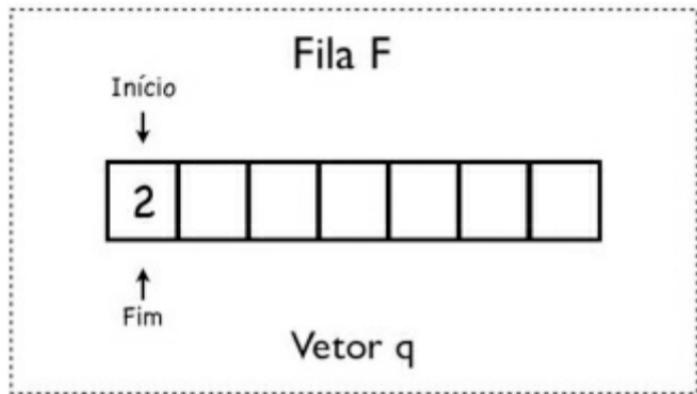


Figura 19 – Fila F após a inserção do elemento dois.

Após a inserção do elemento 2, a variável Fim foi incrementada para zero. Nesta situação, o elemento 2 é o primeiro e também o último elemento da fila.

Ao inserir o elemento quatro na fila, esta ficará com a seguinte configuração:

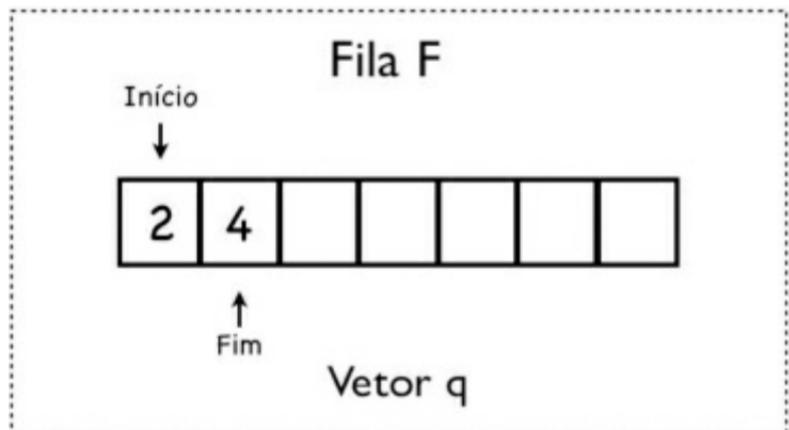


Figura 20 – Fila F após a inserção do elemento quatro.

Após a inserção do elemento quatro, a variável Fim foi incrementada para um como mostra a figura 20. Sempre que um elemento é inserido nesta estrutura a variável Fim será incrementada. Ao inserir mais um elemento na fila, esta ficará com a seguinte configuração:

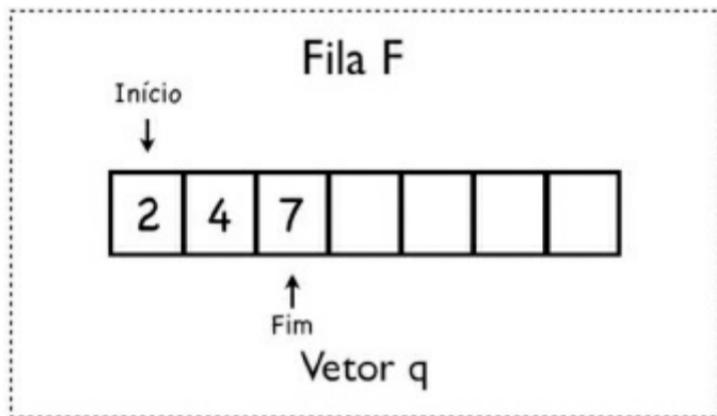


Figura 21 – Fila F após a inserção do elemento sete.

Novamente a variável Fim foi incrementada após a inserção do elemento sete. Após a remoção de um elemento da fila, esta ficará com a seguinte configuração:

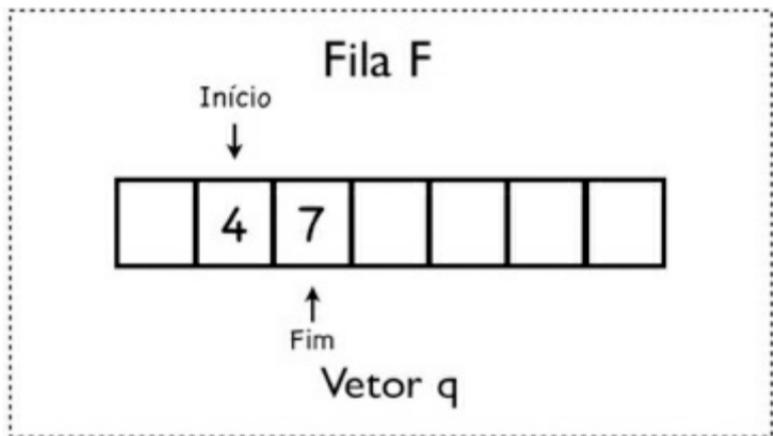


Figura 22 – Fila F após a remoção de um elemento.

Uma remoção em uma fila sempre ocorre no seu início. Desta maneira, ao ser realizada a remoção, foi retirado o elemento 2, que era o elemento que se encontrava no início da fila. Para realizar esta remoção, a variável Início foi incrementada. Após esta remoção, o elemento 4 passou a ser o novo início da fila.

Após a remoção de mais um elemento da fila, esta ficará com a seguinte configuração:

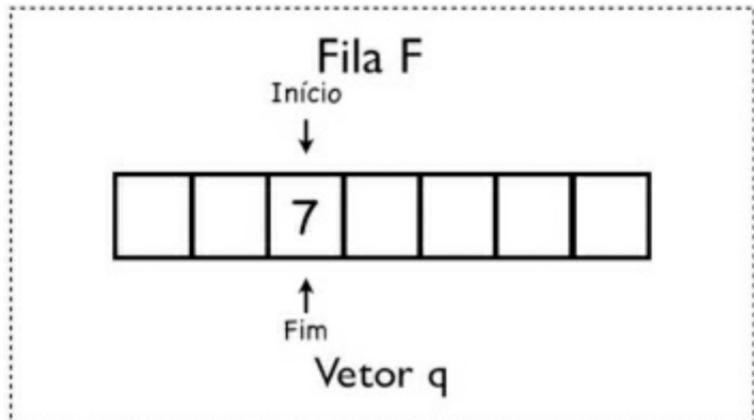


Figura 23 – Fila F após a remoção de um elemento.

Após esta remoção, o elemento 7 passa a ser o único elemento da fila. Sempre que um elemento é removido da fila, a variável `início` é incrementada.

Do ponto de vista de implementação, a classe que implementa uma fila deve possuir os seguintes métodos:

Método	Descrição
public boolean isEmpty()	Verifica se a fila está vazia. Retorna <code>true</code> caso ela esteja, ou <code>false</code> caso contrário.
public boolean isFull()	Verifica se a pilha está fila. Retorna <code>true</code> caso ela esteja, ou <code>false</code> caso contrário.
public void enqueue(Object o)	Insere um elemento no fim da fila. Public
Object dequeue()	Remove e retorna o objeto do início da fila.
public Object front()	Retorna (sem remover) uma referência ao objeto que se encontra no inicio da fila.

Tabela 6 – Métodos da classe que implementa uma fila.

Assim como nas classes de lista sequencial e pilha estudadas nas seções anteriores, esta implementação também assume que a fila armazena objetos do tipo `Object`. O código fonte da classe que implementa a estrutura de fila com alocação estática é mostrado abaixo.

A classe `Queue` mostrada no exemplo 18 define três atributos: um vetor que armazena objetos do tipo `Object` chamado `q` (linha 3) e dois inteiros `first` e `last` que definem o início e o fim da fila respectivamente. O atributo `first` é inicializado com menos um e o atributo `last` é inicializado com zero.

```
1  public class Queue {  
2  
3      protected Object q[];  
4      protected int first = 0, last = -1;  
5  
6      public Queue(int size) {  
7          q = new Object[size];  
8      }  
9  
10     public boolean isEmpty() {  
11         if (last == first - 1)  
12             return true;  
13         return false;  
14     }  
15  
16     public boolean isFull() {  
17         if (last == q.length - 1)  
18             return true;  
19         return false;  
20     }  
21  
22     public void enqueue(Object element) throws OverflowException {  
23         if (isFull()) {  
24             throw new OverflowException();  
25         } else {  
26             last++;  
27             q[last] = element;  
28         }  
29     }  
30  
31  
32     public Object dequeue() throws UnderflowException {  
33         if (isEmpty()) {  
34             throw new UnderflowException();  
35         } else {  
36             Object o = q[first];  
37             q[first] = null;  
38             first++;  
39             return o;  
40         }  
41     }  
42  
43     public Object getFirst() throws UnderflowException {  
44         if (isEmpty()){  
45             throw new UnderflowException();  
46         }  
47         else{  
48             return q[first];  
49         }  
50     }  
51 }
```

Exemplo 18 – Código fonte da classe *Queue*.

O construtor da classe *Queue* (linhas 6 a 8) recebe como parâmetro um número

inteiro que é utilizado para instanciar o vetor *q* com o tamanho máximo de elementos que a fila irá suportar.

O método *isEmpty* (linhas 10 a 14) retorna *true* caso a fila esteja vazia, ou *false* caso contrário. A fila é considerada vazia quando o atributo *last* for igual ao atributo *first* menos um. Quando a fila é instanciada, esta condição é verdadeira.

O método *isFull* (linhas 16 a 20) retorna *true* caso a fila esteja cheia, ou *false* caso contrário. A fila é considerada cheia quando o atributo *last* for igual ao tamanho do vetor menos um, ou seja, quando o atributo *last* estiver com o valor do último índice válido do vetor.

O método *enqueue* (linhas 22 a 29) recebe como parâmetro um objeto do tipo *Object* que será inserido na fila. Este método inicialmente realiza um teste para verificar se a fila está cheia, chamando o método *isFull*. Caso a fila esteja cheia uma exceção de *Overflow* é lançada. Caso a fila não esteja cheia, o atributo *last* é incrementado (linha 26) e o elemento é inserido no vetor na posição indicada por *last* (linha 27).

O método *dequeue* (linhas 32 a 41) remove e retorna um elemento contido na fila. Inicialmente, um teste é realizado para verificar se a fila está vazia através da chamada ao método *isEmpty* (linha 32). Caso a fila esteja vazia, uma exceção de *UnderFlow* é gerada. Caso a fila não esteja vazia, uma referência ao primeiro elemento da fila é armazenada na variável local *o* (linha 36), o valor *null* é armazenado na posição indicada pelo atributo *first* e então este é incrementado. Ao final da execução o método retorna uma referência ao objeto que estava armazenado na primeira posição da fila.

O método *getFirst* (linhas 43 a 50) funciona de maneira semelhante ao método *dequeue*, mas este não remove o elemento da primeira posição da fila. Este método apenas retorna uma referência ao primeiro elemento armazenado na fila (linha 48).

A implementação da classe *Queue* apresentada no exemplo 18 apresenta um problema. Supondo uma fila com capacidade para sete elementos, caso um elemento seja inserido e removido sete vezes da fila, como mostrado no exemplo 19, esta estará com uma configuração como a mostrada na figura 24.

```

1  public class QueueTest {
2      public static void main(String[] args) {
3          Queue q = new Queue(7);
4          try{
5              for (int i = 0; i < 7; i++){
6                  q.enqueue(new Integer(i));
7                  q.dequeue();
8              }
9          }
10         catch (OverflowException e){
11             System.out.println(e);
12         }
13         catch (UnderflowException e){
14             System.out.println(e);
15         }
16
17         if (q.isFull()){
18             System.out.println("Fila Cheia !");
19         }
20     }
21 }
```

Exemplo 19 – Utilização da classe Queue.

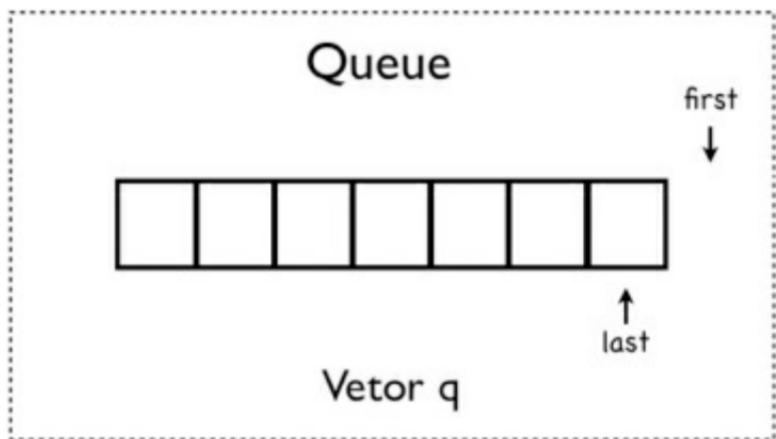


Figura 24 – Configuração da fila após a inserção e remoção de sete elementos.

Nesta situação uma chamada ao método *isFull* irá retornar *true*, indicando uma condição de fila cheia quando a fila na verdade está vazia. Isso acontece porque tanto na inserção quanto na remoção os atributos *first* e *last* são sempre incrementados.

Uma maneira de lidar com este problema é modificar o tratamento dos atributos *first* e *last* tanto na inserção quanto na remoção dos elementos da fila. A classe

CircularQueue (fila circular) mostrada no exemplo 20 tem este tratamento modificado de modo a resolver este problema. Nesta estrutura de dados, toda vez que o início ou o final da fila atinge o final do vetor, estes “dão a volta” de modo a utilizar os espaços vazios do vetor.

Considerando a fila mostrada na figura 25 como sendo uma fila circular que armazena objetos do tipo *Integer*, ao ser inserido mais um elemento nesta fila, esta fica como é mostrado na figura 26, em que o atributo *last* aponta o índice zero do vetor.

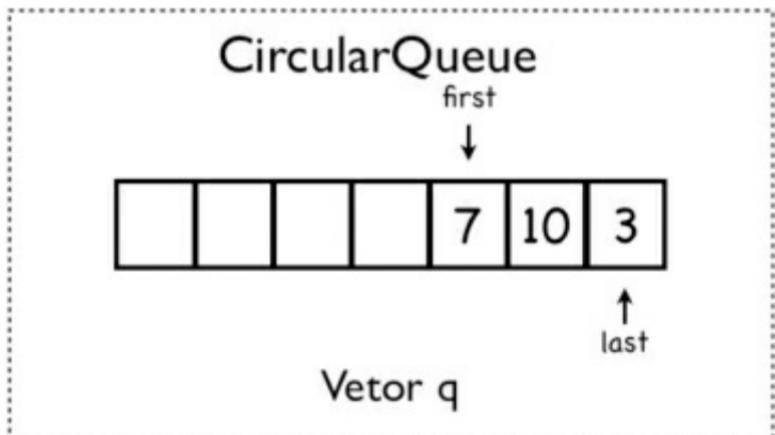


Figura 25 – Fila circular com o último elemento armazenado na última posição do vetor.

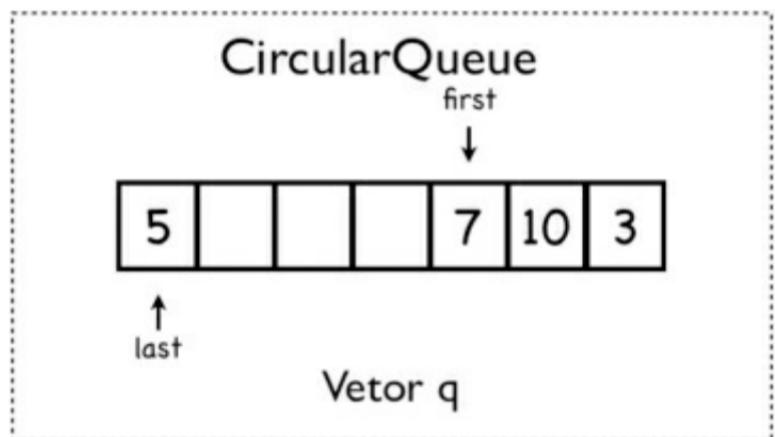


Figura 26 – Fila circular após a inserção de um elemento.

Ao ser inserido o elemento 5, o atributo *last* que estava apontando para a última posição do vetor passa a apontar para a primeira. Deste modo, pode-se dizer que a fila circulou, ou “deu a volta”, por isso o nome fila circular.

O mesmo acontece na remoção de elementos da fila. Considerando a fila mostrada na figura 27 como uma fila circular que armazena elementos do tipo *Integer*, ao ser removido um elemento da fila, esta ficará como mostrado na figura 28.

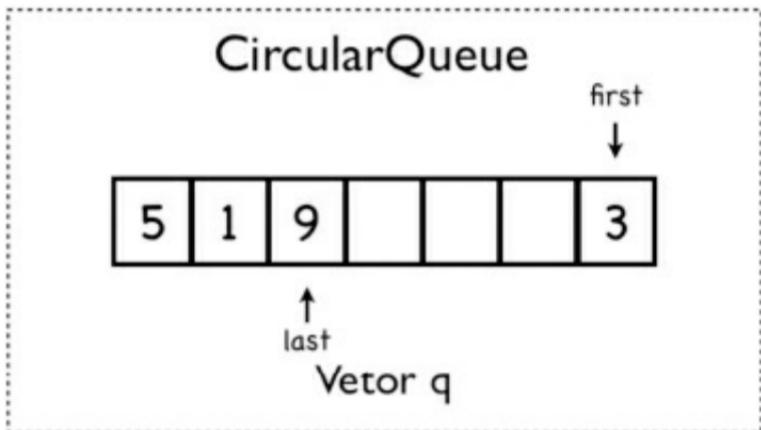


Figura 27 – Fila circular com o primeiro elemento armazenado na última posição do vetor.

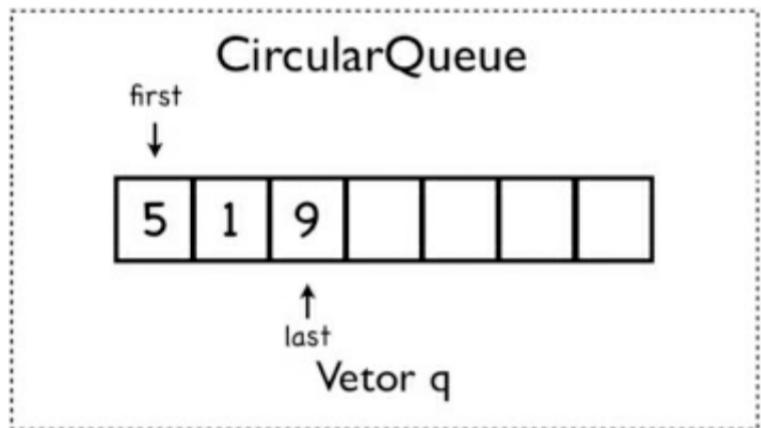


Figura 28 – Fila circular após a remoção de um elemento.

Ao ser removido o elemento 3, o atributo *first* que estava apontando para a última posição do vetor passa a apontar para a primeira. O código fonte da classe *CircularQueue* é mostrado a seguir (exemplo 20).

A classe *CircularQueue* define os mesmos atributos da classe *Queue*: um vetor de objetos do tipo *Object* e as variáveis inteiras *first* e *last*. A inicialização dos atributos inteiros é diferente no caso da fila circular, pois ambos são inicializados com menos um.

O método *isFull* (linhas 10 a 15) testa duas possibilidades para verificar se a fila está cheia. A fila é considerada cheia se o atributo *first* aponta para o primeiro elemento do vetor e o atributo *last* para o último, no caso da fila ainda não ter circulado. No caso da fila ter circulado, o teste verifica se o atributo *last* é igual ao atributo *first* menos um. Este caso indica que não existe mais nenhuma posição livre no vetor.

O método *isEmpty* (linhas 17 a 22) testa se o atributo *first* é igual a menos um. Esta condição indica que a fila está vazia.

A implementação do método *enqueue* (linhas 24 a 37) é mais sofisticada que a implementação do método *enqueue* da classe *Queue*. Inicialmente, o método testa se a fila está cheia. Em caso positivo, uma exceção é lançada. Caso a fila não esteja cheia, um teste é realizado no atributo *last* para verificar se este se encontra na última posição do vetor ou está com o valor menos um (condição de fila vazia). Em caso positivo, é atribuído o valor zero para este atributo. Isso pode ocorrer em duas situações: quando a fila está vazia ou quando a fila circulou após a inserção de um elemento.

```
1  public class CircularQueue {
2
3      private Object[] q;
4      private int first = -1, last = -1;
5
6      public CircularQueue(int n) {
7          q = new Object[n];
8      }
9
10     public boolean isFull() {
11         if ((first == 0 && last == q.length - 1) || (first == last + 1))
12             return true;
13         else
14             return false;
15     }
16
17     public boolean isEmpty() {
18         if (first == -1)
19             return true;
20         else
21             return false;
22     }
23
24     public void enqueue(Object o) throws OverflowException {
25
26         if (isFull())
27             throw new OverflowException();
28         if (last == q.length - 1 || last == -1) {
29             last = 0;
30             q[last] = o;
31             if (first == -1)
32                 first = 0;
33         } else {
34             last = last + 1;
35             q[last] = o;
36         }
37     }
38
39     public Object dequeue() throws UnderflowException {
40         if (!isEmpty()) {
41             Object o = q[first];
42             q[first] = null;
43             if (first == last)
44                 first = last = -1;
45             else if (first == q.length - 1)
46                 first = 0;
47             else
48                 first++;
49             return o;
50         } else
51             throw new UnderflowException();
52     }
53
54     public Object getFirst() throws UnderflowException {
55         if (isEmpty())
56             throw new UnderflowException();
57     }
58     else{
59         return q[first];
60     }
61 }
62 }
```

Exemplo 20 – Código fonte da classe *CircularQueue* que implementa uma fila circular.

Caso o atributo *last* não esteja em nenhuma extremidade do vetor, o código das linhas 34 e 35 é executado.

O método *dequeue* da classe *CircularQueue* (linhas 39 a 52) também possui uma implementação mais sofisticada que na classe *Queue*. Na classe *CircularQueue* ele também remove e retorna uma referência ao objeto removido. Inicialmente, um teste é feito para verificar se a fila encontra-se vazia. Em caso positivo, uma exceção é lançada. Caso a fila não esteja vazia, uma referência ao objeto armazenado na primeira posição da fila é armazenada na variável local *o* e o valor *null* é armazenado na posição apontada pelo atributo *first* (linha 42). O teste realizado na linha 43 verifica se a fila contém apenas um elemento. Neste caso o valor menos um é atribuído a ambos os atributos. O teste realizado na linha quarenta e cinco verifica se o atributo *first* encontra-se na última posição do vetor. Em caso positivo, o valor zero é atribuído ao atributo *first*, fazendo com que a fila “circule”. Se ambos os testes da linha 43 e da linha 45 retornarem *false*, a inserção é realizada da mesma maneira que na classe *Queue*.

A implementação do método *getFirst* é semelhante à implementação da classe *Queue*.

Este capítulo introduziu o conceito de lista linear com alocação estática. Neste tipo de lista normalmente utiliza-se um vetor como estrutura de armazenamento dos elementos, pois estes ficam armazenados em posições contíguas de memória. A primeira seção do capítulo apresentou a estrutura lista sequencial juntamente com sua implementação. As seções seguintes apresentaram as estruturas de pilha e de fila também com suas implementações, utilizando alocação estática. Estas estruturas são listas lineares com políticas específicas de inserção e remoção de elementos.

1 O termo *push* é utilizado em inglês para indicar a inserção de um elemento na pilha.

2 O termo *pop* é utilizado em inglês para indicar a remoção de um elemento da pilha.

CAPÍTULO 4

LISTAS LINEARES COM ALOCAÇÃO DINÂMICA

Este capítulo introduz o conceito de lista linear geral com alocação dinâmica. Este tipo de estrutura, também chamado de lista simplesmente encadeada, não utiliza alocação estática como as estruturas estudadas no Capítulo 3. Uma vantagem desta abordagem é que a lista não possui mais um tamanho fixo como na lista sequencial. A memória para novos elementos é alocada dinamicamente à medida que novos elementos são inseridos na lista. Utilizando esta abordagem, os elementos da lista ficam dispostos aleatoriamente na memória e são interligados por referências.

Classes autorreferenciais

As estruturas de dados que estudaremos no restante deste livro fazem uso de classes autorreferenciais para armazenar seus elementos. Uma classe autorreferencial é uma classe que contém uma variável de instância que referencia outro objeto do mesmo tipo de classe. No caso de listas lineares com alocação dinâmica utilizaremos a classe Node como classe autorreferencial. Esta classe irá representar um nó, ou nodo, em que internamente é armazenado o elemento. A estrutura da classe Node é mostrada na figura 29.

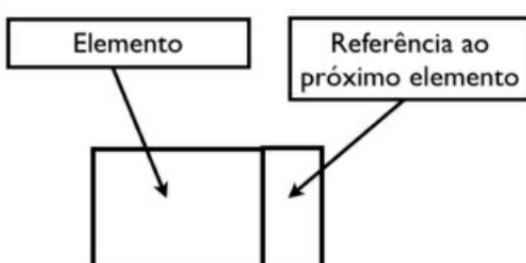


Figura 29 – Estrutura de um nodo de uma lista simplesmente encadeada.

A implementação da classe *Node* que representa um nodo de uma lista simplesmente encadeada é mostrada no exemplo 21.

```
1  public class Node {  
2  
3      private Object data;  
4      private Node nextNode;  
5  
6      public Node( Object object ) {  
7          this( object, null );  
8      }  
9  
10     public Node( Object object, Node node ) {  
11         data = object;  
12         nextNode = node;  
13     }  
14  
15     public Object getData() {  
16         return data;  
17     }  
18     public void setData (Object element){  
19         data = element;  
20     }  
21  
22     public Node getNext() {  
23         return nextNode;  
24     }  
25  
26     public void setNext(Node o) {  
27         nextNode = o;  
28     }  
29 }
```

Exemplo 21 – Código fonte da classe *Node* que implementa um nodo de uma lista simplesmente encadeada.

A classe *Node* mostrada no exemplo 21 define dois atributos: uma referência a um *Object* (linha 3), utilizada para armazenar o dado dentro do nodo e uma referência a um objeto do tipo *Node* (linha 4), que é utilizada para fazer o encadeamento dos nodos.

A classe *Node* também define dois construtores: um que recebe uma referência a um objeto que é o dado a ser armazenado dentro do nodo (linha 6) e outro construtor que recebe uma referência a um objeto e também uma referência ao seu sucessor na lista (linha 10). Ambos os construtores apenas atualizam os atributos da classe.

Também são definidos métodos *get* e *set* para ambos os atributos (linhas 15 a 28).

Antes de iniciarmos o estudo mais detalhado da lista simplesmente encadeada, a próxima seção faz uma breve revisão do gerenciamento de memória empregado pelo

ambiente de execução da linguagem Java. Os conceitos apresentados nesta seção são amplamente estudados nas estruturas de dados vistas no decorrer do livro.

Gere nciamento de memória

Criar e manter estruturas de dados dinâmicas requer alocação dinâmica de memória, que é a capacidade de um programa obter mais espaço em memória em tempo de execução para armazenar novos nodos e também liberar o espaço desnecessário. Os programas escritos em Java não liberam explicitamente a memória alocada dinamicamente. Em vez disso, o ambiente de execução da linguagem realiza a coleta de lixo automática de objetos que não são mais referenciados no programa. Esta coleta é feita por um processo chamado *garbage collector* (coletor de lixo).

O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de espaço em disco no caso de um sistema de memória virtual. Frequentemente estes limites são menores, pois o computador precisa dividir sua memória com diferentes aplicativos. A seguinte linha de código:

```
Node n = new Node(new Integer(10));
```

Aloca a memória para armazenar um objeto do tipo Node e armazena neste objeto um objeto do tipo Integer. Esta linha também retorna uma referência à memória alocada para o objeto, que é atribuída à variável n. As seções seguintes tratam de estruturas de dados que utilizam alocação dinâmica de memória e classes autorreferenciais para criar estruturas de dados dinâmicas.

Lista simplesmente encadeada

Assim como na estrutura lista sequencial, na estrutura de lista encadeada não existe uma política específica para inserção e remoção de elementos. Estes podem ser inseridos e removidos em qualquer posição da lista. Os elementos de uma lista simplesmente encadeada também são chamados de nodos ou nós.

Por definição, uma lista simplesmente encadeada é uma sequência de objetos (nodos), alocados dinamicamente, em que cada objeto possui uma referência ao seu

sucessor na lista, como mostrado na figura 30.

Em uma lista encadeada cada elemento é armazenado dentro de um nodo da lista e, por sua vez, cada nodo da lista referencia seu sucessor na lista, com exceção do último elemento da lista que não referencia nenhum elemento. Neste sentido, uma lista encadeada pode ser vista como uma lista de nodos.

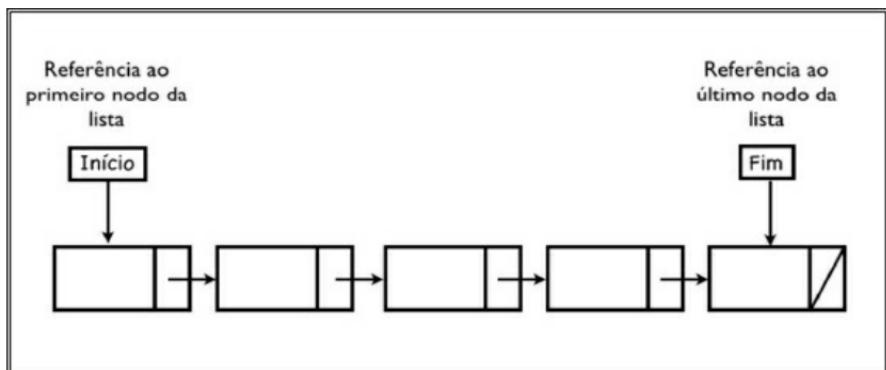


Figura 30 – Estrutura de uma lista simplesmente encadeada.

A estrutura de lista encadeada mostrada na figura 30 possui, além da lista de nodos, uma referência ao primeiro nodo da lista e também uma referência ao último nodo da lista. Estas referências são utilizadas para acessar os elementos da lista. Um programa Java que necessita percorrer toda a lista inicia o processo através da referência ao primeiro elemento da lista. De maneira similar, caso uma inserção precise ser feita no final lista, esta pode ser feita com o auxílio da referência ao último elemento da lista sem a necessidade de percorrer a lista inteira.

Como dito anteriormente, em uma lista encadeada, cada vez que um elemento é inserido, a memória para armazenar este elemento é alocada dinamicamente. Mais precisamente, cada vez que um novo elemento é inserido, um novo nodo da lista é alocado para armazenar o novo elemento e este é inserido na lista.

Para demonstrar o funcionamento de uma lista simplesmente encadeada, suponha uma lista L de números inteiros com os seguintes elementos: 4, 7 e 8 como mostra a figura 31.

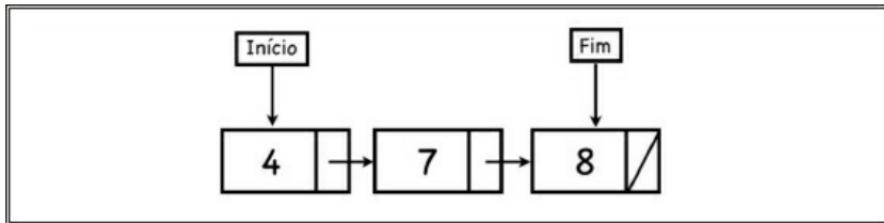


Figura 31 – Lista simplesmente encadeada.

Nesta lista o elemento 4 é o primeiro elemento e o elemento 8 é o último elemento da lista.

Ao inserir o elemento 12 no fim da lista, esta ficará com a seguinte configuração:

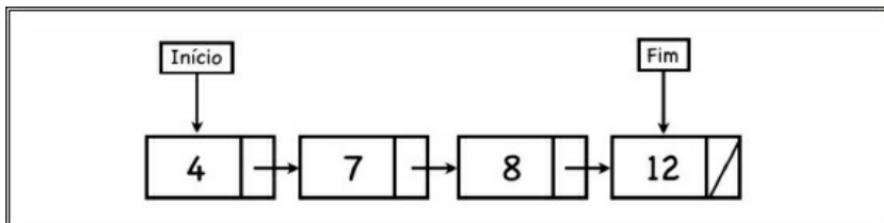


Figura 32 – Lista simplesmente encadeada após a inserção do elemento 12 no final.

Para inserir o elemento 12 na lista L um novo nodo foi criado para armazenar o elemento 12 e este foi inserido no fim da lista. Para realizar a inserção deste novo nodo, o nodo que contém o elemento 8 passou a referenciar o novo nodo. A referência ao último elemento da lista também foi atualizada de acordo, pois após a inserção o nodo com o elemento 12 é o novo último elemento.

Ao inserir o elemento 11 no início na lista, esta ficará com a seguinte configuração:

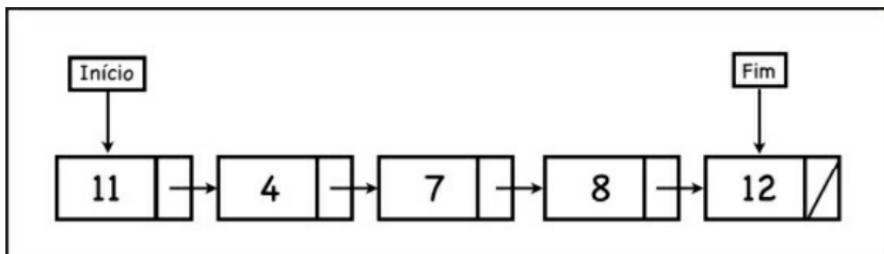


Figura 33 – Lista simplesmente encadeada após a inserção do elemento 11 no início.

Para inserir o elemento 11 no início da lista L um novo nodo foi criado para armazená-lo e então este nodo foi inserido na lista. Este novo nodo tem como sucessor o nodo com o elemento 4 (que anteriormente era o primeiro elemento da lista). Nesta inserção a referência ao primeiro elemento da lista também foi atualizada de acordo, pois o elemento 11 passou a ser o novo primeiro elemento.

Ao remover o elemento 12 da lista, esta ficará com a seguinte configuração:

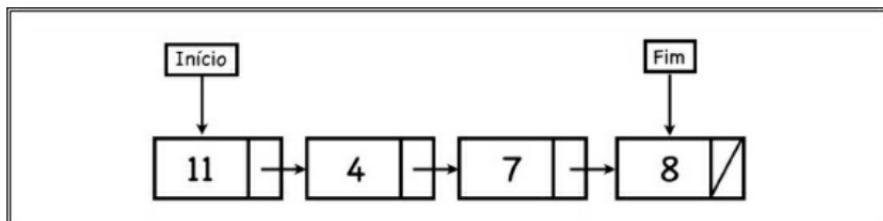


Figura 34 – Lista simplesmente encadeada após a remoção do elemento 12.

Para remover o elemento 12, o valor *null* foi atribuído ao sucessor do nodo que armazena o elemento 8 e a referência ao último elemento da lista foi atualizada de acordo, pois após esta remoção o elemento 8 passou a ser o novo final da lista.

Do ponto de vista de implementação, a classe que representa uma lista simplesmente encadeada possui os seguintes métodos:

Método	Descrição
public boolean isEmpty()	Verifica se a lista está vazia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
public void insertAtFront(Object o)	Insere um elemento na primeira posição da lista.
public void insertAtBack(Object o)	Insere um elemento na última posição da lista.
public Object removeFromFront()	Remove e retorna o objeto contido na primeira posição da lista.
public Object removeFromBack()	Remove e retorna o objeto contido na última posição da lista.
public Node getFirst()	Retorna uma referência ao primeiro nodo da lista.
public Node getLast()	Retorna uma referência ao último nodo da lista.

Tabela 7 – Métodos da classe de lista simplesmente encadeada.

A classe *List* que implementa uma lista simplesmente encadeada é mostrada no exemplo 22.

```
1  public class List {
2
3      protected Node firstNode;
4      protected Node lastNode;
5
6      public List() {
7          firstNode = null;
8          lastNode = null;
9      }
10
11     public Node getFirst(){
12         return firstNode;
13     }
14
15     public Node getLast(){
16         return lastNode;
17     }
18
19     public void insertAtFront( Object insertItem ) {
20         if ( isEmpty() )
21             firstNode = lastNode = new Node( insertItem );
22         else
23             firstNode = new Node( insertItem, firstNode );
24     }
25
26     public void insertAtBack( Object insertItem ) {
27         if ( isEmpty() )
28             firstNode = lastNode = new Node( insertItem );
29
30         else {
31             lastNode.setNext ( new Node( insertItem ) );
32             lastNode = lastNode.getNext();
33         }
34     }
35
36     public Object removeFromFront() throws UnderflowException {
37         if ( isEmpty() )
38             throw new UnderflowException();
39
40         Object removedItem = firstNode.getData();
41
42         if ( firstNode == lastNode )
43             firstNode = lastNode = null;
44         else
45             firstNode = firstNode.getNext();
46
47         return removedItem;
48     }
49
50     public Object removeFromBack() throws UnderflowException {
51         if ( isEmpty() )
52             throw new UnderflowException();
53
54         Object removedItem = lastNode.getData();
55
56         if ( firstNode == lastNode ) {
57             firstNode = lastNode = null;
58         }
59         else {
60             Node current = firstNode;
61             while ( current.getNext() != lastNode ){
62                 current = current.getNext();
63             }
64             lastNode = current;
65             current.setNext(null);
66         }
67         return removedItem;
68     }
69
70     public boolean isEmpty() {
71         return firstNode == null;
72     }
73 }
```

Exemplo 22 – Código fonte da classe *List* – Lista simplesmente encadeada.

A classe *List* mostrada no exemplo 22 define dois atributos: *frstNode* e *lastNode* (linhas 4 e 5). Estes atributos são, respectivamente, referências ao primeiro e ao último nodos da lista simplesmente encadeada.

O construtor da classe *List* (linhas 6 a 9) não recebe nenhum parâmetro e inicializa os dois atributos *firstNode* e *lastNode* para *null*. Diferente da implementação da classe *SequentialList*, em que é necessário especificar um tamanho máximo para a lista, esta implementação, por não utilizar um vetor como estrutura de armazenamento, não possui um número máximo de elementos, pois a memória é alocada sob demanda.

O método *getFirst* (linhas 11 a 13) retorna uma referência ao nodo que armazena o primeiro elemento da lista.

O método *getLast* (linhas 15 a 17) retorna uma referência ao nodo que armazena o último elemento da lista.

O método *insertAtFront* (linhas 19 a 24) insere o objeto recebido como parâmetro no início da lista. Inicialmente, o método testa a condição de lista vazia através da chamada ao método *isEmpty*. Caso este método retorne *true*, a lista encontra-se vazia e os atributos *firstNode* e *lastNode* passam a referenciar um novo nodo, contendo o objeto passado como parâmetro. Caso a lista não esteja vazia, um novo nodo é criado, contendo o objeto a ser inserido, e este novo nodo tem como sucessor o primeiro nodo da lista e então o atributo *firstNode* passa a referenciar este novo nodo. Esta inserção ocorre na linha 23.

O método *insertAtBack* (linhas 26 a 34) insere o objeto recebido como parâmetro no final da lista. Inicialmente, o método testa a condição de lista vazia através da chamada ao método *isEmpty*. Caso este método retorne *true*, a lista encontra-se vazia, e os atributos *firstNode* e *lastNode* passam a referenciar um novo nodo contendo o objeto passado como parâmetro. Caso a lista não esteja vazia, um novo nodo é criado, contendo o objeto passado como parâmetro, e o sucessor do último nodo passa a referenciar este novo nodo e o atributo *lastNode* passa a referenciar este novo nodo.

O método *removeFromFront* (linhas 36 a 48) remove e retorna o elemento da primeira posição da lista. Inicialmente, o método testa a condição de lista vazia através da chamada ao método *isEmpty*. Caso a lista esteja vazia, uma exceção de *Underflow* é lançada. Caso a lista não esteja vazia, uma referência ao elemento armazenado no primeiro nodo da lista é armazenada na variável local *removedItem* (linha 40). Na linha 42 um teste é realizado para verificar se os atributos *firstNode* e *lastNode* referenciam o mesmo nodo. Caso este teste retorne *true* significa que a lista contém apenas um

elemento e tanto o atributo *firstNode* quanto o atributo *lastNode* recebem o valor *null*, ocasionando a remoção deste elemento. Caso a lista não esteja vazia, o atributo *firstNode* passa a referenciar seu sucessor na lista (linha 45), removendo assim o primeiro elemento da lista. Ao final o método retorna a referência ao elemento removido.

O método *removeFromBack* remove e retorna o elemento armazenado na última posição da lista. Caso a lista esteja vazia, uma exceção de *underflow* é lançada (linha 52). Caso a lista não esteja vazia, uma referência ao elemento armazenado no último nodo da lista é armazenada na variável local *removedItem* (linha 54). Caso a lista contenha apenas um elemento, tanto o atributo *firstNode* quanto o atributo *lastNode* recebem o valor *null*, removendo assim este elemento (linha 57). Caso a lista contenha mais de um elemento, uma variável local deve ser utilizada para “caminhar” pela lista de modo a chegar no penúltimo nodo. Isso é preciso, pois uma lista simplesmente encadeada possui apenas uma referência ao próximo elemento da lista e não ao anterior. A variável *current* (linha sessenta) inicialmente referencia o primeiro nodo da lista (*first Node*). O laço das linhas 61 a 63 faz com que esta variável “ande” pela lista até o penúltimo elemento. O laço para sua execução se dá quando o sucessor da variável *current* estiver referenciando o último nodo da lista (linha 61). Uma vez que a variável *current* esteja referenciando o penúltimo nodo da lista, o atributo *lastNode* passa a referenciar o nodo referenciado por *current* (linha 64) e o sucessor do novo *lastNode* recebe o valor *null*, removendo, desta maneira, o último elemento da lista.

O exemplo abaixo mostra como a classe *List* pode ser utilizada em um programa Java.

```

1  public class ListTest {
2
3      public static void main( String args[] ) {
4
5          List list = new List();
6          list.insertAtFront(new Integer(1));
7          list.insertAtBack(new Integer(2));
8          list.insertAtFront(new Integer(3));
9          list.insertAtBack(new Integer(4));
10         list.insertAtFront(new Integer(5));
11         list.insertAtBack(new Integer(6));
12
13         try{
14             List.removeFromBack();
15             List.removeFromFront();
16         }
17         catch (UnderflowException e){
18             System.out.println(e);
19         }
20     }
21 }
```

Exemplo 23 – Classe *List* utilizada em um programa Java.

Ao final do método *main* da classe *ListTest* do exemplo 23, a lista estará com a configuração mostrada na figura 35.

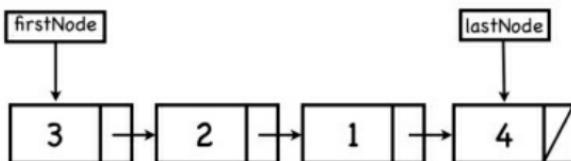


Figura 35 – Configuração do objeto *list* após a execução do exemplo 23.

Esta seção introduziu o conceito de estrutura de dados com alocação dinâmica de memória através do estudo da estrutura de lista simplesmente encadeada. Esta lista difere da lista sequencial apresentada no capítulo anterior por não utilizar um vetor como estrutura de armazenamento. Nesta estrutura a memória é alocada à medida que os elementos são inseridos na lista e também liberada à medida que os elementos são removidos.

Nas seções seguintes serão apresentadas as estruturas de pilha e fila utilizando alocação dinâmica de memória

Pilha

A estrutura de pilha apresentada no Capítulo 3 pode também ser implementada com alocação dinâmica de memória como a estrutura de lista apresentada na seção anterior. Utilizando esta abordagem, cada elemento da pilha é armazenado em um nodo e estes são inseridos e removidos sempre no topo da pilha. Uma vantagem desta abordagem em relação à abordagem que utiliza alocação estática é que não há mais um tamanho fixo para a pilha. Por não utilizar um vetor como estrutura de armazenamento, não existe a necessidade de definir um tamanho máximo no momento da instanciação.

Para demonstrar o funcionamento de uma pilha com alocação dinâmica de memória, suponha uma pilha P de números inteiros com os elementos 2, 3 e 6 como mostra a figura 36.

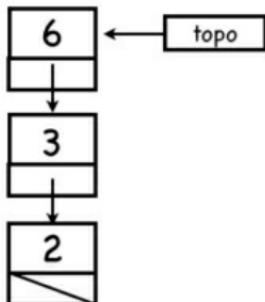


Figura 36 – Pilha com alocação dinâmica de memória.

Nesta pilha cada elemento está armazenado em um nodo. O nodo com o elemento 6 está no topo da pilha e tem como sucessor o nodo com o elemento 3. O nodo com o elemento 2 está na base da pilha e não possui nenhum sucessor.

Após a inserção do elemento 4 na pilha esta ficará com a seguinte configuração:

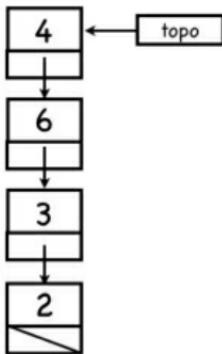


Figura 37 – Pilha após a inserção do elemento 4.

Para realizar esta inserção, um novo nodo foi criado para armazenar o elemento 4, este novo nodo passou a referenciar o topo da pilha (elemento 6) e então o topo da pilha passou a referenciar o novo nodo.

Após a remoção de um elemento da pilha, esta ficará com a seguinte configuração:

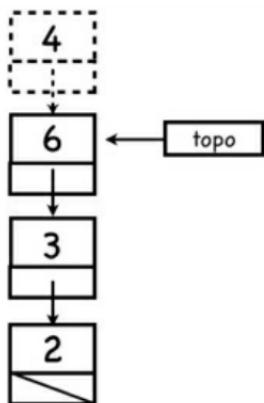


Figura 38 – Pilha após a operação de remoção.

A remoção em uma pilha com alocação dinâmica ocorre quando o topo da pilha passa a referenciar seu sucessor. Na remoção mostrada na figura 38, quando a referência ao topo passa a referenciar seu sucessor, o nodo com o elemento 4 é

removido da memória pelo *garbage collector*.

A classe que implementa uma pilha com alocação dinâmica apresenta os seguintes métodos:

Método	Descrição
public boolean isEmpty()	Verifica se a pilha está vazia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
public void push(Object o)	Insere um elemento no topo da pilha
public Object pop()	Remove e retorna o objeto contido no topo da pilha.
public Object top()	Retorna (sem remover) uma referência ao objeto do topo da pilha.

Tabela 8 – Métodos da classe que implementa uma pilha com alocação dinâmica.

Assim como na classe de lista simplesmente encadeada, esta implementação assume que os nodos da lista armazenam objetos do tipo *Object*.

A classe que implementa a estrutura de pilha com alocação dinâmica é mostrada no exemplo 24.

```

1  public class Stack {
2
3      private Node top;
4
5      public Stack() {
6          top = null;
7      }
8
9      public boolean isEmpty() {
10         return (top == null);
11     }
12
13     public Object top() throws UnderflowException {
14         if (isEmpty()) {
15             throw new UnderflowException();
16         } else {
17             return top.getData();
18         }
19     }
20
21     public void push (Object insertItem) {
22         Node n = new Node(insertItem);
23         n.setNext(top);
24         top = n;
25     }
26
27     public Object pop() throws UnderflowException {
28         if (isEmpty()) {
29             throw new UnderflowException();
30         }
31         Node ret = top;
32         top = top.getNext();
33         return ret.getData();
34     }
35 }
```

Exemplo 24 – Código fonte da classe *Stack* – Pilha com alocação dinâmica de memória.

A classe *Stack* definida no exemplo 24 define um atributo do tipo *Node*, chamado *top*, que é utilizado para referenciar o topo da pilha.

O construtor da classe *Stack* (linhas 5 a 7) não recebe nenhum parâmetro e inicializa o atributo *top* com o valor *null*.

O método *isEmpty* (linhas 9 a 11) retorna *true* caso a pilha esteja vazia, ou *false* caso contrário. A pilha é considerada vazia sempre que o atributo *top* conter o valor *null*.

O método *top* (linhas 13 a 19) retorna uma referência ao objeto armazenado no nodo do topo da pilha. Inicialmente, um teste é realizado para verificar se a pilha está vazia (linha 14). Caso este teste retorne *true*, uma exceção é lançada (linha 15). Caso contrário, uma referência ao objeto armazenado dentro do nodo referenciado pelo

atributo *top* é retornada.

O método *push* (linhas 21 a 25) recebe como parâmetro um objeto a ser inserido na pilha. Para inserir este objeto, um novo nodo é criado (linha 22) para armazenar este objeto. O sucessor deste nodo é então apontado para o topo da pilha (linha 23) e então o topo passa a referenciar o novo nodo (linha 24).

O método *pop* (linhas 27 a 34) remove e retorna o objeto contido no topo da pilha. Inicialmente, um teste é realizado para verificar se a pilha está vazia. Caso este teste retorne *true*, uma exceção é lançada (linha 29). Caso contrário, uma referência ao topo da pilha é armazenada na variável local *ret* (linha 31), e o topo da pilha passa a referenciar seu sucessor (linha 32) e então o objeto armazenado dentro do nodo *ret* é retornado.

O exemplo 25 mostra como a classe *Stack* mostrada no exemplo 24 pode ser utilizada em um programa Java.

```
1 public class StackTest {
2
3     public static void main(String[] args) {
4
5         Stack s = new Stack();
6         s.push(new Integer(1));
7         s.push(new Integer(2));
8         s.push(new Integer(3));
9         s.push(new Integer(4));
10
11         try{
12             s.pop();
13             s.pop();
14         }
15         catch (UnderflowException e){
16             System.out.println(e);
17         }
18     }
19 }
```

Exemplo 25 – StackTest.

Ao final do método *main* da classe *StackTest* do exemplo 25 a configuração da pilha estará como mostra a figura 39:

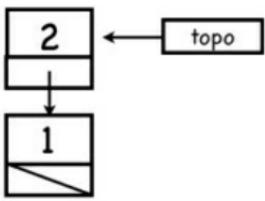


Figura 39 – Conteúdo da pilhas após a execução do exemplo 25.

Esta seção apresentou a estrutura de dados pilha implementada com alocação dinâmica de memória. Nesta implementação cada elemento da pilha é mantido em um nodo e os elementos são sempre inseridos e removidos no topo da pilha segundo a política LIFO (*Last In First Out*). Outra característica desta implementação, quando comparada com a implementação mostrada no capítulo anterior é que não há mais a necessidade de definir um tamanho máximo para a pilha quando esta é instanciada. Isso acontece porque à medida que novos elementos são inseridos na pilha a memória é alocada dinamicamente. O mesmo ocorre na remoção de elementos quando a memória é liberada.

Fila

Assim como a estrutura de pilha apresentada na seção anterior, a estrutura de fila também pode ser implementada com alocação dinâmica de memória. Nesta implementação a fila é uma sequência de nodos interligados por referências e duas referências são utilizadas para marcar o início e o fim da fila. Este esquema é mostrado na figura 40.

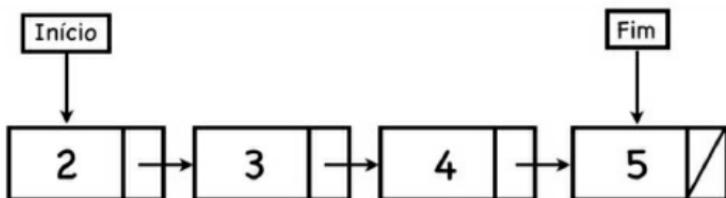


Figura 40 – Fila – Implementação com alocação dinâmica de memória.

A estrutura mostrada na figura 40 é semelhante à estrutura de lista simplesmente encadeada. A única diferença no caso da fila é a política de inserção e remoção de elementos. Conforme visto no Capítulo 3, a fila segue a política FIFO de inserção e remoção de elementos, em que o primeiro elemento a ser removido é sempre o primeiro que foi inserido.

No caso da fila mostrada na figura 40, após a remoção de um elemento da mesma, esta ficará com a seguinte configuração:

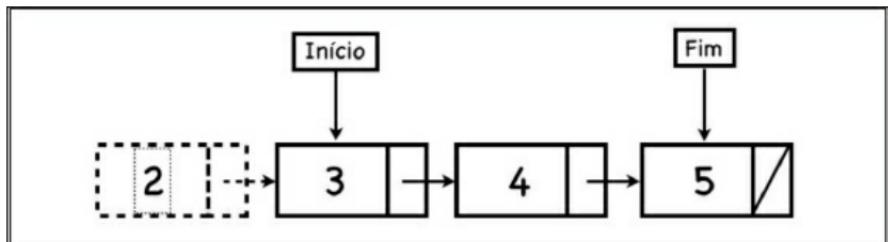


Figura 41 – Fila após a operação de remoção.

Como em uma fila a remoção sempre ocorre no início, o elemento removido foi o elemento 2. Para realizar esta remoção, a referência ao início da fila passou a referenciar seu sucessor (nodo com o elemento 3). O nodo com o elemento dois, mostrado com linha tracejada, é removido da memória pelo *garbage collector* por não possuir mais nenhuma referência ou referenciando.

Após a inserção do elemento 7 na fila, esta ficará com a configuração mostrada na figura 42.

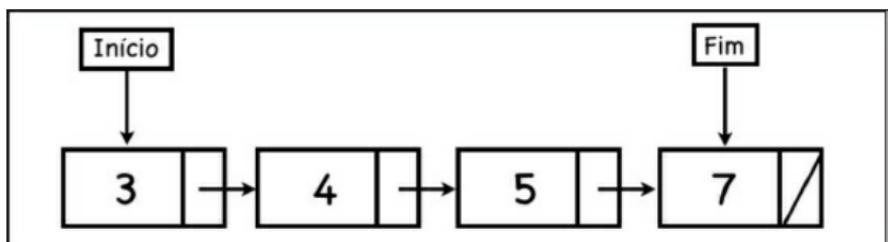


Figura 42 – Fila após a inserção do elemento 7.

Como em uma fila a inserção sempre ocorre no final, o elemento sete foi inserido como sucessor do elemento 5 que era o final da fila antes da inserção. Após esta operação, a referência que aponta o final da fila passou a referenciar o nodo que

contém o elemento 7.

A classe que implementa uma fila com alocação dinâmica de memória possui os métodos mostrados na tabela abaixo:

Método	Descrição
public boolean isEmpty()	Verifica se a fila está vazia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
public void enqueue(Object o)	Insere um elemento no fim da fila.
public Object dequeue()	Remove e retorna o objeto do início da fila.
public Object front()	Retorna (sem remover) uma referência ao objeto que se encontra no início da fila.
public int size()	Retorna o tamanho da fila.

Tabela 9 – Métodos implementados pela classe Queue.

O exemplo 26 mostra o código fonte da classe Queue que implementa uma fila com alocação dinâmica de memória.

```
1  public class Queue {  
2  
3      protected Node firstNode;  
4      protected Node lastNode;  
5  
6      public Queue(){  
7          firstNode = null;  
8          lastNode = null;  
9      }  
10  
11     public boolean isEmpty() {  
12         return firstNode == null;  
13     }  
14  
15     public Object front() throws UnderflowException {  
16         if (isEmpty()) {  
17             throw new UnderflowException();  
18         }  
19         return firstNode.getData();  
20     }  
21  
22     public void enqueue (Object insertItem) {  
23         if (isEmpty()) {  
24             firstNode = lastNode = new Node(insertItem);  
25         } else {  
26             lastNode.setNext(new Node(insertItem));  
27             lastNode = lastNode.getNext();  
28         }  
29     }  
30  
31     public Object dequeue() throws UnderflowException {  
32         if (isEmpty()) {  
33             throw new UnderflowException();  
34         }  
35         Object removedItem = firstNode.getData();  
36         if (firstNode == lastNode)  
37             firstNode = lastNode = null;  
38         Else  
39             firstNode = firstNode.getNext();  
40         return removedItem;  
41     }  
42 }
```

Exemplo 26 – Código fonte da classe Queue – implementação com alocação dinâmica.

A classe *Queue* mostrada no exemplo 26 define dois atributos do tipo *Node*: *firstNode* (linha 3) que é utilizado para marcar o início da fila e *lastNode* (linha 4), que é utilizado para marcar o fim da fila.

O construtor da classe *Queue*, definido nas linhas 6 a 9, atribui o valor *null* para os atributos *firstNode* e *lastNode*.

O método *isEmpty* (linhas 11 a 13) retorna *true* caso a fila esteja vazia, ou *false*

caso contrário. A fila é considerada vazia sempre que o atributo *firstNode* for igual a *null*.

O método *front* (linhas 15 a 20) retorna uma referência ao primeiro elemento da fila sem removê-lo. Este método lança uma exceção de *Underflow* caso a fila esteja vazia (linha 17).

O método *enqueue* (linhas 22 a 29) insere um novo elemento no final da fila. Ele recebe como parâmetro um objeto *e*, caso a fila esteja vazia, ambos os atributos, *firstNode* e *lastNode*, passam a referenciar um nodo contendo o elemento passado como parâmetro (linha 24). Caso a lista não esteja vazia, um novo nodo é criado como sucessor do último nodo da fila (linha 26) e então o atributo *lastNode* passa a referenciar este novo nodo (linha 27).

O método *dequeue* (linhas 31 a 41) remove e retorna um elemento do início da fila. Caso a fila esteja vazia, uma exceção é lançada (linha 33). Caso contrário, uma variável local armazena uma referência ao primeiro elemento da fila *e*, caso a fila contenha apenas um elemento, ambos os atributos da classe recebem o valor *null* (linha 37), fazendo com que a fila fique vazia. Caso a fila contenha mais de um elemento, o atributo *firstNode* passa a referenciar seu sucessor (linha 39), fazendo com que o primeiro elemento da fila seja removido.

O exemplo 27 mostra como a classe *Queue* pode ser utilizada em um programa Java.

```

1  public class QueueTest {
2
3      public static void main(String[] args) {
4
5          Queue queue = new Queue();
6
7          queue.enqueue( new Integer(1) );
8          queue.enqueue( new Integer(2) );
9          queue.enqueue( new Integer(3) );
10         queue.enqueue( new Integer(4) );
11
12         Object removedEl = null;
13
14         try {
15             while (!queue.isEmpty()) {
16                 removedEl = queue.dequeue();
17                 System.out.println(" "+ removedEl + " removido" );
18             }
19         } catch (UnderflowException e) {
20             System.out.println(e);
21         }
22     }
23 }
```

Exemplo 27 – Classe *StackTest*. Exemplo de utilização da classe *Queue* implementada com alocação dinâmica.

Neste exemplo um objeto do tipo *Queue* é instanciado na linha 5 e quatro elementos são inseridos na fila. Após última chamada ao método *enqueue* (linha 10), a fila *queue* estará com a seguinte configuração:

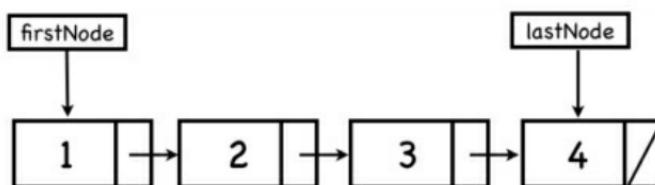


Figura 43 – Fila *queue* após a inserção dos quatro elementos.

Após as operações de inserção, todos os elementos da fila são removidos no laço *while* das linhas 15 a 18. Cada vez que um elemento é removido, este é mostrado na saída (linha 17). A saída do programa é mostrada na figura 44.

```

1 removido
2 removido
```

3 removido
4 removido

Figura 44 – Saída do exemplo 43.

Esta seção apresentou a estrutura de dados fila implementado com alocação dinâmica de memória. A implementação apresentada nesta seção apresenta duas vantagens quando comparada com a implementação mostrada no capítulo anterior. A primeira delas é que não existe a necessidade de especificar um tamanho máximo para a fila quando esta é instanciada, pois a memória é alocada e liberada dinamicamente quando os elementos são inseridos e removidos. Outra vantagem é que, devido à alocação dinâmica de memória, não há o problema de a fila parecer cheia quando na verdade está vazia.

Lista duplamente encadeada

Esta seção apresenta a estrutura de lista duplamente encadeada que é uma variação da estrutura de lista simplesmente encadeada apresentada nas seções anteriores. Nesta estrutura cada nodo da lista possui duas referências: uma ao seu sucessor na lista e outra ao seu antecessor.

Um problema com as listas simplesmente encadeadas é que a travessia delas é facilitada em apenas um sentido devido à estrutura dos nodos que possuem apenas uma referência ao seu sucessor. Toda a vez que um elemento precisa ser removido, é necessário encontrar seu antecessor para realizar a remoção e, como os nodos da lista possuem apenas uma referência ao seu sucessor, esta operação envolve operações adicionais, como, por exemplo, no método *removeFromBack* do exemplo 22.

A classe *Node* mostrada abaixo é uma variação da classe *Node* mostrada no exemplo 21. Esta classe, além de possuir uma referência ao nodo sucessor, possui também uma referência ao nodo antecessor na lista, como mostra a figura 45.

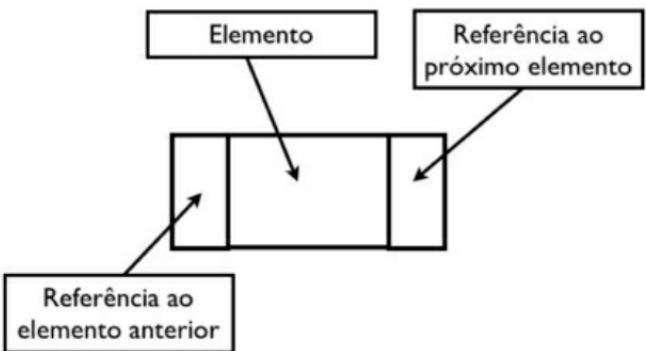


Figura 45 – Estrutura de um nodo de uma lista duplamente encadeada.

O código fonte da classe que implementa um nodo de uma lista duplamente encadeada é mostrado no exemplo 28.

```
1  public class Node {  
2  
3      private Object data;  
4      private Node nextNode;  
5      private Node previousNode;  
6  
7      public Node( Object object ) {  
8          this( object, null );  
9      }  
10  
11     public Node( Object object, Node node ) {  
12         data = object;  
13         nextNode = node;  
14     }  
15  
16     public Object getData() { return data; }  
17  
18     public void setData (Object element){  
19         data = element;  
20     }  
21  
22     public Node getNext() { return nextNode; }  
23  
24     public void setNext(Node o) {  
25         nextNode = o;  
26     }  
27  
28     public Node getPrevious() { return previousNode; }  
29  
30     public void setPrevious(Node o) {  
31         previousNode = o;  
32     }  
33 }
```

Exemplo 28 – Classe *Node*. Utilizada para armazenar elementos de uma lista duplamente encadeada.

A classe *Node* mostrada acima define três atributos: *data* do tipo *Object*, que é utilizado para armazenar o elemento dentro no nodo da lista, e dois atributos do tipo *Node*, *nextNode* e *previousNode*, que são utilizados para referenciar o nodo sucessor e antecessor respectivamente.

A classe *Node* define também dois construtores: um que recebe uma referência ao um objeto do tipo *Object*, que é o dado a ser armazenado dentro do nodo (linhas 7 a 8), e outro que recebe o dado a ser armazenado e uma referência ao seu nodo sucessor (linhas 11 a 14).

O restante da classe *Node* contém os métodos *getter* e *setter* para os atributos definidos. Esta classe difere da classe *Node* apresentada no exemplo 21, pois esta define uma referência adicional (*previousNode*) que é utilizada para referenciar o nodo

antecessor da lista.

Assim como uma lista simplesmente encadeada, uma lista duplamente encadeada também pode ser vista como uma sequência de nodos, como mostra a figura 46.

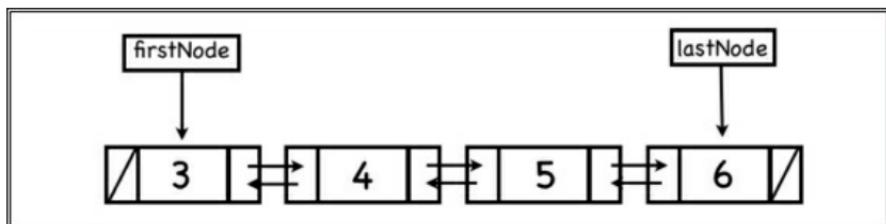


Figura 46 – Lista duplamente encadeada.

Do ponto de vista de implementação, os métodos da classe que implementa uma lista duplamente encadeada são os mesmos métodos da classe que implementa uma lista simplesmente encadeada. O que muda é a implementação dos métodos, que agora tem que lidar com uma referência adicional.

Método	Descrição
public boolean isEmpty()	Verifica se a lista está vazia. Retorna <i>true</i> caso ela esteja, ou <i>false</i> caso contrário.
public void insertAtFront(Object o)	Insere um elemento na primeira posição da lista.
public void insertAtBack(Object o)	Insere um elemento na última posição da lista.
public Object removeFromFront()	Remove e retorna o objeto contido na primeira posição da lista.
public Object removeFromBack()	Remove e retorna o objeto contido na última posição da lista.
public Node getFirst()	Retorna uma referência ao primeiro nodo da lista.
public Node getLast()	Retorna uma referência ao último nodo da lista.

Tabela 10 – Métodos da classe de lista duplamente encadeada.

Comparando esta lista com a lista simplesmente encadeada apresentada anteriormente, cada nodo desta lista ocupa um espaço adicional relativo à referência ao nodo antecessor. Os algoritmos para inserção e remoção também possuem uma complexidade adicional, pois mais uma referência deve ser levada em conta na realização destas operações. Em contrapartida, esta abordagem facilita certas operações, uma vez que a lista pode ser atravessada em ambos os sentidos.

```
1  public class List {
2
3      private Node firstNode;
4      private Node lastNode;
5
6      public List() {
7          firstNode = null;
8          lastNode = null;
9      }
10
11     public Node getFirst(){
12         return firstNode;
13     }
14
15     public Node getLast(){
16         return lastNode;
17     }
18
19     public boolean isEmpty() {
20         return firstNode == null;
21     }
22
23     public void insertAtFront( Object insertItem ) {
24         Node n = new Node( insertItem );
25         if ( isEmpty() )
26             firstNode = lastNode = n;
27         else {
28             firstNode.setPrevious (n);
29             n.setNext(firstNode);
30             firstNode = n;
31         }
32     }
33
34     public void insertAtBack( Object insertItem ) {
35         Node n = new Node( insertItem );
36         if ( isEmpty() )
37             firstNode = lastNode = n;
38
39         else {
40             lastNode.setNext (n);
41             n.setPrevious(lastNode);
42             lastNode = n;
43         }
44     }
45
46     public Object removeFromFront() throws UnderflowException {
47         if ( isEmpty() )
48             throw new UnderflowException();
49
50         Object removedItem = firstNode.getData();
51
52         if ( firstNode == lastNode )
53             firstNode = lastNode = null;
54         else {
55             firstNode = firstNode.getNext();
56             firstNode.setPrevious(null);
57         }
58         return removedItem;
59     }
60
61     public Object removeFromBack() throws UnderflowException {
62         if ( isEmpty() )
63             throw new UnderflowException();
64
65         Object removedItem = lastNode.getData();
66
67         if ( firstNode == lastNode )
68             firstNode = lastNode = null;
69         else {
70             Node beforeLast = lastNode.getPrevious();
71             lastNode = beforeLast;
72             lastNode.setNext(null);
73         }
74         return removedItem;
75     }
76 }
```

Exemplo 29 – Código fonte da classe que implementa uma lista duplamente encadeada.

A classe *List* mostrada no exemplo 29 define dois atributos do tipo *Node*, *firstNode* e *lastNode*. Estes atributos são utilizados para referenciar o início e o fim da lista respectivamente.

O construtor da classe *List* (linhas 6 a 9) inicializa os atributos *firstNode* e *lastNode* com o valor *null*.

Os métodos *getFirst* e *getLast* retornam uma referência ao primeiro e ao último nodo da lista respectivamente.

O método *isEmpty* definido na linha 19 retorna *true* sempre que a lista estiver vazia. A lista é considerada vazia sempre que o atributo *firstNode* for igual a *null*.

O método *insertAtFront* (linhas 23 a 32) insere um novo elemento no inicio da lista. Ele recebe como parâmetro o objeto a ser inserido na lista. Na linha 24 um novo nodo é instanciado para armazenar o objeto recebido como parâmetro. Caso a lista esteja vazia, ambos os tributos (*firstNode* e *lastNode*) passam a referenciar no novo nodo. Caso a lista não esteja vazia, a referência ao antecessor do nodo referenciado por *firstNode* passa a referenciar o novo nodo (linha 28), a referência ao sucessor do novo nodo passa a referenciar *firstNode* (linha 29) e então *firstNode* passa a referenciar o novo nodo (linha 30).

O método *insertAtBack* (linhas 34 a 44) insere um novo elemento no final da lista. Ele recebe como parâmetro o objeto a ser inserido na lista e instancia um novo nodo para armazenar este elemento (linha 35). Caso a lista esteja vazia, ambos os atributos *firstNode* e *lastNode* passam a referenciar o novo nodo (linha 37). Caso a lista não esteja vazia, a referência ao sucessor do atributo *lastNode* passa a referenciar o novo nodo (linha 40), a referência ao antecessor do novo nodo passa a referenciar o atributo *lastNode* (linha 41) e, por fim, o atributo *lastNode* passa a referenciar o novo nodo (linha 42).

O método *removeFromFront* (linhas 46 a 59) remove e retorna o elemento armazenado na primeira posição da lista. Caso a lista esteja vazia, este método lança uma exceção de *Underflow* (linha 63). Uma referência ao objeto armazenado no último nodo da lista é armazenada na variável local *removedItem* (linha 65). Um teste é realizado na linha 67 para verificar se a lista possui apenas um elemento. Neste caso, os atributos *firstNode* e *lastNode* estarão referenciando mesmo nodo da lista e então o valor *null* é atribuído a ambos (linha 68). Caso a lista contenha mais de um elemento, o atributo *firstNode* passa a referenciar seu sucessor (linha 55) e a referência ao seu antecessor recebe o valor *null*, removendo assim o primeiro elemento da lista. Na linha

58 a variável *removedItem* é retornada.

O método *removeFromBack* (linhas 61 a 75) remove e retorna o elemento armazenado na última posição da lista. Uma exceção de *Underflow* é lançada caso a lista esteja vazia (linha 63). Uma referência ao objeto armazenado na última posição da lista é armazenada na variável local *removedItem* na linha 65. Caso a lista contenha apenas um elemento, ambos os atributos, *firstNode* e *lastNode* recebem o valor *null* (linha 68). Caso a lista contenha mais de um elemento, uma referência ao penúltimo nodo da lista é armazenada na variável local *beforeLast* (linha 70), o atributo *lastNode* passa a referenciar seu antecessor (linha 71) e então o sucessor do atributo *lastNode* recebe o valor *null*, removendo assim o último elemento da lista. Por fim o elemento que estava armazenado no último nodo da lista é retornado.

O exemplo 30 mostra como a classe *List* pode ser utilizada em um programa Java.

```
1  public class ListTest {
2      public static void main( String args[] ) {
3
4          List list = new List();
5
6          list.insertAtBack(new Integer(1));
7          list.insertAtFront(new Integer(2));
8          list.insertAtBack(new Integer(3));
9          list.insertAtFront(new Integer(4));
10
11         try {
12             System.out.println("Elemento removido: " + list.removeFromBack());
13             System.out.println("Elemento removido: " + list.removeFromFront());
14         } catch (UnderflowException e) {
15             System.out.println(e);
16         }
17     }
18 }
```

Exemplo 30 – Uso da classe *List* em um programa Java.

Neste exemplo um objeto *List* é declarado e instanciado (linha 4) e então quatro objetos do tipo *Integer* são inseridos na lista. Após a inserção dos quatro elementos, a lista estará com a configuração mostrada na figura 47.

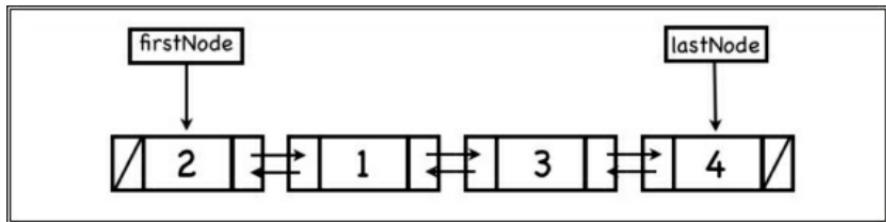


Figura 47 – Conteúdo da lista *list* após as inserções.

Após a inserção dos quatro elementos, o primeiro e o último elemento da lista são removidos nas linhas 12 e 13. Após estas remoções, a lista fica com a configuração mostrada na figura 48.

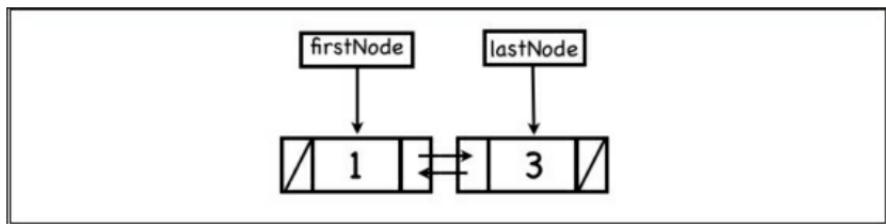


Figura 48 – Conteúdo da lista *list* após as duas remoções.

Esta seção apresentou a estrutura de lista duplamente encadeada que difere da estrutura de lista simplesmente encadeada por cada nodo possuir, além de uma referência ao seu sucessor, também uma referência ao seu antecessor na lista. Esta abordagem faz com que cada nodo da lista ocupe um espaço extra devido à referência adicional, mas em contrapartida facilita a travessia da lista em ambos os sentidos. A introdução de uma referência adicional também torna um pouco mais complicadas as operações de inserção e remoção da lista, uma vez que mais referências devem ser acertadas em cada inserção ou remoção.

Este capítulo apresentou as estruturas de lista linear geral, pilha e fila utilizando alocação dinâmica de memória. As *interfaces* expostas pelas classes apresentadas neste capítulo são semelhantes às *interfaces* expostas pelas classes de lista, pilha e fila apresentadas no capítulo anterior. A principal diferença está na maneira como a memória é manipulada. Nas estruturas apresentadas neste capítulo a memória é alocada e liberada sob demanda a medida em que elementos são inseridos e removidos das estruturas. Nesta abordagem os elementos não se encontram em posições contíguas de memória, mas sim espalhados e interligados por referências. Outra

diferença é que, devido a este esquema de manipulação de memória, não existe a necessidade de especificar um tamanho máximo no momento da instanciação como acontece com as estruturas apresentadas no capítulo anterior.

CAPÍTULO 5

RECURSÃO

Este capítulo introduz o conceito de recursão. Embora este conceito não seja uma estrutura de dados propriamente dita, seu entendimento é bastante importante, pois diversas estruturas de dados estudadas nos próximos capítulos utilizam este conceito nos algoritmos que as manipulam. Além disso, muitos problemas da computação podem ser resolvidos de uma maneira bastante elegante através da utilização de recursão.

A recursão é uma técnica de programação na qual uma função (método) chama a si mesma. Esta técnica também é chamada de dividir para conquistar, uma vez que um problema maior, através da utilização de recursão, pode ser dividido em subproblemas menores.

Para entender melhor o conceito de recursão, considere, por exemplo, a operação de multiplicação entre dois números. Esta operação pode ser definida em termos da operação de adição. Em outras palavras, multiplicar m por n é equivalente a somar m n vezes. Ex.: $3*4$ pode ser calculado somando-se 3 quatro vezes: $3 + 3 + 3 + 3$.

A implementação de um método que realiza a multiplicação em termos da adição pode ser realizada tanto de maneira iterativa quanto de maneira recursiva. O exemplo 31 mostra como este método pode ser implementado de maneira iterativa.

```
1 public static int mult (int m, int n){  
2     int resultado = 0;  
3     for (int i = 1; i <= n; i++ ){  
4         resultado = resultado + m;  
5     }  
6     return resultado;  
7 }
```

Exemplo 31 – Multiplicação implementada de maneira iterativa.

Este mesmo método implementado de maneira recursiva é mostrado no exemplo 32.

```
1  public static int mult(int m, int n){  
2      if (n == 0){  
3          return 0;  
4      }  
5      else{  
6          return m + mult(m, n-1);  
7      }  
8  }
```

Exemplo 32 – Multiplicação implementada de maneira recursiva.

Pode parecer estranho à primeira vista um método que chama a si mesmo, mas este conceito é utilizado em diversas linguagens de programação, não sendo específico à linguagem Java.

O método mult recursivo do exemplo 32 realiza um teste na linha 2 para verificar se o parâmetro n é igual a zero. Se este caso for verdadeiro, o método retorna o valor zero. Caso n seja maior que zero, o método soma o parâmetro m com o retorno da chamada a si mesmo, passando m e n-1 como parâmetro. Note que a cada chamada do método mult o parâmetro n é decrementado em um. Isso implica que, em algum momento, o parâmetro n passado será igual a zero e o método irá parar de chamar a si mesmo.

Embora esta implementação mostrada no exemplo 32 seja bastante simples, ela demonstra bem os dois principais conceitos envolvidos na implementação de um método recursivo, que são:

- ◆ Determinação de um caso básico
 - É o caso onde definimos o menor de todos os subproblemas, que é quando o método irá parar de chamar a si mesmo.
- ◆ Divisão de um problema maior em problemas menores
 - É como o problema maior é dividido em subproblemas menores.

No caso do exemplo 32, o caso básico é testado na linha 2. Quando n for igual a zero, o método para de chamar a si mesmo. A divisão do problema em subproblemas menores ocorre na chamada recursiva na linha seis. A cada nova chamada o número de termos a serem somados diminui, levando o problema em direção ao caso básico.

Outra função simples de ser implementada de maneira recursiva é a função que determina o fatorial de um número. O fatorial de um número n é definido como $n!$ e é calculado através do produto de todos os inteiros positivos menores ou iguais a n. Desta maneira, o fatorial de 4 é definido como $4!$ e é calculado da seguinte maneira: $4!$

$$= 4 * 3 * 2 * 1.$$

A definição recursiva do fatorial de um número é mostrada abaixo:

$$N! = \begin{cases} 1 & \text{se } N = 0; \\ N * (N - 1)! & \text{se } N > 0 \end{cases}$$

Figura 49 – Definição do fatorial de um número inteiro.

Utilizando a linguagem Java, um método que calcula o fatorial de um número inteiro N positivo pode ser implementado tanto de maneira iterativa, através de um laço de repetição, quanto de maneira recursiva. O exemplo 33 mostra a implementação do método que calcula o fatorial de maneira iterativa.

```
1 public static int fatorial(int n){  
2     int resultado = 1;  
3     for (int i = n; i > 0; i--){  
4         resultado = resultado * i;  
5     }  
6     return resultado;  
7 }
```

Exemplo 33 – Método que calcula o fatorial de maneira iterativa.

Este mesmo método implementado de maneira recursiva é mostrado no exemplo 34.

```
1 public static int fatorial(int n){  
2     if (n <= 1){  
3         return 1;  
4     }  
5     else{  
6         return n * fatorial(n-1);  
7     }  
8 }
```

Exemplo 34 – Método que calcula o fatorial de maneira recursiva.

O método fatorial do exemplo 34 realiza o mesmo cálculo do método fatorial mostrado no exemplo 33. Este método define um caso básico na linha 2, onde é

realizado um teste para verificar se o parâmetro n é menor ou igual a um. Caso este teste retorne verdadeiro, o método para de chamar a si mesmo e retorna um. É esta condição que faz com que o método não fique chamando a si mesmo infinitamente. Caso o teste da linha dois retorne falso, o método continua chamando a si mesmo (linha 6) e o retorno da chamada é multiplicado com o valor de n. Note que, novamente, a cada chamada do método o valor de n é decrementado. Isso faz com que, a cada nova chamada, n esteja cada vez mais próximo de um, ou seja, da condição básica testada na linha 2.

Quando chamado com n igual a cinco, o método fatorial do exemplo 34 retorna o valor 120, que é $5 * 4 * 3 * 2 * 1$.

Uma questão-chave no entendimento do conceito de recursão é saber como o ambiente de execução lida com as chamadas de métodos, como os parâmetros são tratados e como o valor de retorno é retornado ao método chamador, entre outras coisas. Todas estas questões são tratadas na próxima seção.

Pilha de chamadas

De modo a entender como o ambiente de execução lida com as chamadas de métodos, considere o exemplo 35.

```
1 public class PilhaDeChamada {  
2  
3     public static void A(int x){  
4         int y = 1;  
5         System.out.println("Valor de x: " + x + " e de y: " + y);  
6         B(x + 1);  
7     }  
8  
9     public static void B(int x){  
10        int y = 2;  
11        System.out.println("Valor de x: " + x + " e de y: " + y);  
12        C(x + 1);  
13    }  
14  
15    public static void C(int x){  
16        int y = 3;  
17        System.out.println("Valor de x: " + x + " e de y: " + y);  
18    }  
19  
20    public static void main(String[] args) {  
21        int local = 5;  
22        A(local);  
23    }  
24 }
```

Exemplo 35 – Classe PilhaDeChamada – Demonstra o funcionamento da pilha de chamada de métodos.

Neste exemplo a classe PilhaDeChamada define três métodos: A, B e C. Todos os métodos recebem como parâmetro formal um inteiro x e todos eles definem também uma variável local chamada y. No método main da classe PilhaDeChamada é definida uma variável local inicializada com o valor 5, que é passada para o método A como parâmetro real na linha 22.

Cada vez que um método é chamado, o ambiente de execução realiza os seguintes passos, entre outras coisas:

- ▶ Reserva de memória para os parâmetros reais e para as variáveis locais. Os parâmetros formais são, na verdade, considerados variáveis locais ao método sendo chamado.
- ▶ Inicialização dos parâmetros formais com os valores definidos pelos parâmetros reais.
- ▶ Armazenamento do endereço de retorno do método. Este endereço é utilizado visto que o ambiente de execução precisa saber qual o próximo comando a ser executado após o término do método

Todos estes passos são realizados com o auxílio de uma estrutura chamada

registro de ativação. Para cada método chamado em um programa Java, incluindo o método main, um registro de ativação é criado. É no registro de ativação que ficam armazenadas as variáveis locais, parâmetros formais, valor de retorno e o endereço de retorno, entre outras coisas. Ainda, sempre que um método chama outro método, como no caso do exemplo 35, o registro de ativação do método sendo chamado é “empilhado” em cima do registro de ativação do método chamador. Da mesma maneira, sempre que um método termina sua execução, seu registro de ativação é “desempilhado” e a memória ocupada por suas variáveis locais e parâmetros é liberada. A estrutura em alto nível de um registro de ativação é mostrada na figura 50.

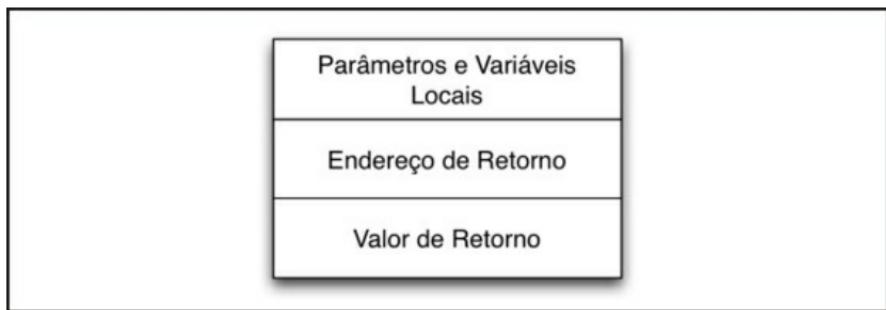


Figura 50 – Estrutura de um registro de ativação.

Voltando ao exemplo 35, quando o método C estiver executando, a pilha de registros de ativação estará como mostra a figura 51. Nesta figura podemos ver que o primeiro registro de ativação (na base da pilha de chamadas) é o registro do método main. Quando o método main chama o método A, o segundo registro de ativação é criado e empilhado em cima do registro de ativação do método main. O mesmo acontece quando o método A chama o método B e quando o método B chama o método C. Cada um dos registros de ativação mantém uma cópia do parâmetro formal x e cada um dos registros de ativação mantém também uma cópia da variável local y. Da mesma maneira que um registro é empilhado, ele também é desempilhado quando o método termina sua execução. Para saber qual o próximo comando a ser executado após a execução de um método, o ambiente de execução utiliza o endereço de retorno armazenado no registro de ativação.

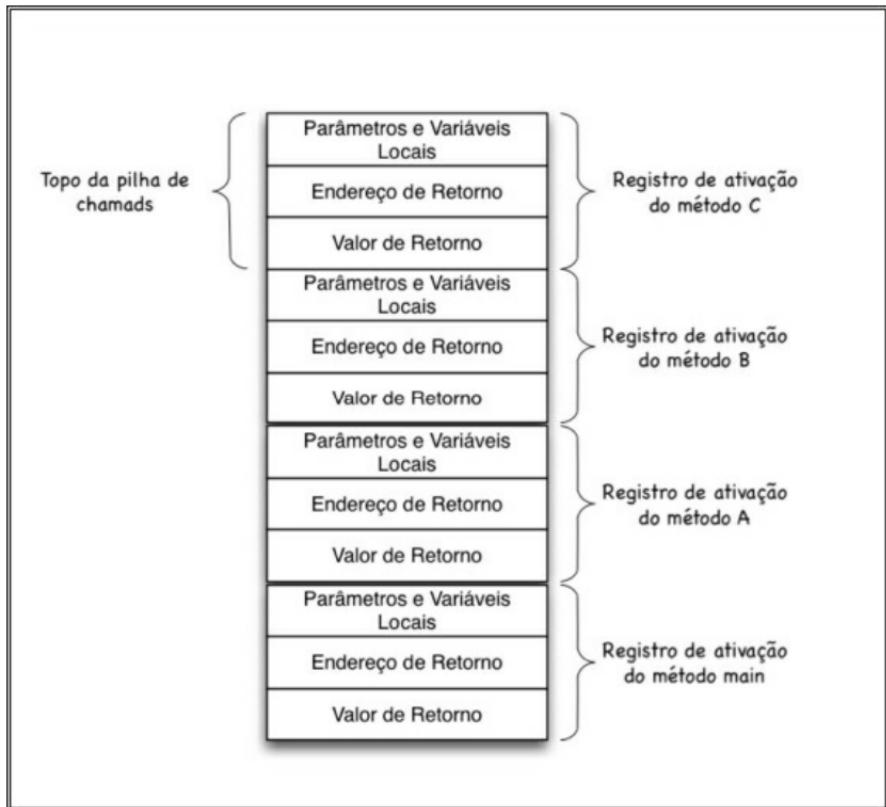


Figura 51 – Pilha de chamadas durante a execução do método C.

Voltando ao exemplo do factorial implementado de maneira recursiva, cada vez que uma nova chamada ao método factorial é feita, um novo registro de ativação é criado e, para cada um destes registros, uma nova cópia do parâmetro formal n é criada.

As chamadas realizadas por um método recursivo também podem ser mostradas como na figura 52. Esta figura mostra a sequência de chamadas realizadas pelo método factorial do exemplo 34, quando este foi chamado com o parâmetro 5 no método main do exemplo 36. Ao final da execução, o método factorial terá sido chamado cinco vezes e o valor 120 é armazenado na variável fat e o valor é mostrado na tela.

```

1  public static void main(String[] args) {
2      int fat = fatorial(5);
3      System.out.println("5! = " + fat);
4  }

```

Exemplo 36 – Chamada ao método fatorial com parâmetro 5.

Cada vez que uma nova chamada é feita ao método fatorial, o valor de n é decrementado, até o momento em que ele fica com o valor 1, quando o método para de chamar a si mesmo. As linhas pontilhadas da figura 52 mostram como o valor de retorno do método fatorial é utilizado no cálculo.

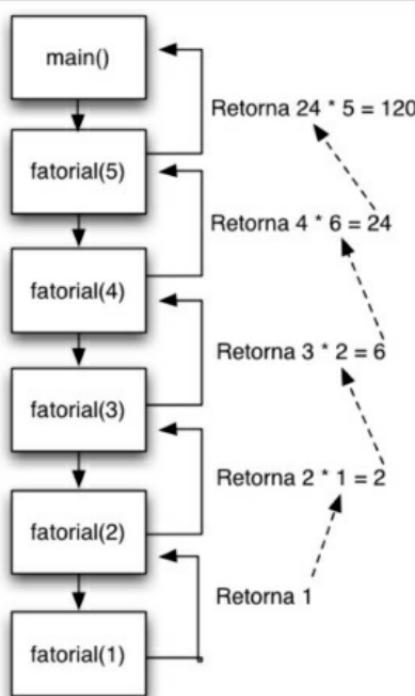


Figura 52 – Chamadas executadas no cálculo recursivo do fatorial de 5.

Outro problema que pode ser resolvido de maneira recursiva é a série de Fibonacci. Esta série inicia com os valores zero e um, e o valor dos termos subsequentes é obtido

através da soma dos dois termos anteriores. Os oito primeiros termos da série de Fibonacci são os seguintes:

0, 1, 1, 2, 3, 5, 8 e 13

Eles são calculados da seguinte maneira:

0+1=1, 1+1=2, 2+1=3, 3+2=5, 5+3=8, 8+5=13

A definição recursiva da série de Fibonacci é mostrada abaixo:

$$\text{Fib}(n) = \begin{cases} 0 & \text{se } n = 0; \\ 1 & \text{se } n = 1; \\ \text{Fib}(n - 2) + \text{Fib}(n - 1) & \text{outros casos} \end{cases}$$

Figura 53 – Definição matemática da série de Fibonacci.

Observe que existem dois casos básicos na definição: quando n é igual a zero e quando n é igual a um. Esta definição pode ser implementada de maneira recursiva conforme mostra o exemplo 37.

```
1 public static int fibonacci(int n){  
2     if ((n == 0) || (n == 1)){  
3         return n;  
4     }  
5     else return fibonacci(n-2) + fibonacci(n-1);  
6 }
```

Exemplo 37 – Método recursivo que calcula a série de Fibonacci.

Neste método ambos os casos básicos da definição são testados na linha 2. Quando n é maior que um, o método chama a si mesmo duas vezes, uma passando $n-2$ e outra passando $n-1$. A figura abaixo demonstra graficamente a sequência de chamadas quando o método é chamado com o valor três.

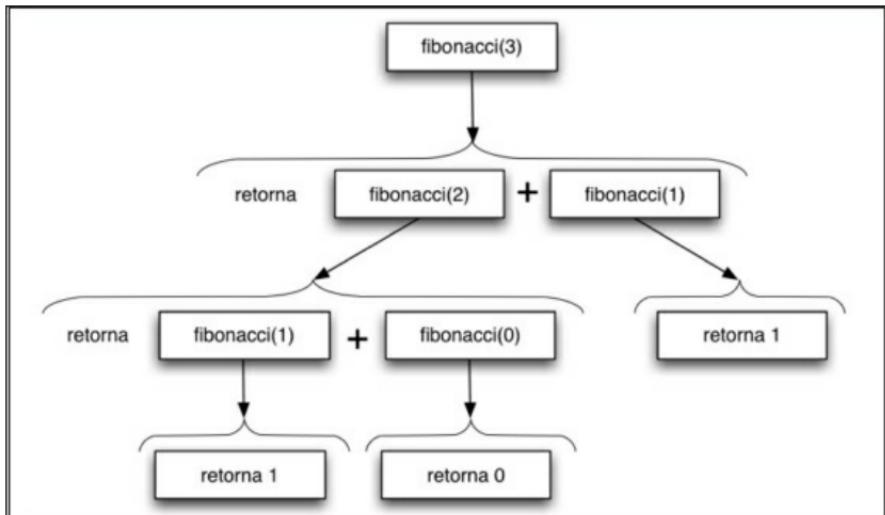


Figura 54 – Conjunto de chamadas recursivas para Fibonacci(3).

Observe que na parte inferior da figura restam apenas os valores um, zero e um. Estes valores são o resultado da execução dos casos básicos. Os primeiros dois valores de retorno (da esquerda para a direita), um e zero são retornados como valores de retorno das chamadas de `fibonacci(1)` e `fibonacci(0)`. A soma um mais zero é retornada como o valor de retorno de `fibonacci(2)`. Este valor então é então adicionado ao valor de retorno da chamada `fibonacci(1)`, produzindo o valor dois. Este valor então é retornado como valor de `fibonacci(3)`.

A estrutura do método que calcula a série de Fibonacci é semelhante à estrutura dos outros métodos recursivos tratados nesta seção, pois eles apresentam as seguintes características:

- ◆ Apresentam pelo menos um caso básico;
- ◆ Chamam a si mesmo para resolver parte do problema.

Na verdade estas características são comuns a todos os métodos recursivos. Na implementação de qualquer método recursivo, deve-se montar uma definição (especificação) do problema de maneira recursiva da seguinte maneira:

1. Definir pelo menos um caso básico;
2. Quebrar o problema maior em subproblemas, definindo o(s) caso(s) recursivo(s);

3. Realizar o teste de finitude, isto é, certificar-se de que as chamadas recursivas levam obrigatoriamente, e numa quantidade finita de vezes, ao(s) caso(s) básico(s).

Uma vez que a especificação esteja definida, basta traduzi-la para a linguagem de programação. Outro ponto que vale ser notado é que todo e qualquer método que é implementado de maneira recursiva pode também ser implementado de maneira iterativa com o uso de algum comando de repetição. Existem diferentes técnicas para a eliminação da recursão, as quais podem ser utilizadas para escrever um método recursivo de maneira iterativa.

Para ambas as abordagens (iterativa ou recursiva) existem vantagens e desvantagens. Dentre as vantagens da recursão destacam-se:

- ◆ Redução do tamanho do código fonte.
 - ◆ Todos os exemplos vistos neste capítulo apresentam um código mais compacto quando comparado com suas versões iterativas.;
- ◆ Maior clareza do algoritmo para problemas de definição naturalmente recursiva.

Dentre as desvantagens destacam-se:

- ◆ Baixo desempenho na execução devido ao gerenciamento das chamadas;
- ◆ Maior dificuldade de depuração, principalmente se a recursão for muito profunda.

Este capítulo apresentou o conceito de recursão, que é uma técnica de resolver um problema dividindo-o em problemas menores para encontrar a solução. Esta técnica é suportada por várias linguagens de programação além da linguagem Java. O entendimento deste conceito é bastante importante, pois diferentes algoritmos estudados nos próximos capítulos apresentam definições naturalmente recursiva.

CAPÍTULO 6

ÁRVORES

As estruturas de dados estudadas até então são estruturas lineares, no sentido que em todas elas existe a notação de antes e depois. Em uma lista encadeada, por exemplo, um elemento pode ter um sucessor e um antecessor. O mesmo é válido para listas sequenciais e também para vetores. Em todas estas estruturas um elemento pode estar antes ou depois de outro elemento, estabelecendo assim uma relação de linearidade.

Neste capítulo estudaremos uma das estruturas mais utilizadas na computação: a estrutura de árvore. Esta estrutura não é linear como as estruturas que estudamos até o momento, mas sim hierárquica, pois o relacionamento entre seus elementos é definido por uma hierarquia, lembrando assim os galhos de uma árvore botânica.

Árvores são normalmente esquematizadas graficamente como mostra a figura 55, a qual representa o organograma de uma empresa.

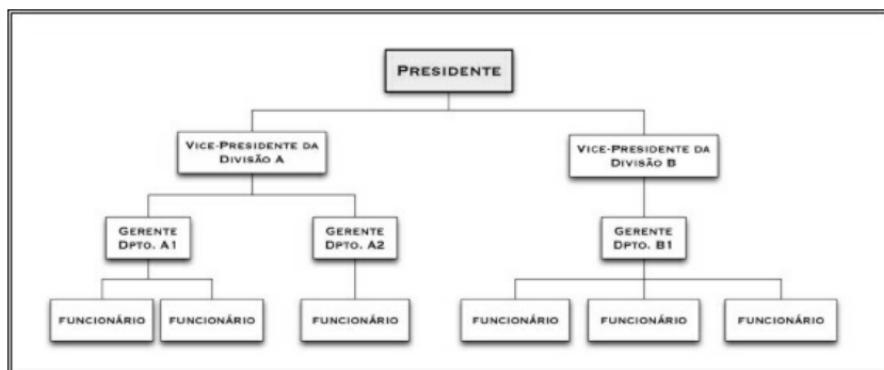


Figura 55 – Representação de hierarquia através de uma árvore.

O presidente da empresa está no topo da árvore e os vice-presidentes estão logo abaixo. Abaixo dos vice-presidentes estão os gerentes e abaixo dos gerentes estão os outros funcionários. Nesta hierarquia cada funcionário se reporta a um gerente, cada gerente se reporta a um vice-presidente e cada vice-presidente se reporta ao presidente. Embora esta árvore esteja de cabeça para baixo (com a raiz no topo), esta é

a maneira mais comum de representar este tipo de estrutura. No exemplo da figura 55 o presidente é chamado de raiz da árvore e os funcionários são chamados de folhas.

Uma árvore é extremamente útil para certos tipos de computação. Ainda considerando a árvore da figura 55, suponha que queiramos determinar o total de salário pago aos empregados, por divisão ou departamento. O total de salários pagos na divisão A pode ser calculado somando-se os salários pagos nos departamentos A1 e A2 mais o salário do vice presidente da divisão A. O total de salários do departamento A1, por sua vez, pode ser calculado somando-se o salário dos dois funcionários deste departamento mais o salário do gerente. Uma implementação desta computação deve visitar sistematicamente cada um dos elementos da árvore de modo a determinar a soma correta dos salários. Um algoritmo que realiza esta visita sistemática é chamado de caminhamento. Neste capítulo, além de estudarmos diferentes tipos de árvores, estudaremos também diferentes algoritmos de caminhamento.

Conforme mostrado na figura 55, em uma árvore alguns elementos estão “acima” e outros “abaixo” de outros, definindo assim uma hierarquia. Na verdade a principal terminologia das estruturas de árvore vem das árvores genealógicas, sendo os termos “pai”, “filho”, “ancestral” e “descendente” os mais utilizados para descrever os relacionamentos.

Neste sentido, uma árvore pode ser vista como um tipo abstrato de dados que armazena elementos de maneira hierárquica. Com exceção do elemento do topo, cada elemento da árvore possui um elemento pai e zero ou mais elementos filhos. Uma árvore é normalmente desenhada como mostrado na figura 55. O elemento mais ao topo da árvore (Presidente) é chamado de raiz da árvore e é sempre desenhado mais ao topo com os outros elementos conectados abaixo (justamente ao contrário de uma árvore real).

Outro exemplo de representação hierarquia por árvore é a hierarquia de classes da API da linguagem Java. Na API da linguagem Java todas as classes são subclasses da superclasse Object, direta ou indiretamente. Desta maneira a classe Object é a raiz da árvore e todas as outras classes são suas descendentes, como mostra a figura 56.

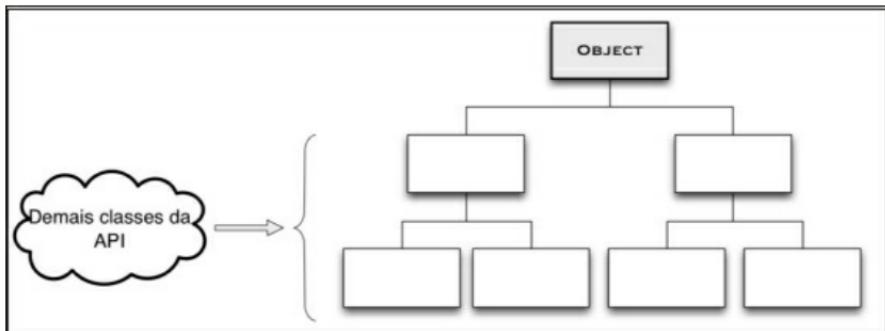


Figura 56 – Hierarquia de classes da API da linguagem Java.

Árvores também podem ser utilizadas para representar composição como, por exemplo, um país que é composto por estados ou um livro que é composto por capítulos. A figura 57 mostra uma árvore representando o Brasil com alguns de seus estados. Na raiz da árvore temos o país, logo abaixo temos os estados e, abaixo dos estados, temos as cidades. Olhando para esta árvore, pode-se verificar que as cidades de Porto Alegre e Canoas estão contidas no estado do Rio Grande do Sul. Da mesma maneira, as cidades de Curitiba e Londrina estão contidas no estado do Paraná.

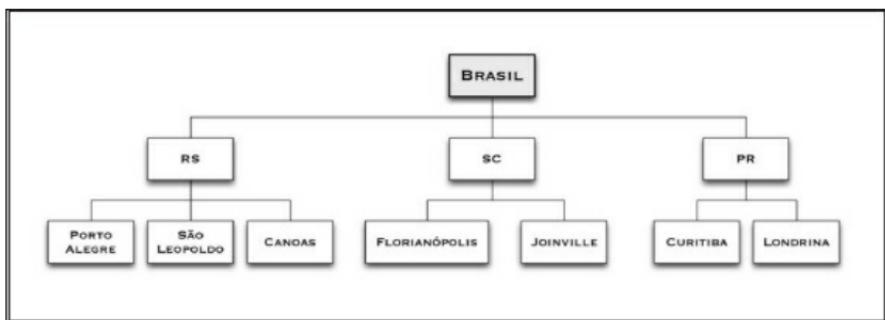


Figura 57 – Árvore utilizada para representar composição.

Outra utilização da estrutura de árvores acontece quando compilamos um programa Java. Toda vez que um programa Java é compilado, o compilador constrói uma árvore que representa o programa. Sempre que o compilador consegue construir esta árvore, significa que o programa está sintaticamente correto. Quando o programa contém um erro de sintaxe, o compilador não consegue montar corretamente esta árvore. Tomando com exemplo a seguinte expressão: $(a * b) + (c / (d + e))$, o

compilador constrói uma árvore como a mostrada na figura 58.

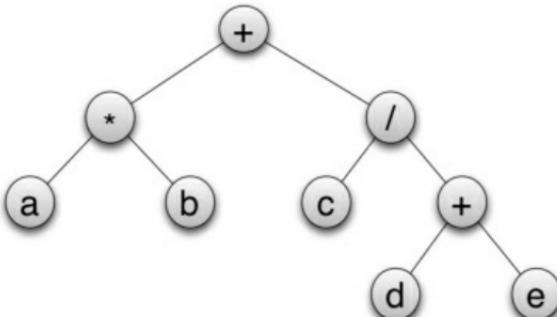


Figura 58 – Árvore de derivação para a expressão $(a * b) + (c / (d + e))$.

Além dos exemplos acima citados, árvores são utilizadas extensamente na computação, como, por exemplo, em sistemas de arquivos, em que temos um diretório raiz que pode conter outros arquivos e também outros diretórios. *Interfaces* gráficas de usuários utilizam árvores para representar hierarquias de menu. A organização das páginas de um *site* também pode ser representada por uma árvore.

A próxima seção introduz o conceito de árvore de uma maneira mais formal juntamente com a terminologia relativa a árvores utilizada no restante do livro.

Definição e terminologia

Formalmente, uma árvore T pode ser definida como um conjunto de nodos que armazenam elementos em um relacionamento *pai-filho* com as seguintes propriedades:

- Se T é não vazia, ela tem um nodo especial chamado raiz de T que não possui pai;
- Cada nodo v de T diferente da raiz tem um único nodo pai, w ; todo nodo com pai w é filho de w .

Observe que, por esta definição, uma árvore pode ser vazia. Isso significa que ela não possui nenhum nodo. Esta convenção permite que uma árvore possa ser definida recursivamente de maneira que uma árvore T ou está vazia, ou consiste em um nodo r ,

chamado raiz de T e um conjunto (possivelmente vazio) de árvores cuja raízes são filhas de r .

A partir desta definição, como mostra a figura 59, a raiz da árvore é o nodo A, que tem como nodos filhos os nodos B, C e D que, por sua vez, são também raízes de três subárvores distintas que também são consideradas árvores. Neste sentido, o nodo K também pode ser considerado raiz de uma árvore que, neste caso, não contém nenhum filho.

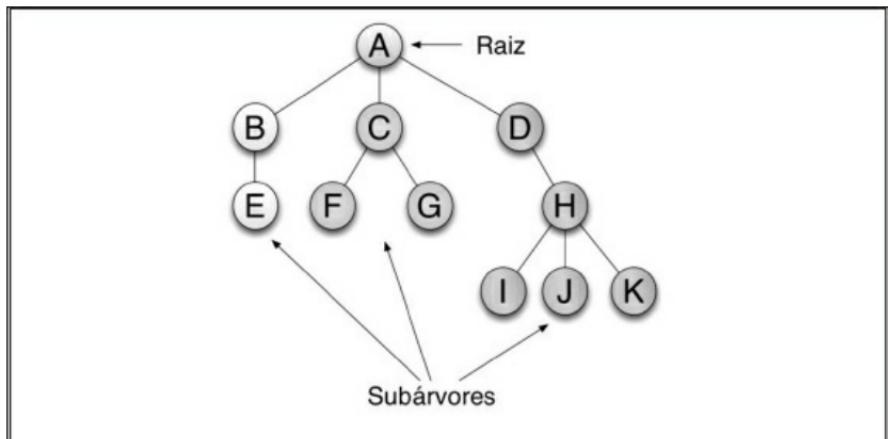


Figura 59 – Árvore com três sub-árvore distintas.

Dois nodos que são filhos do mesmo pai são chamados de irmãos. Um nodo v é dito externo quando este não possui filhos. Um nodo v é interno se possui um ou mais filhos. Nodos externos também são chamados de nodos folha. Na figura 58, por exemplo, todos os nodos que representam operadores são nodos internos e todos os nodos que representam operandos são nodos externos (ou folhas).

Um nodo é considerado irmão de outro nodo quando estes possuem o mesmo nodo pai. A figura 60 mostrada a seguir demonstra outros relacionamentos entre nodos de uma árvore. Todos os relacionamentos mostrados na árvore são relativos ao nodo X.

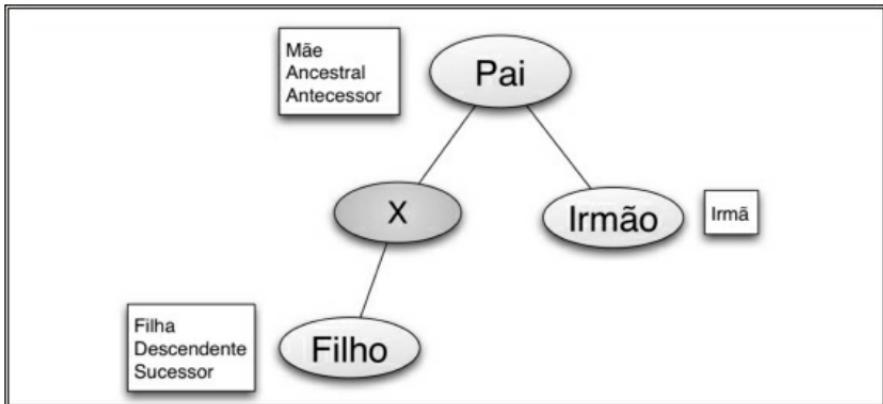


Figura 60 – Relacionamentos entre nodos de uma árvore.

Dependendo do contexto, a terminologia pode variar conforme mostra a figura 60. No caso de uma hierarquia de classes, por exemplo, podemos chamar uma classe de classe mãe ou de classe filha. No caso de uma árvore genealógica, é mais comum utilizar os termos ancestral e descendente.

O grau de um nodo é definido pelo número de subárvores que este contém. Um nodo folha sempre terá grau zero, uma vez que este não possui nenhum filho. O grau de uma árvore é dado pelo máximo entre os graus de seus nodos. Considerando a mesma árvore da figura 59, o grau da árvore é três, pois este é o grau máximo entre seus nodos. Os graus dos nodos internos desta árvore são mostrados na figura 61.

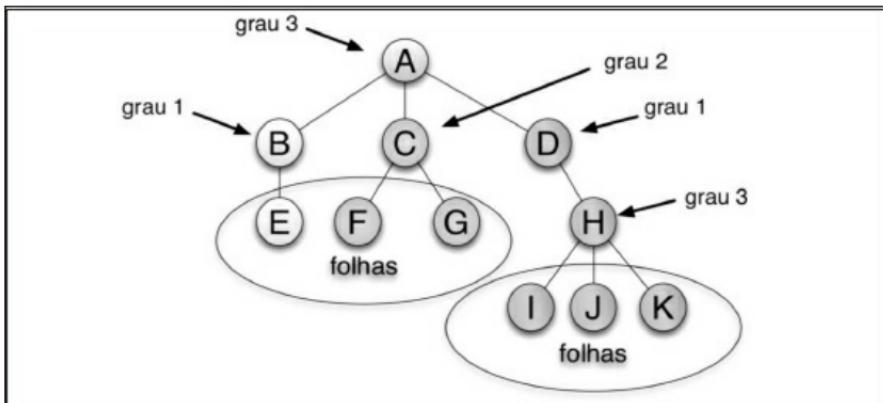


Figura 61 – Graus dos nodos.

Uma floresta é um conjunto de zero ou mais árvores disjuntas, como mostra a figura 62.

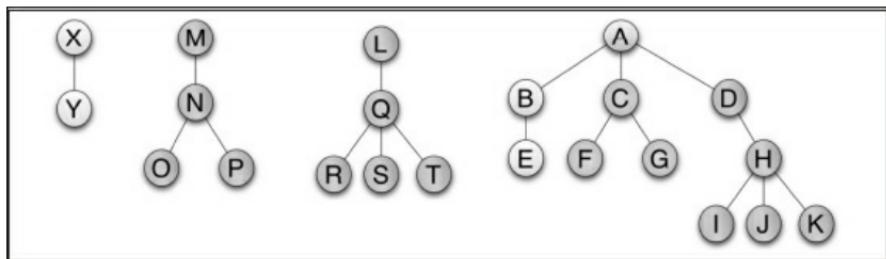


Figura 62 – Floresta – Conjunto de zero ou mais árvores.

Um caminho em uma árvore é uma sequência de nodos distintos, tal que sempre existem nodos consecutivos que possuem entre si a relação de “é filho de” ou “é pai de”. A figura 63 mostra um caminho que inicia em v_1 e termina em v_k .

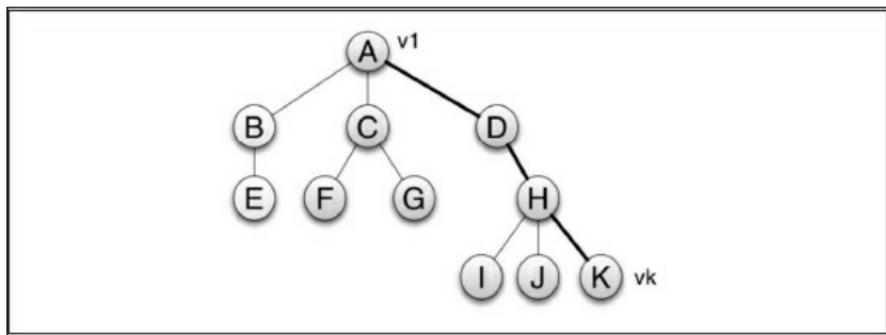


Figura 63 – Caminho em uma árvore.

Este caminho indica que v_1 alcança v_k e que v_k é alcançado por v_1 . O comprimento de um caminho é dado pelo número de nodos do caminho menos um. No caso da figura 63 o caminho que inicia no nodo A (v_1) e termina no nodo K (v_k) possui comprimento três.

O nível de um nodo em uma árvore é definido pelo número de nodos entre ele e a raiz da árvore. O nodo raiz sempre possui nível um, os filhos do nodo raiz nível dois e assim por diante. A altura, ou profundidade, de uma árvore é dada pelo maior nível entre seus nodos. A árvore mostrada na figura 64 possui altura quatro.

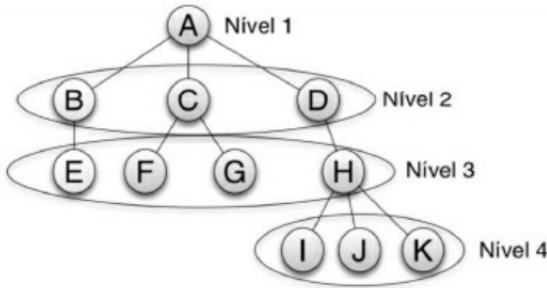


Figura 64 – Níveis em uma árvore. Raiz sempre possui nível 1.

Esta seção apresentou a terminologia relativa a árvores que será utilizada no restante deste livro. Na próximas seções estaremos estudando tipos específicos de árvores a partir de uma perspectiva de implementação da estrutura de dados. Neste sentido, do ponto de vista de implementação, um tipo abstrato de dados que representa uma árvore deve prover as seguintes operações básicas:

- ▶ Criação da árvore
- ▶ Inserção de um novo nodo como raiz
- ▶ Inserção de um novo nodo folha
- ▶ Inserção de um novo nodo em uma posição intermediária
- ▶ Exclusão de um determinado nodo
- ▶ Acesso a determinado nodo (caminhamento na árvore)
- ▶ Destruíção da árvore

No próximo capítulo estudaremos um tipo específico de árvore conhecida como árvore binária.

CAPÍTULO 7

ÁRVORES BINÁRIAS

Uma árvore binária é um tipo específico de árvore onde cada nodo pode conter zero, um ou dois filhos. Em outras palavras, é uma árvore em que o grau máximo é dois. Neste sentido, cada nodo de uma árvore binária possui um valor (chave e informações satélite), uma referência ao nodo esquerdo e outra referência ao nodo direito. A árvore mostrada na figura 65 é uma árvore binária, uma vez que seu grau é dois. Nesta árvore nenhum nodo possui mais de dois filhos.

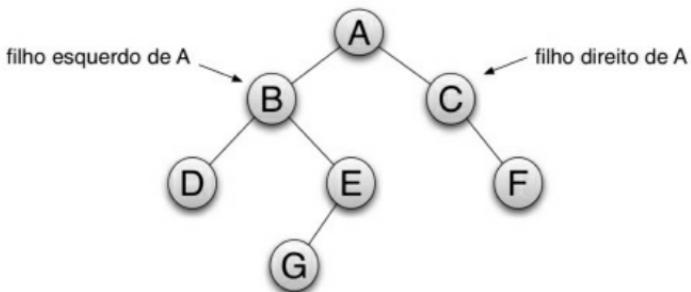


Figura 65 – Árvore binária.

A subárvore enraizada no filho da direita ou no filho da esquerda de um nodo interno v é chamada de subárvore direita ou subárvore esquerda de v respectivamente. No caso da figura 65, o nodo B, que é filho esquerdo do nodo A, é também raiz da subárvore esquerda de A. Da mesma maneira, o nodo C, que é filho direito de A, é também raiz da subárvore direita de A. A partir da terminologia estudada na seção anterior, podemos verificar que os nodos B e C são irmãos, pois estes possuem o mesmo pai.

Uma árvore binária é dita **estritamente binária** quando todo o nodo interno da árvore possuir subárvores direita e esquerda não vazias. Em outras palavras, é uma árvore binária onde todos os nodos possuem zero ou dois filhos como mostra a figura

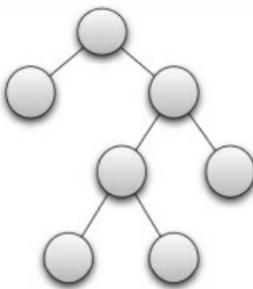


Figura 66 – Árvore estritamente binária.

Uma propriedade interessante de uma árvore estritamente binária é que o número de nodos pode ser obtido através da seguinte fórmula: $2*n - 1$, onde n é o número de nodos folha. Aplicando esta fórmula na árvore da figura 66, que contém quatro nodos folha, temos que o número total de nodos desta árvore é: $2*4 - 1 = 7$.

Uma árvore binária é dita binária completa quando todos os nodos folha encontram-se no último ou no penúltimo nível da árvore, como mostra a figura 67.

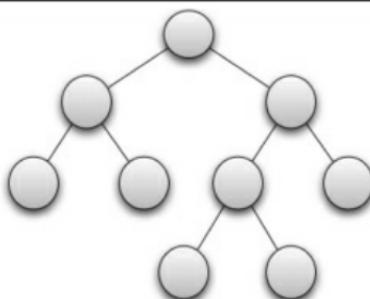


Figura 67 – Árvore binária completa.

Uma árvore binária é dita binária cheia quando esta é uma árvore estritamente binária onde os nodos folha encontram-se todos no último nível da árvore, como mostra a figura 68.

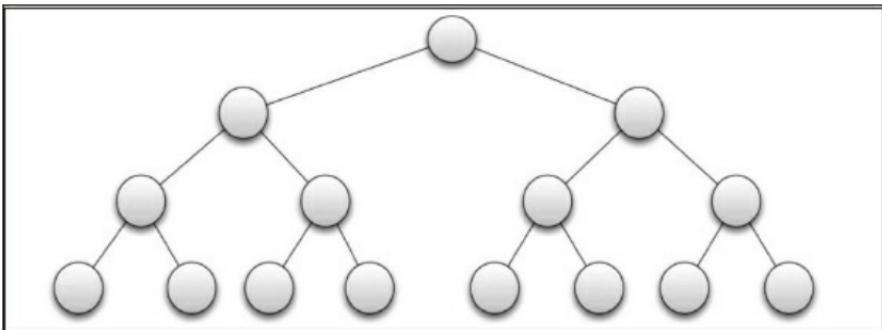


Figura 68 – Árvore binária completa. Nodos folha encontram-se no último nível.

Em uma árvore binária cheia de altura h , o número total de nodos é dado pela fórmula $2^h - 1$. Aplicando esta fórmula na árvore da figura 68, temos que o número de nodos da árvore será igual a $2^4 - 1 = 15$.

Utilizando esta mesma fórmula, podemos afirmar que a altura de uma árvore binária cheia é dada por $\log_2(n+1)$, onde n é o número de nodos da árvore. Uma propriedade interessante de uma árvore binária cheia é que, embora a árvore possa possuir muitos nodos, a distância da raiz até qualquer nodo folha da árvore (profundidade) é relativamente pequena.

Aplicação de árvores binárias

Uma das possíveis aplicações de uma árvore binária de pesquisa é na representação de uma expressão aritmética. Nesta representação, como mostrado na figura 58, cada nodo interno da árvore representa um operador e cada nodo externo representa um operando (constante ou variável). Uma árvore como esta será sempre uma árvore estritamente binária, uma vez que os operadores considerados atuam todos sobre dois operandos. Caso operadores unários (como por exemplo, o operador “ $-$ ”), então a árvore resultante da expressão não será estritamente binária.

Outra característica deste tipo de árvore é que o operador com menor prioridade aparece sempre na raiz da árvore. A subexpressão à esquerda deste operador dá origem à subárvore esquerda e, da mesma maneira, a subárvore direita dá origem à subárvore direita. Através desta abordagem a expressão representada pela árvore apresenta a ordem de prioridade das operações de maneira implícita.

A figura 69 mostra como a expressão $A + B * C$ pode ser representada por uma árvore binária.

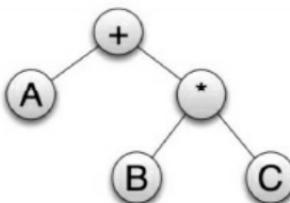


Figura 69 – Árvore binária representando a expressão $A + B * C$.

O caminhamento em uma árvore binária que representa uma expressão pode ser utilizado tanto para avaliar a expressão quanto para reescrever a expressão na forma pós-fixada¹, infixada ou ainda pré-fixada².

A tabela abaixo mostra como, através dos caminhamentos especificados, a expressão da figura 69 pode ser reescrita de diferentes maneiras.

Caminhamento	Expressão resultante
Pós ordem	Notação pós-fixada: A B C * +
Em ordem	Notação infixada (fórmula original): A+B*C
Pré-ordem	Notação polonesa: +A*B*C

Tabela 11 – Caminhamentos e expressões resultantes.

Os caminhamentos citados na tabela 11 serão explicados com mais detalhes no próximo capítulo.

Uma árvore binária que representa uma expressão aritmética contendo apenas os operadores de adição, subtração, multiplicação e divisão pode ser construída de acordo com o pseudocódigo mostrado abaixo.

```
Método Interpretar (String expressão) {
```

1. Construir uma árvore com a expressão contendo um único nodo.
2. Procurar por operadores de menor precedência (+ ou -) que não estejam entre parênteses, pois estes serão considerados operandos. Caso um destes operadores seja encontrado, a árvore deve ser substituída por uma árvore com este operador no nodo raiz e como operandos como nodos folha. Este procedimento deve ser repetido recursivamente no nodo direito da árvore resultante.
3. Repetir o passo 2 com operadores de média precedência (* ou /) somente nos nodos folha.

4. Percorrer os nodos folha (todos conterão operandos).
Se o nodo é um número ou uma variável, nada é feito.
Se o nodo é uma expressão entre parênteses, substituir este nodo por Interpretar(*expressão sem os parênteses*).
}

Exemplo 38 – Pseudocódigo do método que cria uma árvore binária a partir de uma expressão aritmética na forma infixa.

Aplicando o método descrito no exemplo 38 temos que, a partir da expressão $A + B * (C - D)$, a árvore resultante será construída como mostrado abaixo:

Passo Um

Neste passo um nodo é criado contendo a expressão inteira como mostra a figura 70.



A+ $B^*(C-D)$

Figura 70 – Passo Um. Expressão como um único nodo.

Passo Dois

Neste passo a expressão é analisada da esquerda para a direita e o operador de menor precedência (+) é encontrado. Este operador é então a nova raiz da árvore como mostra a figura 71.

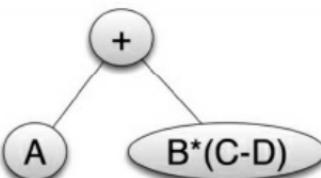


Figura 71 – Passo Dois. Operador de menor precedência vira raiz da árvore.

Passo Três

Neste passo, como não existem mais operadores de menor precedência (+ e -) e parênteses são considerados operandos, o algoritmo passa a procurar operadores de maior precedência (*) ou (/). Este operador é encontrado na subárvore direita e a árvore

resultante é mostrada na figura 72.

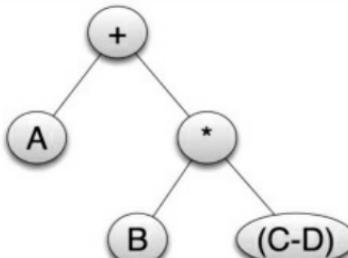


Figura 72 – Passo Três. Passo dois do algoritmo é aplicado em operadores de maior precedência.

Passo Quatro

Como não existem mais operadores de maior precedência, o passo quatro do algoritmo é executado. Neste passo o único nodo que não contém operandos é o nodo que contém a subexpressão $(C - D)$. Neste ponto o algoritmo é chamado recursivamente recebendo a expressão sem os parênteses: $C - D$. A árvore resultante fica então como a árvore mostrada na figura 73.

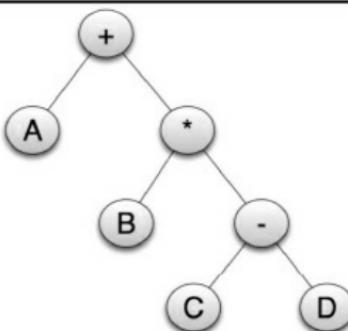


Figura 73 – Passo Quatro. Árvore resultante após o término do algoritmo.

Os parênteses não são necessários na árvore, pois a precedência dos operadores está implícita baseada em sua localização na árvore. Tomando o operador de subtração, como, exemplo, como ele está situado em um nível superior aos outros operadores, a operação $C - D$ será a primeira subexpressão a ser avaliada. A seguda

subexpressão a ser avaliada será B multiplicado pelo resultado de (C – D) e a última subexpressão será A + o resultado da multiplicação de B pelo resultado de (C – D).

O algoritmo que realiza a avaliação de uma expressão representada por uma árvore binária é mostrado abaixo em pseudocódigo:

```
double avalia(BinaryTreeNode v){  
    /* Este algoritmo recebe como parâmetro uma referência à raiz da árvore que  
    armazena a expressão. A classe BinaryTreeNode possui como atributos uma  
    referência ao filho esquerdo, uma referência ao filho direito e também  
    mais dois atributos:  
        char operador  
        double operando */  
    double retorno = 0.0;  
    if (v == null) return 0.0; //Se a raiz é null não existe  
                            //árvore e o algoritmo retorna  
                            //zero.  
    else if (v.operador == '+')  
        return v.operando; //se o nodo não possui operador o  
                            //valor do operando é retornado  
    else if (v.operador == '-')  
        retorno = avalia(v.esquerdo) - avalia(v.direito);  
    else if (v.operador == '*')  
        retorno = avalia(v.esquerdo) * avalia(v.direito);  
    else if (v.operador == '/')  
        retorno = avalia(v.esquerdo) / avalia(v.direito);  
    return retorno;  
}
```

Exemplo 39 – Pseudocódigo do método que avalia uma expressão representada por uma árvore binária.

O método avalia mostrado no exemplo 39 recebe como parâmetro uma referência à raiz da árvore que contém a expressão e utiliza chamadas recursivas para realizar o cálculo da expressão aritmética. A árvore mostrada na figura 73 é mostrada na figura 74 representada com nodos como os utilizados no algoritmo do exemplo 39.

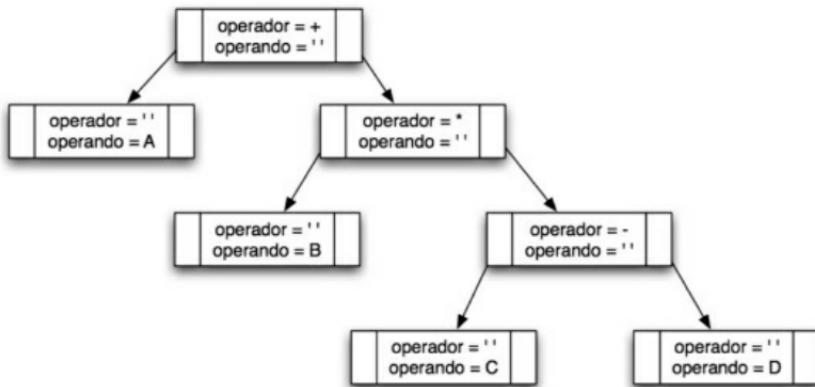


Figura 74 – Árvore que representa uma expressão aritmética.

O código fonte da classe *BinaryTreeNode* é mostrado no exemplo 40.

```
1  public class BinaryTreeNode {
2
3      protected char operando;
4      protected char operador;
5      protected BinaryTreeNode esquerdo;
6      protected BinaryTreeNode direito;
7
8      public BinaryTreeNode(char operando, char operador) {
9          this.operando = operando;
10         this.operador = operador;
11         esquerdo = null;
12         direito = null;
13     }
14
15     public char getOperando() {
16         return operando;
17     }
18
19     public void setOperando(char operando) {
20         this.operando = operando;
21     }
22
23     public char getOperador() {
24         return operador;
25     }
26
27     public void setOperador(char operador) {
28         this.operador = operador;
29     }
30
31     public BinaryTreeNode getEsquerdo() {
32         return esquerdo;
33     }
34
35     public void setEsquerdo(BinaryTreeNode n) {
36         this.esquerdo = n;
37     }
38
39     public BinaryTreeNode getDireito() {
40         return direito;
41     }
42
43     public void setDireito(BinaryTreeNode n) {
44         this.direito = n;
45     }
46 }
```

Exemplo 40 – Código fonte da classe *BinaryTreeNode*.

Esta seção apresentou a estrutura de dados árvore binária. Uma árvore binária é uma árvore em que um nodo pode ter no máximo dois filhos, isto é, uma árvore binária terá sempre um grau inferior ou igual a dois. A próxima seção apresenta a estrutura de dados árvore binária de pesquisa, que é um tipo especial de árvore binária.

1 http://pt.wikipedia.org/wiki/Notação_polonesa_reversa.

2 http://pt.wikipedia.org/wiki/Notação_polonesa.

CAPÍTULO 8

ÁRVORES BINÁRIAS DE PESQUISA

Uma árvore binária de pesquisa é um tipo específico de árvore binária que apresenta uma relação de ordem entre os nodos. Esta relação é definida por um campo específico do nodo, definido como a chave do nodo. Esta estrutura também é chamada de árvore binária ordenada ou árvore binária de busca, pois ela é muito utilizada em aplicações que necessitam de buscas eficientes.

Em uma árvore binária um nodo pode aparecer em qualquer posição da árvore, oferecendo assim uma flexibilidade na construção da árvore. Dependendo da aplicação, esta flexibilidade pode ser bastante útil, como no caso da árvore binária que representa uma expressão. Entretanto, esta abordagem faz com que, caso queiramos procurar por um nodo em específico, tenhamos que percorrer (no pior caso) todos os nodos da árvore para encontrar o nodo ou detectar que o nodo procurado não está presente na árvore.

Em uma árvore binária de pesquisa os nodos estão dispostos em uma ordem específica que visa facilitar a busca de um determinado nodo. Este tipo de árvore foi projetado para suportar operações eficientes de busca. Esta restrição define que, para qualquer nodo da árvore, o filho esquerdo deste nodo deve possuir uma chave menor que a sua e o filho direito deste nodo deve possuir uma chave maior que a sua chave.

A figura 75 mostra uma árvore binária de pesquisa que possui números inteiros como chave.

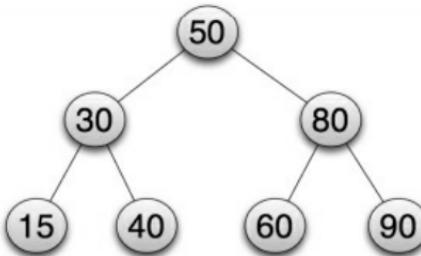


Figura 75 – Árvore binária de pesquisa.

Nesta árvore, para qualquer nodo tem-se que a subárvore esquerda deste nodo possui chaves menores que a sua chave e, da mesma maneira, a subárvore direita possui chaves maiores que a sua chave.

Um nodo em uma árvore binária de pesquisa possui a estrutura mostrada na figura 76. Além da informação armazenada (neste caso apenas a chave é mostrada), o nodo contém uma referência ao filho esquerdo e também uma referência ao filho direito.

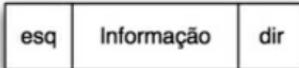


Figura 76 – Estrutura de um nodo de uma árvore binária de pesquisa.

O código fonte da classe que representa um nodo de uma árvore binária de pesquisa é mostrado no exemplo 41. Neste exemplo é considerado que o nodo armazena uma chave do tipo inteiro.

```
1  public class BSTNode {  
2      protected int key;  
3      protected BSTNode left, right;  
4  
5      public BSTNode() {  
6          left = right = null;  
7      }  
8  
9      public BSTNode(int num) {  
10         this(num,null,null);  
11     }  
12  
13     public BSTNode(int num, BSTNode lt, BSTNode rt) {  
14         this.key = num; left = lt; right = rt;  
15     }  
16  
17     protected int getKey(){  
18         return key;  
19     }  
20 }
```

Exemplo 41 – Código fonte da classe *BSTNode*.

A classe *BSTNode* possui três atributos: a chave (linha 2) e uma referência ao filho esquerdo e uma referência ao filho direito (linha 3). Esta classe possui três construtores (linhas 5 a 15) e um método que retorna a chave armazenada no nodo (linhas 16 a 19).

A árvore binária de pesquisa mostrada na figura 75, quando representada com a estrutura definida pela classe *BSTNode*, é mostrada na figura 77.

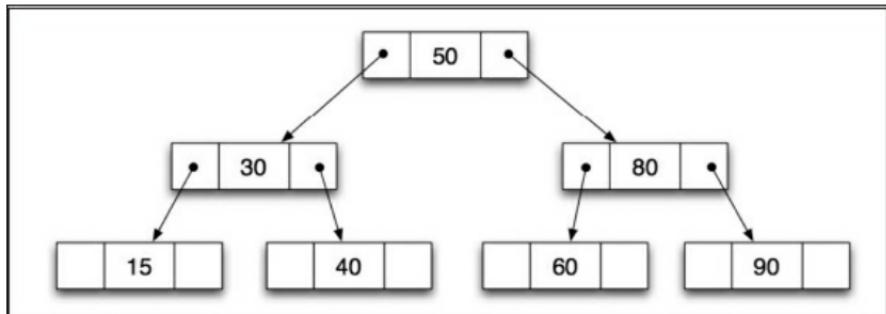


Figura 77 – Árvore binária de pesquisa representada com encadeamento de nodos.

Utilizando esta representação, cada elemento da árvore é um objeto do tipo *BSTNode*. Os nodos folha da árvore possuem as referências *left* e *right* com valor *null*, indicando que estes não possuem filhos. Os nodos internos da árvore possuem estas referências referenciando seus filhos esquerdos e direitos. Note que a árvore mantém a condição de árvore binária de pesquisa, isto é, para cada nodo tem-se que seu filho esquerdo possui uma chave menor e seu filho direito possui uma chave maior.

Construção de uma árvore binária de pesquisa

A construção de uma árvore binária de pesquisa deve garantir que os nodos inseridos mantenham a propriedade da árvore. Desta maneira, ao inserir um nodo, este deve ser comparado com nodos já existentes na árvores a fim de ser inserido corretamente. Considere as seguintes chaves: 15, 12, 20, 7, 14, 31 e 19. Supondo que estas chaves sejam inseridas em uma árvore binária de pesquisa (inicialmente vazia), teremos os seguintes passos executados:

Passo 1: Com a árvore inicialmente vazia, o nodo com chave 15 é inserido como raiz, como mostra a figura 78.

15

Figura 78 – Árvore binária de pesquisa após a inserção do nodo com chave 15.

Passo 2: Ao inserir o nodo com chave 12, primeiramente este é comparado com o nodo raiz. Como este é menor, será inserido à esquerda do nodo raiz (nodo com chave quinze), como mostra a figura 79.



Figura 79 – Árvore binária de pesquisa após a inserção do nodo com chave 12.

Passo 3: Ao inserir o nodo com chave 20, este é inicialmente comparado com o nodo raiz. Como ele é maior que o nodo raiz e a posição a direita do nodo raiz está vazia, o nodo é inserido à direita do nodo com chave 15, como mostra a figura 80.

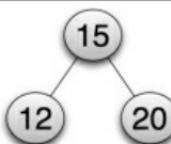


Figura 80 – Árvore binária de pesquisa após a inserção do nodo com chave 20.

Passo 4: Ao inserir o nodo com chave 7, este inicialmente é comparado com a raiz da árvore. Como ele é menor, é então comparado com o filho esquerdo da raiz (nodo com chave 12). Como sua chave também é menor, o nodo acaba sendo inserido como filho esquerdo do nodo com chave 12, como mostra a figura 81.

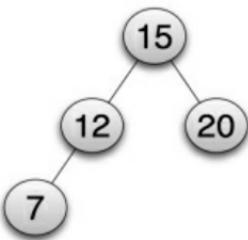


Figura 81 – Árvore binária de pesquisa após a inserção do nodo com chave 7.

Passo 5: Ao inserir o nodo com chave 14, este é comparado com a raiz. Como ele é menor, é então comparado com o nodo com chave 12. Como ele é maior, é inserido à direita deste como mostra a figura 82.

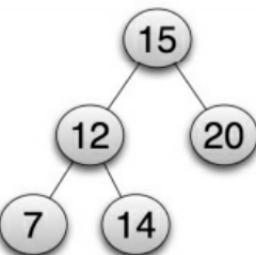


Figura 82 – Árvore binária de pesquisa após a inserção do nodo com chave 14.

Passo 6: Ao inserir o nodo com chave 31, este é comparado com a raiz. Como ele é maior, é então comparado com o nodo com chave 20. Como ele também é maior, ele acaba sendo inserido à direita deste, como mostra a figura 83.

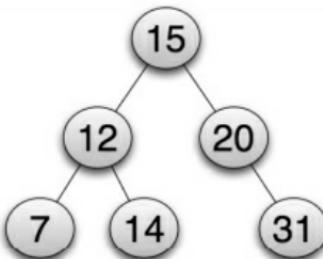


Figura 83 – Árvore binária de pesquisa após a inserção do nodo com chave 31.

Passo 7: Ao inserir o nodo com chave 19, a mesma lógica é seguida. Inicialmente, ele é comparado com a raiz da árvore. Como ele é maior, ele é então comparado com o filho direito (nodo com chave 20); como ele é menor, acaba sendo inserido como filho esquerdo do nodo com chave 20, como mostra a figura 84.

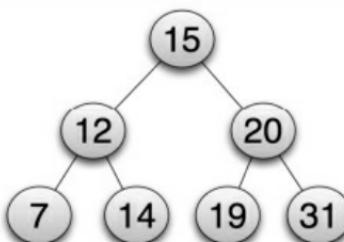


Figura 84 – Árvore binária de pesquisa após a inserção do nodo com chave 19.

A árvore final mostrada na figura 84 é uma árvore binária completa. Note que, para qualquer nodo desta árvore temos que o filho esquerdo sempre possui uma chave menor e o filho direito sempre possui uma chave maior.

Classe BST (Binary Search Tree)

A implementação da classe que representa uma árvore binária de pesquisa é mostrada abaixo. Esta implementação não está completa, pois, à medida que as operações em árvores binárias de pesquisa forem sendo explicadas, os métodos correspondentes serão adicionados a esta classe. A implementação do exemplo 42

mostra apenas alguns métodos presentes na classe BST.

```
1  public class BST {  
2  
3      protected BSTNode root = null;  
4  
5      public void clear() {  
6          root = null;  
7      }  
8      public boolean isEmpty() {  
9          return root == null;  
10     }  
11  
12     public BSTNode getRootNode () {  
13         return root;  
14     }  
15     ...
```

Exemplo 42 – Código fonte da classe BST (*Binary Search Tree*) que representa uma árvore binária de pesquisa.

Nesta implementação a classe BST possui como atributo uma referência ao nodo raiz da árvore (linha 3). O método *clear* (linhas 5 a 7) elimina todos os nodos da árvore ao atribuir o valor *null* para o atributo *root*.

O método *isEmpty()*, linhas 8 a 10, retorna *true* caso a árvore esteja vazia (não possui nenhum nodo), ou *false* caso contrário. A árvore não possuirá nenhum nodo quando o atributo *root* estiver com o valor *null*.

O método *get rootNode()* (linhas 12 a 14) retorna uma referência ao nodo raiz da árvore.

Busca

Devido à ordem em que os nodos são dispostos em uma árvore binária de pesquisa, a busca de um determinado valor é bastante otimizada quando comparada à busca de um elemento em uma lista, por exemplo. Sempre que um valor for procurado em uma árvore binária de pesquisa, o valor é comparado com o nodo corrente (que inicialmente é o nodo raiz da árvore). Se a chave procurada for menor, o novo nodo corrente é o filho esquerdo e a comparação é feita novamente. Se a chave procurada é maior, o novo nodo corrente é o filho direito. A busca para quando for encontrado o nodo com a chave procurada ou quando não existir mais meios de continuar a busca (nodo corrente é um nodo folha), pois a chave procurada não está na árvore.

Utilizando a árvore da figura 84 como exemplo, em uma busca pelo nodo com chave 19, os nodos visitados seriam os nodos com chave 15, 20 e então o nodo com chave 19. A figura 85 destaca os nodos visitados na busca pelo nodo com chave 19.

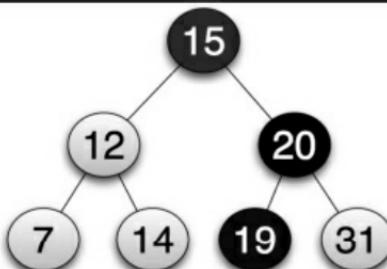


Figura 85 – Nodos visitados na busca pelo nodo com chave 19.

Note que, embora a árvore binária de pesquisa possua sete nodos, com apenas três comparações pudemos encontrar o nodo procurado.

O método que realiza a busca de determinada chave em uma árvore binária de pesquisa é mostrado no exemplo 43.

```
16  public BSTNode search(int el) {  
17      return search(root,el);  
18  }  
19  protected BSTNode search(BSTNode p, int el) {  
20      while (p != null) {  
21          if (el==p.key) return p;  
22          else if (el<p.key) p = p.left;  
23          else p = p.right;  
24      }  
25      return null;  
26  }
```

Exemplo 43 – Métodos de busca em uma árvore binária de pesquisa.

O método de busca (*search*) possui uma versão pública que recebe como parâmetro a chave a ser procurada (nesta implementação estamos considerando que as chaves armazenadas são números inteiros) e também uma versão protegida que recebe a chave e também uma referência a um nodo a partir de onde a busca deve ser realizada. Desta maneira, quem for utilizar a classe BST irá utilizar o método que recebe apenas a chave procurada e retorna (caso encontrado) uma referência ao nodo da árvore que possui a chave procurada, caso ele exista na árvore, ou o valor *null* caso

contrário.

O método *search* que recebe apenas a chave a ser procurada (linhas 16 a 18) chama o método *search* que recebe a chave e uma referência a um nodo passando a chave e a referência à raiz da árvore. Este método (linhas 19 a 26) então entra em um laço em que, para cada nodo visitado, um teste é realizado para verificar se a chave do nodo é a chave sendo procurada (linha 21). Caso a chave seja a chave procurada, uma referência a este nodo é retornada. Caso ela não seja a chave procurada, um teste é realizado para verificar se a chave sendo procurada é menor que a chave corrente. Caso ela seja, a busca prossegue pela subárvore esquerda (linha 22). Caso ela não seja, a busca prossegue pela subárvore direita (linha 23). Este laço continua sua execução até que a chave encontrada seja procurada ou até que a referência *p* receba o valor *null*, indicando que a chave procurada não está presente na árvore.

Inserção

A inserção em uma árvore binária de pesquisa segue uma lógica semelhante ao algoritmo de busca. Sempre que um nodo for inserido em uma árvore binária de pesquisa, a ordem entre os nodos da árvore deve ser mantida, isto é, a inserção deve fazer com que, para cada nodo da árvore, todos os nodos presentes na subárvore esquerda deste nodo possuam chaves menores e todos os nodos presentes na subárvore direita possuam chaves maiores.

Tomando como exemplo a árvore da figura 86, após a inserção de um nodo com a chave 23, esta ficará como mostra a figura 87.

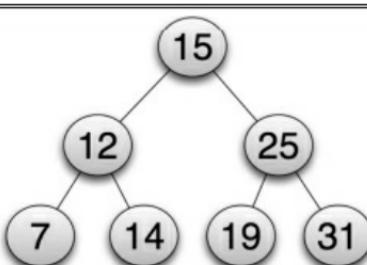


Figura 86 – Árvore binária de pesquisa onde será inserido um nodo com chave 23.

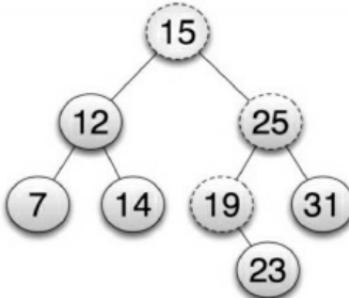


Figura 87 – Árvore binária de pesquisa após a inserção do nodo com chave 23.

Os nodos com linhas tracejadas são os nodos percorridos para encontrar o lugar correto do nodo.

De modo a encontrar o lugar correto na árvore do nodo com chave 23, inicialmente a este é comparado com a raiz da árvore (nodo com chave 15). Como a chave 23 é maior, ela é comparada com o filho direito (nodo com chave 25). Como este é menor, a próxima comparação é feita com o nodo com chave 19 (filho esquerdo do nodo com chave 25). A próxima comparação seria realizada com o filho direito do nodo com chave 19, mas como este não existe (na verdade o valor dele é *null*), este é o lugar onde é inserido o nodo com chave 23.

O método de inserção em uma árvore binária de pesquisa é mostrado no exemplo 44.

```

27 public boolean insert(int el) {
28     BSTNode p = root, prev = null;
29
30     if (search(el)!=null){
31         return false;
32     }
33
34     while (p != null) {
35         prev = p;
36         if (el<p.key) p = p.left;
37         else p = p.right;
38     }
39
40     if (root == null)  root = new BSTNode(el);
41     else if (prev.key<el) prev.right = new BSTNode(el);
42     else prev.left = new BSTNode(el);
43     return true;
44 }
```

Exemplo 44 – Método de inserção em uma árvore binária de pesquisa.

O método *insert* mostrado no exemplo 44 recebe como parâmetro a chave a ser inserida na árvore e retorna *true* caso a inserção tenha ocorrido com sucesso, ou *false* caso contrário. Duas variáveis locais são utilizadas para realizar a inserção: *p*, inicializado com a raiz da árvore e *prev*, inicializado com o valor *null*. Um teste é realizado na linha 30 para verificar se a chave já está presente na árvore. Neste caso o método retorna *false* (esta implementação não aceita nodos com chaves repetidas).

O laço das linhas 34 a 38 localiza o lugar correto para a inserção da chave. Ao final da execução deste laço a referência *p* estará com o valor *null* e a referência *prev* estará referenciando o pai do nodo a ser inserido.

O código da linha 40 verifica se a raiz da árvore contém o valor *null*. Neste caso o novo nodo é inserido como raiz da árvore. O código das linhas 41 e 42 verifica em que posição (filho esquerdo ou filho direito) o novo nodo deve ser inserido em relação ao nodo *prev* que é o nodo pai do nodo sendo inserido.

Caminhamento

O caminhamento em uma árvore binária de pesquisa é o processo de visitar cada nodo da árvore exatamente uma vez. Diferentes algoritmos projetados com árvores em mente tem como característica em comum visitar sistematicamente todos os nodos da árvore. Isto é, o algoritmo caminha através da estrutura de dados e faz algumas computações em cada nodo da árvore.

O caminhamento, também chamado de percurso, pode ser interpretado como colocar todos os nodos da árvore em uma linha. Este processo também é conhecido como linearização da árvore.

Existem dois métodos essencialmente diferentes no que diz respeito à visita sistemática dos nodos de uma árvore, ou percurso:

- ◆ Percursos em extensão;
- ◆ Percursos em profundidade.

Os percursos em extensão visitam todos os nodos de cada nível da árvore, nível por nível, indo do nível mais alto ao nível mais baixo ou vice-versa.

Os percursos em profundidade percorrem os caminhos da árvore percorrendo inicialmente o caminho mais à esquerda e assim por diante. Neste livro estudaremos três tipos de caminhamento em profundidade em uma árvore binária:

- ◆ Caminhamento em ordem;

- Caminhamento em pré-ordem;
- Caminhamento em pós-ordem.

As próximas seções detalham os caminhamentos em profundidade tanto como exemplo a árvore binária de pesquisa da figura 86.

Caminhamento em ordem

Em um caminhamento em ordem os nodos de uma árvore binária são visitados na seguinte ordem:

1. Percorre a subárvore esquerda;
2. Visita o nodo;
3. Percorre a subárvore direita.

O algoritmo que realiza este caminhamento pode ser definido de maneira recursiva e é mostrado no exemplo 45.

```
45     public void inorder() {  
46         inorder(root);  
47     }  
48     protected void inorder(BSTNode p) {  
49         if (p != null) {  
50             inorder(p.left);  
51             System.out.print(p.key));  
52             inorder(p.right);  
53         }  
54     }
```

Exemplo 45 – Métodos que realizam o caminhamento em ordem em uma árvore binária de pesquisa.

O algoritmo de caminhamento em ordem é implementado no exemplo 45 na forma de dois métodos da classe BST. Uma versão do método *inorder* não recebe parâmetro nenhum e chama uma versão sobrecarregada deste método passando como parâmetro a referência ao nodo raiz da árvore. É neste método sobrecarregado que o caminhamento ocorre de maneira recursiva.

Aplicando este algoritmo na árvore da figura 86, os nodos serão visitados na seguinte ordem:

7, 12, 14, 15, 19, 25 e 31

Este caminhamento, quando aplicado a uma árvore binária de pesquisa como a da figura 86, visita os nodos de maneira crescente.

Caminhamento em pré-ordem

No caminhamento em pré-ordem os nodos de uma árvore binária são visitados da seguinte maneira:

1. Visita o nodo;
2. Percorre a subárvore esquerda;
3. Percorre a subárvore direita.

O algoritmo do caminhamento também pode ser definido de maneira recursiva e é mostrado no exemplo 46.

```
55     public void preorder() {  
56         preorder(root);  
57     }  
58     protected void preorder(BSTNode p) {  
59         if (p != null) {  
60             System.out.print(p.key);  
61             preorder(p.left);  
62             preorder(p.right);  
63         }  
64     }  
65 }
```

Exemplo 46 – Métodos que realizam o caminhamento em pré-ordem em uma árvore binária de pesquisa.

O algoritmo de caminhamento em pré-ordem é implementado no exemplo 46 na forma de dois métodos da classe BST. Uma versão do método *preorder* não recebe parâmetro nenhum e chama uma versão sobrearcregada, passando como parâmetro uma referência à raiz da árvore. É neste método sobrearcregado que o caminhamento ocorre de maneira recursiva.

Aplicando este algoritmo na árvore binária de pesquisa da figura 86, os nodos são visitados na seguinte ordem:

15, 12, 7, 14, 25, 19 e 31

Caminhamento em pós-ordem

No caminhamento em pós-ordem os nodos de uma árvore binária são visitados da seguinte maneira:

1. Percorre a subárvore esquerda;
2. Percorre a subárvore direita;
3. Visita o nodo.

O algoritmo que implementa o caminhamento em pós-ordem também é definido de maneira recursiva e é mostrado no exemplo 47.

```
65  public void postorder() {  
66      postorder(root);  
67  }  
68  
69  protected void postorder(BSTNode p) {  
70      if (p != null) {  
71          postorder(p.left);  
72          postorder(p.right);  
73          System.out.print(p.key);  
74      }  
75  }
```

Exemplo 47 – Métodos que implementam o caminhamento em pós-ordem em uma árvore binária de pesquisa.

O algoritmo de caminhamento em pós-ordem é implementado no exemplo 47 na forma de dois métodos da classe BST. Uma versão do método *postorder* não recebe parâmetro nenhum e chama uma versão sobrecarregada do mesmo método passando como parâmetro uma referência à raiz da árvore. Assim como nos outros caminhamentos, é neste método que recebe uma referência a um nodo que o caminhamento acontece de maneira recursiva.

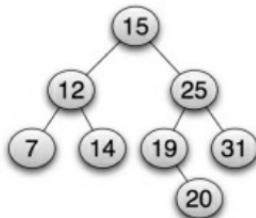
Aplicando este algoritmo na árvore binária de pesquisa da figura 86, os nodos são visitados na seguinte ordem:

7, 14, 12, 19, 31, 25 e 15

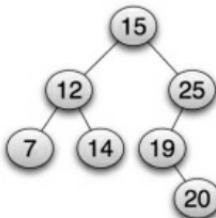
Remoção

A remoção de nodos de uma árvore binária de pesquisa deve garantir que os nodos da árvore continuem ordenados após a operação de remoção. Se o nodo a ser removido é um nodo folha (possui grau zero), então esta operação é bastante simples uma vez

que esta remoção não altera a ordem dos nodos. Considere, por exemplo a remoção do nodo com chave trinta e um mostrado na figura 88 (a). A árvore resultante após a remoção deste nodo é mostrada na figura 88 (b). Este é o caso mais simples de remoção, pois a ordem dos nodos da árvore não foi alterada apóas a remoção.



(a)

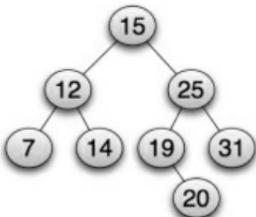


(b)

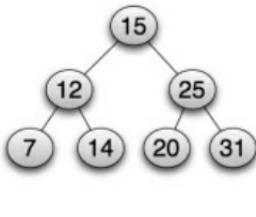
Figura 88 – (a) Árvore binária de pesquisa antes da remoção do nodo com chave 31.
(b) Árvore binária de pesquisa após a remoção do nodo com chave 31.

Um caso mais complexo na remoção de um nodo de uma árvore binária de pesquisa ocorre quando o nodo a ser removido possui um filho. Tomando como exemplo a figura 88 (a), caso o nodo com chave 19 seja removido, temos que este não pode ser removido diretamente, pois possui o nodo com chave 20 como filho direito. Neste caso a operação de remoção deve fazer com que o nodo com chave 20 se torne o filho esquerdo do nodo com chave 25. Para isso a operação de remoção, além de localizar o nodo a ser removido, deve também localizar o pai deste nodo.

A figura 89 (a) mostra a árvore binária de pesquisa antes da remoção do nodo com chave 19 e a figura 89 (b) mostra a árvore binária apóas a remoção deste nodo.



(a)



(b)

- Figura 89 – (a) Árvore binária de pesquisa antes da remoção do nodo com chave 19.
(b) Árvore binária de pesquisa após a remoção do nodo com chave 19. Após a remoção do nodo com chave 19, o nodo com chave 20, que era filho direito do nodo 19, passou a ser o filho esquerdo do nodo com chave 25.

O método que localiza o pai de um determinado nodo em uma árvore binária de pesquisa é mostrado no exemplo 48.

```
76 protected BSTNode searchFather (int el) {  
77     BSTNode p = root;  
78     BSTNode prev = null;  
79     while (p != null && !(p.key==el)) {  
80         prev = p;  
81         if (p.key<el)  
82             p = p.right;  
83         else p = p.left;  
84     }  
85     if (p!=null && p.key==el) return prev;  
86     return null;  
87 }
```

Exemplo 48 – Método que retorna uma referência ao pai do nodo que contém a chave passada como parâmetro.

O método *searchFather* recebe como parâmetro uma chave e retorna uma referência ao nodo pai do nodo que contém esta chave. Para isso uma referência auxiliar (*prev* definida na linha 78) é utilizada. Caso a chave passada como parâmetro esteja contida no nodo raiz ou a chave não esteja presente na árvore, o método retorna o valor *null*.

O caso mais complexo ocorre quando o nodo a ser removido contém dois filhos. Neste caso o nó deve ser removido deve ser substituído pelo maior valor contido em sua subárvore esquerda (isso faz com que a ordem dos nodos da árvore seja mantida). Uma vez que esta substituição, ou cópia, tenha sido feita, a remoção do nodo que foi copiado será um dos casos simples: nodo com um ou com nenhum filho.

Considere como exemplo a árvore mostrada na figura 90. Os passos necessários para a remoção do nodo com chave 90 são mostrados na figura 91.

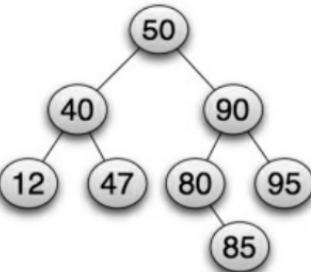


Figura 90 – Árvore binária de pesquisa antes da remoção do nodo com chave 90.

Os passos realizados para a remoção do nodo com chave 90 são mostrados na figura 91.

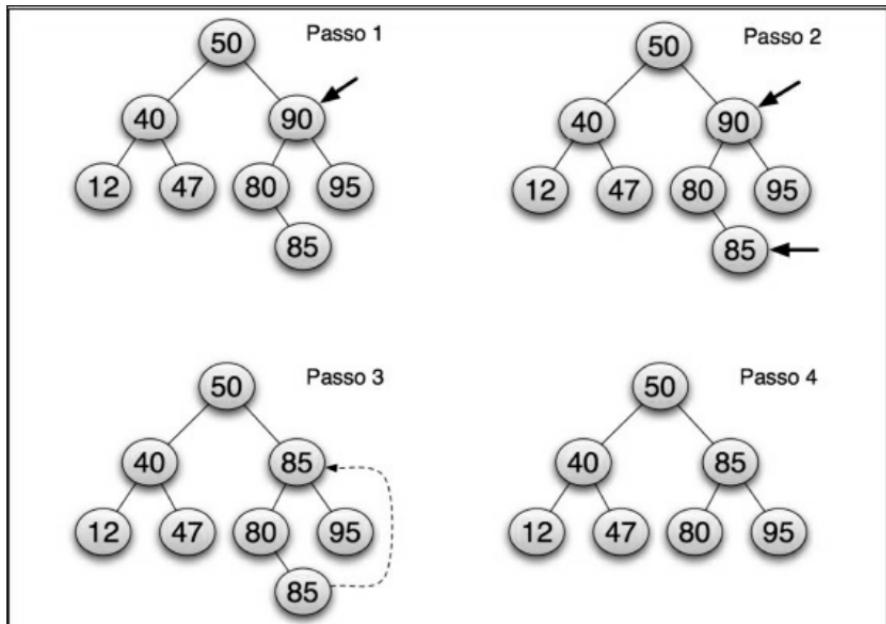


Figura 91 – Passos realizados para a remoção do nodo com chave 90.

No passo um o nodo a ser removido é localizado (indicado pela seta). Como ele possui dois filhos, o maior nodo de sua subárvore esquerda é localizado no passo dois

(também indicado pela seta). A chave deste nodo é então copiada para o nodo a ser removido no passo três. Esta abordagem faz com que a ordem imposta pela árvore binária de pesquisa seja mantida. No passo quatro o nodo localizado no passo dois é removido. A remoção deste nodo sempre será um caso simples (nodo com um ou com nenhum filho).

O método que realiza a remoção de um nodo de uma árvore binária de pesquisa é mostrado no exemplo 49.

```
88     public void delete (int el) {
89         BSTNode node, father = null;
90         node = search (el) ;
91         if (node != null) {
92             if (node!=root) father = searchFather (el);
93             if (node.right == null){
94                 if (node==root) root= node.left;
95                 else if (father.left == node) father.left = node.left;
96                 else father.right = node.left;
97             }
98             else if (node.left == null) {
99                 if (node==root) root= node.right;
100                else if (father.left == node) father.left = node.right;
101                else father.right = node.right;
102            }
103            else {
104                BSTNode tmp = node.left;
105                while (tmp.right != null) {
106                    tmp = tmp.right;
107                }
108                delete(tmp.key);
109                node.key = tmp.key;
110            }
111        }
112        else if (root != null)
113            System.out.println("A chave " + el + " não está na árvore");
114        else System.out.println("A árvore está vazia");
115    }
```

Exemplo 49 – Método que realiza a remoção de um nodo de uma árvore binária de pesquisa.

O método mostrado no exemplo 49 recebe como parâmetro a chave a ser removida da árvore. Na linha 90 o nodo a ser removido é procurado e, caso este seja encontrado, uma referência a ele é armazenada na variável local *node*. Caso este nodo tenha sido encontrado, outro teste é realizado na linha 92 para verificar se este nodo não é a raiz da árvore. Caso ele não seja, uma referência ao pai deste nodo é armazenada na variável local *father*. Na linha 93 um teste é realizado para verificar se o nodo não possui filho direito. Caso este teste seja verdadeiro, a remoção é um dos casos simples (nodo com zero ou um filho). Na linha 98 um teste é feito para verificar se o nodo a

ser removido não possui filho esquerdo. Caso este teste seja verdadeiro, a remoção também é um dos casos mais simples onde o nodo possui apenas um filho. Caso ambos os testes (linhas 93 e 97) sejam falsos, significa que o nodo a ser removido possui ambos os filhos e os passos mostrados na figura 91 devem ser realizados. Inicialmente, uma referência ao filho esquerdo do nodo é armazenada na variável local *tmp* (linha 104). Após, o laço das linhas 105 a 107 faz com que esta referência “caminhe” até a extremidade direita da subárvore esquerda do nodo a ser removido, que é onde está armazenada a maior chave desta subárvore. Uma vez que esta chave tenha sido localizada, o método é chamado recursivamente para que esta chave seja removida. Note que neste caso a remoção é um caso simples (nodo com zero ou com um filho). Quando a chamada recursiva retornar a chave armazenada na referência *tmp*, será copiada para a chave do nodo a ser removido. Neste ponto o maior valor da subárvore esquerda já terá sido removido e a árvore continua sendo uma árvore binária de pesquisa, isto é, para qualquer nodo armazenado da árvore, sua subárvore esquerda possui apenas nodos com chaves menores e sua subárvore direita possui apenas nodos com chaves maiores.

Este capítulo apresentou a estrutura de árvore binária de pesquisa, também chamada de árvore binária de busca. Esta estrutura é bastante utilizada na busca de dados, pois a ordem em que os nodos são dispostos na árvore facilita este tipo de operação. O próximo capítulo apresenta a estrutura de árvore AVL, que é um tipo mais específico de árvore binária de pesquisa.

CAPÍTULO 9

ÁRVORES AVL

As árvores binárias de pesquisa apresentadas no capítulo anterior são comumente utilizadas em operações de busca, pois estas apresentam uma boa eficiência para este tipo de operação. Entretanto, a busca em uma árvore binária de pesquisa é eficiente somente quando os nodos estão distribuídos uniformemente na árvore. Considere, por exemplo, uma árvore binária de pesquisa com os seguintes nodos (inseridos nesta ordem): 10, 12, 15, 20, 28, 30 e 32.

A árvore resultante é mostrada na figura 93. Esta árvore claramente não apresenta muita eficiência na busca por elementos. Não só na busca, mas também nas operações de inserção e remoção, uma vez que estas também dependem da forma da árvore. Considere, por exemplo, a busca pelo elemento 32 nesta árvore. Esta busca necessitará de n comparações, onde n é o número de nodos da árvore.

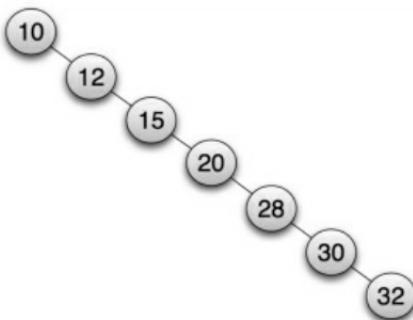


Figura 92 – Árvore binária de pesquisa resultante das inserções das chaves 10, 12, 15, 20 e 28 (nesta ordem).

A topologia desta árvore é semelhante a uma lista simplesmente encadeada. Isso acontece porque esta árvore não está balanceada. Em uma árvore binária de pesquisa isso acontece visto que esta estrutura não armazena nenhuma informação sobre a forma da árvore. Esta mesma árvore é mostrada na figura 93.

Esta árvore contém os mesmos nodos da árvore anterior, porém esta árvore está balanceada. Uma árvore binária de pesquisa é dita balanceada quando a distância média dos nodos até a raiz for mínima.

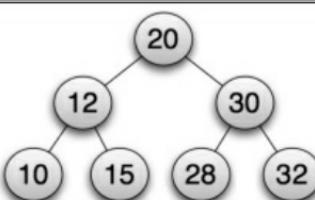


Figura 93 – Árvore binária de pesquisa balanceada.

Note que nesta árvore a subárvore esquerda possui o mesmo número de nodos que a subárvore direita. Ainda, esta árvore é uma árvore binária completa, uma vez que os nodos folha encontram-se todos no último nível da árvore.

Se dissermos que uma árvore binária é balanceada se as subárvores esquerda e direita de cada nodo tiverem a mesma altura, então as únicas árvores binárias balanceadas serão as árvores binárias completas, que são aquelas que possuem 2^{h-1} nodos, onde h é a altura da árvore. Neste caso somente poderíamos criar árvores binárias balanceadas com n nodos, para $n = 1, 3, 7, 15, 21, 63, \dots$. Claramente, esta condição de balanceamento não é adequada, pois ela não permite que árvores binárias sejam criadas para qualquer n .

Uma condição de balanceamento menos restritiva foi proposta por Adelson-Velski e Landis em 1962. Esta condição diz que em uma árvore binária balanceada a altura da subárvore esquerda deve diferir da altura da subárvore direita de no máximo 1. Claramente esta condição é menos restritiva que a condição anterior, uma vez que nem todas as árvores binárias que satisfazem esta condição são árvores binárias completas, mas todas as árvores binárias completas satisfazem essa condição.

As árvores binárias de pesquisa que satisfazem essa condição são chamadas árvores AVL e utilizam um atributo adicional em seus nodos de modo a manter informação a respeito da forma da árvore. Este atributo adicional armazena a altura da subárvore cujo nodo é raiz e é utilizado pelos algoritmos que manipulam a árvore para verificar a condição de balanceamento. Mais precisamente, este atributo é utilizado para calcular o fator de balanceamento do nodo, ou simplesmente fator do nodo. No contexto de uma árvore AVL o fator de um nodo n é dado pela seguinte fórmula:

$$\text{fator}(n) = h(\text{esq}(n)) - h(\text{dir}(n))$$

Figura 94 – Fórmula para obtenção do fator de um nodo n.

Onde h é uma função que retorna a altura de um determinado nodo, esq é uma função que retorna a raiz da subárvore esquerda de um determinado nodo e dir é uma função que retorna a subárvore direita de um determinado nodo.

O Exemplo quarenta e nove mostra o código fonte da classe `AVLNode` que representa um nodo de uma árvore binária de pesquisa AVL.

```
1  public class AvlNode {
2
3      protected int height;
4      protected int key;
5      protected AvlNode left, right;
6
7      public AvlNode ( int theElement ) {
8          this( theElement, null, null );
9      }
10
11     public AvlNode ( int theElement, AvlNode lt, AvlNode rt ) {
12         key = theElement;
13         left = lt;
14         right = rt;
15         height = 0;
16     }
17 }
```

Exemplo 50 – Código fonte da classe `AVLNode` que representa um nodo de uma árvore AVL.

Esta classe é semelhante à classe `BSTNode`, que representa um nodo de uma árvore binária de pesquisa. A classe `AVLNode`, entretanto, define um atributo adicional, `height`, na linha 3, que é utilizado para o cálculo do fator do nodo.

De modo a demonstrar como os fatores dos nodos são obtidos, considere a seguinte árvore binária de pesquisa com os fatores de cada nodo exibidos à direita dos mesmos.

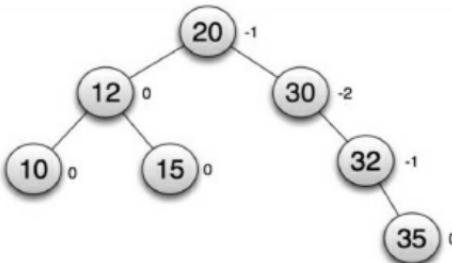
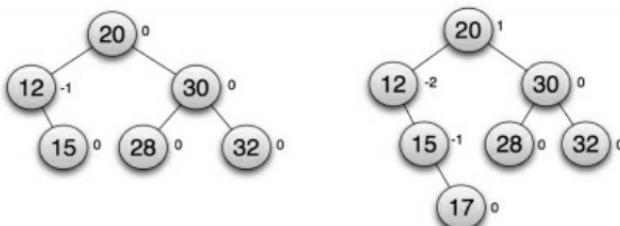


Figura 95 – Árvore binária de pesquisa com os fatores dos nós.

Esta árvore não é uma árvore AVL, pois o fator do nodo com chave 30 está fora do intervalo $-1, 0$ e 1 . O fator de cada um dos nodos foi obtido através da fórmula definida na figura 94. Tomando o nodo com chave 30 como exemplo, note que sua subárvore esquerda possui altura zero e sua subárvore direita possui altura dois. Logo, seu fator é menos dois. Da mesma maneira, se tomarmos o nodo com chave 20 (a raiz da árvore), temos que sua subárvore esquerda possui altura dois e sua subárvore direita possui altura três, logo seu fator é menos um.

Considere agora a árvore mostrada na figura 96 (a). Esta árvore é uma árvore AVL, pois os fatores de seus nodos estão no intervalo $-1, 0$ e 1 . Esta mesma árvore é mostrada na figura 96 (b) após a inserção do nodo com chave 17. Note que, após esta inserção, a árvore deixou de ser AVL, pois o nodo com chave 12 ficou com fator menos dois.

Todas as operações em uma árvore binária de pesquisa AVL devem garantir que a árvore mantenha a propriedade AVL. Desta maneira, após a inserção se um novo elemento na árvore, esta deve ser reestruturada caso necessário.



(a)

(b)

Figura 96 – (a) Árvore binária AVL; (b) Árvore binária após a inserção do nodo com chave 17.

Em uma árvore binária de pesquisa do tipo AVL a reestruturação da árvore ocorre através de operações de rotação. Estas operações têm por característica preservar a ordem das chaves e seu objetivo é fazer com que a árvore volte a ficar AVL novamente. Para isso existem quatro situações específicas, e para cada uma delas existe uma operação de rotação específica. São elas:

- ◆ Rotação simples à esquerda;
- ◆ Rotação simples à direita;
- ◆ Rotação dupla à esquerda;
- ◆ Rotação dupla à direita.

Estas rotações serão estudadas com mais detalhes nas próximas seções. Antes de entrarmos em detalhes das implementação da operações de rotação, considere o trecho de código mostrado no exemplo 51. Este exemplo mostra o código fonte dos métodos auxiliares utilizados pela classe AVLTree.

```

1  public class AvlTree {
2
3      private AvlNode root = null;
4
5      public AvlTree( ) {
6          root = null;
7      }
8
9      public void clear() {
10         root = null;
11     }
12     public boolean isEmpty() {
13         return root == null;
14     }
15
16     public AvlNode getRootNode () {
17         return root;
18     }
19
20     private static int height( AvlNode t ) {
21         if (t == null)
22             return -1;
23         else
24             return t.height;
25     }
26
27     private static int max( int lhs, int rhs ) {
28         if (lhs > rhs)
29             return lhs;
30         else
31             return rhs;
32     }
33
34     private int getFactor (AvlNode t) {
35         return height( t.left ) - height( t.right );
36     }

```

Exemplo 51 – Trecho do código fonte da classe AVLTree.

O método *height* definido nas linhas 20 a 25 retorna a altura da subárvore cuja raiz é o nodo passado como parâmetro. Esta altura é obtida através do atributo *height* definido na classe AVLNode.

O método *max* definido nas linhas 27 a 32 retorna o maior valor entre os dois valores passados como parâmetro.

O método *getFactor* definido nas linhas 34 a 36 recebe como parâmetro um nodo da árvore e retorna o fator deste nodo. Este fator é calculado com o auxílio do método

height.

Rotação simples à direita

Esta rotação deve ser aplicada toda vez que uma subárvore fica com um fator positivo fora do intervalo $-1, 0$ e 1 , e sua subárvore esquerda também possuir um fator positivo.

Considere a árvore mostrada na figura 98 (a). Ela é uma árvore binária do tipo AVL. Esta mesma árvore é mostrada na figura 98 (b) após a inserção do nodo com chave 8 . Note que o nodo com chave 11 ficou com seu fator igual a dois. Como ele está fora do intervalo aceito por uma árvore AVL e como seu fator é positivo e sua subárvore esquerda também possui um fator positivo, a rotação a ser aplicada é a rotação simples à direita.

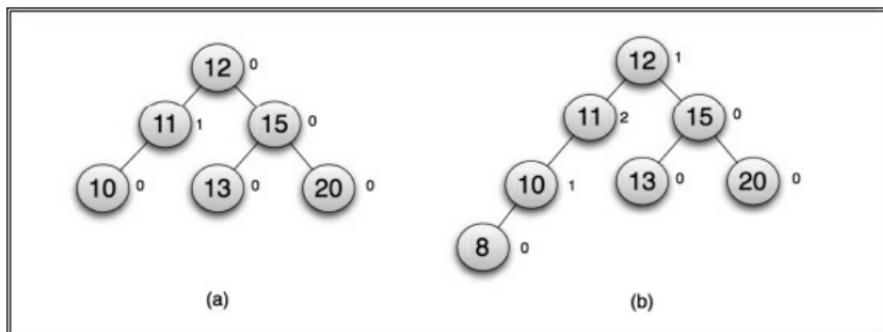


Figura 97 – (a) Árvore AVL antes da inserção do elemento com chave 8 e (b) Após a inserção do elemento com chave 8 .

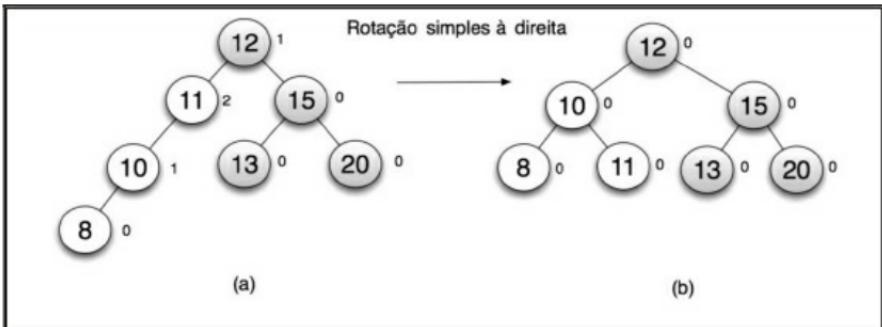


Figura 98 – (a) Árvore AVL antes da rotação simples à direita e
 (b) Árvore AVL após a rotação simples à direita realizada no nodo com chave 11.

Os passos realizados em uma rotação simples à direita são mostrados na figura 99. Nesta rotação o nodo com chave 11 é raiz da rotação, ou seja, é a raiz da subárvore na qual a rotação será aplicada.

Note que após a rotação a árvore voltou a ficar AVL novamente com todos os nodos com fator zero. Por definição, uma rotação simples à direita com raiz em um nodo K2 deve realizar os seguintes passos:

1. Seja K1 o filho esquerdo de K2;
2. Tornar o filho direito de K1 o filho esquerdo de K2;
3. Tornar K2 o filho direito de K1.

A figura 99 demonstra uma árvore AVL antes e depois da aplicação destes passos. Considere que s1, s2 e s3 são subárvores vazias ou não e que o nodo K2 é a raiz da rotação.

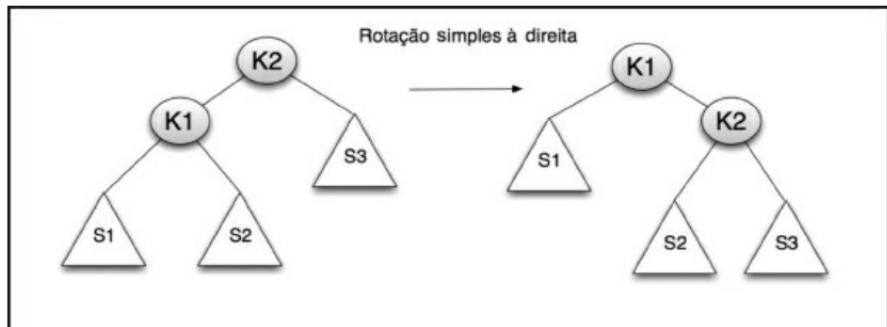


Figura 99 – Rotação simples à direita em uma árvore AVL.

O código fonte que implementa esta rotação em uma subárvore cuja raiz é o nodo K2 é mostrado no exemplo 52.

```
38 private static AvlNode doRightRotation( AvlNode k2 ) {  
39     AvlNode k1 = k2.left;  
40     k2.left = k1.right;  
41     k1.right = k2;  
42     k2.height = max( height( k2.left ), height( k2.right ) ) + 1;  
43     k1.height = max( height( k1.left ), k2.height ) + 1;  
44     return k1;  
45 }
```

Exemplo 52 – Método que implementa a rotação simples à direita.

Este método recebe como parâmetro a raiz da rotação e retorna a nova raiz da árvore após a operação de rotação. Nas linhas 41 e 42 as novas alturas dos nodos K1 e K2 são acertadas.

Os passos realizados em uma rotação simples à direita são mostrados na figura 100.

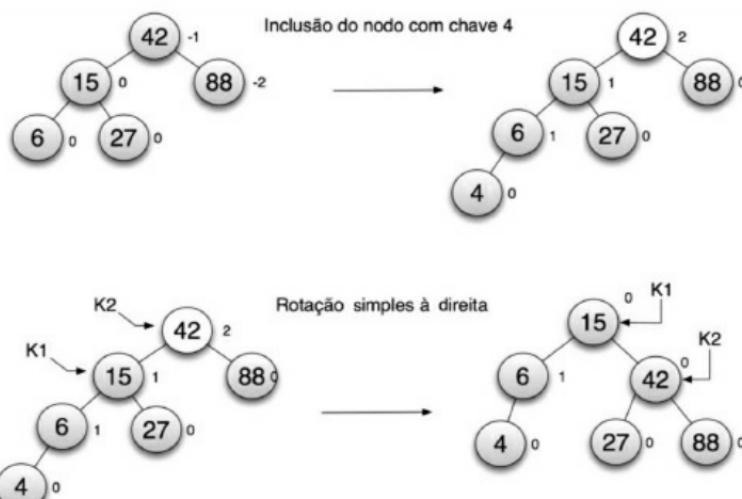


Figura 100 – Passos realizados em uma rotação simples à direita.

Rotação simples à esquerda

Esta rotação deve ser aplicada toda vez que uma subárvore ficar com um fator negativo fora do intervalo $-1, 0$ e 1 e sua subárvore direita também possuir um fator negativo.

Considere a árvore mostrada na figura 101 (a). Ela é uma árvore binária do tipo AVL. Esta mesma árvore é mostrada na figura 101 (b) após a inserção do nodo com chave 20. Note que o nodo com chave 13 ficou com seu fator igual a menos dois. Como ele está fora do intervalo aceito por uma árvore AVL e, como seu fator é negativo e sua subárvore direita também possui um fator negativo, a rotação a ser aplicada é a rotação simples à esquerda.

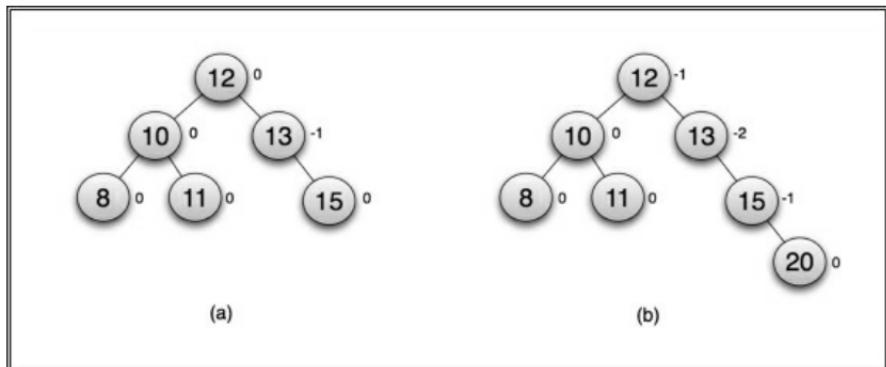


Figura 101 – (a) Árvore binária AVL antes da inserção do nodo com chave 20 e b) Após a inserção do nodo com chave 20.

Os passos realizados em uma rotação simples à esquerda são mostrados na figura 102. Nesta rotação o nodo com chave 13 é raiz da rotação, ou seja, é a raiz da subárvore na qual a rotação será aplicada.

Rotação simples à esquerda

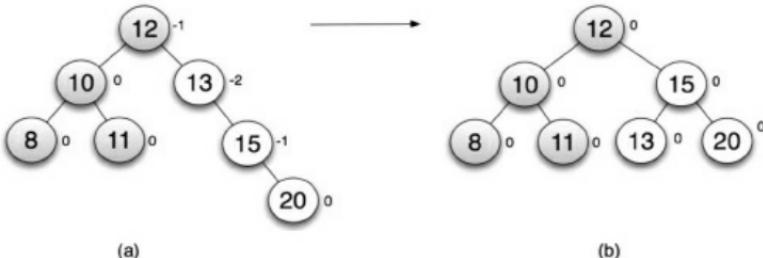


Figura 102 – (a) Árvore AVL antes da rotação simples à esquerda.

(b) Árvore AVL após a rotação simples à esquerda realizada no nodo com chave 13.

Note que após a rotação a árvore voltou a ficar AVL novamente com todos os nodos com fator zero. Por definição, uma rotação simples à esquerda com raiz em um nodo K1 deve realizar os seguintes passos:

1. Seja K2 o filho direito de K1;
2. Tornar o filho esquerdo de K2 o filho direito de K1;
3. Tornar K1 o filho esquerdo de K2.

A figura 103 mostra uma árvore AVL antes e depois da aplicação destes passos. Considere que s1, s2 e s3 são subárvores vazias ou não e que o nodo K1 é a raiz da rotação.

Rotação simples à esquerda

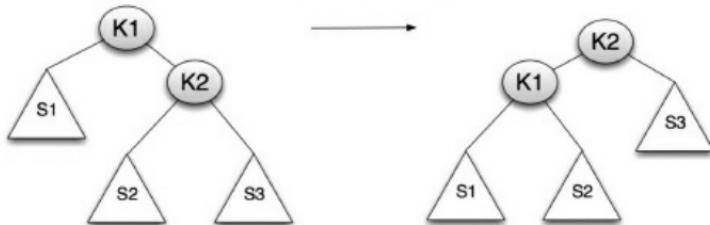


Figura 103 – Rotação simples à esquerda em uma árvore AVL

O código fonte que implementa esta rotação em uma subárvore cuja raiz é o nodo

K1 é mostrado no exemplo 53.

```
46     private static AvlNode doLeftRotation( AvlNode k1 ) {  
47         AvlNode k2 = k1.right;  
48         k1.right = k2.left;  
49         k2.left = k1;  
50         k1.height = max( height( k1.left ), height( k1.right ) ) + 1;  
51         k2.height = max( height( k2.right ), k1.height ) + 1;  
52         return k2;  
53     }
```

Exemplo 53 – Método que implementa uma rotação simples à esquerda.

Este método recebe como parâmetro a raiz da rotação e retorna a nova raiz da árvore após a operação de rotação. Nas linhas 50 e 51 as novas alturas dos nodos k1 e k2 são acertadas.

Os passos realizados em uma rotação simples à esquerda são mostrados na figura 104.

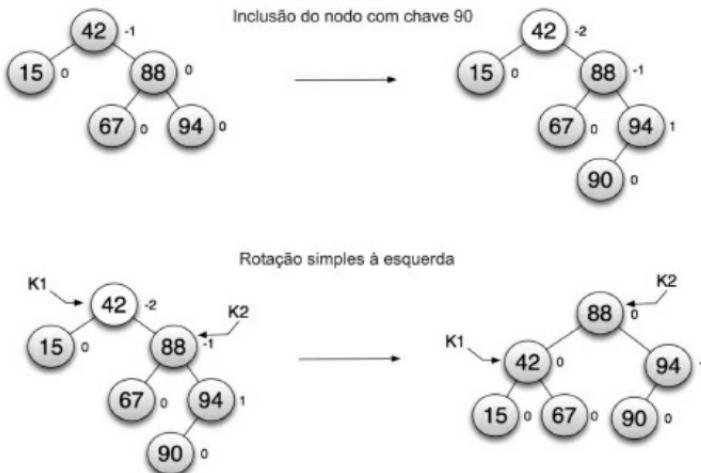
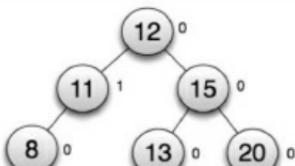


Figura 104 – Passos realizados em uma rotação simples à esquerda.

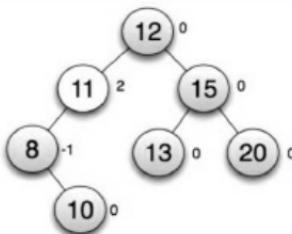
Rotação dupla à direita

Esta rotação deve ser aplicada sempre que uma subárvore ficar com um fator positivo, fora do intervalo $-1 \leq fator \leq 1$, e sua subárvore esquerda ficar com um fator negativo.

Uma rotação dupla tem como base as operações de rotação simples estudadas nas seções anteriores. Desta maneira, uma rotação dupla consiste da aplicação de duas rotações simples. Considere a árvore mostrada na figura 105 (a). Esta mesma árvore é mostrada na figura 105 (b) após a inserção do nodo com chave 10. Note que após esta inserção o nodo com chave 11 ficou com fator dois e sua subárvore esquerda ficou com fator negativo (menos um). Esta configuração requer uma rotação dupla à direita.



(a)



(b)

Figura 105 – (a) Árvore AVL antes da inserção do nodo com chave 10 e

(b) Após a inserção do nodo com chave 10.

Uma rotação dupla à direita consiste de uma rotação simples à esquerda na subárvore esquerda do nodo e, em seguida, uma rotação simples a direita no nodo. A figura 106 mostra estes passos aplicados na árvore da figura 105 (b).

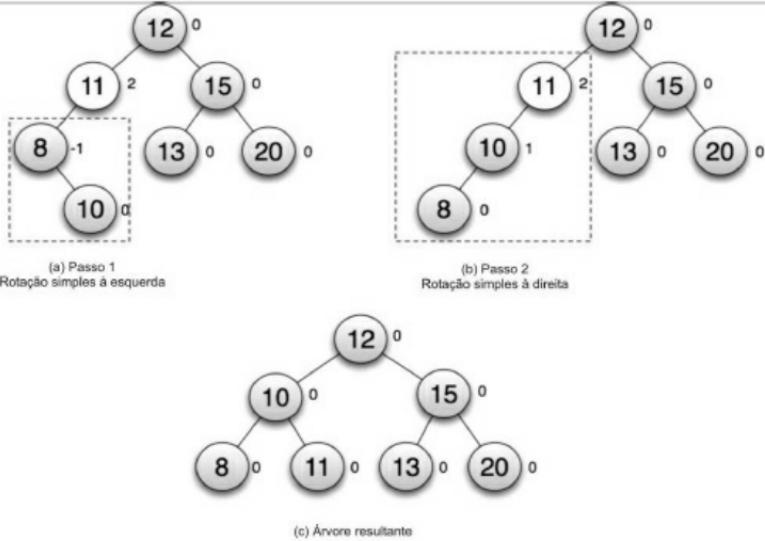


Figura 106 – Passos realizados em uma rotação dupla à direita.

- (a) Uma rotação simples à esquerda é realizada na subárvore esquerda do nodo com chave 11.
- (b) Uma rotação simples a direita é realizada no nodo com chave 11.
- (c) Árvore resultante após a operação de rotação.

Na figura 106 (a) uma rotação simples à esquerda é realizada no nodo com chave 8. A árvore resultante desta operação é mostrada na figura 106 (b). No segundo passo uma rotação simples à direita é aplicada no nodo com chave 11. A árvore resultante após a rotação dupla à direita é mostrada na figura 106 (c).

Por definição, uma rotação dupla a direita com raiz em um nodo K3 deve realizar os seguintes passos:

1. Seja K1 o filho esquerdo de K3;
2. Realizar uma rotação simples à esquerda em K1;
3. Realizar uma rotação simples a direita em k2.

A figura 107 mostra uma árvore AVL antes e depois da aplicação destes passos. Considere que S1, S2, S3 e S4 são subárvore vazias ou não e que o nodo K3 é a raiz da rotação.

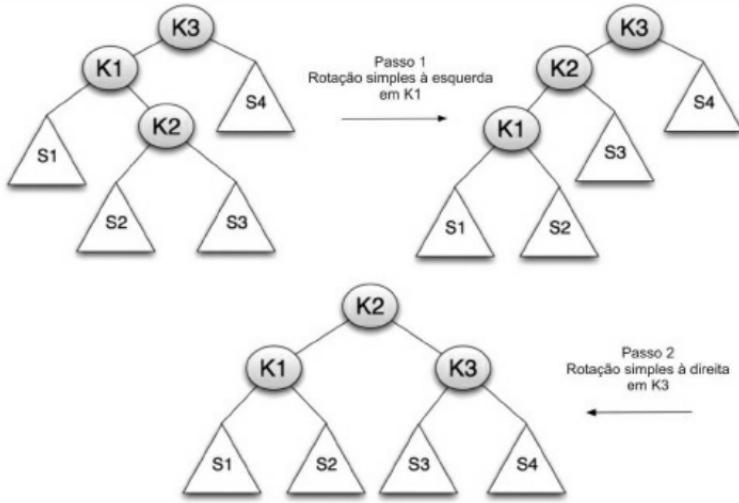


Figura 107 – Rotação dupla à direita em uma árvore AVL.
K3 é a raiz da rotação e S1, S2, S4 e S4 são subárvore vazias ou não.

O código fonte que implementa esta rotação em uma subárvore cuja raiz é o nodo K3 é mostrado no exemplo 54.

```

55 private static AvlNode doDoubleRightRotation( AvlNode k3 ) {
56     k3.left = doLeftRotation( k3.left );
57     return doRightRotation( k3 );
58 }
```

Exemplo 54 – Método que implementa uma rotação dupla à direita.

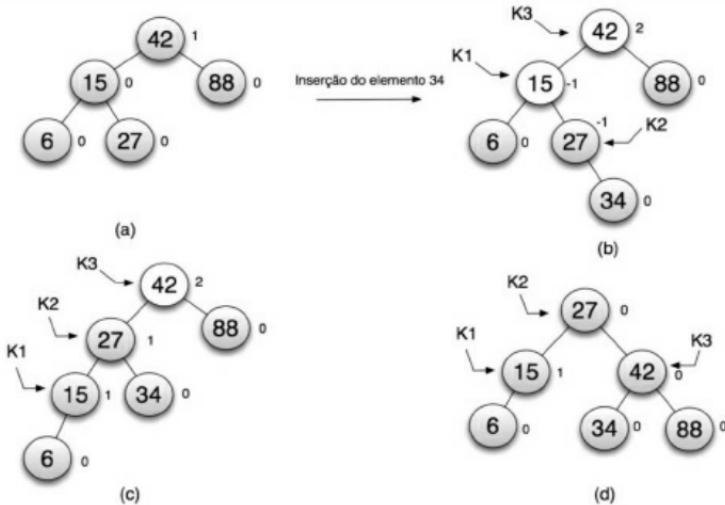


Figura 108 – Passos realizados em uma rotação dupla à direita.

Este método recebe como parâmetro a raiz da rotação e reutiliza os métodos previamente definidos para realizar as duas rotações simples em sequência e retorna a nova raiz da árvore após a operação de rotação.

Os passos realizados em uma rotação dupla à direita são mostrados na figura 108.

A árvore mostrada na figura 108 (a) é uma árvore AVL. Quando o nodo com chave 34, é inserido, a árvore deixa de ser AVL pois o fator do nodo raiz ficou com um valor fora do intervalo $-1, 0$ e 1 . Como o nodo com chave 42 possui fator positivo e sua subárvore esquerda possui fator negativo, uma rotação dupla à direita deve ser aplicada a este nodo. O primeiro passo desta rotação é mostrado na figura 108 (c), onde uma rotação simples à esquerda foi realizada no nodo com chave 15. A árvore resultante é mostrada na figura 108 (d), após a aplicação da rotação simples à direita no nodo com chave 42.

Rotação dupla à esquerda

Esta rotação deve ser aplicada sempre que uma subárvore ficar com um fator

negativo, fora do intervalo -1 , 0 e 1 , e sua subárvore direita ficar com um fator positivo. Esta rotação também utiliza como base as operações de rotação simples definidas anteriormente. Desta maneira, uma rotação dupla à esquerda consiste de uma rotação simples à direita seguida de uma rotação simples à direita. Considere a árvore mostrada na figura 109 (a). Esta mesma árvore é mostrada na figura 109 (b) após a inserção do nodo com chave 15 . Note que a árvore resultante desta inserção não é mais uma árvore AVL, pois o nodo com chave treze possui fator menos dois. Esta condição implica uma rotação dupla à esquerda, pois sua subárvore esquerda possui fator positivo.

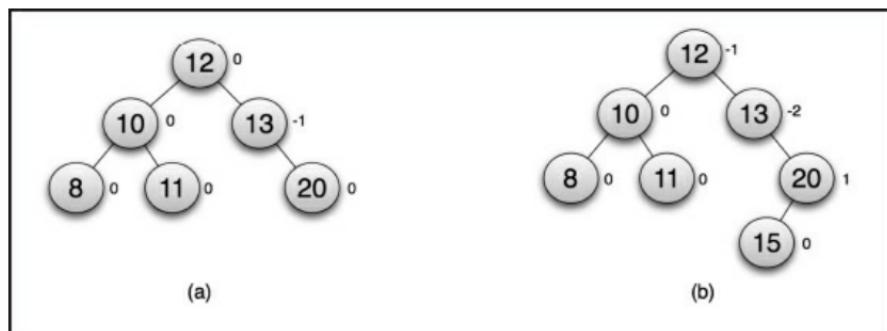


Figura 109 – {a) Árvore AVL antes da inserção do nodo com chave 15 e (b) Após a inserção.

Uma rotação dupla à esquerda consiste de uma rotação simples à direita no nodo e, em seguida, uma rotação simples à esquerda no nodo. A figura 110 mostra estes passos aplicados na árvore da figura 109 (a).

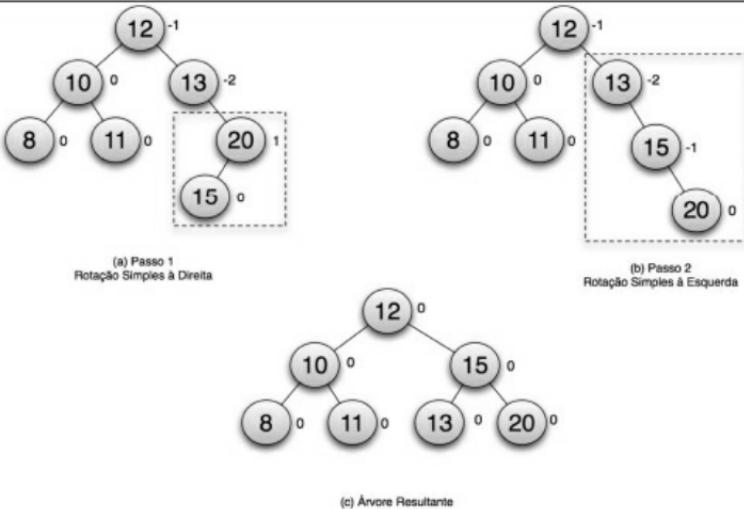


Figura 110 – Passos realizados em uma rotação dupla à esquerda. (a) Uma rotação simples à direita é realizada no nodo com chave 20. (b) Uma rotação simples à esquerda é realizada no nodo com chave 13. (c) Árvore resultante após a operação de rotação.

Na figura 110 (a) uma rotação simples à direita é realizada no nodo com chave 20. A árvore resultante desta operação é mostrada na figura 110 (b). No segundo passo uma rotação simples à esquerda é aplicada no nodo com chave 13. A árvore resultante após a rotação dupla à direita é mostrada na figura 110 (c).

Por definição, uma rotação dupla à direita com raiz em um nodo K1 deve realizar os seguintes passos:

1. Seja K3 o filho direito de K1;
2. Realizar uma rotação simples à direita em K3;
3. Realizar uma rotação simples à esquerda em K1.

A figura 111 mostra uma árvore AVL antes e depois da aplicação destes passos. Considere que S1, S2, S3 e S4 são subárvores vazias ou não e que o nodo K1 é a raiz da rotação.

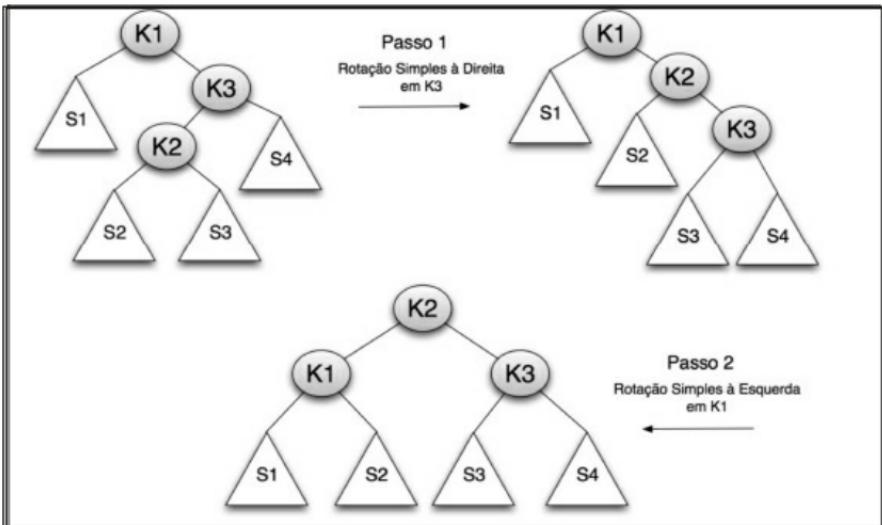


Figura 111 – Rotação dupla à esquerda em uma árvore AVL.
K1 é a raiz da rotação e S1, S2, S3 e S4 são subárvores vazias ou não.

O código fonte que implementa esta rotação em uma subárvore cuja raiz é o nodo K1 é mostrado no exemplo 55.

```

60 private static AvlNode doDoubleLeftRotation( AvlNode k1 ) {
61     k1.right = doRightRotation( k1.right );
62     return doLeftRotation( k1 );
63 }
```

Exemplo 55 – Método que implementa uma rotação dupla à esquerda.

Este método recebe como parâmetro a raiz da rotação e reutiliza os métodos previamente definidos para realizar as duas rotações simples em sequencia e retorna a nova raiz da árvore apôs a operação de rotação.

Os passos realizados em uma rotação dupla à direita são mostrados na figura 112.

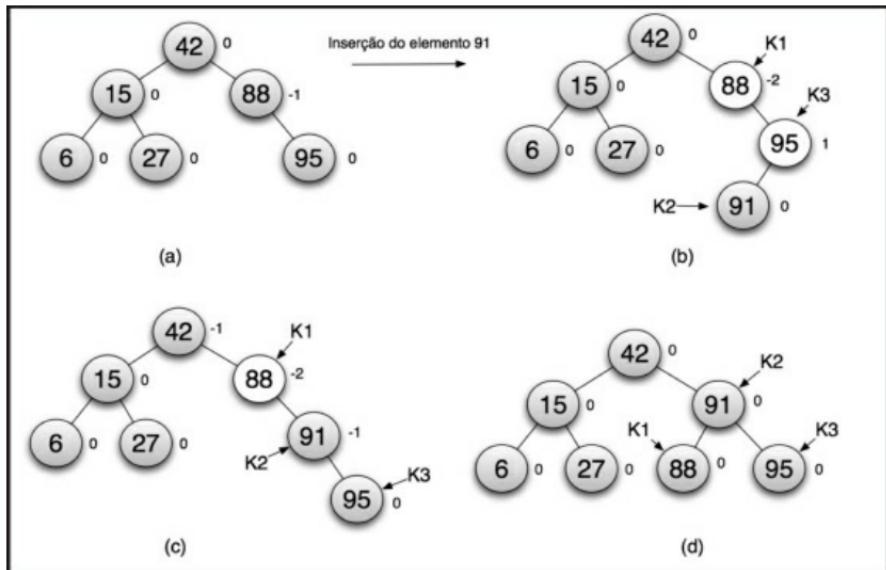


Figura 112 – Passos realizados em uma rotação dupla à esquerda.

A árvore mostrada na figura 112 (a) é uma árvore AVL. Quando o nodo com chave 91 é inserido, a árvore deixa de ser AVL, pois o fator do nodo com chave 88 ficou com um valor fora do intervalo $-1, 0$ e 1 . Como este nodo possui fator negativo e sua subárvore direita possui fator positivo, uma rotação dupla à esquerda deve ser aplicada a este nodo. O primeiro passo desta rotação é mostrado na figura 113 (c), em que uma rotação simples à direita foi realizada no nodo com chave 95. A árvore resultante é mostrada na figura 112 (d), após a aplicação da rotação simples à esquerda no nodo com chave 88.

Inserção em uma árvore AVL

O processo de inserção de um elemento em uma árvore binária de pesquisa AVL é dividido em duas partes: inicialmente, o elemento é inserido usando o método de inserção utilizado em árvores binárias de pesquisa e, após a inserção, a árvore precisa ser verificada a fim de verificar se ela continua平衡ada. Caso ela não continue平衡ada, as rotações apropriadas precisam ser aplicadas.

O algoritmo de inserção é mostrado no exemplo 56. Uma vez que o nodo foi

inserido na árvore, o fator de todos os nodos visitados durante a inserção precisa ser verificado. Este algoritmo implementa esta verificação de maneira recursiva, visitando todos os nodos pertencentes ao percurso realizado no processo de inserção e, para cada um deles, verificando o fator. Caso o fator de algum destes nodos esteja fora do intervalo $-1, 0, 1$, a rotação apropriada deve ser aplicada.

O código fonte do método da classe AVLTree que implementa o algoritmo de inserção em uma árvore AVL é mostrado no exemplo 56.

```
65     private AvlNode insert (int x, AvlNode t) {
66
67         if( t == null ) {
68             t = new AvlNode( x, null,null );
69         }
70         else if( x<t.key ) {
71             t.left = insert( x, t.left );
72         }
73         else if( x>t.key) {
74             t.right = insert( x, t.right );
75         }
76
77         if ( getFactor(t) == 2 ) {
78             if (getFactor (t.left)>0) t = doRightRotation( t );
79             else t = doDoubleRightRotation( t );
80         }
81         else if ( getFactor(t) == -2 ) {
82             if ( getFactor(t.right)<0 ) t = doLeftRotation( t );
83             else t = doDoubleLeftRotation( t );
84         }
85
86         t.height = max( height( t.left ), height( t.right ) ) + 1;
87
88     }
```

Exemplo 56 – Método que implementa a inserção em uma árvore AVL.

O lugar correto para a inserção do nodo é encontrado através das chamadas recursivas realizadas nas linhas 71 e 74. A inserção ocorre efetivamente quando um nodo folha (que será o pai do novo nodo sendo inserido) é encontrado. Neste ponto um novo nodo é instanciado com a chave a ser inserida na árvore (linha 78) e o método para se chamar a si mesmo. Após a inserção do novo elemento, o código das linhas 77 a 87 é executado para todos os nodos visitados durante o percurso realizado pelas chamadas recursivas. Este código verifica os fatores de todos os nodos e aplica as rotações apropriadas, caso seja necessário. Note que este algoritmo garante que o fator

de um nodo nunca será maior que dois e também nunca será menor que menos dois. O código na linha 86 modifica o atributo *height* de todos os nodos visitados durante o percurso realizado pela inserção.

Os passos realizados por este algoritmo em uma árvore AVL são mostrados na figura 113.

Na figura 113 (a) temos uma árvore AVL. Na figura 113 (b) o nodo com chave 99 é inserido. Os nodos visitados durante o processo de inserção deste nodo estão com linhas tracejadas. Após a inserção, o fator de todos estes nodos precisa ser recalculado. Isso acontece quando o algoritmo, após inserir o novo nodo, para de chamar a si mesmo, e o código das linhas 77 a 87 é executado. Este código vai ser executado para todos os nodos com linhas tracejadas na figura 113 (b). Na figura 113 (c), ao recalcular o fator do nodo com chave 88, o algoritmo detecta que este está fora do intervalo permitido e aplica a rotação correta. A árvore resultante é mostrada na figura 113 (d).

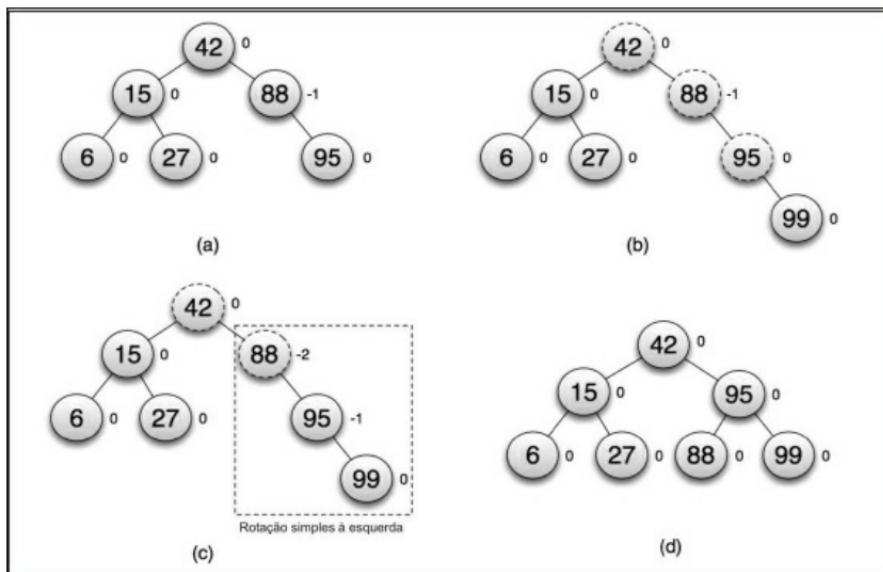


Figura 113 – Passos realizados pelo algoritmo de inserção.

No código fonte da classe AVLTree o método mostrado no exemplo 57 é utilizado para realizar a inserção de uma nova chave na árvore.

```
90 public boolean insert (int x) {  
91     root = insert (x, root);  
92     return true;  
93 }
```

Exemplo 57 – Versão sobre carregada do método insert que recebe como parâmetro apenas a chave a ser inserida.

Remoção em uma árvore AVL

A remoção em uma árvore AVL deve também garantir que árvore continue balanceada. Para isso é necessário que, ao remover um elemento da árvore, o fator de todos os nodos que pertencem ao caminho que inicia no nodo pai do nodo removido e termina na raiz da árvore seja recalculado. Caso algum destes fatores esteja fora do intervalo permitido a árvore deve ser balanceada.

Os passos realizados na deleção de um nodo de uma árvore AVL são mostrados abaixo. Considere a árvore mostrada na figura 114. Esta mesma árvore é mostrada na figura 115 após a deleção do nodo com chave 8.

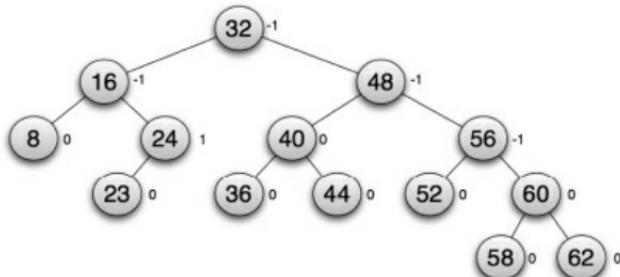


Figura 114 – Árvore AVL antes da remoção do nodo com chave oito.

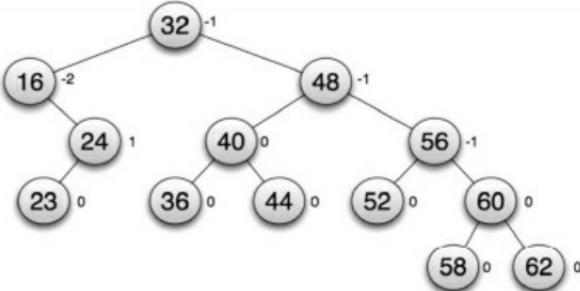


Figura 115 – Árvore AVL antes da deleção do nodo com chave oito.

Note que após a deleção do nodo com chave 8 a árvore deixou de ser AVL, pois o nodo com chave 16 ficou com um fator menos dois. Após a operação de rotação adequada ser aplicada (dupla à esquerda), a árvore resultante é mostrada na figura 116.

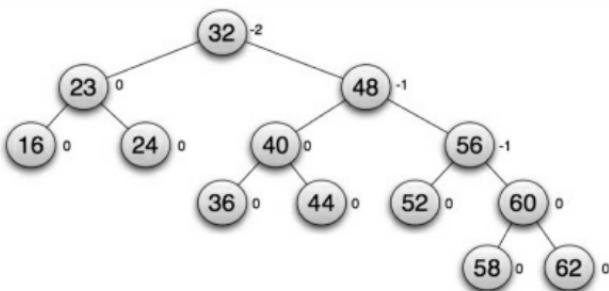


Figura 116 – Árvore AVL após a rotação dupla à esquerda realizada no nodo com chave 16.

Após esta rotação, note que a raiz da árvore ficou com o fator fora do intervalo AVL (menos dois). Como sua subárvore direita também possui um fator negativo, uma rotação simples à esquerda deve ser aplicada a este nodo. A árvore resultante após a aplicação desta rotação é mostrada na figura 117.

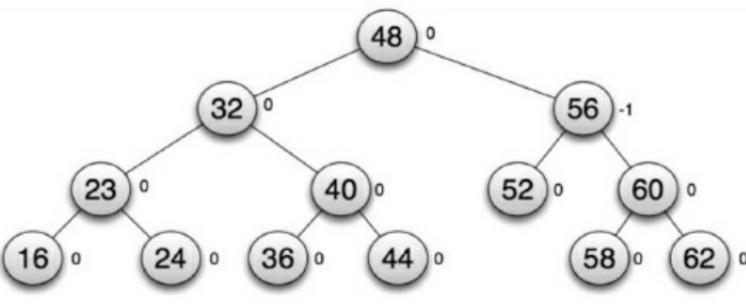


Figura 117 – Árvore AVL após a rotação simples à esquerda realizada no nodo raiz.

Este capítulo apresentou a estrutura de árvore AVL, que é um tipo de árvore binária de pesquisa balanceada. Para manter a árvore balanceada esta estrutura mantém informação sobre a forma da árvore através do cálculo do fator de cada um dos nodos da árvore. Este fator deve estar no intervalo $-1, 0, 1$ para que a árvore seja considerada balanceada. Esta restrição acrescenta uma complexidade aos algoritmos que manipulam a estrutura, mas garante que a árvore estará sempre balanceada, favorecendo assim as operações de busca, inserção e remoção.

O código fonte completo da classe AVLTree é mostrado no exemplo 58.

```
1  public class AvlTree {
2
3      private AvlNode root = null;
4
5      public AvlTree( ) {
6          root = null;
7      }
8
9      public void clear() {
10         root = null;
11     }
12     public boolean isEmpty() {
13         return root == null;
14     }
15
16     public AvlNode getRootNode () {
17         return root;
18     }
19
20     private static int height( AvlNode t ) {
21         if (t == null)
22             return -1;
23         else
24             return t.height;
25     }
26
27     private static int max( int lhs, int rhs ) {
28         if (lhs > rhs)
29             return lhs;
30         else
31             return rhs;
32     }
33
34     private int getFactor (AvlNode t) {
35         return height( t.left ) - height( t.right );
36     }
37
38     public boolean insert (int x) {
39         root = insert (x, root);
40         return true;
41     }
42
43     private AvlNode insert (int x, AvlNode t) {
44
45         if( t == null ) {
46             t = new AvlNode( x, null,null);
47         }
48         else if( x<t.key ) {
49             t.left = insert( x, t.left );
50         }
51         else if( x>t.key ) {
52             t.right = insert( x, t.right );
53         }
54
55         if ( getFactor(t) == 2 ) {
56             if (getFactor (t.left)>0) t = doRightRotation( t );
57             else t = doDoubleRightRotation( t );
58         }
59         else if ( getFactor(t) == -2 ) {
60             if ( getFactor(t.right)<0 ) t = doLeftRotation( t );
61             else t = doDoubleLeftRotation( t );
62         }
63
64         t.height = max( height( t.left ), height( t.right ) ) + 1;
65         return t;
66     }
67 }
```

```
68     private static AvlNode doRightRotation( AvlNode k2 ) {
69         AvlNode k1 = k2.left;
70         k2.left = k1.right;
71         k1.right = k2;
72         k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
73         k1.height = max( height( k1.left ), k2.height ) + 1;
74         return k1;
75     }
76
77     private static AvlNode doLeftRotation( AvlNode k1 ) {
78         AvlNode k2 = k1.right;
79         k1.right = k2.left;
80         k2.left = k1;
81         k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
82         k2.height = max( height( k2.right ), k1.height ) + 1;
83         return k2;
84     }
85
86     private static AvlNode doDoubleRightRotation( AvlNode k3 ) {
87         k3.left = doLeftRotation( k3.left );
88         return doRightRotation( k3 );
89     }
90
91     private static AvlNode doDoubleLeftRotation( AvlNode k1 ) {
92         k1.right = doRightRotation( k1.right );
93         return doLeftRotation( k1 );
94     }
95
96     public AvlNode search(int el) {
97         return search(root,el);
98     }
99     protected AvlNode search(AvlNode p, int el) {
100        while (p != null) {
101            if (el==p.key) return p;
102            else if (el<p.key) p = p.left;
103            else p = p.right;
104        }
105        return null;
106    }
107
108    public void inorder() {
109        inorder(root);
110    }
111    protected void inorder(AvlNode p) {
112        if (p != null) {
113            inorder(p.left);
114            System.out.print(p.key+" - ");
115            inorder(p.right);
116        }
117    }
118
119    public void preorder() {
120        preorder(root);
121    }
122    protected void preorder(AvlNode p) {
123        if (p != null) {
124            System.out.print(p.key + " ");
125            preorder(p.left);
126            preorder(p.right);
127        }
128    }
```

Exemplo 58 – Código fonte da classe AVLTree.

CAPÍTULO 10

ÁRVORES GENÉRICAS

As estruturas de árvores estudadas nos capítulos anteriores impõem uma restrição quanto ao grau máximo da árvore. Tanto árvores binárias quanto árvores binárias de pesquisa e também árvores AVL permitem que cada nodo da árvore possua no máximo dois filhos, ou seja, grau dois. Neste capítulo estudaremos uma estrutura de árvore que não apresenta esta restrição, possibilitando assim que cada nodo da árvore possua um número arbitrário de filhos. Esta estrutura é conhecida como árvore genérica e pode ser utilizada, por exemplo, para representar uma árvore de diretórios.

A figura 118 mostra uma árvore genérica. Nesta figura, o nodo com chave G possui grau três, o nodo com chave F possui grau quatro e o nodo com chave L possui grau zero (folha).

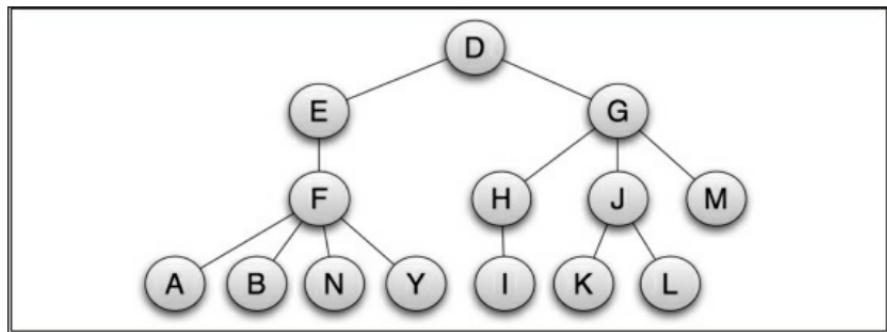


Figura 118 – Árvore Genérica.

Nesta estrutura não existe o conceito de filho esquerdo e filho direito, uma vez que o grau de cada nodo é arbitrário.

Uma árvore genérica pode ser definida como um conjunto T , finito e não vazio de nodos, com as seguintes propriedades:

1. Um nodo especial da árvore, r , é chamado raiz da árvore; e
2. O restante dos nodos é particionado em $n \geq 0$ subconjuntos T_1, T_2, \dots, T_n , cada um dos quais é uma árvore.

Esta definição, embora não permita uma árvore genérica vazia, não limita o uso desta estrutura em aplicações reais. Uma árvore de diretórios, por exemplo, nunca é vazia, uma vez que sempre existirá um diretório raiz.

Um nodo folha em uma árvore genérica é um nodo que não possui nenhuma subárvore. No caso da figura 118, os nodos A, B, N, Y, I, K, L e M são nodos folha, pois estes não possuem nenhuma subárvore.

A implementação de uma árvore genérica pode ser realizada de diferentes maneiras. Iremos considerar uma implementação em que a chave associada a cada nodo é uma variável do tipo *Object* e os filhos são armazenados em uma lista simplesmente encadeada. A figura 119 mostra como a árvore da figura 118 fica armazenada na memória. A ideia básica é que a cada nodo da árvore está associada uma lista simplesmente encadeada que irá armazenar suas subárvore.

Os métodos da classe *GeneralTree*, que implementa uma árvore genérica, são mostrados na tabela 12.

Método	Descrição
public Object getKey()	Retorna a chave associada à árvore.
public GeneralTree getSubtree(int i)	Retorna a i-ésima subárvore desta árvore
public void attachSubtree (GeneralTree t)	Insere a subárvore especificada por t como subárvore da árvore corrente.
public GeneralTree detachSubtree(GeneralTree t)	Remove e retorna a subárvore especificada pelo parâmetro t.

Tabela 12 – Métodos da classe *GeneralTree* que implementa uma árvore genérica.

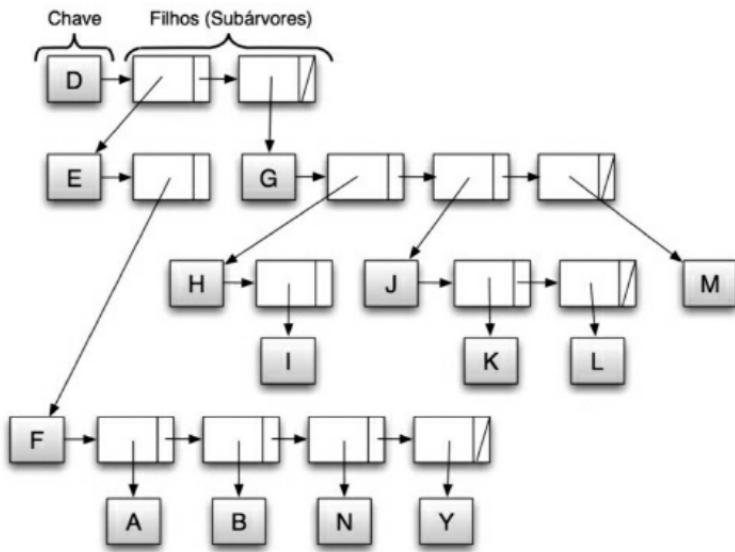


Figura 119 – Árvore genérica da figura 118 representada na memória.

A implementação da classe *GeneralTree* é mostrada no exemplo 59.

```
1  public class GeneralTree {
2
3      protected Object key;
4      protected int degree;
5      protected List list;
6
7      public Object getKey () {
8          return key;
9      }
10
11     public GeneralTree (Object key) {
12         this.key = key;
13         degree = 0;
14         list = new List ();
15     }
16
17     public GeneralTree getSubtree (int i) {
18         if (i < 0 || i >= degree)
19             throw new IndexOutOfBoundsException ();
20         Node ptr = list.getFirst ();
21         for (int j = 0; j < i; ++j)
22             ptr = ptr.getNext ();
23         return (GeneralTree) ptr.getData ();
24     }
25
26     public void attachSubtree (GeneralTree t) {
27         list.insertAtBack (t);
28         ++degree;
29     }
30
31     public GeneralTree detachSubtree (GeneralTree t) {
32         list.remove (t);
33         --degree;
34         return t;
35     }
36
37     public boolean equals(Object obj) {
38         GeneralTree t = (GeneralTree) obj;
39         if (t.getKey ().equals(this.getKey ())) {
40             return true;
41         }
42         return false;
43     }
44 }
```

Exemplo 59 – Código fonte da classe *GeneralTree* que implementa uma árvore genérica.

A classe *GeneralTree* define três atributos: *key* do tipo *Object*, que é utilizado

para armazenar uma referência à chave do nodo na linha 3, *degree* do tipo inteiro, que é utilizado para armazenar o grau da árvore (quantidade de subárvores ou filhos), e List na linha 5, que é uma lista simplesmente encadeada utilizada para armazenar as subárvores da árvore especificada.

O construtor da classe *GeneralTree*, definido nas linhas 11 a 15, recebe como parâmetro a chave a ser armazenada na árvore, inicializa o atributo *de-gree* com o valor zero e instancia a lista simplesmente encadeada utilizada para armazenar as subárvores.

O método *getKey*, definido nas linhas 7 a 9, é um método de acesso que retorna o objeto armazenado como chave na árvore.

O método *getSubTree*, definido nas linhas 16 a 24, recebe como parâmetro um inteiro que representa o índice da subárvore a ser retornada. Caso este índice seja menor que zero ou maior que o atributo *degree*, uma exceção é lançada. Do contrário, a lista encadeada de subárvores é percorrida e a subárvore armazenada na posição *i* da lista é retornada. Em sua implementação o método *getSubTree* faz uso do método *getSize* da classe de lista simplesmente encadeada. Este método retorna um número inteiro que representa a quantidade de elementos contidos na lista. O código fonte deste método é mostrado no exemplo 60.

```
1  public int getSize(){
2      Node current = firstNode;
3      int size = 0;
4      while(current != null){
5          current = current.getNext();
6          size = size + 1;
7      }
8      return size;
9  }
```

Exemplo 60 – Código fonte do método *getSize* que retorna o tamanho de uma lista simplesmente encadeada.

O método *attachSubtree* adiciona a árvore passada como parâmetro como subárvore da árvore que chamou o método. Para isso ele insere a árvore passada como parâmetro na lista encadeada de subárvores da árvore. Ao final de sua execução, o método *attachSubtree* incrementa o atributo *degree*, indicando que uma nova subárvore foi adicionada.

O método *detachSubtree* remove a árvore passada como parâmetro da lista da subárvore que chamou o método. Para isso ele percorre a lista encadeada de

subárvores à procura da árvore t através da chamada do método $remove$ da lista simplesmente encadeada. O código fonte deste método é mostrado no exemplo 61. Ao final de sua execução, o método $detachSubtree$ decrementa o atributo $degree$ indicando que uma subárvore foi removida.

```
1 public boolean remove(Object o){  
2  
3     Node currentNode = firstNode;  
4     Node previous = firstNode;  
5  
6     while (currentNode != null){  
7         if (currentNode.getData().equals(o)){  
8             if (currentNode == firstNode){  
9                 firstNode = firstNode.getNext();  
10                return true;  
11            }  
12            previous.setNext(currentNode.getNext());  
13            if (currentNode == lastNode){  
14                lastNode = previous;  
15            }  
16            currentNode = null;  
17            return true;  
18        }  
19        previous = currentNode;  
20        currentNode = currentNode.getNext();  
21    }  
22    return false;  
23 }
```

Exemplo 61 – Código fonte do método $remove$ que remove um elemento de uma lista simplesmente encadeada.

O método $equals$, definido nas linhas 37 a 43, é utilizado pelo método $remove$ da lista simplesmente encadeada. Este método é utilizado para verificar se duas árvores genéricas contêm a mesma chave.

Este capítulo apresentou a estrutura de árvore genérica, que é uma estrutura de árvore que não apresenta restrição quanto ao número de filhos. Foi apresentada também uma classe que implementa esta estrutura onde uma lista simplesmente encadeada é utilizada para armazenar subárvores.

CAPÍTULO 11

MÉTODOS DE PESQUISA

A manipulação de uma grande quantidade de dados é uma tarefa bastante comum em diferentes tipos de programas. Em muitas situações os dados estão organizados em estruturas chamadas *tabelas*.

Uma tabela é formada por uma coleção de entradas, cada uma delas formada por um conjunto de campos. Normalmente, uma tabela armazena informações sobre vários itens de um mesmo tipo, onde cada um corresponde a uma entrada da tabela. Considere, por exemplo, uma tabela que armazena os dados de uma turma de alunos. Nesta tabela, cada entrada corresponderia aos dados de um aluno em específico como mostra a tabela 13.

Núm. Mat.	Nome	Data Nasc.	E-mail
1001	José	25.07.1980	1001@unisinos.br
1002	Paulo	23.12.1081	1002@unisinos.br
1003	Pedro	15.06.1991	1003@unisinos.br
1004	Bruno	22.04.1978	1004@unisinos.br
1005	Lucas	14.08.1987	1005@unisinos.br
1006	Felipe	25.05.1979	1006@unisinos.br

Tabela 13 – Tabela com dados de uma turma de alunos.

Na tabela mostrada acima as entradas são divididas em quatro campos:

- Núm. Mat., associado ao atributo número de matrícula do aluno;
- Nome, associado ao atributo nome do aluno;
- Data Nasc., associado ao atributo data de nascimento do aluno;
- E-mail, associado ao atributo e-mail do aluno.

Normalmente, pelo menos um dos campos da tabela constitui o que chamamos de *chave primária*, ou seja, um campo que apresenta um valor distinto para todas as

entradas da tabela. No caso da tabela de exemplo, podemos dizer que este campo é o campo “Núm. Mat.”, visto que cada aluno possui um número de matrícula diferente. Desta maneira, dada uma chave primária, podemos identificar de maneira não ambígua uma única entrada da tabela que corresponde a esta chave, caso ela esteja presente na tabela.

Uma das operações mais comuns e importantes no processamento de dados é a consulta, ou pesquisa, a um determinado dado em uma tabela. Esta operação consiste em acessar uma determinada entrada da tabela a partir de uma determinada chave. Neste capítulo trataremos de dois métodos de pesquisa: pesquisa sequencial e pesquisa binária.

Para efeitos de implementação nos exemplos apresentados, por estarmos utilizando a linguagem de programação Java, consideraremos que uma tabela é um vetor de elementos, normalmente números inteiros. Cada um destes números representa uma chave da tabela. Em uma aplicação real, esta chave denotada por um número inteiro pode, por exemplo, ser substituída por um objeto do domínio da aplicação, sendo desenvolvida sem maiores problemas.

Pesquisa sequencial

A pesquisa sequencial é um método de pesquisa bastante simples que consiste em percorrer de maneira serial a tabela à procura da chave desejada. Durante a procura a chave passada como argumento é comparada com cada uma das chaves contidas na tabela. A procura termina quando a chave é encontrada ou quando a procura chega ao final da tabela, indicando assim que a chave procurada não está na tabela.

O exemplo 62 mostra a implementação do método de pesquisa sequencial em um vetor de números inteiros.

```
1 public static int pesquisaSequencial (int tab[], int arg) {
2     int i = 0;
3     while (i < tab.length) {
4         if (tab[i] == arg)
5             return i;
6         else i = i + 1;
7     }
8     return -1;
9 }
```

Exemplo 62 – Método que implementa pesquisa sequencial em um vetor de inteiros.

Este método recebe como parâmetro uma referência ao vetor e o argumento a ser procurado. O laço das linhas 3 a 7 executa até que todas as entradas do vetor tenham sido verificadas ou até que o argumento seja encontrado. Caso o argumento seja encontrado, a sua posição no vetor (índice) é retornado. Caso contrário, o valor menos um é retornado. Considerando o vetor da figura 120, caso quiséssemos procurar pelo elemento 7, teríamos que realizar seis comparações para afirmar que este elemento se encontra no vetor.

5	20	12	1	2	7	32	22	31	16
0	1	2	3	4	5	6	7	8	9

Figura 120 – Vetor de inteiros com dez elementos (não ordenado).

Ainda, caso quiséssemos procurar pelo elemento 6, teríamos que realizar dez comparações para afirmar que o elemento não se encontra no vetor. O desempenho deste método é bastante modesto, visto que o número médio de comparações (entradas examinadas) para a localização de uma entrada qualquer é dado pela seguinte fórmula, onde N é o tamanho do vetor:

$$EE = \frac{N + 1}{2}$$

Este algoritmo apresenta um desempenho um pouco melhor se considerarmos um vetor ordenado pelo valor da chave. Desta maneira, a procura por uma chave pode encerrar quando esta for encontrada, quando uma chave com um valor maior que o valor da chave procurada for encontrada ou quando a procura chegar ao final da tabela.

O exemplo 63 mostra a implementação do método de pesquisa sequencial em um vetor com números inteiros, considerando que as chaves estão ordenadas de maneira crescente.

```
1 public static int pesquisaSequencialOrdenada (int tab[], int arg) {  
2     int i = 0;  
3     while ((i < tab.length) && (tab[i] <= arg)) {  
4         if (tab[i] == arg) return i;  
5         else i = i + 1;  
6     }  
7     return -1;  
8 }
```

Exemplo 63 – Método que implementa a pesquisa sequencial em um vetor de inteiros com as chaves ordenadas de maneira crescente.

Considerando o mesmo vetor da figura 120, agora com as chaves ordenadas de maneira crescente, caso quiséssemos procurar pelo elemento 7, teríamos que realizar quatro comparações para afirmar que este elemento se encontra no vetor. No caso da procura pelo elemento 6, ainda teríamos que realizar dez comparações para afirmar este não se encontra no vetor.

1	2	5	7	12	16	20	22	31	32
0	1	2	3	4	5	6	7	8	9

Figura 121 – Vetor de inteiros com dez elementos (ordenado de maneira crescente).

O método mostrado no exemplo 63 ainda pode ser otimizado. Um teste adicional pode ser realizado para verificar se o elemento sendo procurado é maior que o último elemento do vetor, o que indica que o elemento procurado não está no vetor.

Pesquisa binária

A pesquisa binária é um método de pesquisa que deve ser aplicado a uma tabela ordenada. O método consiste na comparação do argumento de pesquisa com a chave localizada no meio da tabela. Se o argumento for igual a esta chave, a busca termina. Se o argumento for maior, o processo é repetido para a metade superior. Se o argumento for menor, o processo é repetido para a metade inferior. Utilizando esta abordagem, a cada comparação a área de pesquisa é sempre reduzida à metade do número de elementos.

O exemplo 64 mostra a implementação do método de pesquisa binária em um vetor de números inteiros ordenado de maneira crescente.

```
1  public static int pesquisaBinaria (int tab[], int arg) {
2      int inf, sup, med;
3      inf = 0;      sup = tab.length-1;
4      while (inf <= sup) {
5          med = (inf + sup)/2;
6          if (arg == tab[med]) return med;
7          else if (arg > tab[med]) inf = med + 1;
8          else if (arg < tab[med]) sup = med - 1;
9      }
10     return -1;
```

Exemplo 64 – Método que implementa a pesquisa binária em um vetor de números inteiros ordenado de maneira crescente.

O método pesquisa binária define três variáveis locais do tipo inteiro, utilizadas para armazenar os endereços inferior, superior e médio respectivamente. Inicialmente, a variável inf é inicializada com zero (primeiro elemento do vetor), e a variável sup é inicializada com o último índice válido do vetor ($length - 1$). A cada iteração do laço das linhas 4 a 9 o índice médio da área de busca é calculado e atribuído à variável med. Caso o valor neste índice seja igual a arg, a busca termina com sucesso (linha 6). Caso arg seja maior que o elemento armazenado na posição med, a busca continua na metade superior do vetor. Caso ele seja menor, a busca continua na metade inferior. Caso a busca termine sem sucesso, o método retorna o valor menos um.

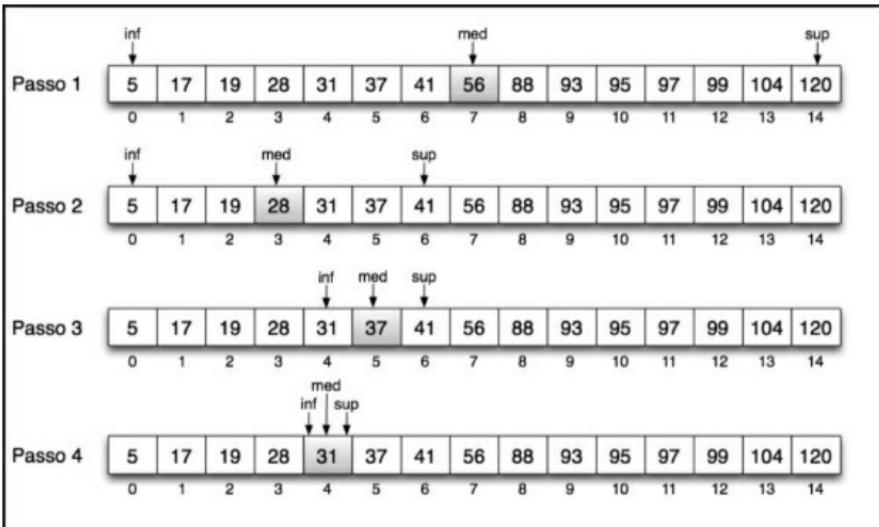


Figura 122 – Posições examinadas pelo algoritmo de pesquisa binária na procura pela chave 31.

A figura 122 mostra os passos realizados pelo método de busca binária em um vetor de quinze posições na procura pela chave 31. As posições visitadas estão indicadas pelo índice med em cada um dos passos.

A figura 123 mostra os passos realizados pelo método de busca binária no mesmo vetor de quinze posições na procura pela chave 98. As posições visitadas estão indicadas pelo índice med em cada um dos passos.

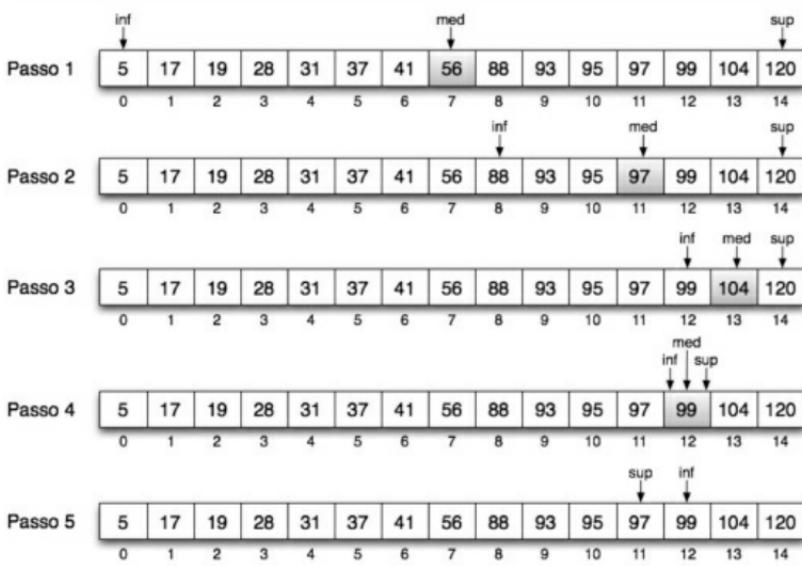


Figura 123 – Posições examinadas pelo algoritmo de pesquisa binária na procura pela chave 98.

No exemplo da figura 123 o elemento procurado não foi encontrado no vetor. Note que no passo cinco nenhuma posição é examinada, pois a condição do laço ficou falsa e o algoritmo terminou sua execução. Nesta caso, com apenas quatro comparações pudemos afirmar que o elemento com chave 98 não se encontra no vetor.

O desempenho do algoritmo de pesquisa binária é muito melhor que o algoritmo de pesquisa sequencial, pois com poucas comparações pode-se afirmar se uma determinada chave está ou não na tabela. Analisando este algoritmo, podemos verificar que na primeira iteração a área de pesquisa é n , onde n é o tamanho do vetor. Na segunda iteração esta área é $n/2$. Na terceira é $n/4$, e assim sucessivamente. Desta maneira, na $(k+1)$ -ésima iteração a área de busca será de aproximadamente $n/2^k$. Quando $k > \log_2 n$, temos que $n/2^k < 1$ e a execução para. Sendo assim, o número aproximado de iterações (comparações) para um vetor de n posições será de $2 * \log_2 n$. Isso porque a cada iteração estamos fazendo duas comparações.

A figura 124 mostra um comparativo entre o número médio de comparações para ambos os métodos, pesquisa sequencial e pesquisa binária, considerando diferentes tamanhos de tabelas.

Tamanho da tabela	Número médio de comparações	
	Pesquisa sequencial (n/2)	Pesquisa binária $2 \cdot \log_2 n$
10	5	4
100	50	7
500	250	9
1000	500	10
10000	5000	14
100000	50000	17
1000000	500000	20
10000000	5000000	24

Figura 124 – Comparativo entre os métodos de pesquisa binária e de pesquisa sequencial

Este capítulo apresentou os métodos de pesquisa sequencial e de pesquisa binária. O método de pesquisa sequencial, embora menos eficiente, não exige nenhuma preparação dos dados. Estes podem estar sem nenhuma ordem definida. Este método é uma alternativa para tabelas de tamanho pequeno. Entretanto, quando o tamanho da tabela cresce, o método de pesquisa binária mostra-se muito mais eficiente como mostra a figura 124.

CAPÍTULO 12

CLASSIFICAÇÃO DE DADOS

Este capítulo introduz o conceito de classificação de dados, que constitui uma das tarefas mais frequentes e importantes no processamento de dados. O acesso a dados é uma tarefa que programas executam frequentemente, e a classificação, ou ordenação, destes dados pode ser vista como uma preparação, ou um pré-processamento que objetiva tornar mais simples as tarefas posteriores que um programa venha a realizar. Considere o processo de ordenação, por exemplo, como um pré-processamento para a posterior aplicação da pesquisa binária, que assume que os dados estejam ordenados. Ou, ainda, imagine um dicionário onde as palavras não estivessem classificadas alfabeticamente. Com certeza o trabalho para procurar uma palavra em específico seria muito maior!

Neste capítulo continuaremos utilizando as definições do capítulo anterior, onde uma tabela representa um conjunto de registros, às vezes também chamado de arquivo. Cada um destes registros possui um campo distinto, chamado de chave, e também informações satélite associadas. A classificação de dados é o processo de organizar os registros em ordem crescente ou decrescente, segundo algum critério. Para os exemplos estudados neste capítulo, para efeitos didáticos, estaremos considerando registros compostos apenas uma chave numérica. Para isso utilizaremos vetores que armazenam números inteiros.

O resultado de um processo de classificação é a tabela ordenada. Existem diferentes formas de representar o resultado de um processo de ordenação. A seguir três formas serão apresentadas:

- ◆ Reorganização física;
- ◆ Vetor indireto de ordenação;
- ◆ Encadeamento.

Quando a reorganização física é utilizada, as chaves são fisicamente rearranjadas. Ao final da execução as chaves estarão fisicamente ordenadas. A figura 125 mostra um vetor antes e depois da reorganização física. Note que esta abordagem envolve a movimentação das chaves no vetor, o que pode significar um custo bastante elevado se as entradas forem grandes (registros com muitos dados satélite)

0	7
1	5
2	3
3	9
4	4

(a)

0	3
1	4
2	5
3	7
4	9

(b)

Figura 125 – (a) Vetor antes da ordenação. (b) Após a reorganização física com as chaves rearranjadas.

Utilizando um vetor indireto de ordenação, as chaves não são fisicamente rearranjadas. Elas são mantidas em suas posições originais e o vetor ordenado é especificado por um segundo vetor (chamado vetor indireto de ordenação), que é criado durante o processo de ordenação. A figura 126 mostra um vetor e o vetor indireto de ordenação resultante do processo de ordenação.

0	7
1	5
2	3
3	9
4	4

(a)

0	2
1	4
2	1
3	0
4	3

(b)

Figura 126 – (a) Vetor não ordenado. (b) Vetor indireto de ordenação gerado.

O valor do i -ésimo elemento do vetor indireto de ordenação é igual ao i -ésimo índice do vetor original.

Utilizando encadeamento, as chaves também são mantidas em suas posições originais, sendo o resultado do processo de ordenação o encadeamento das chaves do vetor original. Embora esta abordagem não utilize um segundo vetor, um campo adicional em cada entrada é necessário para indicar a sequência ordenada.

Início: 2

	0	7		0	7	3
	1	5		1	5	0
	2	3		2	3	4
	3	9		3	9	-1
	4	4		4	4	1
	(a)			(b)		

Figura 127 – (a) Vetor original. (b) Vetor ordenado com encadeamento. A primeira posição está armazenada em uma variável externa. A lista inicia no índice dois.

Nesta abordagem o endereço da primeira entrada pode ser indicado por uma variável externa.

A classificação de dados foi assunto de extensa pesquisa na ciência da computação e alguns métodos muito sofisticados foram desenvolvidos. Neste capítulo veremos quatro métodos de ordenação. Três deles são mais simples, apropriados para conjuntos pequenos de dados, e um mais sofisticado, mais apropriado a conjuntos maiores de dados.

Inserção direta

O algoritmo de inserção direta é um algoritmo que pertence à classe dos algoritmos de ordenação por inserção. A ordenação dos dados ocorre porque estes são inseridos na posição correta. É um método bastante simples, normalmente utilizado para um conjunto pequeno de dados por apresentar uma baixa eficiência.

Este algoritmo divide o vetor em dois segmentos. O primeiro segmento contém os elementos já ordenados e o segundo segmento contém os elementos ainda não ordenados. O algoritmo inicia com o primeiro segmento contendo apenas uma posição, consequentemente já ordenado e com o segundo segmento contendo o restante do vetor. A seguir, por meio de sucessivas iterações, cada elemento do segundo segmento é inserido ordenadamente no primeiro, até que todos os elementos estejam no primeiro segmento.

O código fonte do método que implementa o método de inserção direta é mostrado no exemplo 65.

```
1  public static void insertionSort (int a[]) {  
2      for (int i = 1; i < a.length; i++) {  
3          int j = i;  
4          int B = a[i];  
5          while ((j > 0) && (a[j-1] > B)) {  
6              a[j] = a[j-1];  
7              j--;  
8          }  
9          a[j] = B;  
10     }  
11 }
```

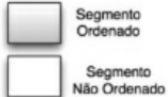
Exemplo 65 – Método que implementa o algoritmo de inserção direta em um vetor de números inteiros.

O método recebe como parâmetro um vetor de números inteiros e já considera o primeiro do elemento do vetor como ordenado (primeiro segmento). A atribuição da linha 3 guarda o índice do primeiro elemento do segmento não ordenado na variável j. A atribuição da linha 4 guarda o valor do primeiro elemento do segmento não ordenado na variável B. O laço das linhas 5 a 8 encontra a posição correta do primeiro elemento do segmento não ordenado no segmento ordenado.

A figura 128 mostra o funcionamento passo a passo deste método em um vetor de números inteiros. Cada passo mostrado na figura representa uma iteração do laço mais externo do algoritmo.

O algoritmo de inserção direta, apesar de simples, apresenta uma baixa eficiência. Se considerarmos um vetor ordenado de maneira decrescente, temos que para cada elemento do segmento não ordenado sendo analisado, todos os elementos do segmento ordenado precisarão ser movidos. Neste caso, para um vetor de N elementos teremos N^2 comparações. No caso do vetor já estar ordenado, teremos N comparações.

Passo 1	<table border="1"> <tr><td>18</td><td>15</td><td>7</td><td>9</td><td>23</td><td>16</td><td>14</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	18	15	7	9	23	16	14	0	1	2	3	4	5	6	
18	15	7	9	23	16	14										
0	1	2	3	4	5	6										
Passo 2	<table border="1"> <tr><td>15</td><td>18</td><td>7</td><td>9</td><td>23</td><td>16</td><td>14</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	15	18	7	9	23	16	14	0	1	2	3	4	5	6	
15	18	7	9	23	16	14										
0	1	2	3	4	5	6										
Passo 3	<table border="1"> <tr><td>7</td><td>15</td><td>18</td><td>9</td><td>23</td><td>16</td><td>14</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	7	15	18	9	23	16	14	0	1	2	3	4	5	6	
7	15	18	9	23	16	14										
0	1	2	3	4	5	6										
Passo 4	<table border="1"> <tr><td>7</td><td>9</td><td>15</td><td>18</td><td>23</td><td>16</td><td>14</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	7	9	15	18	23	16	14	0	1	2	3	4	5	6	
7	9	15	18	23	16	14										
0	1	2	3	4	5	6										
Passo 5	<table border="1"> <tr><td>7</td><td>9</td><td>15</td><td>18</td><td>23</td><td>16</td><td>14</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	7	9	15	18	23	16	14	0	1	2	3	4	5	6	
7	9	15	18	23	16	14										
0	1	2	3	4	5	6										
Passo 6	<table border="1"> <tr><td>7</td><td>9</td><td>15</td><td>16</td><td>18</td><td>23</td><td>14</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	7	9	15	16	18	23	14	0	1	2	3	4	5	6	
7	9	15	16	18	23	14										
0	1	2	3	4	5	6										
Passo 7	<table border="1"> <tr><td>7</td><td>9</td><td>14</td><td>15</td><td>16</td><td>18</td><td>23</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	7	9	14	15	16	18	23	0	1	2	3	4	5	6	
7	9	14	15	16	18	23										
0	1	2	3	4	5	6										



 Segmento Ordenado



 Segmento Não Ordenado

Figura 128 – Passos realizados pelo algoritmo de inserção direta em um vetor de números inteiros.
Cada passo representa uma iteração do laço mais externo do algoritmo.

Método bolha (*bubble sort*)

O algoritmo *bubble sort* pertence à classe dos algoritmos de ordenação por troca. A ordenação dos dados ocorre através da troca de pares de elementos, caso eles estejam fora da ordem desejada. Este também é um método bastante simples, onde a cada passo cada elemento do vetor é comparado com seu elemento sucessor. Caso eles estejam fora da ordem, eles são trocados. Sucessivas iterações são executadas até que em uma delas nenhuma troca de pares é realizada, indicando que o vetor já está ordenado.

O código fonte do método que implementa o algoritmo de ordenação *bubble sort* é mostrado no exemplo 66.

```

1  public static void bubbleSort(int a[]) {
2      for (int i = a.length; --i>=0; ) {
3          boolean flipped = false;
4          for (int j = 0; j<i; j++) {
5              if (a[j] > a[j+1]) {
6                  int T = a[j];
7                  a[j] = a[j+1];
8                  a[j+1] = T;
9                  flipped = true;
10             }
11         }
12     }
13 }
```

Exemplo 66 – Método que implementa o algoritmo *bubble sort* em um vetor de números inteiros.

O método recebe como parâmetro um vetor de números inteiros e itera sobre estes elementos. A cada iteração do laço mais externo (linhas 2 a 12), uma variável boolean é inicializada com o valor *false*. O laço mais interno (linhas 4 a 11) realiza a comparação dos pares de elementos e, caso um par esteja fora de ordem, a troca é realizada e o valor *true* é atribuído à variável *flipped*, indicando que uma troca foi realizada. O algoritmo executa estas sucessivas varreduras no vetor até que nenhuma troca ocorra em uma delas. Neste ponto o vetor está ordenado.

A figura 129 e um mostra a execução passo a passo do algoritmo em um vetor de números inteiros.

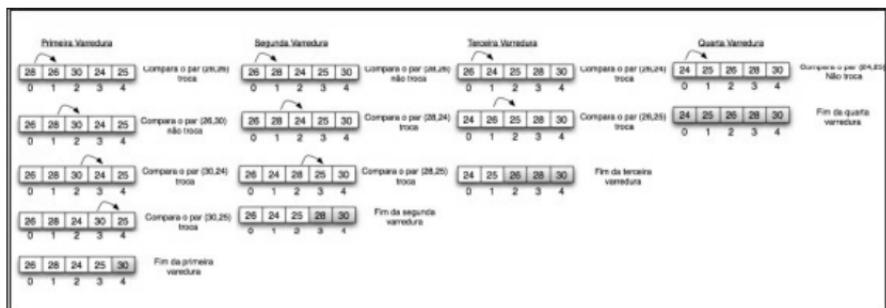


Figura 129 – Execução passo a passo do algoritmo *bubble sort* em um vetor de números inteiros.

Uma característica do algoritmo *bubble sort* é que ele garante que, ao final da primeira varredura, o maior elemento estará na última posição do vetor. Ao final da segunda varredura o segundo maior estará na penúltima e assim por diante. Este

algoritmo também apresenta uma baixa eficiência. Se considerarmos um vetor de N elementos ordenados de maneira decrescente, serão necessárias N^2 comparações para ordenar o vetor. Se considerarmos um vetor já ordenado, serão necessárias N comparações, pois na primeira varredura nenhuma troca será feita e o algoritmo termina.

Seleção direta

O algoritmo de seleção direta pertence à classe de algoritmos de ordenação por seleção. Nesta classe de algoritmos a ordenação é realizada pela seleção sucessiva do menor (ou maior) valor do vetor. A cada passo o elemento selecionado é colocado em sua posição definitiva e o processo é repetido para os segmentos que contêm os elementos ainda não selecionados.

No algoritmo de seleção direta a seleção da menor chave é feita por uma pesquisa sequencial. A menor chave encontrada é então permutada com a chave que ocupa a posição inicial do vetor, que fica reduzido de um elemento.

O código fonte do método que implementa o algoritmo de seleção direta é mostrado no exemplo 67.

```
1 public static void selectionSort (int a[]) {  
2     int min=0, ch;  
3     for (int i=0; i<a.length-1; i++) {  
4         min = i;  
5         for (int j = i + 1; j<a.length; j++)  
6             if (a [ j ] < a [ min ]) min = j;  
7         ch = a [ i ];  
8         a [ i ] = a [ min ];  
9         a [ min ] = ch;  
10    }  
11 }
```

Exemplo 67 – Método que implementa o algoritmo de seleção direta em um vetor de números inteiros.

O método recebe como parâmetro um vetor de números inteiros e a cada iteração do laço mais externo a menor chave é localizada pelo laço mais interno (linhas 5 e 6). Uma vez que esta chave tenha sido encontrada ela é permutada, na primeira iteração com a primeira posição, na segunda iteração com a segunda e assim por diante.

A figura 130 mostra uma execução passo a passo deste algoritmo em um vetor de números inteiros.

Iteração	Vetor	Chave Selecionada	Troca														
1	<table border="1"><tr><td>18</td><td>15</td><td>7</td><td>9</td><td>23</td><td>16</td><td>14</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	18	15	7	9	23	16	14	0	1	2	3	4	5	6	7	7 e 18
18	15	7	9	23	16	14											
0	1	2	3	4	5	6											
2	<table border="1"><tr><td>7</td><td>15</td><td>18</td><td>9</td><td>23</td><td>16</td><td>14</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	7	15	18	9	23	16	14	0	1	2	3	4	5	6	9	9 e 15
7	15	18	9	23	16	14											
0	1	2	3	4	5	6											
3	<table border="1"><tr><td>7</td><td>9</td><td>18</td><td>15</td><td>23</td><td>16</td><td>14</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	7	9	18	15	23	16	14	0	1	2	3	4	5	6	14	14 e 18
7	9	18	15	23	16	14											
0	1	2	3	4	5	6											
4	<table border="1"><tr><td>7</td><td>9</td><td>14</td><td>15</td><td>23</td><td>16</td><td>18</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	7	9	14	15	23	16	18	0	1	2	3	4	5	6	15	Não troca
7	9	14	15	23	16	18											
0	1	2	3	4	5	6											
5	<table border="1"><tr><td>7</td><td>9</td><td>14</td><td>15</td><td>23</td><td>16</td><td>18</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	7	9	14	15	23	16	18	0	1	2	3	4	5	6	16	16 e 23
7	9	14	15	23	16	18											
0	1	2	3	4	5	6											
6	<table border="1"><tr><td>7</td><td>9</td><td>14</td><td>15</td><td>16</td><td>23</td><td>18</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	7	9	14	15	16	23	18	0	1	2	3	4	5	6	18	18 e 23
7	9	14	15	16	23	18											
0	1	2	3	4	5	6											
7	<table border="1"><tr><td>7</td><td>9</td><td>14</td><td>15</td><td>16</td><td>18</td><td>23</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr></table>	7	9	14	15	16	18	23	0	1	2	3	4	5	6		
7	9	14	15	16	18	23											
0	1	2	3	4	5	6											

Figura 130 – Execução passo a passo do algoritmo de seleção direta em um vetor de números inteiros.

Uma característica deste método de ordenação é que, independente da ordem em que os elementos estejam, ele sempre executará no mesmo número de comparações. Se considerarmos um vetor de N posições, serão necessárias N^2 comparações para qualquer caso: vetor já ordenado ou ordenado de maneira decrescente.

Quicksort

O algoritmo *quicksort*, também chamado de algoritmo de troca e partição, também pertence à classe de algoritmos de ordenação por troca, em que o processo de ordenação se dá através da troca de pares de elementos. Ele apresenta um desempenho consideravelmente melhor que os métodos apresentados anteriormente e a ideia básica deste método é dividir o problema de ordenar um conjunto de N itens em dois problemas menores. Em seguida, estes problemas menores são ordenados independentemente e ao final os resultados são combinados para produzir a solução do problema original.

O primeiro passo realizado pelo algoritmo é o particionamento. Este processo consiste em particionar um vetor V com N elementos em três segmentos como mostra a figura 131.

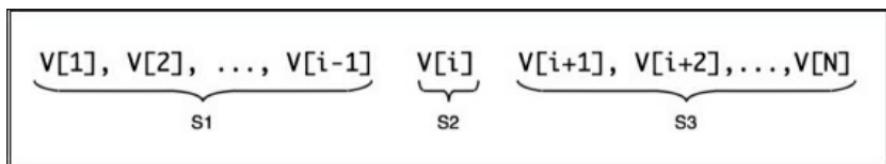


Figura 131 – Particionamento do vetor.

Este particionamento faz com que o segmento S2 tenha comprimento um e contenha uma chave denominada particionadora, ou pivô. O segmento S1 terá comprimento maior ou igual a zero e conterá todas as chaves cujos valores são menores ou iguais ao da chave particionadora. O segmento S3 também terá comprimento maior ou igual a zero e conterá todas as chaves cujos valores são maiores do que o da chave particionadora.

O algoritmo segue sua execução reaplicando o processo de particionamento aos segmentos S1 e S2 e a todos os segmentos resultantes que tiverem comprimento maior ou igual a um. Quando não restarem segmentos a serem particionados, o vetor estará ordenado.

O código fonte do método que implementa o algoritmo *quicksort* é mostrado no exemplo 68.

```
1  public static void quickSort (int[] a, int lo, int hi){  
2  
3      int i=lo, j=hi, h;  
4      int x=a[(lo+hi)/2];  
5  
6      do {  
7          while (a[i]<x) i++;  
8          while (a[j]>x) j--;  
9          if (i<=j)  
10             {  
11                 h=a[i]; a[i]=a[j]; a[j]=h;  
12                 i++; j--;  
13             }  
14         } while (i<=j);  
15  
16         if (lo<j) quickSort(a, lo, j);  
17         if (i<hi) quickSort(a, i, hi);  
18     }  
}
```

Exemplo 68 – Método que implementa o algoritmo *quicksort* em um vetor de números inteiros.

O algoritmo recebe como parâmetro um vetor de números inteiros e dois inteiros: *lo* e *hi*, que são utilizados para delimitar a região sendo classificada. Na linha 4 a chave particionadora é escolhida. Este método sempre escolhe a chave posicionada no meio do vetor. O laço da linha 7 percorre o vetor da esquerda para a direita até que seja encontrada uma chave maior ou igual à chave particionadora. Quando encontrada, a posição desta chave estará na variável *i*. O laço da linha 8 percorre o vetor da direita para a esquerda até que seja encontrada uma chave menor ou igual à chave particionadora. Quando encontrada a posição desta chave, estará na variável *j*. O código das linhas 9 a 13 troca de posição as chaves localizadas anteriormente. O laço mais externo termina quando os índices *i* e *j* se cruzarem. Neste ponto o processo de particionamento está completo. Em seguida, o algoritmo segue chamando a si mesmo para o segmento S1 (linha 16) e para o segmento S2 (linha 17).

A figura 132 mostra as chamadas recursivas executadas pelo algoritmo *quicksort* em um vetor de números inteiros.

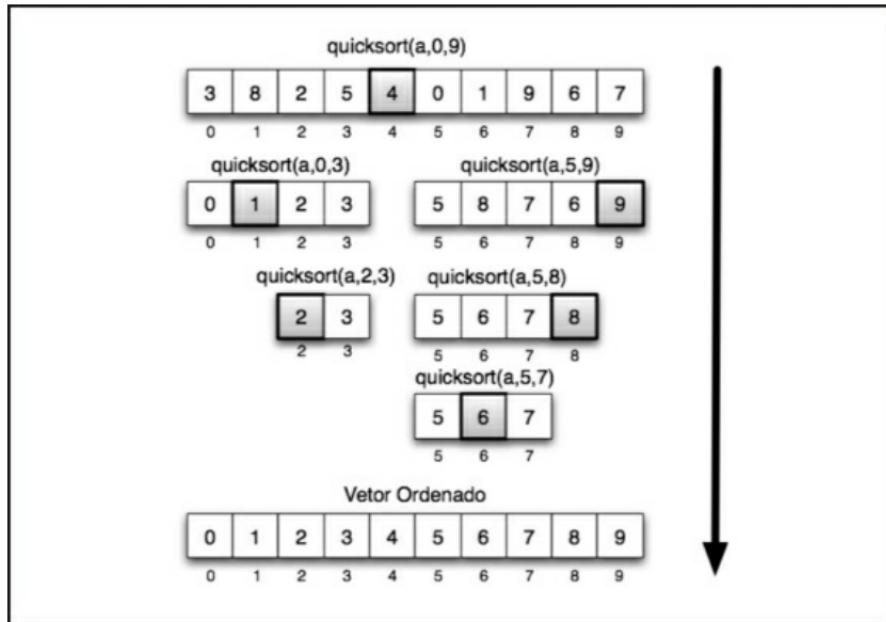


Figura 132 – Chamadas recursivas executadas pela execução ao algoritmo *quicksort* em um vetor de números inteiros.

Note que a cada chamada do método *quicksort* o elemento escolhido como chave particionadora já se encontra em sua posição correta no vetor.

Uma questão-chave no algoritmo *quicksort* é a escolha da chave particionadora, pois esta influencia diretamente no desempenho do algoritmo. No exemplo mostrado a chave posicionada exatamente no meio do vetor foi selecionada, mas nada garante que esta seja a melhor escolha. Uma boa escolha de chave particionadora só pode ser feita se possuirmos um conhecimento prévio a respeito das chaves.

O algoritmo *quicksort* tem seu melhor desempenho sempre que o vetor é dividido em duas partes iguais, cada uma com $N/2$ elementos pelo processo de particionamento. Nesta caso a profundidade da recursão será de $N \cdot \log_2 N$.

Este capítulo apresentou o conceito de classificação de dados. Este conceito é um dos mais importantes na área da computação, pois é aplicado em muitos domínios de aplicações. Foram apresentados quatro algoritmos de ordenação. Três mais simples, adequados a conjuntos pequenos de dados, e um mais elaborado, adequado a conjuntos maiores de dados.

CAPÍTULO 13

APÊNDICE A – EXCEÇÕES

O tratamento de exceções é um mecanismo da linguagem Java utilizado para a manipulação de problemas que ocorrem durante a execução de um programa como, por exemplo, o acesso a um índice ilegal de um vetor ou uma conversão ilegal entre tipos primitivos ou até mesmo referências a objetos.

A linguagem Java disponibiliza em sua API uma hierarquia de classes de exceção, onde cada classe, ou conjunto de classes, compreende um tipo específico de problema. Através da utilização de exceções, um programa pode ser escrito de modo a continuar funcionando mesmo após a ocorrência de erros. Neste sentido, uma exceção é um objeto que descreve uma condição de exceção que ocorreu em alguma parte do programa durante sua execução.

Neste apêndice trataremos questões específicas a respeito de exceções. Iniciaremos estudando como uma exceção pode ser tratada por um programa. Em seguida veremos como a API da linguagem Java organiza as classes de exceção e como podemos lançar exceções ao detectar condições problemáticas em nossos programas. Também veremos como podemos criar nossas próprias classes de exceção.

Tratando exceções

Considere o código mostrado abaixo, onde a classe “TesteExcecao” contém em seu método main uma divisão por zero.

```
1 public class TesteExcecao {  
2     public static void main(String[] args) {  
3         int i = 0;  
4         int x = 10/i;  
5         System.out.println("Execução continua.");  
6     }  
7 }
```

Exemplo 69 – Exceção de divisão por zero não tratada.

Ao executar a linha 4, o programa termina sua execução mostrando na saída a causa

do problema. Esta saída é mostrada na figura abaixo.

```
Exception in thread "main" java.lang.ArithmetricException: / by zero at
Excecoes.TesteExcecao.main(TesteExcecao.java:4)
```

Figura 133 – Saída gerada pelo exemplo 68.

Note que o programa terminou sua execução e a linha de código cinco não foi executada. A mensagem mostrada indica que uma exceção do tipo *ArithmetricException* ocorreu devido a uma divisão por zero na linha 4 do método *main*. Como esta exceção não foi tratada, o programa terminou sua execução.

Este mesmo código é mostrado abaixo, agora com o tratamento de exceções. Para que uma exceção seja tratada, o código que a lança deve estar dentro de um bloco *try-catch*, como mostrado no exemplo 70.

```
1  public class TesteTratamentoExcecao {
2      public static void main(String[] args) {
3          try {
4              int i = 0;
5              int x = 10/i;
6          } catch (ArithmetricException e) {
7              System.out.println("Aconteceu uma divisão por zero!!");
8          }
9          System.out.println("Execução continua.");
10     }
11 }
```

Exemplo 70 – Divisão por zero com tratamento de exceção.

Este programa, ao executar a divisão por zero na linha 5, faz com que o fluxo de execução seja desviado para o bloco das linhas 6 a 8. O código dentro deste bloco é então executado e o programa não termina sua execução. A saída deste código é mostrada na figura abaixo.

```
Aconteceu uma divisão por zero!!
Execução continua.
```

Figura 134 – Saída gerada pelo exemplo sessenta e nove.

Ao executar a divisão por zero, dizemos que o programa lançou uma exceção. Quando isso acontece, o fluxo de execução é desviado para o ambiente de execução,

que inicia uma procura por um tratador para a exceção do tipo da exceção lançada. No caso do exemplo 70 este tratador foi encontrado no bloco *catch*, que é para onde o fluxo de execução foi desviado. Neste caso, como a exceção foi tratada, o programa não terminou sua execução.

Podemos ainda ter mais de uma cláusula *catch* associada a um bloco *try*. Isso faz com que possamos tratar mais de um tipo de exceção que possa vir a ser lançada em um bloco de código. Considere o exemplo abaixo, onde dois tipos de exceções são tratadas para um único bloco *try*.

```
1 public class MultiCatch {
2     public static void main(String[] args) {
3         Try {
4             Teclado t = new Teclado();
5             int a = t.leInt();
6             int b = 42/a;
7             int c [ ] = {1};
8             c[42] = 99;
9         }
10        catch (ArithmetricException e) {
11            System.out.println ("Divisão por 0: "+e);
12        }
13        catch (ArrayIndexOutOfBoundsException e) {
14            System.out.println ("Índice inválido do array: "+e);
15        }
16        System.out.println("Execução continua...");
17    }
18 }
```

Exemplo 71 – Múltiplas clausulas *catch* para um único bloco *try*.

Neste exemplo, dentro do bloco *try* um número inteiro é solicitado ao usuário. Este número é utilizado então na divisão da linha 6. Caso este número seja zero, uma divisão por zero irá ocorrer na linha 6, uma exceção será lançada e capturada pelo bloco *catch* das linhas 10 a 12. Caso esta exceção não seja lançada, um índice inválido do vetor *c* é acessado na linha 8. Este acesso inválido irá gerar uma exceção do tipo *ArrayIndexOutOfBoundsException*, que será tratada pelo bloco *catch* das linhas 13 a 15.

Quando o valor zero é passado pelo usuário, a seguinte saída é produzida:

```
Divisão por 0: java.lang.ArithmetricException: / by zero
Execução continua...
```

Figura 135 – Saída do exemplo 71, quando o valor zero é informado pelo usuário.

Quando um valor diferente de zero é informado pelo usuário, a seguinte saída é produzida:

```
Índice inválido do array:  
java.lang.ArrayIndexOutOfBoundsException: 42  
Execução continua...
```

Figura 136 – Saída do exemplo 71, quando um valor diferente de zero é informado pelo usuário.

Um ponto importante a ser mencionado é que, uma vez que uma exceção ocorre dentro de um bloco *try*, o restante do código dentro do bloco não é executado, uma vez que o fluxo de execução é desviado para o tratador da exceção. No exemplo 71, ao ocorrer uma divisão por zero, as linhas 7 e 8 nunca serão executadas, pois o fluxo de execução desvia para o tratador específico.

O exemplo 71 também demonstra uma vantagem do uso de exceções ao tratamento tradicional de erros em que o fluxo normal de execução e o tratamento de erros fica misturado na mesma região de código. Utilizando exceções, temos o fluxo normal de execução completamente separado do tratamento de erros, o que deixa o programa mais fácil de ser lido e, consequentemente, mantido.

Um bloco *catch* deve sempre especificar entre parênteses um parâmetro de exceção que identifica o tipo da exceção que ele irá tratar. Sempre que uma exceção ocorre dentro de um bloco *try*, o bloco *catch* executado é aquele que possui um parâmetro cujo tipo corresponde ao tipo da exceção que ocorreu (isto é, o tipo corresponde exatamente ao tipo da exceção gerada ou alguma superclasse dela). O nome do parâmetro de exceção permite que tratador interaja com o objeto de exceção gerado, como, por exemplo, chamando o método *toString* do objeto, que exibe informações sobre a exceção gerada.

Em algumas situações pode não ser apropriado tratar a exceção no método onde ela ocorreu. Neste caso, podemos apenas especificar na assinatura de um método que este pode vir a lançar uma ou mais exceções através da cláusula *throws*. Considere o exemplo mostrado abaixo, onde uma divisão é executada pelo método *divide*.

```
1  public class DemoThrows {
2
3      public static double divide(int a, int b) throws ArithmeticException {
4          return a / b;
5      }
6
7      public static void main(String[] args) {
8          try{
9              divide(4,0);
10         }
11         catch(ArithmeticException e){
12             System.out.println("Divisão por zero: " + e);
13         }
14     }
15 }
```

Exemplo 72 – Utilização da cláusula *throws*.

No exemplo 72 o método *divide* especifica em sua assinatura que uma exceção do tipo *ArithmeticException* pode ser lançada em seu corpo. Se esta exceção for gerada, o fluxo de execução é desviado e o ambiente de execução da linguagem Java irá procurar por um tratador em algum dos métodos que tenha chamado o método *divide*. Neste caso este tratador é encontrado no método *main* e a exceção é capturada pelo *catch* da linha onze. Esta abordagem permite que uma exceção seja passada por diferentes métodos através da pilha de chamadas. Para isso basta que os métodos especifiquem em suas assinaturas a exceção que pode ser lançada através da cláusula *throws*. Note que mais de um tipo de exceção pode ser especificado. Para isso basta separar por vírgula os tipos de exceção.

Hierarquia de exceções

Todas as classes de exceção definidas na API da linguagem Java herdam (direta ou indiretamente) da classe *Throwable*. A classe *Throwable* possui duas subclasses diretas: *Error* e *Exception*, como mostra a figura 137.

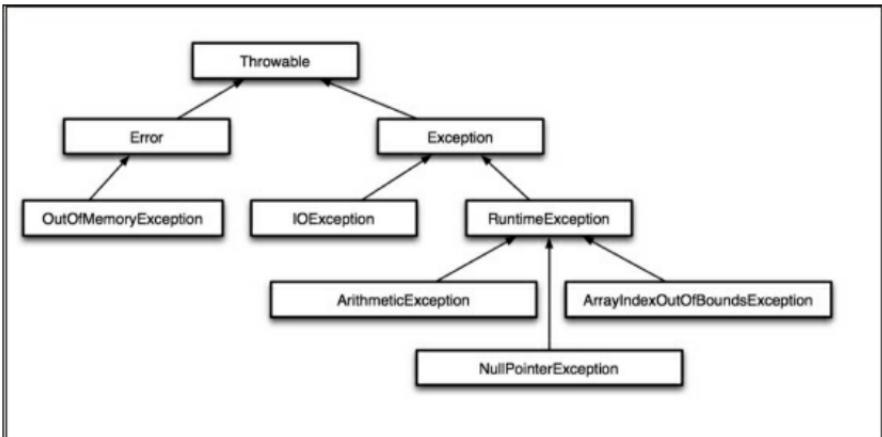


Figura 137 – Algumas classes da hierarquia de classes de exceções definidas na API da linguagem Java.

A classe *Exception* e todas as suas subclasses representam situações excepcionais que podem ocorrer em um programa e podem ser capturadas. A classe *Error* e suas subclasses representam situações anormais que podem ocorrer no ambiente de execução (JVM). As exceções desse tipo acontecem raramente e geralmente sinalizam problemas nos quais não há como contornar, como, por exemplo, a falta de memória.

A hierarquia de exceções da linguagem Java é imensa. A figura 137 mostra apenas algumas subclasses das classes *Exception* e *Error*. Mais informações sobre estas classes e suas subclasses podem ser encontradas na documentação da API.

As classes de exceção da linguagem Java possuem duas categorias:

- ◆ Exceções verificadas (*checked exceptions*);
- ◆ Exceções não verificadas (*unchecked exceptions*).

Uma exceção verificada deve, obrigatoriamente, ser tratada ou especificada. Caso ela não seja, o compilador emite um erro. Uma exceção não verificada não precisa ser tratada. Todas as exceções que são subclasses de *RuntimeException* são do tipo não verificadas e, consequentemente, não precisam ser tratadas. Todas as exceções que são subclasses de *Exception*, mas não de *RuntimeException*, são exceções verificadas e precisam ser tratadas ou especificadas na assinatura do método com a cláusula *throws*. Elas são chamadas de exceções verificadas, pois o compilador verifica cada chamada de método e declaração de método para determinar se o método lança exceções verificadas.

Diferentes classes de exceção podem ser derivadas de uma mesma superclasse. Desta maneira, um bloco *catch* escrito para capturar objetos de exceção de uma superclasse pode também capturar qualquer objeto de exceção que seja do tipo de uma subclasse deste. Isso possibilita que um bloco *catch* trate erros relacionados de uma maneira concisa, permitindo também o processamento polimórfico de exceções relacionadas. Note que esta abordagem deve ser utilizada somente se o comportamento do tratamento do erro for o mesmo para todas as subclasses. Um erro comum de programação é colocar um bloco *catch* para um tipo de exceção de superclasse antes de outros blocos *catch* que capturam tipos de exceção de subclasse. Isso torna o bloco *catch* da subclasse inalcançável e gera um erro de compilação.

Lançando uma exceção

Uma exceção é lançada através do comando *throw*. Este comando cria o objeto de exceção e faz com que o ambiente de execução da linguagem Java procure por um tratador para a exceção lançada.

No exemplo mostrado abaixo o método *metodoA* recebe dois parâmetros, um número inteiro e um vetor de inteiros. Na assinatura deste método está especificado que ele pode lançar duas exceções: *IllegalArgumentException* e *ArrayIndexOutOfBoundsException*. A exceção *ArrayIndexOutOfBoundsException* é lançada na linha 6 caso o valor armazenado na variável *a* esteja fora do intervalo permitido pelo vetor. A exceção *IllegalArgumentException* é lançada na linha 10 se o valor do parâmetro *a* for menor que dez. Note que no método *main* a chamada ao método está dentro de um bloco *try* e existe um bloco *catch* para cada uma das exceções que podem ser geradas.

```
1 public class DemoThrow {
2
3     public static void metodoA(int a, int v[]) throws IllegalArgumentException,
4                                         ArrayIndexOutOfBoundsException{
5         if ((a < 0) || (a > v.length-1)){
6             throw new ArrayIndexOutOfBoundsException();
7         }
8
9         if (v[a] < 10){
10            throw new IllegalArgumentException();
11        }
12
13        System.out.println("v[" + a + "] = " + v[a]);
14    }
15
16    public static void main(String[] args) {
17        try{
18            int v[] = {1,2,3,4,5,6,7,8,9,10};
19            metodoA(12,v);
20        }
21        catch(IllegalArgumentException e){
22            System.out.println("Argumento ilegal: " + e);
23        }
24        catch(ArrayIndexOutOfBoundsException e){
25            System.out.println("índice inválido: " + e);
26        }
27    }
28 }
```

Exemplo 73 – Utilização do comando *throw* para lançar exceções.

Ao executar este programa, uma exceção *ArrayIndexOutOfBoundsException* é gerada, pois o argumento passado para o método está fora do intervalo permitido pelo vetor. Embora simples, este exemplo demonstra como uma exceção pode ser lançada com o comando *throw*.

Criando uma classe de exceção

Embora a API da linguagem Java disponibilize uma grande quantidade de classes de Exceção, às vezes, quando estamos programando classes específicas, é interessante criar nossas próprias classes de exceção, adequadas ao domínio da aplicação sendo desenvolvida.

Ao definir uma classe de exceção, uma boa prática é estudar as classes de exceção

já definidas na linguagem Java de modo a tentar estender a classe sendo desenvolvida a partir de uma destas classes. Por exemplo, se você estiver criando uma nova classe para tratar uma divisão por zero, você poderia estender sua classe da classe *ArithmaticException*, pois a divisão por zero ocorre durante uma operação aritmética. Caso as classes existentes não sejam apropriadas, você ainda deve decidir se a sua exceção será verificada ou não verificada. A nova classe de exceção deve ser verificada, isto é, ser subclasse da classe *Exception*, mas não de *RuntimeException*, se possíveis clientes precisarem tratar a exceção. Do contrário, a exceção pode ser não verificada, ou seja, derivada de *RuntimeException*.

Os exemplos abaixo demonstram como uma classe de exceção pode ser criada e utilizada em um programa Java.

```
1 public class IdadeInsuficienteException extends RuntimeException{  
2  
3     public IdadeInsuficienteException (String mensagem) {  
4         super(mensagem);  
5     }  
6 }
```

Exemplo 74 – Classe IdadeInsuficienteException.

```
1 public class CarteiraDeMotorista {  
2     private String nome;  
3     private int idade;  
4  
5     public CarteiraDeMotorista (String n, int i){  
6         nome = n;  
7         this.idade = i;  
8         if (this.idade > 18) {  
9             System.out.println ("A idade suficiente!");  
10        }  
11        else {  
12            throw new IdadeInsuficienteException ("A idade insuficiente!");  
13        }  
14    }  
15 }
```

Exemplo 75 – Classe CarteiraDeMotorista. Uma exceção é gerada no construtor caso a idade passada como parâmetro seja menor que dezoito.

```
1 public class TestaMinhaExcecao {
2     public static void main (String args[]) throws IdadeInsuficienteException
3     {
4         try{
5             CarteiraDeMotorista c1 = new CarteiraDeMotorista ("Fulano", 31);
6             CarteiraDeMotorista c2 = new CarteiraDeMotorista ("Betrano", 15);
7         }
8         catch (IdadeInsuficienteException e){
9             System.out.println(e.getMessage());
10        }
11    }
12 }
```

Exemplo 76 – Classe de teste que captura a exceção `IdadeInsuficiente`.

Bloco `finally`

O bloco *finally* é utilizado para liberar recursos alocados dentro de um bloco *try*. Este bloco de código é sempre executado, havendo ou não uma exceção. Alguns recursos como, por exemplo, arquivos abertos, conexões com bancos de dados ou conexões de rede podem não ser liberados na ocorrência de uma exceção. Desta maneira, como o bloco *finally* é sempre executado, este é um bom lugar para liberar estes recursos.

O exemplo abaixo demonstra a utilização do bloco *finally*.

```

1  class DemoFinally {
2      static void procA() throws Exception {
3          try {
4              System.out.println("dentro de procA");
5              throw new Exception("demo");
6          } finally {
7              System.out.println("finally de procA");
8          }
9      }
10     static void procB() {
11         try {
12             System.out.println("dentro de procB");
13             return;
14         } finally {
15             System.out.println("finally de procB");
16         }
17     }
18     public static void main(String args[]) {
19         try {
20             procA();
21         } catch (Exception e) {
22             System.out.println("Tratando exceção que ocorreu no método procA");
23         }
24         procB();
25     }
26 }
```

Exemplo 77 – Utilização do bloco *finally*.

Ao executar o programa do exemplo 77, a saída mostrada na figura 138 é gerada.

```

dentro de procA
finally de procA
Tratando exceção que ocorreu no método procA
dentro de procB
finally de procB
```

Figura 138 – Saída gerada pelo exemplo 77.

O método main do exemplo 77 chama o método procA. Este método gera uma exceção intencionalmente na linha 5. Note que antes do fluxo de execução ser desviado o código dentro do bloco *finally* foi executado. Em seguida, a exceção gerada dentro do método procA é capturada na linha 21. Depois de tratada, o método main chama o método procB, que apenas imprime uma mensagem na tela e retorna. Note que antes de retornar, o código dentro do bloco *finally* também é executado.

Este exemplo demonstra que o bloco *finally* é sempre executado, mesmo na

ocorrência de um comando *return* como o da linha 13.

Método printStackTrace

O método *printStackTrace* pertence à classe *Throwable* (superclasse de todas as exceções) e é bastante útil em processos de depuração de programas. Ao ser chamado, este método imprime toda a pilha de chamadas a partir de onde a exceção foi lançada. Muitas vezes a exceção é lançada em um método que foi chamado posteriormente, possivelmente após muitas outras chamadas de métodos. Utilizando este método, podemos visualizar o “rastreamento” da pilha a partir de onde a exceção ocorreu.

O exemplo abaixo demonstra a utilização do método *printStackTrace*.

```
1  class DemoPrintStackTrace {
2      public static void main(String args[]) {
3          try{
4              metodo1();
5          }
6          catch (Exception e) {
7              System.out.println("Erro:" + e);
8              e.printStackTrace();
9          }
10     }
11     public static void metodo1() throws Exception {
12         metodo2();
13     }
14     public static void metodo2() throws Exception {
15         metodo3();
16     }
17     public static void metodo3() throws Exception {
18         throw new Exception();
19     }
20 }
```

Exemplo 78 – Utilização do método *printStackTrace*.

Ao ser executado, o programa do exemplo 78 gera a saída mostrada na figura 139.

```
Erro:java.lang.Exception
java.lang.Exception
at Excecoes.Demo.PrintStackTrace.metodo3(DemoPrintStackTrace.java:18)
at Excecoes.Demo.PrintStackTrace.metodo2(DemoPrintStackTrace.java:15)
at Excecoes.Demo.PrintStackTrace.metodo1(DemoPrintStackTrace.java:12)
```

at Excecoes.Demo.PrintStackTrace.main(DemoPrintStackTrace.java:14)

Figura 139 – Saída gerada pelo Exemplo setenta e sete.

Note que a exceção foi gerada no método `metodo3` e foi passada pelos métodos `metodo2` e `metodo1` para ser capturada no método `main`. Através do método `printStackTrace` conseguimos visualizar toda a pilha de chamadas, desde o ponto onde a exceção foi gerada até onde ela foi capturada.

REFERÊNCIAS BIBLIOGRÁFICAS

- DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. 6. Edição. São Paulo: Pearson, 2007.
- GOODRICH, Michael T.; TAMASSIA, Roberto. *Estruturas de dados e algoritmos em Java*. 4. Edição. Porto Alegre: Bookman, 2007.
- LAFORE, Robert. *Estruturas de dados e algoritmos em Java*. 2. Edição. Rio de Janeiro: Ciência Moderna, 2005.
- MCONNEL, Steve. *Code Complete. A practical handbook of software construction*. 2nd Edition. New York: Microsoft Press, 2004.
- PREISS, Bruno R. *Data structures and algorithms: with object-oriented design patterns in Java*. New York: John Wiley e Sons, 1999.
- SERSON, Roberto Rubinstein. *Programação orientada a objetos com Java*. Rio de Janeiro: Brasport, 2007.
- VELOSO, Paulo, et. al. *Estruturas de Dados*. Rio de Janeiro: Campus, 1983.

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

Reitor

Pe. Marcelo Fernandes de Aquino, SJ

Vice-reitor

Pe. José Ivo Follmann, SJ

EDITORIA UNISINOS

Diretor

Pe. Pedro Gilberto Gomes, SJ



Editora da Universidade do Vale do Rio dos Sinos

EDITORA UNISINOS

Av. Unisinos, 950

93022-000 São Leopoldo RS Brasil

Telef: 51.35908239

Fax: 51.35908238

editora@unisinos.br

2010 Direitos de publicação e comercialização da
Editora da Universidade do Vale do Rio dos Sinos
EDITORAS UNISINOS

L616e Lermen, Gustavo.

Estruturas de dados e algoritmos / Gustavo Lermen. – São Leopoldo, RS : Ed. UNISINOS, 2010.

168 p.: il. – (EAD Unisinos)

ISBN 978-85-7431-391-7

1. Estruturas de dados (Computação). 2. Algoritmos. I. Título. II. Série.

CDD 005.73
CDU 004.422.63

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecário Flávio Nunes, CRB 10/1298)

Esta obra segue as normas do Acordo Ortográfico da Língua Portuguesa vigente desde 2009.



Editor
Carlos Alberto Gianotti

Acompanhamento editorial
Mateus Colombo Mendes

Revisão
Renato Deitos
Editoração
Décio Remígius Ely
Capa
Isabel Carballo
Impressão, inverno de 2010

danosa à cultura.
Foi feito o depósito legal.

Sobre o autor

GUSTAVO LERMEN. Mestre em Computação Aplicada pela Universidade do Vale do Rio dos Sinos – UNISINOS. Bacharel em Ciência da Computação pela UNISINOS/RS. Professor dos cursos de Ciência da Computação e Sistemas de Informação da UNISINOS. Também atua como Analista Desenvolvedor no SAP *Custom Development*.

Edição digital: janeiro 2014

Arquivo ePub produzido pela **Simplíssimo Livros**
