

LINGUAGENS DE PROGRAMAÇÃO E ESTRUTURA DE DADOS

Prof^ª. Edna da Luz Lampert

Prof. Elton Giovani Gretter



UNIASSELVI

2014



UNIASSELVI

Copyright © UNIASSELVI 2014

Elaboração:

Prof^ª. Edna da Luz Lampert

Prof. Elton Giovani Gretter

Revisão, Diagramação e Produção:

Centro Universitário Leonardo da Vinci – UNIASSELVI

Ficha catalográfica elaborada na fonte pela Biblioteca Dante Alighieri
UNIASSELVI – Indaial.

005.133

L237l

Lampert, Edna da Luz

Linguagens de programação e estrutura de dados / Edna da Luz Lampert, Elton Giovani Gretter. Indaial : Unias-selvi, 2014.

237 p. : il

ISBN 978-85-7830-857-5

1. Linguagens de programação

I. Centro Universitário Leonardo da Vinci.

APRESENTAÇÃO

Caro(a) acadêmico(a)!

Seja bem-vindo(a) aos estudos da disciplina de Linguagem de Programação e Estruturas de dados.

O estudo desta disciplina é muito importante para ampliar seus conhecimentos acerca da estrutura de dados e sua relevância para a linguagem de programação e como esses dois conceitos estão muito atrelados, para o desenvolvimento de programas de tecnologia da informação.

Desta forma será possível verificar quais são as responsabilidades de um programador perante o desenvolvimento de um *software*, como realizar a estruturação de dados, para que o sistema seja rápido, confiante, dinâmico e seguro. Verificar que a estrutura de dados tem um papel fundamental em organizar os dados e informações, armazenar espaço de memória, realizar funções, executar códigos e gerenciar toda estrutura de dados que um programa envolve.

Este material foi desenvolvido com objetivo de disponibilizar as melhores práticas, formas de como estruturar os dados, este é um processo de aprendizagem que visa instigar sua consciência para a pesquisa e investigativa, buscar novas formas e melhores ferramentas tecnológicas para uma aprendizagem significativa.

Desejo sucesso nessa nova busca de informações, principalmente no que tange ampliar seus conhecimentos!

Prof. Edna da Luz Lampert



Você já me conhece das outras disciplinas? Não? É calouro? Enfim, tanto para você que está chegando agora à UNIASSELVI quanto para você que já é veterano, há novidades em nosso material.

Na Educação a Distância, o livro impresso, entregue a todos os acadêmicos desde 2005, é o material base da disciplina. A partir de 2017, nossos livros estão de visual novo, com um formato mais prático, que cabe na bolsa e facilita a leitura.

O conteúdo continua na íntegra, mas a estrutura interna foi aperfeiçoada com nova diagramação no texto, aproveitando ao máximo o espaço da página, o que também contribui para diminuir a extração de árvores para produção de folhas de papel, por exemplo.

Assim, a UNIASSELVI, preocupando-se com o impacto de nossas ações sobre o ambiente, apresenta também este livro no formato digital. Assim, você, acadêmico, tem a possibilidade de estudá-lo com versatilidade nas telas do celular, *tablet* ou computador.

Eu mesmo, UNI, ganhei um novo *layout*, você me verá frequentemente e surgirei para apresentar dicas de vídeos e outras fontes de conhecimento que complementam o assunto em questão.

Todos esses ajustes foram pensados a partir de relatos que recebemos nas pesquisas institucionais sobre os materiais impressos, para que você, nossa maior prioridade, possa continuar seus estudos com um material de qualidade.

Aproveito o momento para convidá-lo para um bate-papo sobre o Exame Nacional de Desempenho de Estudantes – ENADE.

Bons estudos!



BATE SOBRE O PAPO ENADE!



Olá, acadêmico!

Você já ouviu falar sobre o **ENADE**?

Se ainda não ouviu falar nada sobre o ENADE, agora você receberá algumas informações sobre o tema.

Ouviu falar? Ótimo, este informativo reforçará o que você já sabe e poderá lhe trazer novidades.



Vamos lá!

Qual é o significado da expressão ENADE?

EXAME NACIONAL DE DESEMPENHO DOS ESTUDANTES

Em algum momento de sua vida acadêmica você precisará fazer a prova ENADE.



Que prova é essa?

É **obrigatória**, organizada pelo INEP – Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Quem determina que esta prova é obrigatória... O **MEC – Ministério da Educação**.

O objetivo do MEC com esta prova é o de avaliar seu desempenho acadêmico assim como a qualidade do seu curso.



Fique atento! Quem não participa da prova fica impedido de se formar e não pode retirar o diploma de conclusão do curso até regularizar sua situação junto ao MEC.

Não se preocupe porque a partir de hoje nós estaremos auxiliando você nesta caminhada.

Você receberá outros informativos como este, complementando as orientações e esclarecendo suas dúvidas.



Você tem uma trilha de aprendizagem do ENADE, receberá e-mails, SMS, seu tutor e os profissionais do polo também estarão orientados.

Participará de webconferências entre outras tantas atividades para que esteja preparado para #mandar bem na prova ENADE.

Nós aqui no NEAD e também a equipe no polo estamos com você para vencermos este desafio.

Conte sempre com a gente, para juntos mandarmos bem no ENADE!



SUMÁRIO

UNIDADE 1 – INTRODUÇÃO AO ESTUDO DE LINGUAGEM DE PROGRAMAÇÃO E ESTRUTURAS DE DADOS.....	1
TÓPICO 1 – CONCEITOS FUNDAMENTAIS - INTERPRETAÇÃO VERSUS COMPILAÇÃO.....	3
1 INTRODUÇÃO.....	3
2 CONCEITOS FUNDAMENTAIS DE INTERPRETAÇÃO E COMPILAÇÃO.....	5
2.1 INTERPRETAÇÃO	5
2.2 COMPILAÇÃO	7
3 LINGUAGEM DE PROGRAMAÇÃO C.....	8
3.1 TIPOS PRIMITIVOS DE DADOS	9
3.2 CONSTANTES E VARIÁVEIS	10
3.2.1 Constantes.....	10
3.2.2 Variáveis.....	11
3.3 ATRIBUIÇÃO	12
3.4 OPERADORES ARITMÉTICOS	13
3.5 FUNÇÕES	15
3.5.1 Uso de Funções	16
4 EXEMPLO DE CÓDIGO EM C.....	19
4.1 PALAVRA RESERVADA	20
4.2 COMENTÁRIOS	20
4.2.1 Utilizando Barra dupla	21
4.2.2 Comentários com /* */	22
4.3 IDENTIFICAÇÃO	22
5 COMPILAÇÃO DE PROGRAMAS EM C.....	23
6 LINGUAGEM DE PROGRAMAÇÃO JAVA.....	27
6.1 EXEMPLOS DE CÓDIGOS	28
6.2 APPLETs JAVA	29
6.3 SEGURANÇA.....	29
6.4 PORTABILIDADE	30
6.5 J2ME	30
7 EXEMPLO DE CÓDIGO EM JAVA.....	31
8 COMPILAÇÃO DE PROGRAMAS EM JAVA.....	32
9 CICLO DE DESENVOLVIMENTO.....	33
RESUMO DO TÓPICO 1.....	35
AUTOATIVIDADE	36
TÓPICO 2 – FUNÇÕES.....	37
1 INTRODUÇÃO.....	37
2 FUNÇÕES.....	38
2.1 FUNÇÕES EM JAVA	39
2.1.1 Criando funções sem Argumentos.....	40
2.1.2 Criando funções com Argumentos	40
2.1.3 Criando Funções sem Retorno.....	41

2.1.4 Criando Funções com Retorno	41
2.2 UTILIZANDO FUNÇÕES DE PRIMEIRA CLASSE	42
2.2.1 Funções Internas	42
2.2.2 Usando páginas Web em JavaScript.....	42
3 DEFINIÇÃO DE FUNÇÕES	43
4 PILHA DE EXECUÇÃO	43
4.1 CRIANDO UMA PILHA	46
4.2 INSERÇÃO DE UM NODO NA PILHA	46
4.3 ACESSO À PILHA	48
4.4 CRIAÇÃO DE PILHA ENCADEADA	48
5 PONTEIRO DE VARIÁVEIS.....	49
6 RECURSIVIDADE.....	51
7 VARIÁVEIS ESTÁTICAS DENTRO DE FUNÇÕES	53
7.1 VARIÁVEL LOCAL	54
7.2 VARIÁVEL GLOBAL	55
RESUMO DO TÓPICO 2.....	57
AUTOATIVIDADE	58
 TÓPICO 3 – VETORES E ALOCAÇÃO DINÂMICA.....	 59
1 INTRODUÇÃO	59
2 VETORES E ALOCAÇÃO DINÂMICA.....	59
2.1 VETORES (ARRAYS).....	59
2.2 MÉTODOS UTILIZADOS EM ARRAYS	61
3 ALOCAÇÃO DINÂMICA.....	63
3.1 ALOCAÇÃO ESTÁTICA	67
3.2 ALOCAÇÃO DINÂMICA	67
LEITURA COMPLEMENTAR.....	68
RESUMO DO TÓPICO 3.....	71
AUTOATIVIDADE	72
 UNIDADE 2 – ESTRUTURAS DE DADOS	 73
 TÓPICO 1 – CADEIA DE CARACTERES	 75
1 INTRODUÇÃO	75
2 CADEIA DE CARACTERES.....	75
2.1 CARACTERES	75
2.2 CADEIA DE CARACTERES (STRINGS).....	80
2.2.1 Constantes do tipo caracteres	80
2.2.2 Variáveis do tipo caracteres.....	82
2.2.3 Cadeia de caracteres e operações	83
2.2.4 Strings.....	84
2.3 VETOR DE CADEIA DE CARACTERES	86
2.3.1 Vetores unidimensionais	89
2.3.2 Leitura de vetores	91
2.3.3 Escrita de vetores	91
RESUMO DO TÓPICO 1.....	93
AUTOATIVIDADE	95
 TÓPICO 2 – TIPOS DE ESTRUTURA DE DADOS.....	 97
1 INTRODUÇÃO	97
2 TIPOS ESTRUTURADOS.....	98
2.1 TIPO ESTRUTURA	98

2.1.1 Variáveis do tipo heterogênea.....	100
2.1.2 Variáveis do tipo homogênea.....	101
2.1.3 Arranjos unidimensionais	101
2.1.4 Arranjos multidimensionais.....	104
2.2 PONTEIRO PARA ESTRUTURAS.....	106
2.2.1 Operadores	108
2.3 DEFINIÇÃO DE “NOVOS” TIPOS.....	110
2.3.1 Comando typedef.....	113
2.4 VETORES DE ESTRUTURAS.....	113
2.5 VETORES DE PONTEIROS PARA ESTRUTURAS	114
2.5.1 Funções de vetores de ponteiros para estruturas.....	116
2.6 TIPO UNIÃO	119
2.7 TIPO ENUMERAÇÃO	120
LEITURA COMPLEMENTAR.....	121
RESUMO DO TÓPICO 2.....	122
AUTOATIVIDADE	124
 TÓPICO 3 – MATRIZES.....	 125
1 INTRODUÇÃO	125
2 MATRIZES	125
2.1 MATRIZES ESPECIAIS.....	128
2.2 VETORES BIDIMENSIONAIS – MATRIZES.....	129
2.3 MATRIZES DINÂMICAS	132
2.4 REPRESENTAÇÃO DE MATRIZES.....	134
2.5 REPRESENTAÇÃO DE MATRIZES SIMÉTRICAS.....	137
2.6 TIPOS ABSTRATOS DE DADOS.....	142
2.6.1 Tipo de dado abstrato Ponto.....	144
2.6.2 Tipo de dado abstrato Matriz	145
RESUMO DO TÓPICO 3.....	148
AUTOATIVIDADE	149
 UNIDADE 3 – ESTRUTURAS DE DADOS AVANÇADAS.....	 151
 TÓPICO 1 – LISTAS ENCADEADAS	 153
1 INTRODUÇÃO	153
2 A LISTA ENCADEADA.....	155
2.1 NÓS E LISTA ENCADEADA	157
2.2 DEFININDO A INTERFACE.....	159
2.3 IMPLEMENTAÇÃO DOS MÉTODOS.....	160
2.3.1 Método adicionaNoComeco().....	160
2.3.2 Método adiciona().....	162
2.3.3 Método adicionaPosicao()	163
2.3.4 Método pega()	165
2.3.5 Método removeDoComeco().....	165
2.3.6 Método removeDoFim()	167
2.3.7 Método remove()	169
2.3.8 Método contem().....	170
2.3.9 Método tamanho()	171
2.4 VERIFICANDO O FUNCIONAMENTO DOS MÉTODOS.....	171
2.4.1 Classe TesteAdicionaNoFim	172
2.4.2 Classe TesteAdicionaPorPosicao	172
2.4.3 Classe TestePegaPorPosicao.....	173

2.4.4 Classe TesteRemovePorPosicao	174
2.4.5 Classe TesteTamanho	174
2.4.6 Classe TesteContemElemento	175
2.4.7 Classe TesteAdicionaNoComeco	175
2.4.8 Classe TesteRemoveDoComeco	176
2.4.9 Classe TesteRemoveDoFim	176
2.5 CÓDIGO COMPLETO DA CLASSE LISTA ENCADEADA	177
RESUMO DO TÓPICO 1.....	182
AUTOATIVIDADE	183
 TÓPICO 2 – LISTA DUPLAMENTE ENCADEADA	 185
1 INTRODUÇÃO	185
2 MÉTODO ADICIONANOCOMEÇO()	185
3 MÉTODO ADICIONA().....	187
4 MÉTODO ADICIONAPOSICAO().....	188
5 MÉTODO REMOVEDOCOMEÇO()	188
6 MÉTODOS REMOVEDOFIM() E REMOVE().....	189
RESUMO DO TÓPICO 2.....	191
AUTOATIVIDADE	192
 TÓPICO 3 – PILHAS	 193
1 INTRODUÇÃO	193
2 O CONCEITO DE PILHAS.....	194
3 A IMPLEMENTAÇÃO DA CLASSE PEÇA	194
3.1 SOLUÇÃO DOS PROBLEMAS DAS PEÇAS	195
3.2 OPERAÇÕES EM PILHAS	198
3.2.1 Inserir uma peça.....	198
3.2.2 Remover uma peça	198
3.2.3 Informar se a Pilha está vazia	199
3.2.4 Generalização	200
RESUMO DO TÓPICO 3.....	204
AUTOATIVIDADE	205
 TÓPICO 4 – FILAS.....	 207
1 INTRODUÇÃO	207
2 O CONCEITO DE FILAS	208
3 INTERFACE DE USO.....	208
3.1 OPERAÇÕES EM FILA.....	210
3.1.1 Inserir uma pessoa.....	210
3.1.2 Remover uma pessoa	211
3.1.3 Informar se a Fila está vazia	212
3.2 GENERALIZAÇÃO.....	212
LEITURA COMPLEMENTAR.....	216
RESUMO DO TÓPICO 4.....	227
AUTOATIVIDADE	228
REFERÊNCIAS.....	229

INTRODUÇÃO AO ESTUDO DE LINGUAGEM DE PROGRAMAÇÃO E ESTRUTURAS DE DADOS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade, você será capaz de:

- conceituar estruturas de dados;
- entender sua aplicabilidade na linguagem de programação;
- utilizar suas funcionalidades para estruturar dados

PLANO DE ESTUDOS

Esta unidade do caderno de estudos está dividida em três tópicos. No final de cada um deles você encontrará atividades visando à compreensão dos conteúdos apresentados.

TÓPICO 1 - CONCEITOS FUNDAMENTAIS - INTERPRETAÇÃO VERSUS
COMPILAÇÃO

TÓPICO 2 - FUNÇÕES

TÓPICO 3 - VETORES E ALOCAÇÃO DINÂMICA

CONCEITOS FUNDAMENTAIS - INTERPRETAÇÃO VERSUS COMPILAÇÃO

1 INTRODUÇÃO

A estrutura de dados surgiu com o objetivo de auxiliar a tecnologia da informação a estruturar a linguagem de programação, nesse sentido podem ser utilizados vários tipos de associações entre estruturas de dados e linguagem de programação. A estrutura de dados fica com a responsabilidade de organizar e armazenar os dados e os algoritmos com a finalidade de desenvolver, implementar e utilizar os melhores códigos para cada tipo de estrutura de dados. Veloso; Santos; Azeredo (1983), definem a estruturação da informação como “fabula”, onde computadores servem para armazenar informações e programas para manipulá-los. Seguindo, colocam a estrutura de dados com a definição de relações lógicas existentes entre os dados, de modo analógico ao uso de um modelo matemático para espelhar alguns aspectos de uma realidade física.

Este é um contexto em que se definem a importância e utilização da linguagem de programação e estrutura de dados, as estruturas de dados são organizadas, seguras, quando são utilizadas de forma adequada pelos programadores, pois a execução e estruturação de dados são responsabilidade dos programadores, não basta conhecer apenas como desenvolver um programa, utilizar a melhor linguagem para um projeto, precisa conhecer como manipular e estruturar os dados de um programa.

A Estrutura de dados é um processo muito importante, segundo Edelweiss e Galante (2009, p. 36), “um fator que determina o papel dessas estruturas no processo de programação de aplicações é a identificação de quão bem as estruturas de dados coincidem com o domínio do problema a ser tratado”. Por isso é essencial à linguagem de programação o suporte no desenvolvimento do programa e auxílio absoluto das estruturas de dados, esta lógica permite que sejam agregados valor na especificação dos dados, relacionando de forma estruturada os dados, as variáveis e seus elementos. Guimarães e Lages, (2013, p. 52), afirmam:

(...) decisões sobre estruturas dos dados não podem ser feitas sem conhecimento dos algoritmos aplicados a eles, e vice-versa: a estrutura e a escolha dos algoritmos dependem muitas vezes fortemente da estrutura dos dados, e, a arte de programar consiste na arte de organizar e dominar a complexidade.

Faz parte, como processo de estrutura de dados, o estudo sobre a interpretação e compilação de dados. A interpretação e compilação são formas de como um computador pode interpretar e compilar um código fonte de um programa, podemos, assim dizer, que um computador precisa identificar, interpretar códigos e linguagens, após esta sequência de processos de interpretação o computador irá realizar a execução das tarefas solicitadas. Aguilar (2011) apresenta compilação como um processo de tradução de programas-fonte para um programa-objeto. O programa-objeto obtido da compilação é traduzido normalmente para código fonte da máquina. Conforme Costa (2010, p. 26), na técnica de interpretação, um programa é convertido em outro de forma dinâmica, automática.

As etapas que envolvem a linguagem de programação e estruturas de dados pode ser tudo que envolve a lógica de programação, desde a utilização de dados, organização de memória, utilização de códigos e linguagens específicas para cada necessidade. A linguagem de programação em C, por exemplo, é considerada uma linguagem muito rápida para ser executada e possui seu próprio compilador de código. A linguagem C é composta por várias características, como Tipos Primitivos de dados, Constantes e Variáveis, Atribuição, Operadores aritméticos e Funções.

A linguagem de programação em C possui uma característica muito interessante, em sua portabilidade é permitido converter e executar programas, sistemas operacionais e *hardware* diferentes do seu código-fonte. A linguagem C realiza a compilação de programas através do Código-Fonte, Código-Objeto e Código-Executável. Segundo Aguilar (2011, p. 42), o programa-fonte deve ser traduzido para linguagem de máquina, processo realizado com o compilador e o sistema operacional, que se encarrega, na prática, da compilação.

A programação Java, por sua vez, é uma linguagem que atualmente está em ascensão, é muito utilizada para vários tipos de aplicações, principalmente para aplicações *web*. Sua principal característica está na segurança que sua aplicação disponibiliza, sua agilidade para ser executado e confiabilidade em ser utilizado para vários tipos de aplicações. Como suas aplicações estão voltadas principalmente para produtos que imitam o mundo real podemos citar os aplicativos e programas para jogos de computadores.

2 CONCEITOS FUNDAMENTAIS DE INTERPRETAÇÃO E COMPILAÇÃO

Especificamente para que um computador funcione, receba e execute funções, primeiramente este precisa entender os códigos que formam os programas, um computador funciona e executa suas tarefas através de programas de computador. Alguns processos auxiliam os computadores nessa função de entender as linguagens de programação e para que esses executem de forma correta as atividades advindas da programação.

Para entender melhor será apresentada a contextualização e as diferenças entre Interpretação de Compilação. Na arquitetura de linguagens de programação, a interpretação e compilação são processos que ocorrem de formas distintas, no entanto, as duas somam para o processo da linguagem de programação.

Acadêmico, analisando as palavras Interpretação e Compilação o que consegue subentender sobre cada uma?

Interpretação vem de interpretar, já compilar vem de reunir, considerando que estamos falando sobre a computação, podemos perceber que o computador precisa interpretar os códigos e linguagens, reunir os conjuntos de códigos, para assim executar tarefas, analisando por este ângulo fica mais fácil entender. Vamos aos conceitos destes dois termos da arquitetura de linguagens de programação. Iniciando pela definição de programa de Interpretação.

2.1 INTERPRETAÇÃO

O interpretador possui como função executar o código-fonte, traduzindo o programa a cada linha de código que vai sendo escrito (digitado), assim que o computador consegue interpretar os códigos, o programa vai sendo executado, a cada execução do código o programa precisa ser novamente interpretado pelo computador. Conforme Costa (2010, p. 26), na técnica de interpretação, um programa é convertido em outro, dinamicamente. Toda vez que uma execução for necessária, o código-fonte será traduzido para uma linguagem que possa ser executada, direta ou indiretamente, pelo sistema operacional.

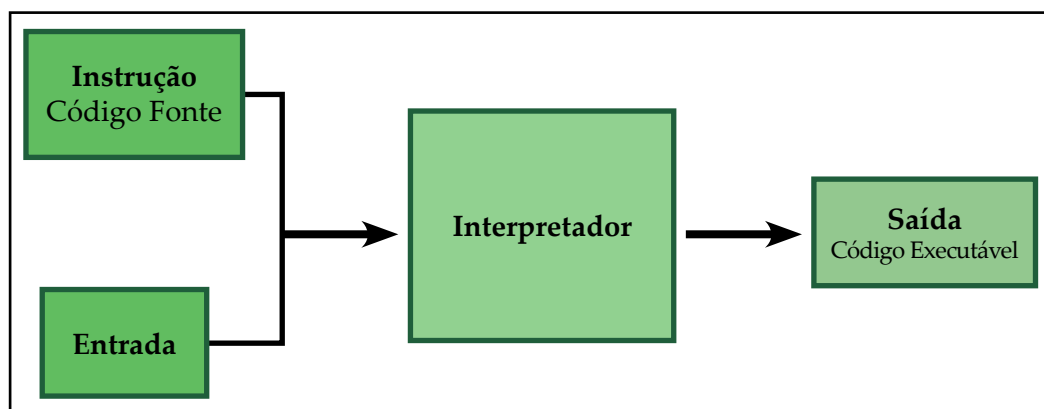


Conforme Costa (2010, p. 26), “quando escrevemos programas de um computador em uma linguagem de programação, o conjunto de instruções que formam o programa é chamado de código-fonte. O código-fonte define as instruções do programa de computador, que, para estarem corretas, devem atender à estrutura léxica, sintática e semântica da linguagem de programação utilizada”.

Conforme Lopes (1997), um interpretador compila e executa um programa fonte. No entanto, esse processo pode ocorrer de forma mais lenta que no programa compilador, pois um interpretador precisa traduzir/executar cada código que está sendo inserido no programa, isso torna o processo de verificação da existência de problemas em uma execução mais demorada, pois, só após cada execução do programa o computador vai identificar a existência de problemas no código-fonte. O programa de interpretação possui a característica de executar cada linha de código digitada no programa de computador, necessariamente o código é interpretado e executado pelo programa em sequência.

Os programas interpretados tornam o processo de tradução para o sistema máquina mais rápido para sua execução, possuem maior dinamicidade e podem ser utilizados em vários sistemas operacionais, possuem maior portabilidade. No entanto, seu desempenho na execução dos códigos para a linguagem da máquina é inferior aos programas compilados.

FIGURA 1- EXECUÇÃO DO PROGRAMA INTERPRETAÇÃO



FONTE: A autora

2.2 COMPILAÇÃO

A seguir será exposto a definição do contexto da compilação, este precisa de um programa fonte, como, por exemplo, um sistema operacional que realiza a função de executar, gerar outro programa, mas neste caso uma linguagem denominada de alto nível. Segundo Costa (2010, p. 27),

(...) a técnica mais comumente utilizada para a tradução de programas é a compilação. Nessa técnica, um programa especial, chamado compilador, recebe como entrada o código-fonte de um programa e produz como saída o mesmo programa, porém escrito em outra linguagem. O resultado da compilação pode ser chamado de código objeto, código intermediário ou código executável, de acordo com o processo de compilação realizado e a terminologia utilizada.

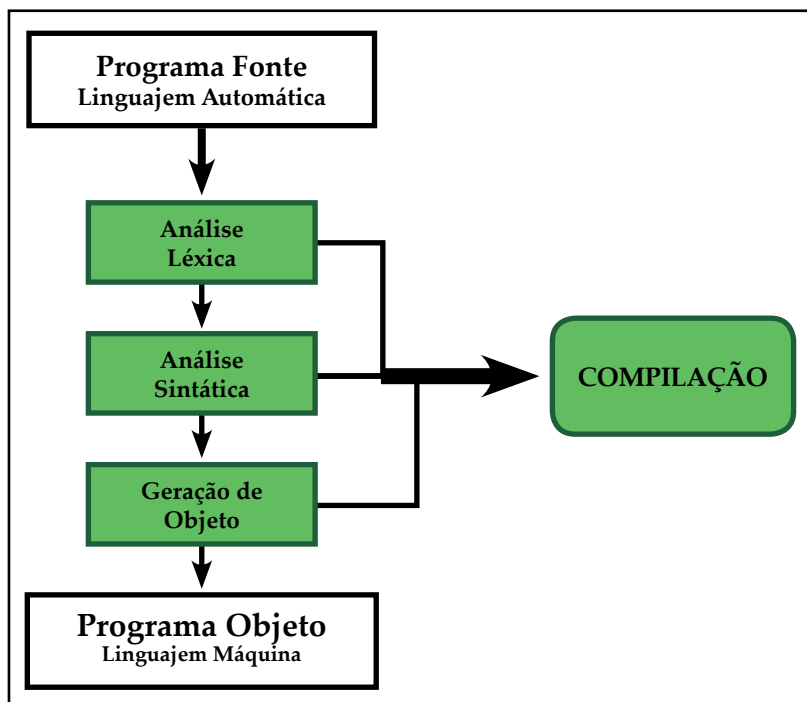


Conforme Lorenzi e Lopes (2000) o Fortran e Cobol foram as primeiras linguagens de programação de alto nível, cujas construções não mostram qualquer semelhança com a linguagem de máquina.

Conforme Lopes (1997, p. 52), “um compilador tem a função de transformar um algoritmo codificado em uma linguagem X para a linguagem da máquina equivalente”. Conforme Sebesta (2010), os programas podem ser traduzidos para linguagem da máquina, a qual pode ser executada diretamente no computador. Esse método é chamado de implementação baseada em compilação, com a vantagem de ter uma execução de programas muito mais rápidos, uma vez que o processo de tradução estiver completo. Uma função muito específica do compilador, para que o mesmo seja executado, todo o programa fonte precisa estar codificado e digitado, após este processo ocorre a execução do programa, resumidamente, para que o programa seja executado, todo o código precisa ser compilado, para ser executado pelo computador.

Costa (2010) afirma, quando compilamos programas de computador utilizamos resultado desse processo de compilação para executar os programas. Depois de compilado, o programa pode ser executado indefinidamente, sem a necessidade de repetição do processo de tradução. Uma compilação só será necessária quando uma mudança no código-fonte do programa for refletida no programa compilado.

FIGURA 2 - EXECUÇÃO DO PROGRAMA COMPILADOR



FONTE: A autora

3 LINGUAGEM DE PROGRAMAÇÃO C

Caro acadêmico(a), recorda-se que anteriormente estávamos falando sobre linguagens de programação que são consideradas de alto nível, a linguagem de programação C possui essa característica. Outra característica da linguagem C pode ser observada a partir do seu próprio compilador, este por sua vez executa os códigos, gerando programas executáveis muito menores e muito mais rápidos do que outras linguagens de programação.

Como uma das funcionalidades da linguagem C, precisa executar de forma repetida suas funções, realiza um processo de separar as atividades definidas e codificadas, com isso tornando o processo de compilação e execução muito mais rápidos.

Conforme Cocian (2004), uma das grandes vantagens da linguagem C é a sua característica de “alto nível” e de “baixo nível” ao mesmo tempo, permitindo o controle total da máquina (*hardware* e *software*) por parte do programador, permitindo efetuar ações sem depender do sistema operacional utilizado. Isso ocorre porque foi desenvolvida com o objetivo de facilitar a vida dos engenheiros na prática de programas, sistemas robustos muito extensos, tentando ao máximo minimizar a quantidade de erros que podem ocorrer.

Cocian (2004, p. 32) destaca as vantagens da linguagem de programação C tornar-se cada vez mais conhecida e utilizada, como:

1. A portabilidade do compilador.
2. O conceito de bibliotecas padronizadas.
3. A quantidade e variedade de operadores poderosos.
4. A sintaxe elegante.
5. O fácil acesso ao *hardware* quando necessário.
6. A facilidade com que as aplicações podem ser otimizadas, tanto na codificação, quanto na depuração, pelo uso de rotinas isoladas e encapsuladas.

A linguagem C é considerada uma linguagem tanto de alto nível, quanto médio e baixo nível, no entanto sua utilização se deve ao grande reaproveitamento dos códigos na programação de um produto. Utiliza a forma de linguagem de programação baseada nas metodologias Imperativas e Procedural, isso quer dizer que são ações e comando dos códigos que sofrem alterações, pode mudar de estado, conforme como deve ser executado o código, como segue etapas e caminhos para alcançar o objetivo final do produto em desenvolvimento.

3.1 TIPOS PRIMITIVOS DE DADOS

A definição de dados pode ser compreendida, estruturada e formada por inúmeros dados que são estipulados para cada objetivo em que o computador tenha que entender e executar alguma tarefa. Conforme Sebesta (2010, p. 268):

Um tipo de dado define uma coleção de valores de dados e um conjunto de operações pré-definidas sobre eles. Programas de Computador realizam tarefas quão bem os tipos de dados disponíveis na linguagem usada casam com os objetivos no espaço do problema do mundo real. Logo é crucial uma linguagem oferecer suporte para uma coleção apropriada de tipos e estruturas de dados.

Com essa afirmação pode-se perceber que as estruturas de dados permitem que a linguagem de computador possa interpretar e compilar os dados e códigos para uma determinada linguagem de programação. Edelweiss e Galante (2009), descrevem os tipos de dados primitivos, em que não possuem uma estrutura sobre seus valores, ou seja, não é possível decompor o tipo primitivo em partes menores. Os tipos básicos são, portanto, indivisíveis.

Consideravelmente este tipo de dado não pode sofrer alterações, pois é definido por ter campos de dados infinitos, podemos citar como exemplo desse dado o CPF de um usuário.

Os tipos de dados primitivos são formados por números inteiros, números, ponto flutuantes, caracteres e tipos nulos, conforme pode ser observado na tabela a seguir, onde possui o código e a função de cada um deles em uma estrutura de dados primitivos.

TABELA 1: TIPO DE DADO PRIMITIVO

Palavra (Código)	Tipo Primitivo
<i>char</i>	caracter
<i>int</i>	inteiro
<i>float</i>	real de precisão simples
<i>double</i>	real de precisão dupla
<i>void</i>	vazio (sem valor)

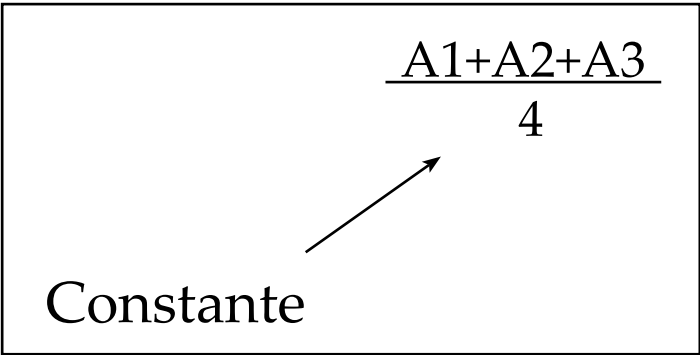
FONTE: A autora

3.2 CONSTANTES E VARIÁVEIS

3.2.1 Constantes

Segundo Almeida (2008, p. 27), “constantes são dados cujos valores são fixos e que não se alteram ao longo da execução de um programa”. Exemplos: números, letras, palavras, entre outros. Geralmente, a constante é definida no início do programa e sua utilização acontece no decorrer dele.

FIGURA 3 - CONSTANTES



FONTE: A autora

Os dados CONSTANTES são definidos por um único valor, desde o início do programa, até o final de sua execução, isto resultando em um processo, em que não é possível os dados já definidos serem alterados, sejam alterações por comandos ou funções.

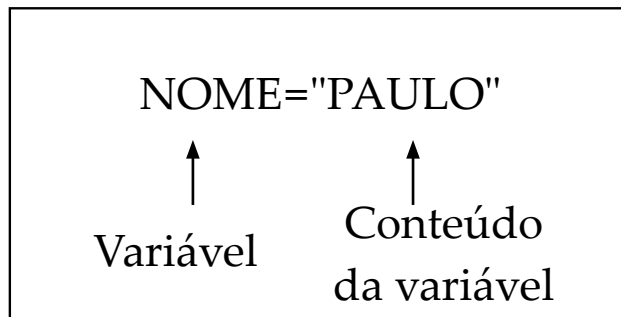
3.2.2 Variáveis

Variáveis por sua vez, conforme define Cocian (2004, p. 99), “devem ser declaradas antes de serem utilizadas”. Existem dois tipos de variáveis: Globais e Locais. Variáveis globais devem ser declaradas no início do programa e fora de qualquer função. As variáveis locais devem ser declaradas no início da função, onde elas serão válidas. Os dados variáveis seguem uma lógica muito parecida com as constantes, no entanto elas são declaradas no início do programa e no início das funções.

Almeida (2008, p. 27), por sua vez destaca que “as variáveis usadas em algoritmos devem ser declaradas antes de serem utilizadas, visto que os compilados precisam reservar um espaço na memória para cada um deles”. Resumindo, uma variável são as formas de como o computador disponibiliza um espaço para armazenar os dados, estas variáveis ficam abertas, disponíveis para receber qualquer dado, podem ser alteradas quando existir a necessidade de mudança. Por isso esses dados, são denominados de variáveis, pois podem sofrer alterações e serem mudadas quando for necessário.

Cocian (2004, p. 100), afirma que “as variáveis podem ser de vários tipos, como: *int* (interias), *float* (real)”. Os nomes das variáveis indicam o endereço de memória onde está um determinado dado. Conforme Almeida (2008), variável é a representação simbólica dos elementos de certo conjunto. Cada variável ocupa uma posição de memória, cujo conteúdo pode ser modificado durante a execução de um programa. Mesmo que uma variável assuma diferentes valores, ela só poderá armazenar um valor por vez.

FIGURA 4 - VARIÁVEL



FONTE: A autora

Especificamente um dado variável pode obter um único valor, isso desde o início até o final das execuções do programa, esse procedimento ocorre devido a um processo em que o computador armazena espaço na memória, para os dados manipulados. São dois tipos de variáveis que podem ser encontradas, Variável Numérica e Variável Alfanumérica. A variável numérica é definida por receber dados numéricos, com a funcionalidade de realizar cálculos. A variável

alfanumérica é definida por receber dados como caracteres e dados em texto. Outra característica dos dados variáveis está relacionada com a declaração da variável, pois ao realizar este procedimento de declarar a variável existe a necessidade de informar o nome e qual o tipo de dado que esta receberá.

FIGURA 5 - IDENTIFICADOR DA VARIÁVEL

Ao "batizarmos uma variável, ou seja, ao atribuirmos um nome à uma variável, devemos escolher um nome sugestivo, que indique o conteúdo que ela receberá. Por exemplo:

Valor 1 → Variável que armazenará o primeiro valor

Valor 2 → Variável que armazenará o segundo valor

Resultado → Variável que armazenará o resultado de um cálculo qualquer

FONTE: Ancibe (2014)

3.3 ATRIBUIÇÃO

Um sistema possui como principal função executar e processar os dados, conforme solicitações e comandos estipulados. Nesse sentido o operador de atribuição possui a responsabilidade de atribuir valor para um dado variável, o símbolo deste pode ser identificado por: (=), porém o sinal de igual, neste caso não está relacionado com a matemática de igualar algum valor, ou cálculo de valores. Para a linguagem C, o cálculo realizado está entre uma expressão variável identificada, segue exemplo:

identificador = expressão;

O identificado é definido pelo nome da variável e a expressão é o identificador.

Segundo Edelweiss e Livi (2014, p. 78), “o operador de atribuição atribui o valor que está à direita da variável localizada à esquerda. O valor da expressão à direita pode ser uma constante, uma variável ou qualquer combinação válida de operadores”.

Exemplo:

TABELA 2: ATRIBUIÇÃO

$x = 4;$	Significa que a variável x recebe o valor de 4
$val = 2.5;$	Variável val recebe valor de 2,5
$y = x + 2;$	A variável y recebe valor de x mais valor de 2
$y = y + 4;$	Variável y recebe valor que estava armazenado nela mesma, mais valor de 4

FONTE: A autora.

Pode-se perceber que os valores atribuídos, sempre ocorrem da direita para a esquerda, sucintamente, o valor das expressões podem ser tanto constantes, quanto variáveis, outro fator muito importante para a atribuição é sua disponibilidade de realizar a atribuição de um valor, para várias outras variáveis, isso ocorre através de um encadeamento de atribuições.

Conforme Exemplo:

$$\text{total_um} = \text{total_dois} = \text{somatorio} = 0;$$

3.4 OPERADORES ARITMÉTICOS

Os operadores aritméticos realizam o processo de cálculo diferentemente da atribuição, neste caso os cálculos seguem as regras estabelecidas na álgebra, onde os operadores compilam os valores da esquerda para a direita. Segundo Rebollo (2013), operadores aritméticos podem ser usados em qualquer tipo de dado escalar sejam números inteiros, reais e caractere. Quando realizamos operações com o mesmo tipo de dado o resultado é do mesmo tipo. Se forem do tipo diferente, a conversão será automática e tentará converter para o tipo mais apropriado para a operação.

Os operadores binários são: $*$ (multiplicação), $/$ (divisão), $\%$ (modulo), $+$ (soma), $-$ (subtração).

Os operadores unários são: $+$ (positivo) e $-$ (negativo), $++$ (incremento) – (decremento).



Dados Escalar são valores individuais sem componentes interno, assim como os exemplos de NUMBER, DATE ou BOOLEAN.

Já os tipos de dados escalares são definidos nas seguintes categorias da Tabela 2.

Tipos de dados	Descrição
Numeric	Valores numéricos em que as operações aritméticas são realizadas.
Character	Valores alfanuméricos que representam caracteres simples ou strings de caracteres.
Boolean	Valores lógicos nos quais as operações lógicas são realizadas.
Datetime	Datas e horas basicamente.

FONTE: Disponível em: Tipos de dados Escalar e LOB <<http://www.devmedia.com.br/pl-sql-tipos-de-dados-escalar-e-lob/29824#ixzz3CwQmVhuz>>

TABELA 3: OPERADORES ARITMÉTICOS

Operador	Símbolo	Descrição
Adição	+	Soma duas ou mais variáveis.
Subtração	-	Subtrai duas ou mais variáveis.
Multiplicação	*	Multiplica os valores de duas ou mais variáveis.
Divisão	/	Divide o valor de duas ou mais variáveis.
Inteiro da Divisão entre dois números	\	Retoma a parte inteira da divisão entre dois números.
Exponenciação	^	$x^y \rightarrow$ É o valor de x elevado à potência y.
Módulo	Mod	Retoma o resto da divisão de dois números.

FONTE: Fialho (2011)

Exemplos de Operadores Aritméticos:

A seguir um Exemplo de utilização de variáveis e operadores aritméticos, segundo o autor Fialho (2011):


```

        ' Declaração de variáveis.
        Dim x, y, z As Integer
x = 10
y = 25
z = x*y
        ' Nesse exemplo a variável z conterà o valor 250

```

3.5 FUNÇÕES

Analisando que a linguagem C é constituída por vários blocos de construção, logo, esses blocos são considerados como funcionais, isso permite que um programa seja desenvolvido através desses blocos funcionais, formando assim as estruturas das funções.

Como na utilização da matemática, as funções em C recebem vários parâmetros e esses geram um valor como resultado de uma função, lógico esse processo ocorre com auxílio dos programadores que executam as funções, esse processo ocorre de forma ordenada, resultando em várias funções executadas pelo próprio programa.

Segundo Aguilar (2011, p. 197), “as funções formam um miniprograma dentro de um programa. As funções economizam espaço, reduzindo repetições e tornando mais fácil a programação, proporcionando um meio de dividir um projeto grande em pequenos módulos mais manejáveis”. Seguindo essa linha, Mokarzel e Soma (2008, p. 44), salientam que:

todos os módulos de um programa em C são denominados funções e cada função é composta de um cabeçalho e um escopo. O principal elemento do cabeçalho é o nome da função, mas ele pode conter também o tipo de valor que ela produz e os parâmetros sobre os quais ele deve atuar.

Analisando as funcionalidades e características das funções em C, pode ser observado que sua responsabilidade é executar comandos e executar uma ou várias tarefas, consideradas como sub-rotinas dentro de um programa. Essas sub-rotinas são responsáveis por receber as informações ou as solicitações, realiza o processamento dessas informações retornando a solicitação com outra informação. Conforme Dornelles (1997, p.74), “existem dois tipos de funções: funções de biblioteca e funções de usuário. Funções de biblioteca são funções escritas pelos fabricantes do compilador e já estão pré-compiladas, isto é, já estão escritas em código de máquina. Funções de usuário são funções escritas pelo programador”.

3.5.1 Uso de Funções

Para que se possa utilizar uma função, existe a necessidade de realizar a declaração desta função, devem ser definidos os parâmetros que a função irá receber e também quais são os parâmetros que esta função deve retornar como resposta para o programa, essas informações ficam armazenadas no manual do usuário do compilador do sistema.

Segue exemplo, conforme Dornelles (1997), de uma sintaxe declarando uma função:

(tipo_ret nome(tipo_1, tipo_2, ...))

Onde nome é o nome da função, *tipo_1, tipo_2, ...* são os tipos (e quantidade) de parâmetros de entrada da função e *tipo_ret* é o tipo de dado de retorno da função. Além dos tipos usuais vistos existe ainda o tipo *void* (vazio, em inglês) que significa que aquele parâmetro é inexistente.

Na linguagem C, pode ser encontrado um conjunto mínimo de instruções, estas possuem como objetivo desenvolver programas executáveis com tamanho em proporções, quanto menor forem melhor será o objetivo alcançado. No entanto só podem ser adicionadas novas funcionalidades para a linguagem C, com inclusões feitas através de bibliotecas, em que a estrutura dessas bibliotecas são formadas por classe de funções. Cada classe da biblioteca possui uma responsabilidade específica para o tratamento dos dados solicitados. Outro fator muito importante em se tratando de linguagens de programação em C é que a inserção de bibliotecas devem ser as primeiras instruções do programa a ser desenvolvido.

FIGURA 6 - EXEMPLO DE FUNÇÕES DE BIBLIOTECA

```
#include <stdio.h>    // biblioteca de funções de entrada e saída
#include <stdlib.h>    // biblioteca de funções do sistema operacional

int main()
{
    printf("Primeiro programa em C!\n");
    system("PAUSE");
    return(0);
}
```

FONTE: Pereira (1996)

A seguir vamos analisar um exemplo, também do autor Dornelles (1997), com relação a função da biblioteca, demonstrando como a mesma pode ser declarada.

A função *cos ()* da biblioteca *math.h* é declarada como:
double cos (double);

Isto significa que a função tem um parâmetro de entrada e um parâmetro de saída, ambos são do tipo *double*.

Exemplo: A função *getch ()* da biblioteca *conio.h* é declarada como:
int getch (void);

Isto significa que a função não tem parâmetros de entrada e tem um parâmetro *int* de saída. Para podermos usar uma função de biblioteca devemos incluir a biblioteca na compilação do programa. Esta inclusão é feita com o uso da diretiva *#include* colocada antes do programa principal.

Exemplo: Assim podemos usar a função no seguinte trecho de programa:

FIGURA 7 - FUNÇÃO DE INCLUIR DE BIBLIOTECA

```
#include <math.h>           // inclusão de biblioteca
void main() {              // início do programa principal
    double h = 5.0;        // hipotenusa
    double co;              // cateto oposto
    double alfa = M_PI_4;   // angulo:  $\pi/4$ 
    co = h * cos(alfa);     // calculo: uso da funcao cos()
}                           // fim do programa
```

FONTE: Dornelles (1997)

FIGURA 8 - PRINCIPAIS FUNCIONALIDADES DE BIBLIOTECA

Biblioteca	Principais funcionalidades
stdio.h	entrada e saída de dados.
stdlib.h	alocação de memória e comandos para o sistema operacional.
math.h	funções matemáticas.
time.h	manipulação de dados nos formatos de data e hora.
ctype.h	manipulação de caracteres.
string.h	manipulação de cadeias de caracteres.
conio.h	manipulação do cursor na tela.

FONTE: Pereira (1996)

Seguem alguns exemplos das funções disponíveis nas Bibliotecas C, isso segundo a ideia de Dornelles (1997): Biblioteca *math.h* e Biblioteca *stdlib.h*.

Biblioteca *math.h*:

int **abs** (*int i*);
double **fabs** (*double d*);

Calcula o valor absoluto do inteiro *i* e do real *d*, respectivamente.

double **sin** (*double arco*);
double **cos** (*double arco*);
double **tan** (*double arco*);
double **asin** (*double arco*);
double **acos** (*double arco*);
double **atan** (*double arco*);

Funções trigonométricas do ângulo **arco**, em radianos.

double **ceil** (*double num*);
double **floor** (*double num*);

Funções de arredondamento para inteiro.

Ceil () arredonda para cima. Ex.: *Ceil* (3.2) = 3.0;
floor () arredonda para baixo. Ex.: *Floor* (3.2) = 4.0;

double **log** (*double num*);
double **log10** (*double num*);

Funções logarítmicas: *log* () é logaritmo natural (base e), *log10* () é logaritmo decimal (base 10).

double **pow** (*double base, double exp*);
 Potenciação: *pow* (3.2,5.6) = 3.25.6.

double **sqrt** (*double num*);
 Raiz quadrada: *sqrt* (9.0) = 3.0.

Biblioteca *stdlib.h*

int **random** (*int num*);

Gera um número inteiro aleatório entre 0 e *num* - 1.

Nos próximos estudos veremos mais alguns exemplos de funções de bibliotecas e conheceremos mais sobre os exemplos de código em C e posteriormente como realizar a Compilação de programas em C, até para entender as funcionalidades dessas funções na linguagem de programação C.

4 EXEMPLO DE CÓDIGO EM C

Segundo Cocian (2004), a linguagem C possui várias características positivas, como, por exemplo, podemos citar sua portabilidade que indica a facilidade de se converter um programa feito para um *hardware* específico e sistema operacional, em um equivalente que possa ser executado em outro *hardware* específico ou sistema operacional. Ainda segundo o autor a estrutura de um programa em C é composto por Cabeçalho, Bloco de Instruções e por Comentários do Programa.

No cabeçalho possui as diretivas de compilador onde se definem o valor de constantes simbólicas, declaração de variáveis, inclusão de bibliotecas, macros, entre outros. Já no bloco de instruções, chamados de função principal (*main*) e outros blocos de funções secundárias. Na estrutura de comentários do programa, consistem a documentação *is situ*. (COCIAN, 2004, p. 97).

Segundo Celes e Rangel (2002), um programa em C tem que, obrigatoriamente, conter a função principal (*main*). A execução de um programa começa pela função principal (a função *main*). As funções auxiliares são chamadas, direta ou indiretamente, a partir da função principal.

Celes e Rangel (2002), demonstram outras características positivas que tornam a linguagem de programação C muito útil para o desenvolvimento de vários tipos de produtos, vamos verificar isso a seguir:

Em C, como nas demais linguagens “convencionais”, devemos reservar área de memória para armazenar cada dado. Isto é feito através da declaração de variáveis, na qual informamos o tipo do dado que iremos armazenar naquela posição de memória. Assim, a declaração *float t1*; do código mostrado, reserva um espaço de memória para armazenarmos um valor real (ponto flutuante – *float*). Este espaço de memória é referenciado através do símbolo *t1*.

Uma característica fundamental da linguagem C diz respeito ao tempo de vida e à visibilidade das variáveis. Uma variável (local) declarada dentro de uma função “vive” enquanto esta função está sendo executada, e nenhuma outra função tem acesso direto a esta variável. Outra característica das variáveis locais é que devem sempre ser explicitamente inicializadas antes de seu uso, caso contrário conterão “lixo”, isto é, valores indefinidos. (CELES; RANGEL, 2002, p. 95 e 96).

4.1 PALAVRA RESERVADA

As palavras reservadas possuem como objetivo permitir que as tarefas a serem executadas e solicitadas ocorram de forma correta pelo computador, isso pode definir como os objetos, variáveis, constantes e subprogramas devem ser criados para esta linguagem de programação.

Uma palavra reservada, em uma linguagem de programação, já possui um significado definido na linguagem, não podendo ser utilizada para criação de outros objetos, tais como variáveis, constantes ou subprogramas, pois não permitirá a correta execução da tarefa solicitada ao computador.

FONTE: Disponível em: <http://cae.ucb.br/conteudo/programar/ltp1/ltp1_conceitosfundamentais.html>. Acesso em: 08 out. 2014

Conforme Cocian (2004, p. 97), “a linguagem de programação em C, possui 32 palavras reservadas, no entanto isso está definido pelos padrões da *American National Standards Institute (ANSI)*”. Na tabela a seguir vamos conhecer estas palavras reservadas:

FIGURA 9 - PALAVRAS RESERVADAS

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

FONTE: Souza (2009)

4.2 COMENTÁRIOS

Os comentários permitem que o computador compreenda de forma mais rápida as tarefas executadas e solicitadas pelo programa. Segundo Rebollo (2013), os comentários podem ser de uma linha ou comentários de múltiplas linhas. Comentários de múltiplas linhas começam com “/*” e terminam com “*/” tudo que estiver entre esses símbolos é considerado comentário. Comentário de uma linha inicia com “//” e termina no final da linha. Esse processo é necessário para agilizar o tempo de respostas, fazendo com que os comentários sejam compilados para que seja compreensível pelo computador e que auxilie o programador

a entender a solicitação. Os comentários servem para auxiliar no processo de documentar o código fonte, esse processo facilita a compreensão, principalmente se este programa possui centenas de linhas de códigos.

4.2.1 Utilizando Barra dupla

Na inclusão de um comentário que ocorra em uma única linha, dentro de um código é utilizado duas barras (//), como por exemplo:

FIGURA 10 - BARRA DUPLA

```
#include <stdio.h>

int main ( ) {

    //Mensagem de saudação
    printf("Bem vindo!");
    printf("\nEstou a aprendendo Linguagem c!");
    printf("\nBye, bye...");

    //Esperando o usuário pressionar <ENTER> para sair do programa
    printf("\n\nPressione <ENTER> para sair do programa.");
    getchar( );

    return 0;

}
```

FONTE: Rebollo (2013)

Exemplo de um comentário com barra dupla contendo uma linha no final do código:

FIGURA 11 - BARRA DUPLA – LINHA NO FINAL

```
#include <stdio.h>

int main ( ) {

    printf("Bem vindo!");
    printf("\nEstou a aprendendo Linguagem c!");
    printf("\nBye, bye...");
    printf("\n\nPressione <ENTER> para sair do programa.");

    getchar( ); //Esperando o <ENTER>
    return 0;

}
```

FONTE: Rebollo (2013)

4.2.2 Comentários com /* */

Neste exemplo de inserção de comentário é que podem ser inseridas várias linhas, geralmente utilizadas no cabeçalho do programa, dentro do código.

FIGURA 12 - BARRA DUPLA – CONTENDO DOIS ASTERÍTCOS

```
/*
Programa exemplo
Linguagem C - Uma Introdução
--
Diego M. Rodrigues
*/

#include <stdio.h>

int main ( ) {

printf("Bem vindo!");
printf("\nEstou a aprendendo Linguagem c!");
printf("\Bye, bye...");
printf("\nPressione <ENTER> para sair do programa.");
getchar( );

return 0;

}
```

FONTE: Rebollo (2013)

4.3 IDENTIFICAÇÃO

Identação nada mais é do que a organização da escrita de códigos-fonte, essa organização ocorre de forma horizontal, definindo a ordem das linhas de códigos em forma hierarquizada. A seguir vamos analisar um exemplo da linha de código, conforme organização da identificação:

```
Variavel A = 0
SE (Variavel A < 1) ENTÃO
    ESCREVA "Variável A é menor que 1."
FIM SE
```

Os comandos que fazem parte do escopo são SE – ENTÃO – FIM SE, analisando o comando acima pode-se perceber que ESCREVA está alinhado mais à direita, indicando que não pertence ao escopo principal deste programa. A identificação não possui especificações para uma única linguagem de programação, no entanto ela pode ser utilizada em qualquer linguagem, porém ela pode sofrer comportamentos diferentes, dependendo do uso das linguagens de programação.

FIGURA 13 - EXEMPLO DE COMANDO EM IDENTAÇÃO

```
#include <stdio.h>

int main ( ) {

    int A=0/
    if ( A < 1 ) {
        printf("Variável A é menor que 1.");
    }

    getch ( );
    return 0;

}
```

FONTE: Rodrigues (2014)

5 COMPILAÇÃO DE PROGRAMAS EM C

Como já visto no início deste tópico de estudos, os compiladores são identificados como programas que realizam a tradução do código-fonte para uma linguagem compilada e entendível para o computador. O processo de tradução (compilação) implica na conversão de programa, expresso em código-fonte, em um programa equivalente, expresso em código-executável. A seguir vamos conhecer esses três componentes da compilação de programas.

Código-fonte: é um código escrito em uma linguagem de programação. Os programas-fontes são normalmente compostos de diversos códigos-fontes, armazenados em vários arquivos.

Código-objeto: é o código gerado na linguagem de máquina da arquitetura-alvo. Esse código, entretanto, não pode ser diretamente executado pelo processador, já que nem todas as referências necessárias à execução estão resolvidas. Pode faltar, por exemplo, referências a endereços de funções que estão em outros códigos-objetos.

Código executável: é o código gerado na linguagem de máquina da arquitetura-alvo, com todas as referências resolvidas, que pode ser diretamente executado pelo processador. O arquivo contendo esse código é chamado de programa executável. (PINHEIRO, 2012, p. 6)

Segundo Aguilar (2011, p. 33), o processamento de execução de um programa escrito em uma linguagem de programação e por meio de um compilador costuma obedecer os 7 (sete) seguintes passos:

1. Escrever o programa-fonte com um editor e guardá-lo em um dispositivo de armazenamento, como, por exemplo, um disco.
2. Introduzir o programa-fonte em memória.
3. Compilar o programa com o compilador C.
4. Verificar e corrigir erros de compilação.
5. Obtenção do programa-objeto.
6. O montador obtém o programa executável.
7. Executa-se o programa e, se não houver erros, se obterá a saída do programa.

Seguindo a análise de Pinheiro (2012, p. 6), o compilador C realiza a compilação dos programas-fonte em quatro etapas, como descritos a seguir:

Pré-processamento: nesta etapa o texto do programa é transformado lexicamente. Ocorre a supressão de espaços desnecessários, substituição de macros e, em especial, a inclusão de outros textos indicados pelas diretivas de pré-processamento *#include*. O texto resultante é chamado de unidade de compilação.

Compilação: nesta etapa ocorre a análise sintática e semântica da unidade de compilação. Caso não haja erros, é gerado o código *assembler* correspondente.

Montagem: nesta etapa ocorre a geração do código-objeto. Os comandos *assembler* são transformados em linguagem de máquina, faltando, entretanto, resolver as referências a (endereços de) objetos e funções implementadas em outros códigos-objetos, como, por exemplo, as referências às funções das bibliotecas do sistema.

Ligação: nesta etapa ocorrem a combinação de todos os códigos-objetos que compõem o programa e a resolução das referências não resolvidas na etapa anterior. O resultado é um código executável.

O compilador possui como responsabilidade transformar os programas fontes em programas objetos e assim executar de forma compreensível em uma linguagem máquina.

Os programas-fontes, em geral, são armazenados em arquivos cujo nome tem a extensão “*.c*”.

Os programas executáveis possuem extensões que variam com o sistema operacional: no *Windows*, tem extensão “*.exe*”; no *Unix (Linux)*, em geral, não tem extensão.

Para exemplificar o ciclo de desenvolvimento de um programa simples, consideremos que o código apresentado na seção anterior tenha sido salvo num arquivo com o nome *prog.c*.

Devemos então compilar o programa para gerarmos um executável. Para ilustrar este processo, usaremos o compilador *gcc*. Na linha de comando do sistema operacional, fazemos:

```
> gcc -o prog prog.c
```

Se não houver erro de compilação no nosso código, este comando gera o executável com o nome *prog* (*prog.exe*, no *Windows*). Podemos então executar o programa:

```
> prog
```

Digite a temperatura em Celsius: 10

A temperatura em Fahrenheit vale: 50.000000

```
>
```

Obs.: Em *itálico* está representado as mensagens do programa e, em **negrito**, exemplificamos um dado fornecido pelo usuário via teclado.

Conforme *site* Thoth, a compilação traduz os programas escritos em linguagem de alto nível (por exemplo, a linguagem) para instruções interpretáveis pelo processador (código máquina):

```
Gcc -g -c -o stack.o stack.c
```

- *c* apenas compilar não invocar o editor de ligações (linker).
- *I* caminho de procura dos ficheiros de inclusão, indicados pelas directivas `#include`.
- *g* incluir, juntamente com o código gerado, informação para debugging.
- *O* otimizar o código gerado.
- *o* para indicação do nome do ficheiro com o código gerado.



Conheça o compilador GCC – GNU Compiler Collection em: <http://www.debian.org/releases/stable/ia64/ch01s02.html.pt>
 Passo a passo para compilar usando sistema operacional Linux ou Windows, segue informações: https://linux.ime.usp.br/~lucasmmg/livecd/documentacao/documentos/terminal/Compilando_um_arquivo_em_C.html
<http://fig.if.usp.br/~esdobay/c/gcc.html>

Vamos exemplificar de forma bem simples como executar uma compilação utilizando o sistema operacional *Linux*. Abra uma arquivo kwrite e insira o seguinte comando:

1. Criar o código-fonte abaixo e salvá-lo com o nome de “ola.c), se não colocar o “c” não irá funcionar:

FIGURA 14 - COMPILANDO

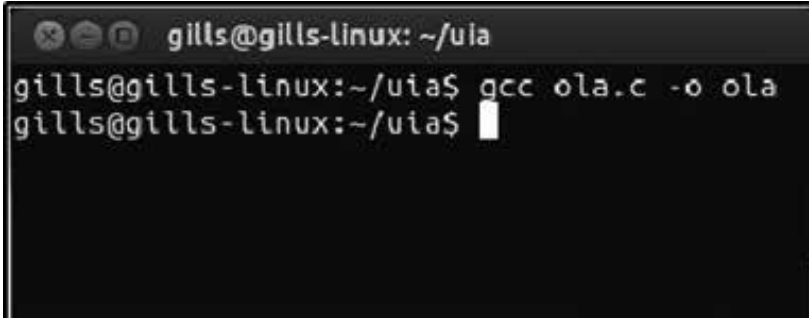
```
#include <stdio.h>

int main(){
    printf("\nola, nola, mundo!\n\n");
    return 0;
}
```

FONTE: A autora

2. Ir até o terminal, navegar (cd) até o diretório-raiz, onde o arquivo-fonte foi salvo, e dar o seguinte comando: “gcc ola.c -o ola” [ENTER]. Este segundo “ola” que aparece no comando, é o nome do programa executável, que pode ser qualquer outro nome, mas no nosso caso será “ola”. Se o comando possuir qualquer erro, o mesmo irá aparecer na tela, se estiver tudo correto com o comando, a tela abaixo irá aparecer:

FIGURA 15 - COMPILANDO


A terminal window with a dark background. The title bar shows window control icons and the text 'gills@gills-linux: ~/uia'. The terminal text shows the user 'gills' at the prompt 'gills@gills-linux:~/uia\$' typing the command 'gcc ola.c -o ola'. The command is executed, and the prompt returns to 'gills@gills-linux:~/uia\$' with a cursor on the next line.

```
gills@gills-linux: ~/uia
gills@gills-linux:~/uia$ gcc ola.c -o ola
gills@gills-linux:~/uia$
```

FONTE: Disponível em: <http://www.academia.edu/4297165/Minitutorial_Compilando_e_rodando_programas_C_no_GNU_Linux> acesso em: 08 set. 2014

3. Para confirmar se realmente o comando do programa foi compilado, navegue pelo diretório-raiz do arquivo-fonte, verificando que foi criado o arquivo “ola”, com permissões diferentes das do ola.c:

FIGURA 16 - COMPILANDO

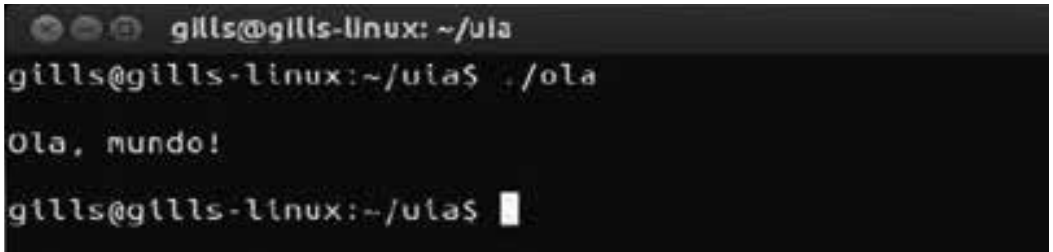
A terminal window with a dark background. The title bar shows window control icons and the text 'gills@gills-linux: ~/uia'. The terminal text shows the user 'gills' at the prompt 'gills@gills-linux:~/uia\$' typing the command 'ls -l'. The output shows the permissions and details for files 'ola' and 'ola.c'. The command is executed, and the prompt returns to 'gills@gills-linux:~/uia\$' with a cursor on the next line.

```
gills@gills-linux: ~/uia
gills@gills-linux:~/uia$ ls -l
total 12
-rwxrwxr-x 1 gills gills 7157 Oct  3 21:51 ola
-rw-rw-r-- 1 gills gills  92 Oct  3 21:45 ola.c
gills@gills-linux:~/uia$
```

FONTE: Disponível em: <http://www.academia.edu/4297165/Minitutorial_Compilando_e_rodando_programas_C_no_GNU_Linux> acesso em: 08 set. 2014

4. Para executar o comando do programa, digite ./ola [ENTER]. Como resposta de saída do programa a seguinte mensagem irá aparecer na tela: ola, nola, mundo!

FIGURA 17 - COMPILANDO



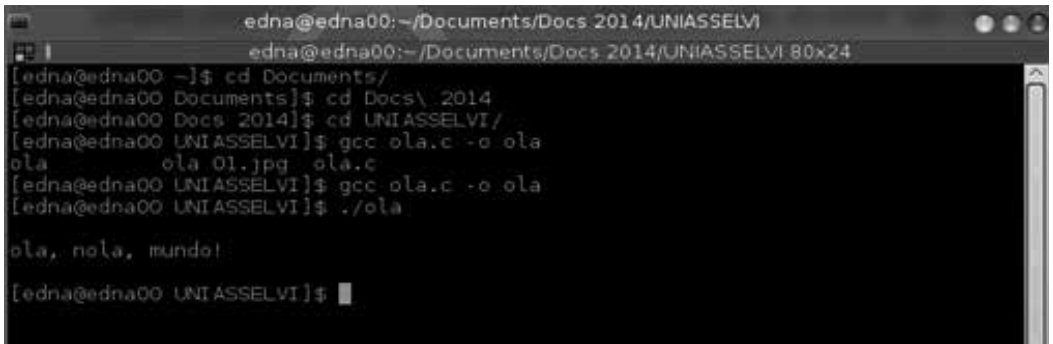
```
gills@gills-linux: ~/uia
gills@gills-linux:~/uia$ ./ola

Ola, mundo!

gills@gills-linux:~/uia$
```

FONTE: Disponível em: <http://www.academia.edu/4297165/Minitutorial_Compilando_e_rodando_programas_C_no_GNU_Linux> acesso em: 08 set. 2014

FIGURA 18 - COMPILANDO



```
edna@edna00:~/Documents/Docs 2014/UNIASSELVI
edna@edna00:~/Documents/Docs 2014/UNIASSELVI 80x24
[edna@edna00 ~]$ cd Documents/
[edna@edna00 Documents]$ cd Docs\ 2014
[edna@edna00 Docs 2014]$ cd UNIASSELVI/
[edna@edna00 UNIASSELVI]$ gcc ola.c -o ola
ola      ola 01.jpg  ola.c
[edna@edna00 UNIASSELVI]$ gcc ola.c -o ola
[edna@edna00 UNIASSELVI]$ ./ola

ola, nola, mundo!

[edna@edna00 UNIASSELVI]$
```

FONTE: A autora

6 LINGUAGEM DE PROGRAMAÇÃO JAVA

Conforme *site* JAVABR, o Java é uma linguagem de programação e plataforma computacional lançada pela primeira vez pela *Sun Microsystems* em 1995. O Java é rápido, seguro e confiável. De *laptops* a *datacenters*, *consoles de games* a supercomputadores científicos, telefones celulares à Internet, o Java está em todos os lugares! A programação Java possui como objetivo programar em alto nível, onde seus comandos e códigos de programas são compilados diretamente em uma máquina virtual, tornando o processo muito mais rápido e não consumindo, assim, memória e processamento de um computador. O grande diferencial desta linguagem, além de ser popularmente muito utilizada, é sua aplicação *web*, pois possui como objetivo disponibilizar ambientes muito parecidos com o mundo real, por ser uma linguagem rápida e inteligente.

Conforme dados do *site* JavaBr, pode-se observar algumas facilidades e vários pontos positivos da utilização desta linguagem, como seguem as descrições:

A base da programação Java são as classe e seus objetos, que 'imita', o mundo real, o que facilita bastante a programação. Por exemplo, os carros são uma classe, já um gol é um objeto da classe carro. As classes possuem métodos e características que são comuns a todos os objetos. Essa associação com o mundo real ajuda bastante na hora abstração, de criar aplicações complexas.

O Java é bastante flexível, por conta da possibilidade de expansão através das bibliotecas, ou APIs, além das extensões do Java, voltada especificamente para desenvolvimento de aplicações para *desktop*, para celulares, para empresas, para áudio, para gráficos 3D, banco de dados, para aplicações de Internet, criptografia, computação/sistemas distribuídos, linguagem de marcação e várias outras.

FONTE: Disponível em: <http://www.java.com/pt_BR/download/faq/whatis_java.xml>. Acesso em: 08 Set. 2014

A linguagem Java é considerada uma linguagem que possui segurança e portabilidade. Segundo Horstmann (2008, p. 38), a linguagem possui vários recursos de segurança que garantem que nenhum *applet* nocivo possa ser executado no seu computador. Com este benefício extra, esses recursos também ajudam a aprender a linguagem mais rapidamente.

Outro fator muito interessante e importante da linguagem Java foi sua grande contribuição para impulsionar a forma como eram desenvolvidos os programas, possui uma especificidade de desenvolver seus códigos orientados a objetos. Boraks (2013, p. 11) afirma que o Java simplificou a programação geral na *web*, ela inovou com um tipo de programa de rede chamado *applet* que, na época, mudou a maneira do mundo *on-line* pensar em conteúdo.

6.1 EXEMPLOS DE CÓDIGOS

FIGURA 19 - PRIMEIRO PROGRAMA EM JAVA

```
/*
 * O primeiro programa em Java: Hello World
 * Autor: Jacques Sauv  
 */
// Todo programa tem um ponto de entrada: o "m  todo" main de
alguma "classe"
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

FONTE: Feij  ; Silva e Clua (2010)

6.2 APPLETS JAVA

Os *applets* possuem a característica de possibilitar e desenvolver programas dentro de páginas em HTML, tornam as páginas da *web* muito mais intuitivas, pois disponibilizam formas animadas, dinâmicas e interativas para a *web*. Conforme Boraks (2013, p. 11), um “*applet* é um tipo especial de programa Java que é projeto para ser transmitido pela internet e executado automaticamente por um navegador *web* compatível com o Java”. Basicamente, os *applets* permitem que uma funcionalidade seja movida do servidor para o cliente. Schutzer e Massago (2010, p. 16), afirmam que:

(...) *applets* são pequenos programas Java que podem ser inseridos dentro de uma página HTML. Com este recurso, uma página torna-se dinâmica, podendo interagir com o usuário que a consulte. Um *applet* pode ainda executar tarefas complexas, como realizar cálculos e apresentar gráficos, sons e imagens em movimento.

Schutzer e Massago (2010), apresentam uma inserção de *applet* em uma página HTML, como exemplo, usando a diretiva **<applet>**, a qual deve apresentar pelo menos três parâmetros: **code**, **width** e **height**, veja a seguir o exemplo:

```
<applet  
  code = [java applet]  
  width = [largura]  
  height = [altura]>
```

6.3 SEGURANÇA

Um fator muito importante para a linguagem de programação Java é a segurança que fica a cargo do *applet*, não permitindo que programas não confiáveis e não autorizados acessem arquivos de computador. Ainda segundo Boraks (2013, p. 11), “para o Java permitir que o *applet* fosse baixado e executado com segurança no computador cliente, era necessário impedir que ele iniciasse esse tipo de ataque, não permitindo que sejam acessadas outras partes do computador”.

Outro fator observado por Horstmann (2008), com relação à segurança na utilização da linguagem Java, onde a máquina virtual pode capturar muitos erros de iniciantes e informá-los de uma maneira precisa.

6.4 PORTABILIDADE

Uma característica muito positiva da linguagem de programação Java é sua portabilidade de ser utilizada em qualquer tipo de equipamento tecnológico, seja ele um equipamento móvel ou equipamento estático fisicamente, suas funcionalidades vão além do uso de apenas um computador, mas podem ser vistos em várias tecnologias, como computadores, *laptop*, *tablets* e *smartphones*. O fator de portabilidade é um aspecto muito importante da internet e dos equipamentos, porque há muitos tipos de computadores e muitos tipos de sistemas, a portabilidade permite que o mesmo código funcione em vários computadores.

Boraks (2013, p. 12) afirma que, “algum meio de gerar código executável e portátil era necessário”. Felizmente, o mesmo mecanismo que ajuda a manter a segurança dos sistemas e computadores, permite também auxiliar a gerar portabilidade, neste caso estamos falando do *applet*. Segundo Horstmann (2008, p. 38),

a portabilidade também é um requisito para *applets*. Quando você visita uma página *web*, o servidor *web* que disponibiliza o conteúdo da página não faz ideia do computador que você está utilizando para navegar pela *web*. Ele simplesmente retorna o código portátil que foi gerado pelo compilador Java. A máquina virtual no seu compilador é quem executa esse código portátil que foi gerado pelo compilador Java.

Sua portabilidade é fazer com que haja comunicação entre vários equipamentos tecnológicos, e que consigam se comunicar de forma segura, gerando assim um ponto muito positivo em termos de portabilidade e de segurança para a linguagem Java.

6.5 J2ME

Pode-se afirmar que a plataforma Java 2 Micro *Edition* – J2ME revolucionou a comunicação entre os equipamentos móveis, tornando o processo de comunicação muito mais eficaz, como também permite, tanto equipamentos, quanto sistemas e aplicativos de empresas diferentes consigam realizar a comunicação entre si. Segundo Mattos (2005), a plataforma Java 2 Micro *Edition* – J2ME é destinada a dispositivos com recursos limitados de memória, vídeos e processamento, possui como objetivo fazer com que qualquer produto possa interagir com recursos de maneira única e simplificada.

Somera (2006, p. 6), diz que “J2ME é um conjunto de tecnologias e especificações destinadas ao consumidor, com dispositivos como telefones móveis, PDAs e impressoras, sendo estas com sistemas Mobile ou que utilizam sistemas *Wireless*”. Mattos (2005, p. 18), afirma ainda que “cada dispositivo executa internamente uma máquina virtual Java desenvolvida por seu fabricante, e os aplicativos em Java de terceiros interagem apenas com a máquina virtual e não com o dispositivo real”. Fiorese (2005), classifica a plataforma J2ME, onde

sua implementação à linguagem de programação Java, como qualquer outra linguagem, é composta por um grupo de comandos predefinidos que, após escritos e compilados podem ser executados resultando em ações específicas.

Resumidamente é uma tecnologia de programação em Java que permite sua aplicabilidade em equipamentos móveis, como celulares, *laptop* e outros dispositivos que tenham como objetivo a navegação de dados e informações pela internet, nas páginas *web*.

7 EXEMPLO DE CÓDIGO EM JAVA

A linguagem de programação Java possui vários fatores que a classificam como positivo, como, por exemplo, a sua portabilidade que significa a facilidade de realizar a comunicação entre equipamentos, sistemas e aplicativos de empresas diferentes. Outro fator positivo é sua segurança que garante que a comunicação seja eficiente entre esses vários equipamentos e sistemas, conseguindo proteger os equipamentos de invasões que possam prejudicar e danificar os sistemas e máquinas. Sua estrutura de código pode ser considerada como uma grande vantagem para o desenvolvimento e programação, pois sua estrutura e linguagem básica facilitam, assim, a escrita dos códigos de forma mais simplificada.

A seguir vamos conhecer a estrutura de código em Java, alguns exemplos de comandos que são utilizados na programação Java, conforme diretrizes do autor Gaúcho (2014):

```
1. // Duas barras significam comentário
2. /* comentários também podem seguir o formato de C++ */
3.
4. public class NomeDoPrograma
5. {
6.     // O método main sempre deve estar presente para que um código
7.     // Java possa ser executado:
8.     static public void main(String[] args)
9.     {
10. // aqui virão os comandos, que são parecidos com C++
11. }
12. }
```

Conforme Gaúcho (2014, p. 27), podemos analisar os comandos à cima da seguinte forma:

Linhas 1 e 2: representam comentários, pode ser uma informação, comportamento do programa, do autor, ou da versão do programa.

Linha 3: está em branco, pois Java permite linhas em branco entre os comandos.

Linha 4: é a declaração do "nome do programa", que é *case-sensitive* (existe diferença entre maiúsculas e minúsculas). O arquivo deve ser salvo com o mesmo nome que aparece após a declaração *public class* e mais a extensão .java, deve ser salvo: NomeDoPrograma.java.

Linha 5 e 9: a abertura de chave { indica início de bloco.

Linha 8: essa linha deve aparecer em todos os códigos Java. Quando um programa Java é executado, o interpretador da JVM executa os comandos que estiverem dentro do bloco indicado pelo método "*static public void main(String)*".

Linha 10: aqui seria escrito o código propriamente dito. Instruções como *for-next* e *print*.

Linha 11 e 12: o fechamento da chave } indica início do bloco.

8 COMPILAÇÃO DE PROGRAMAS EM JAVA

Vamos analisar os comandos básicos de Java para a aplicação a seguir, relacionando novamente o exemplo do comando "ola mundo" e vamos compilar este exemplo:

FIGURA 20 - COMPILAÇÃO JAVA

```
/**
 * Instituto de Software do Ceará - INSOFTEC
 * XI Semana tecnológica de férias
 * Primeiro programa - escrever a mensagem alô mundo na tela.
 */
public class AloMundo
{
    static public void main(String[] args)
    {
        System.out.println("Alo Mundo");
    }
}
```

FONTE: Disponível em: <<http://www.milfont.org/blog/wp-content/upload/Manual.pdf>> Acesso em: 10 set. 2014

Acadêmico, vamos apresentar agora como rodar o comando do programa acima, conforme exemplo do autor Gaúcho (2014):

1. Salve o código acima em um arquivo nomeado: AloMundo.Java (não esqueça o *case-sensitive*);
2. Digite no console:
 - a. C:\fic>javac AloMundo.java
3. Caso não ocorra nenhuma mensagem de erro, digite:
 - a. C:\fic>java AloMundo

FONTE: Disponível em: <<http://www.milfont.org/blog/wp-content/upload/Manual.pdf>> Acesso em: 10 set. 2014

Analizando o código acima referente à compilação do programa:

As primeiras 5 linhas representam um bloco de comentário, que tem por objetivo identificar a função do programa, seu autor, versão etc.

A linha seguinte (`public class AloMundo`) declara uma classe chamada AloMundo. Após compilado, esse código gerará um arquivo AloMundo.class no mesmo diretório em que se encontra o código-fonte.

Algumas dicas em relação à linguagem, os códigos e seus comandos Java:

- Lembrando que um código-fonte em Java pode descrever mais de uma classe.
- Após a compilação do comando, cada descrição de classe gerará um arquivo .class de forma separada.
- Lembre-se, pode haver no máximo uma classe *public* dentro de cada código-fonte Java.
- Caso você, inadvertidamente, declare mais de uma classe como *public* dentro de um código-fonte Java, ocorrerá um erro de compilação e este erro será apresentado na tela.
- Todo o corpo da classe (o código que define a classe) deve ser delimitado por chaves, assim como toda a estrutura de dados.

FONTE: Adaptado de: <<http://www.milfont.org/blog/wp-content/upload/Manual.pdf>> Acesso em: 10 set. 2014

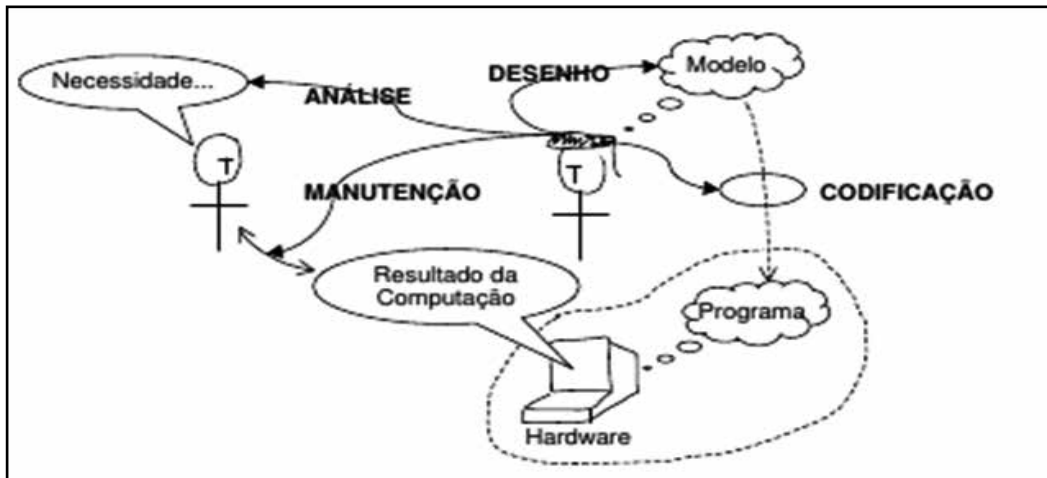
9 CICLO DE DESENVOLVIMENTO

O ciclo de desenvolvimento de um sistema em Java segue basicamente uma metodologia imposta para o desenvolvimento de *softwares* em outras linguagens, onde todas elas precisam passar pelo caminho da identificação de uma necessidade do usuário, precisa modelar o sistema, isso quer dizer, desenhar

o sistema conforme as necessidades do usuário e por fim precisa codificar o sistema. Codificar, necessariamente, é definir o código de linguagem que será utilizado para a programação e desenvolvimento do sistema.

A seguir será apresentado um exemplo de ciclo de desenvolvimento de um sistema.

FIGURA 21 - CICLO DE DESENVOLVIMENTO DE UM SISTEMA



FONTE: Vega (2004)

Segundo Akita (2006, p. 398), “todo desenvolvimento, ou mesmo correções, devem necessariamente, seguir o mesmo ciclo: codificar em ambiente de desenvolvimento, versionar, instalar em ambiente de testes, realizar todos os Casos de Teste”.

RESUMO DO TÓPICO 1

Nesse tópico, você viu:

- Deve-se observar que tanto a linguagem de programação em C quanto Java, são linguagens de programação de alto nível. A linguagem C possui como característica executar as funções de forma repetitiva, no entanto executa de forma rápida e eficiente, isso torna o processo de compilação muito rápido.
- Um fator muito importante para a linguagem C é a interpretação que possui a função de executar a código-fonte, traduzir o programa para que seja interpretado pela linguagem máquina (linguagem de computador).
- A atribuição na linguagem C possui como responsabilidade atribuir, processar e executar um dado variável.
- A linguagem de programação JAVA possui como característica ser rápida para sua execução, é confiável, segura e possui portabilidade para a comunicação entre vários equipamentos e sistemas de vários fabricantes diferentes. Seu principal ponto positivo é sua aplicabilidade para uso na internet e desenvolvimento de dispositivos móveis e aplicativos *web*.
- A compilação tanto na linguagem C, quanto na linguagem Java, possui o mesmo objetivo, que é executar os códigos em uma linguagem de fácil entendimento para o computador.



1 Simplificando, a compilação é um processo em que o programa compila o recebimento de entrada de um código-fonte, produzindo uma saída do mesmo programa, porém escrito em outra linguagem. Esse processo ocorre para que o código seja compreendido pelo computador que recebe este código-fonte. Conforme as características da compilação em linguagem C é correto afirmar:

- a) () Método chamado de implementação, baseada em compilação.
- b) () Processo de decodificação, resultando de um algoritmo função.
- c) () Portabilidade de sistemas, execução de comandos e dados móveis.
- d) () Execução de ciclo de vida, o código-executável possui início e fim.

2 A portabilidade da linguagem Java é uma característica muito positiva, possibilitando que qualquer tipo de equipamento tecnológico consiga conversar um com os outros, permitindo desta forma uma evolução para a área de tecnologia *web*. Esse processo de portabilidade ocorre através do:

- a) () Compilador Java.
- b) () Código de máquina.
- c) () Servidor de dados.
- d) () Atribuição de código.

1 INTRODUÇÃO

A estrutura de dados é composta por muitas funcionalidades e características, as quais têm como objetivo auxiliar a linguagem de programação e aos programadores, tornando os programas mais estruturados. Partindo dessa análise podemos falar de um elemento muito importante para as estruturas de dados que são as Funções, estas possuem como finalidade realizar o processo de chamada de uma referência, de um valor. O valor de uma função fica armazenado em uma variável. A função é considerada um executor de funções, com isso divide as grandes tarefas em subtarefas, para que uma função não se repita em uma chamada a mesma é transformada em uma nova função e que poderá ser chamada de forma repetida.

A pilha de execução está classificada como estrutura de dados, esta possui como objetivo estruturar a linguagem de programação, o procedimento para que essa estrutura aconteça disponibiliza confiança, pois todo o processo de inclusão de elementos, funções e códigos precisam necessariamente passar pelo topo da pilha. Possui como funcionalidade específica utilizar as funções *Last In, First Out*.

Os ponteiros de variáveis também fazem parte da estrutura de dados, sendo estas que cumprem um papel muito importante dentro dessa estrutura, pois os ponteiros possuem como responsabilidade armazenar espaço na memória do programa para elementos e valores. Este procedimento de armazenar espaço na memória ocorre após ser declarada uma variável, após essa declaração da variável a memória armazena o tamanho e o espaço que essa variável vai ocupar dentro do programa.

A recursividade é um elemento que tem a função, dentro da estrutura de dados, de executar a chamada da mesma função, esta pode ser tanto uma recursividade direta, quanto uma recursividade indireta.

2 FUNÇÕES

As funções são procedimentos em uma linguagem de programação no recebimento dos blocos de códigos, as funções possuem como responsabilidade apresentar algum valor sobre a ação solicitada. Precisamente, a função deve retornar sempre um valor para o comando solicitado no programa, esse processo pode ser chamado de passagem de valor ou de referência. Na passagem de valor, o valor atribuído fica contido em um argumento ou copiado para uma variável de parâmetro formal. Na passagem por referência ocorre o processo da chamada da mesma função onde é apontada para mesma posição na memória. Outra característica da função, é que pode ser considerado como um subprograma, e assim agindo sobre os dados e posteriormente retornar para o programa principal, um único valor.

Esse processo de ser atribuído à função um subprograma se deve à divisão que ocorre de grandes tarefas que são divididas em tarefas menores, realiza o processamento de sub-rotina e retorna para o programa principal outra informação, outro valor. Esse procedimento de dividir as tarefas maiores em tarefas menores proporciona outra vantagem para a função, isso evita que os códigos sejam repetidos, e que o procedimento solicitado se transforme em uma função e esta possa ser chamada por diversas vezes. A função realiza o trabalho de agrupar conjuntos de comandos da linguagem de programação, esse processo disponibiliza uma resposta muito mais rápida, pois os parâmetros ficam armazenados e podem ser chamados e consultados diversas vezes. A função possui a responsabilidade de agilizar as chamadas, organizar os parâmetros, realizando desta forma uma economia do código-fonte de um programa.

Como descrito anteriormente uma função é considerada subprograma, esta precisa receber um valor numérico, realizar o cálculo e por fim retornar com um valor único deste comando. São encontrados dois tipos de funções, a de biblioteca e de usuário. A função de biblioteca são funções já existentes, estipuladas, escritas pelo fabricante do compilador, as funções já estão pré-compiladas, já estão escritas em código máquina. A função de usuário são funções escritas pelos próprios programadores de um sistema.

Outras características da função é que a mesma possui recursividade, isso ocorre quando a função invoca-se a si mesma, esse procedimento realiza economia de espaço na memória do computador, estes são mais compactos e seus demais dados e parâmetros também são recursivos.

Mendes (2011, p. 114), coloca que “a função recursiva são funções que obtêm um resultado através de várias chamadas à própria função. Entretanto, as chamadas recursivas devem ser limitadas para evitar o uso excessivo de memória. Nesse sentido, deve-se estar atento para que a função verifique a condição de término de uma recursão”.

A seguir um exemplo de comando Recursivo:

FIGURA 22 - FUNÇÃO RECURSIVA

<u>versão não recursiva:</u>	<u>versão recursiva:</u>
<pre>static int soma (int n) { int i; int soma =0; for (i=n; i>=0; i--) soma += i; return soma; }</pre>	<pre>static int soma (int n) { if(n>0) return n+soma(n-1); else return 0; }</pre>

FONTE: Braz (2014)

A função possui formas de passagem de argumentos, estes são definidos como mecanismos que permitem a transmissão de dados para uma função e são divididos em: argumentos pretendidos e argumentos de procedimento. O argumento possui como funcionalidade receber a chamada e tornar genérico esses dados recebidos para o uso adequado da função, essas passagens de argumentos podem ser identificadas de duas formas, sendo a Passagem de Valor e a Passagem de Referência.

Na passagem por valor em uma chamada de função é realizado o processo de pegar o valor desse dado que está contido em um argumento ou parâmetro real, realizar a transmissão para a variável do parâmetro formal. Já na passagem por referência, o processo ocorre em uma chamada da função, de um parâmetro real, formal para apontar a mesma posição da memória.

2.1 FUNÇÕES EM JAVA

As funções são classificadas como rotinas e sub-rotinas automatizadas, isso define o processo de reutilização de código, pois se existe a necessidade de utilizar uma codificação já existente, apenas precisa ser chamada a função. As funções podem ser utilizadas sempre que existir a necessidade de utilizar um código específico, apenas precisa ser criada ou utilizada a mesma função. Outra característica de funções é que elas são muito úteis para a codificação de um sistema, são adaptáveis a vários tipos de métodos.

2.1.1 Criando funções sem Argumentos

Um ponto a ser observado é que uma função é estática, pode ser sempre reutilizada, pode ser citada, como exemplo, a função mais básica utilizada para executar uma rotina, conhecida como função *main*, esse comando é utilizando em um código, dentro de uma classe.

A seguir vamos apresentar um exemplo de uma função sem argumento, de acordo com Xavier (2010):

```
01. public class ExemploFuncao {  
02.     //criando a função  
03.     public static void mostrarMensagem() {  
04.         System.out.println("Minha Mensagem");  
05.     }  
06.  
07.     public static void main(String[] args) {  
08.         //chamando a função dentro do programa  
09.         mostrarMensagem();  
10.     }  
11. }
```

2.1.2 Criando funções com Argumentos

As funções com argumentos também são funções estáticas, podem ser reutilizadas, no entanto ela se difere por possuir apenas as informações necessárias para que a função consiga executar o processo com argumentos, estas ficam descritas dentro de parênteses. Levando em conta um fator muito importante nesta função é que a função com argumento pode ter entre um e até vários argumentos, desde que estes sejam separados por vírgulas. Outro fator que deve ser observado neste caso é que cada argumento deve ter seu tipo declarado, conforme seguem os exemplos abaixo, de acordo com o autor Xavier (2010, p. 35):

```
public static void funcao1 (String arg1) {}  
public static void funcao2 (int arg1, int arg2) {}  
public static void funcao3 (String arg1, char arg2, int arg3, float arg4, Object  
arg5) {}
```

A seguir será apresentado um exemplo desta função de resultado fatorial, será aplicada uma função de 1 a 10, resultando em um número no fator final.

```

01. public class FatorialComFuncao {
02.     public static void fatorar(int numero) {
03.         int fator = 1;
04.         for (int i = numero; i > 1; i--) {
05.             fator *= i;
06.         }
07.         System.out.println(numero + "! = " + fator);
08.     }
09.
10.     public static void main(String args[]) {
11.         for (int x=1; x<=10; x++)
12.             fatorar (x);
13.     }
14. }

```

FONTE: Xavier (2010)

2.1.3 Criando Funções sem Retorno

Vamos utilizar para criar funções o *JavaScript*, neste caso é utilizada a palavra-chave *Function*, após esse procedimento precisa ser dado um nome para esta função. Vamos apresentar o exemplo da *Function* a seguir, conforme autor Xavier (2010, p. 35):

```
function nomeDaFuncao ( parâmetros ) { código da função }
```

Vamos analisar a seguir outro exemplo do autor Xavier (2010), em relação à função sem retorno, o comando apresenta um *script* que pode ser utilizado diversas vezes durante a execução do código:

Por exemplo, a função irá mostrar na tela a mensagem *Bem-vindo ao JavaScript*.

```

1. <SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
2. function ola () {
3.     document.write ("Bem vindo ao JavaScript<br>");
4. }
5. </SCRIPT>

```

FONTE: Xavier (2010)

2.1.4 Criando Funções com Retorno

O principal cargo de funções com retorno é desenvolver um resultado para um determinado *script*, onde a solução da soma é gerada e utilizada para desenvolver este resultado para o *script* de um programa.

A seguir vamos analisar um exemplo do autor Xavier (2010), em relação à funções com retorno, usando a função *return*:

```
1. <SCRIPT LANGUAGE = "JavaScript" TYPE = "text/javascript" >
2. function somar () {
3.     return 5+5;
4. }
5.
6. document . write ( "A soma de 5 + 5 é " + somar () );
7. </SCRIPT>
```

2.2 UTILIZANDO FUNÇÕES DE PRIMEIRA CLASSE

A utilização de *JavaScript* nas funções de primeira classe são classificadas como objetos que são compostos por propriedades e métodos, podem ser utilizadas para atribuir funções em variáveis e até mesmo retornar as solicitações como um outro objeto.

2.2.1 Funções Internas

São definidas como funções que pertencem internamente a outras funções, possuem como finalidade serem criadas em cada situação em que uma função externa é invocada. Outra característica é que ela pode ser criada a partir de funções externas, em que as constantes, as variáveis e os valores dos argumentos são transformados em uma função interna.

2.2.2 Usando páginas Web em JavaScript

Uma das principais funções de *JavaScript* é escrever funções para que sejam incluídas em páginas *web*, desenvolvidas em linguagem HTML, onde podem interagir diretamente com Modelo de Objeto de Documento. O *JavaScript* torna a execução das ações muito mais rápidas, pois realizam a execução do código diretamente no navegador do usuário, outra vantagem é detectar de forma autônoma as ações e solicitações do usuário.

3 DEFINIÇÃO DE FUNÇÕES

As funções são definidas por desempenhar o papel de dividir as grandes tarefas em várias outras sub-tarefas, auxiliando de forma significativa os programas, tornando-os ainda mais rápidos em seus comando e execuções. Outra funcionalidade das funções é fazer com que os códigos não sejam repetidos, quando ocorre de um código se repetir o mesmo será transformado em uma nova função no programa, e está poderá ser chamada por diversas vezes. Mendes (2011, p. 108), afirma que:

A função agrupa instruções em uma unidade (ou entidades) e lhe atribui um nome. Cabe destacar que essa unidade pode ser chamada de outras partes do programa. Além disso, qualquer sequência de instruções que apareça em um programa mais de uma vez é candidata para se tornar uma função. Fazendo isso, você consegue também reduzir o tamanho do programa.

“A função deve ser utilizada sempre que seja necessário obter um único valor. Esse uso corresponde à noção matemática de função. Consequentemente, é muito estranho que uma função realize uma tarefa diferente de desenvolver um valor, e não deve fazê-lo”. (AGUILAR 2011, p. 268).

4 PILHA DE EXECUÇÃO

Pilhas são classificadas como uma estrutura de dados, muitas vezes utilizada para realizar a implementação de sua estrutura na linguagem de programação, é uma estrutura muito utilizada, pois permite que todo acesso que é realizado em um programa ou ação em código precisa passar pelo topo da estrutura de dados. A pilha segue uma lógica de programação, onde cada código ou um novo elemento é incluído em uma pilha, este passa a ser pertencente da estrutura de dados do topo, outra característica muito positiva para a programação é que nesta estrutura, apenas o topo pode ser excluído. Nesse sentido, seguindo os princípios e funcionalidades da pilha, quer dizer que os elementos da estrutura de dados são retirados em ordem inversa, indicando que o primeiro elemento a sair é o último que entrou, este procedimento é conhecido como *Last In, First Out*.

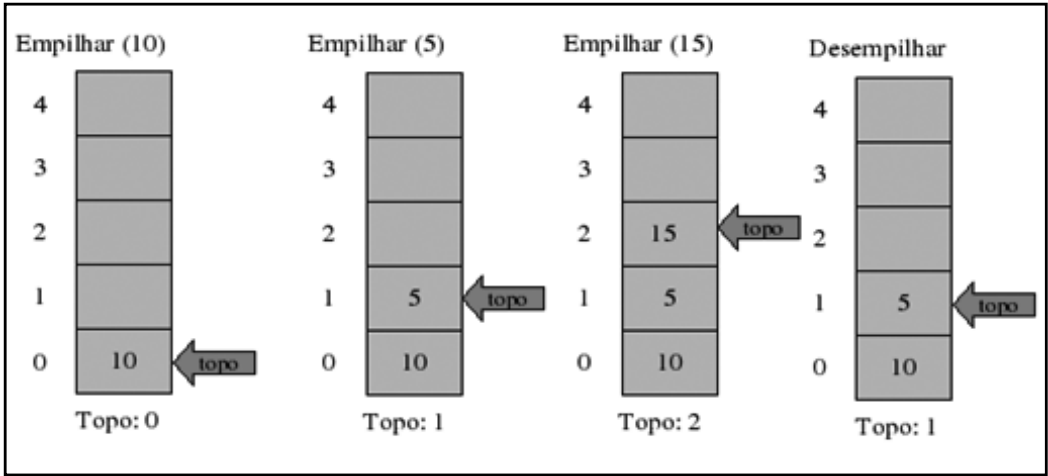
Edelweiss e Galante (2009, p. 126), colocam a definição de pilhas, “sendo listas nas quais o acesso somente pode ser feito em uma das extremidades, denominada topo da pilha. Todas as consultas, alterações, inclusões e remoções de nodos somente podem ser realizadas sobre um nodo, que é aquele que está na extremidade considerada o topo da pilha”.

A estrutura da pilha é composta por duas operações básicas, isso quer dizer que para realizar o empilhamento de uma estrutura de pilha precisa realizar uma operação para empilhar um novo elemento e a segunda operação básica é desempilhar um elemento. Na operação de empilhamento de um novo elemento, o processo ocorre quando são inseridos os elementos no topo do empilhamento.

Na operação de desempilhar um elemento, pega-se o elemento e remove-o do empilhamento.

Segundo os autores Edelweiss e Galante (2009, p. 127), “as pilhas são estruturas de dados fundamentais, sendo utilizadas em muitas aplicações em computação”. Como, por exemplo, os navegadores de internet armazenam os endereços mais recentemente visitados em uma estrutura de dados do tipo pilha. Cada vez que o navegador visita um novo *site*, o endereço é armazenado na pilha de endereços. Utilizando a operação de retorno (“back”), o navegador permite que o usuário retorne ao último *site* visitado, retirando seu endereço da operação pilha.

FIGURA 23 - EXEMPLO DE EMPILHAR E DESEMPILHAR



FONTE: Farias (2009)

Pode ser percebido na figura acima, como ocorre o processo de empilhamento, na primeira pilha é incluído o elemento 10, este fica como sendo o último a ser incluído, na segunda pilha, é incluído o elemento 5, este agora passa a ser o último elemento a ser incluído na pilha, na terceira pilha pode ser percebida a inclusão do elemento 15, este passa então a ser o último elemento a ser incluído na pilha, por isso, se perceber na última coluna, chamada de desempilhar, o número 15 foi excluído da pilha, por ser o último elemento a ser incluído no topo da pilha.

Horowitz e Sahni (1987, p. 77), apresentam a definição de pilha, como “Stack – Pilha é uma lista ordenada na qual todas as inserções e retiradas são feitas numa extremidade, chamada topo”. Como já mencionado, este processo é conhecido como elementos que são retirados do topo da pilha, isso quer dizer que sempre o último elemento a entrar na pilha será o primeiro elemento a ser retirado do topo da pilha.

A partir dessa premissa, em que mesmo sendo o último elemento inserido na pilha, será o primeiro a ser removido, este termo é conhecido como LIFO (*Last In, First Out*). A lista LIFO é considerada uma estrutura dinâmica, denominada

uma coleção que pode aumentar e diminuir durante sua existência. Esta denominação de LIFO vem justamente por serem listas que crescem e diminuem. Ainda, segundo Pereira (1996, p. 18):

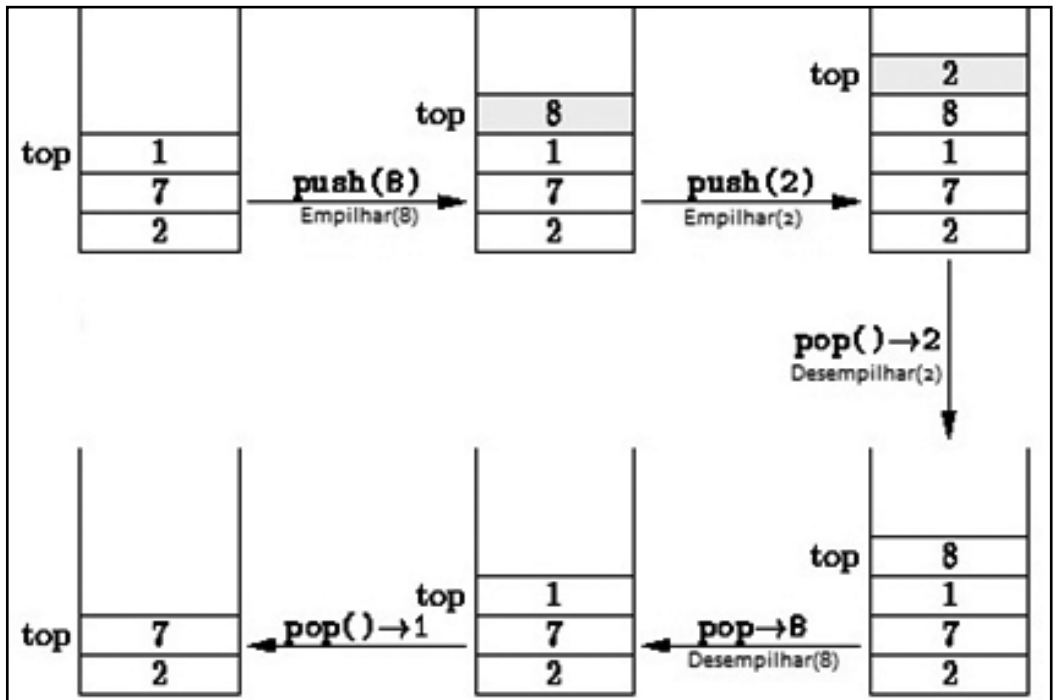
Uma pilha pode suportar até três operações básicas, essas operações básicas são conhecidas como:

TOP: Acessa o elemento que está posicionado no topo da pilha.

PUSH: Este possui como função inserir um novo elemento no topo da pilha.

POP: Este remove um elemento que está no topo da pilha.

FIGURA 24 - EXEMPLO DE PILHA



FONTE: Jprofessor (2014)

Analisando a figura acima podemos perceber que Top está indicando os elementos que estão no topo da pilha, por outro lado a função Push está inserindo os elementos na pilha, e a função Pop exclui os elementos da pilha, como podem ser vistos nas pilhas inferiores da figura, isso pode ser percebido nas três pilhas em que um elemento foi excluído.

4.1 CRIANDO UMA PILHA

Quando existe a necessidade de criar uma nova pilha muitos procedimentos devem ser levados em consideração e alguns cuidados precisam ser observados. Quando se cria uma nova pilha a mesma aparecerá vazia, deve ser informado que a pilha está vazia. Esses procedimentos precisam ser considerados para que a pilha tenha sua utilidade e seja utilizada pelos elementos que a completam. Edelweiss e Galante (2009, p. 129), apresentam o seguinte raciocínio em relação à criação da pilha.

A estratégia é indicar que a pilha está vazia, este procedimento é feito quando o índice do topo da pilha estiver indicando uma unidade a menos do que o índice de sua base. Outra estratégia é os algoritmos que manipulam as pilhas (inserção, remoção e consulta) utilizam esta estratégia para reconhecer que a pilha está vazia.

4.2 INSERÇÃO DE UM NODO NA PILHA

A inserção de um nodo na pilha só pode ocorrer se houver espaço na pilha, isso pode ser verificado pelo valor que limita o espaço da pilha. Esse procedimento de inserção de um nodo na pilha segue a lógica da pilha, onde esse nodo será inserido no topo, essa função é conhecida também pela função Push. Segundo Edelweiss e Galante (2009, p. 130), “caso a inserção seja realizada, o valor do topo da pilha é incrementado, sendo este agora o nodo do topo, consequentemente, o único ao qual se tem acesso”. Para verificar se existe espaço na pilha, e a inserção ocorra no nodo, precisa ser inserido o valor da posição que o nodo assumirá, se não existir mais espaço na pilha a função sucesso apresenta falso a inserção do nodo.

Vamos analisar a seguir um exemplo de um algoritmo sobre a inserção de um novo nodo em um arranjo de pilha.

FIGURA 25 - INSERÇÃO DE NODO NA PILHA

```

InserirPilhaArr
Entradas: Pilha ( TipoPilha)
           Lim (inteiro)
           Topo (inteiro)
           Valor (TipoNodo)
Saidas: Pilha (TipoPilha)
           Topo (inteiro)
           Sucesso (logico)

Inicio
  se Topo < Lim
    entao inicio
      Topo ← Topo + 1
      Pilha [Topo] ← Valor
      Sucesso ← verdadeiro
    fim
  senao Sucesso ← falso
fim

```

FONTE: A autora

Para realizar a remoção de um nodo da pilha é necessário que este esteja no topo da pilha, esta operação é reconhecida como Pop, essa remoção pode acontecer se a pilha apresentar pelo menos um nodo, esse procedimento faz com que o valor da pilha diminua, considerando que podem a partir desse ponto, incluir novos elementos na pilha, sem espaço não é possível incluir, por isso a função de remoção do nodo. Abaixo vamos analisar um exemplo de um algoritmo Remoção de um Nodo da Pilha:

FIGURA 26 - REMOÇÃO DE NODO NA PILHA

```

RemoverPilhaArr
Entradas: Pilha ( TipoPilha)
           Topo (inteiro)
           Base (inteiro)
Saidas: Pilha (TipoPilha)
           Topo (inteiro)
           Sucesso (logico)
           ValorRemovido (TipoNodo)

Inicio
  se Topo ≤ Base
    entao inicio
      ValorRemovido ← Pilha[Topo]
      Topo ← Topo - 1
      Sucesso ← verdadeiro
    fim
  senao Sucesso ← falso
fim

```

FONTE: A autora

4.3 ACESSO À PILHA

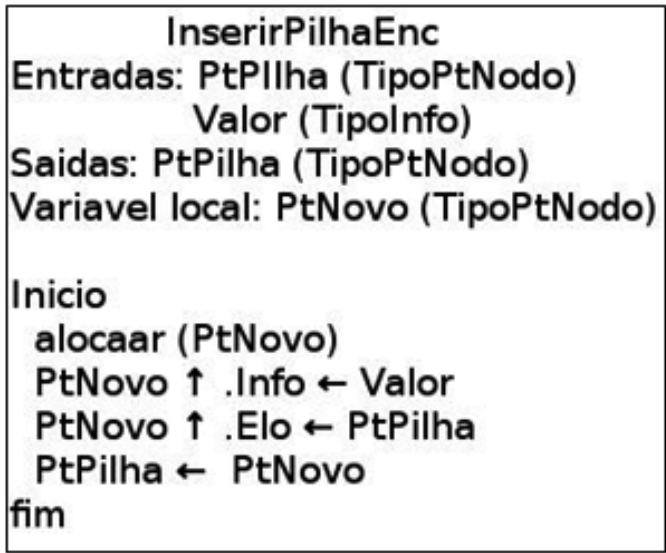
O acesso ao nodo de uma pilha pode ser feito apenas com o nodo que está no topo da pilha, se existir a necessidade de acessar outros nodos da pilha, precisa ser removido os nodos que estão antes do nodo a ser acessado, iniciando a remoção sempre pelo topo da pilha.

4.4 CRIAÇÃO DE PILHA ENCADEADA

O processo de encadeamento de uma pilha exige que os nodos estejam divididos em dois processos, como no campo denominado *Info*, nesse campo são armazenadas todas as informações inerentes ao nodo. O outro campo dividido é chamado *Elo*, neste campo são realizados os encadeamentos de todos os nodos da pilha. Para que uma pilha seja encadeada é realizada a atribuição de um valor nulo para a variável ponteiro, e esta variável irá armazenar este valor no topo da pilha, isso se a pilha estiver com espaço livre para guardar este encadeamento.

Já no processo de inserir um nodo em uma pilha encadeada, o processo ocorre seguindo a lógica da pilha, em que os elementos são incluídos no topo dela, nesse sentido o novo nodo deve ser encadeado com o elemento que estiver no topo da pilha, passando este nodo a ser o novo topo. Vamos analisar a seguir um exemplo de como executar a inserção de um nodo em uma pilha encadeada:

FIGURA 27 - INSERIR NODO EM UMA PILHA



FONTE: A autora

A remoção de um nodo em uma pilha encadeada ocorre em um processo que a variável ponteiro aponta para o nodo que está no topo da pilha, removendo-a e, assim, liberando espaço para a inclusão de novos elementos no topo. O processo de acesso a uma pilha encadeada se torna muito mais fácil e de forma lógica, pois como a variável ponteiro já possui armazenado o endereço do nodo no topo da pilha, o acesso é indicado pela variável ponteiro, essa indicação apresenta o único nodo da pilha que pode ser tanto acessado, quanto alterado.

5 PONTEIRO DE VARIÁVEIS

Pode-se dizer que para cada tipo de memória, existe um tipo de ponteiro, pois este possui como característica armazenar endereços de memória, isso em situações de existência de valores que sejam correspondentes com os valores dos ponteiros. Um exemplo de valores é a declaração variável como: *int a*, ao realizar o processo de declarar a variável, com o nome de *a*, estamos dizendo que esta variável pode conter números inteiros, essa declaração permite que automaticamente seja reservado um espaço na memória para valores inteiros. Com a declaração da variável já apontada e armazenado seu valor em um espaço na memória desenvolve-se um ponteiro, necessitando apenas direcionar este ponteiro para a variável criada. Segundo Martin (2014, p. 277), “os ponteiros permitem que sejam representadas estruturas de dados complexas, permite a alteração de valores passados como argumentos para funções e métodos”.

Os ponteiros possuem como objetivo otimizar a utilização das funcionalidades de um programa a ser desenvolvido, como também garantir a segurança dos dados e uma forma dinâmica de organização dos dados desse programa. Um exemplo do funcionamento do ponteiro é: qualquer dado inserido e executado por um programa será armazenado na memória, vamos citar o acesso pela primeira vez em um sistema de *e-mail*, após inserir o *login* e a senha e este processo for executado, esses dados serão armazenados automaticamente na memória.

Feofiloff (2009, p. 148), define ponteiros sendo este “um tipo especial de variável destinado a armazenar endereços. Todo ponteiro pode ter um valor Null, que é um endereço inválido”.

Sebesta (2010, p. 314),

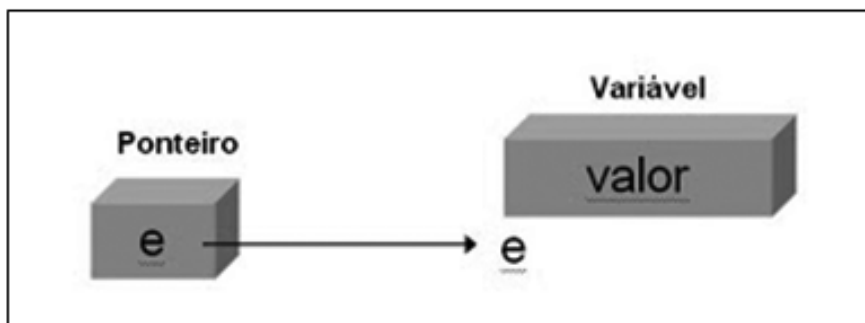
Apresenta as operações de ponteiros como linguagens que fornecem um tipo ponteiro, normalmente incluem duas operações de ponteiros fundamentais, como: Atribuição e Desreferenciamento. A atribuição realiza o processo de modificar o valor de uma variável de ponteiro para algum endereço útil. O desreferenciamento é um procedimento que leva uma referência por meio de um nível de indireção, é a segunda operação fundamental dos ponteiros.

Os ponteiros são definidos por armazenar valores em espaços de memória, estes valores são reconhecidos como *Int*, que possuem como finalidade armazenar os valores inteiros, já o *Float* possui como funcionalidade armazenar valores de ponto flutuante, como números como 1,5, com relação aos valores *Char* estes possuem como função armazenar caracteres, vamos analisar o exemplo:

```
int j;  
j = 22;
```

Perceba que o ponteiro declara a variável *j* como sendo um número inteiro, como no exemplo o número inteiro *j = 22*.

FIGURA 28 - PONTEIRO



FONTE: Schepp (2011)



CASAVELLA, Eduardo. Ponteiros em C. A utilização de ponteiros em linguagem C é uma das características que tornam a linguagem tão flexível e poderosa. Para descobrir mais sobre Ponteiros, acesse o Link: <<http://linguagemc.com.br/ponteiros-em-c/>>.

Percebe-se que na figura fica muito clara a disposição do funcionamento do Ponteiro, onde o ponteiro declara a variável, com isto o valor ocupa um espaço na memória e fica armazenada, este espaço é destinado para o valor desta variável.

6 RECURSIVIDADE

Recursividade se define por executar chamadas de uma mesma função, esse processo ocorre dentro do corpo de uma função existente e declarada, a recursividade chama novamente a própria função. Podemos exemplificar de forma mais simples, onde a função declarada de C chama a sua própria função C, isso é denominado de recursividade direta, no entanto podem ser chamadas também funções indiretas, como, por exemplo, a função C chama a função D, e esta função D, chama a função C, isso é caracterizado como recursividade indireta.

No entanto ao desenvolver uma função recursiva devem ser tomadas algumas medidas, mesmo ela sendo uma solução simples, executando tarefas repetitivas, isso ocorre sem usar estruturas de repetição, caracterizando-se em processo nada dinâmico. Todo processo de desenvolvimento ao utilizar a recursividade exige muita atenção na sua aplicação, para que não tenha problema na declaração das funções e não torne o programa demorado no momento de devolver as respostas das chamadas.

Os cuidados se devem às propostas das funções recursivas. Toda vez que uma função é chamada de forma recursiva, estas são armazenadas na memória do parâmetro de cada chamada, isso ocorre para que os valores não se percam.

Levando em consideração que cada instância de uma função pode acessar as instâncias criadas para ela mesma, não podendo esta instância acessar os parâmetros de outras instâncias e a chamada de uma função é conhecida como registro de ativação, devendo esta chamada obter os seguintes retornos: deve retornar um endereço, registrar o estado e as *Flags* do computador, transformar as variáveis como argumentos e o retorno da variável.

A seguir temos um exemplo de como ocorre uma função recursiva utilizando a linguagem de programação em C. A função recursiva precisa criar um programa que peça um número inteiro ao usuário e retorne a soma de todos os números de 1 até o número que o usuário introduziu, sendo os seguintes valores $1 + 2 + 3 + \dots + n$:

Vamos criar uma função *soma(int n)*.

Se $n=5$, essa função deve retornar: $soma(5) = 5 + 4 + 3 + 2 + 1$

Se $n=4$, essa função deve retornar: $soma(4) = 4 + 3 + 2 + 1$

Se $n=3$, essa função deve retornar: $soma(3) = 3 + 2 + 1$

Isso é feito de uma maneira muito simples, através de um simples teste condicional do tipo *IF ELSE*. Veja a seguir como ficou nosso código em C:

```

#include <stdio.h>

int soma(int n)
{
    if(n == 1)
        return 1;
    else
        return ( n + soma(n-1) );
}

int main()
{
    int n;
    printf("Digite um inteiro positivo: ");
    scanf("%d", &n);

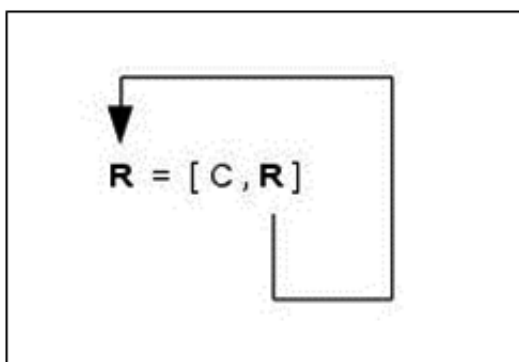
    printf("Soma: %d\n", soma(n));
}

```

FONTE: site Progressivo

Segundo Pereira (1996), em geral, uma rotina recursiva R pode ser expressa como uma composição formada por um conjunto de comando C (que não contém chamadas a R) e uma chamada (recursiva) à rotina R . Exemplo de uma recursão Direta:

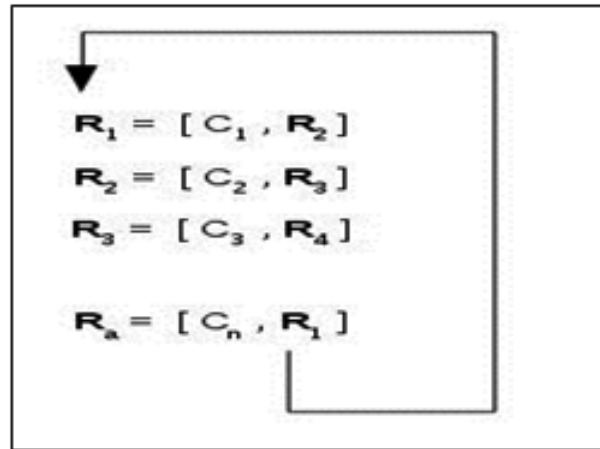
FIGURA 29 - FUNÇÃO RECURSIVA DIRETA



FONTE: A autora

Pereira (1996), apresenta ainda uma forma indireta de recursão, na qual rotinas são conectadas através de uma cadeia de chamadas sucessivas que acabam retornando a primeira que foi chamada.

FIGURA 30 - FUNÇÃO RECURSIVA INDIRETA



FONTE: A autora

Analizando os exemplos acima percebemos que a função recursiva direta chama ela mesma, com R chamando a função R, já na função recursiva indireta a função R chama a função C, e a função C realiza a chamada da função R.



Conheça mais acerca da Recursividade e suas funcionalidades dentro das funções acessando o link: <<http://geraldoferraz.blogspot.com.br/2011/11/recursividade.html>> e também o link: <<http://linguagemc.com.br/recursividade-em-c/>>

7 VARIÁVEIS ESTÁTICAS DENTRO DE FUNÇÕES

A variável estática é definida por ser um valor que está disponibilizado dentro de um escopo local, em uma função, toda vez que a mesma é inicializada pela primeira vez o fluxo de controle atinge o bloco ao qual corresponde seu valor, como não perde também seu valor, quando o nível de execução do programa deixa o escopo. Um procedimento que ocorre em uma variável estática é que esta não fica armazenada em uma pilha, no entanto fica armazenada em uma memória estática, este procedimento ocorre devido a um processo realizado pelo próprio funcionamento do programa. Segundo Junior, et al. (2012, p. 376), “as variáveis estáticas são variáveis locais cujos valores persistem dentro da função, e permanecem a cada vez que a função é chamada”.

Com isso a memória estática continua existindo após a função ser executada, pois seu valor fica armazenado, definindo que essa variável que

foi declarada para esta função, apenas pode ser visualizada dentro da mesma função à qual foi declarada. Outra função muito relevante que a variável estática desempenha é a recuperação do valor de uma variável atribuída, quando a função foi executada pela última vez, é uma vantagem que se deve novamente ao armazenamento da memória estática. Conforme Aguilar (2011, p. 268), “as variáveis estáticas mantêm sua informação, inclusive depois de terminada a função”. Quando a função é novamente chamada a variável adota o valor que tinha quando foi chamada anteriormente.

A variável estática precisa ser declarada dentro de uma função, esta função possui como responsabilidade realizar a impressão de números que são reais, a grande vantagem desta função declarada pela variável estática está em imprimir um número real por vez, cada número separado e colocando até cinco números em cada linha impressa. Essa impressão ocorre da seguinte forma, na primeira linha são impressos cinco números, na segunda linha são impressos mais cinco números e assim consequentemente.

Segundo Celes e Rangel (2002) os números são declarados e impressos em funções com variáveis estáticas, repare o exemplo:

```
Void imprime ( float a )  
{  
    static int n = 1;  
    printf( " %f ", a );  
    if (( n % 5 ) == 0 ) printf( " \n " );  
    n++;  
}
```

7.1 VARIÁVEL LOCAL

A variável local é definida especificamente para uma função que possui um armazenamento destinado para uma variável que está definida dentro de uma função, isso acontece devido as variáveis locais serem declaradas para apenas uma função. Esta variável guarda os valores de várias chamadas de uma função, esse procedimento é muito importante quando existe a necessidade de conter um valor na saída da função, sem necessitar que uma variável global seja criada. Lorenzi, Mattos e Carvalho (2007, p. 44), apresentam as variáveis locais como “declaradas na seção de declaração da sub-rotina, fazendo que somente essa sub-rotina possa utilizá-lo. O restante do programa não reconhece essas variáveis”.

Segundo Xavier (2010), as variáveis locais são aquelas nas quais apenas a função que a declarou pode usá-la, conforme o exemplo da declaração de um

valor *int* *x*, dentro de uma função *main*, apenas a função *main* pode utilizar esta variável, vejamos o exemplo:

```

01. #include <iostream>
02. #include <cstdlib>
03. using namespace std;
04.
05. int obterY (void) // criando a função
06. {
07.     int y; // apenas a função obterY pode utilizar
08.     y=10*5;
09.     return y;
10. }
11.
12. int main (void){
13.     int valorDeY, x; // apenas a função MAIN pode utilizar
14.     cout <<"digite o valor de X: ";
15.     cin >> x;
16.     cin.ignore ();
17.     valorDeY=obterY();
18.     cout <<"\nX="<<x<<"\tY="<<valorDeY<<"\n\n";
19.     system ("pause");
20. }

```

FONTE: Xavier (2010)

Xavier (2010), coloca que a variável *x* só pode ser utilizada por *main* e a variável *y* só pode ser utilizada por *obterY*. Para conseguirmos utilizar o valor guardado em *y* na função *obterY*, tivemos que retornar (por na saída da função) o valor de *y* (**return y**) e guardamos dentro de outra variável que estava em *main* - **valorDeY**.

7.2 VARIÁVEL GLOBAL

A variável global, por sua vez, é declarada fora das funções, externamente a uma função, por isso sua disponibilidade está associada globalmente, para que as funções estejam localizadas abaixo das variáveis declaradas. Um fator muito interessante de uma variável global é a característica de ficar escondida dentro do módulo ao qual foi declarada, e pode ser visível apenas para este módulo. Conforme Lorenzi, Mattos e Carvalho (2007, p. 43), colocam que “as variáveis globais são declaradas na seção de declaração do programa principal. Isso que essas variáveis possam ser empregadas em todo o programa, inclusive dentro das sub-rotinas, pois todo o programa é capaz de ‘enxergá-las”.

Conforme o autor Xavier (2010), a variável é declarada fora de qualquer função. Variáveis globais podem ser utilizadas por qualquer função. E qualquer função pode alterar o valor, utilizá-la em um processo ou até mesmo atribuir o valor que quiser, vamos analisar o exemplo a seguir:

```
01. #include <iostream>
02. #include <cstdlib>
03. using namespace std;
04.
05. int _iniciado=0; // variável global começa em 0
06.
07. void start (void) {
08.     _iniciado++;
09. }
10.
11. int main (void){
12.     start (); // chamando a função start
13.     int x=0, opcao; // variáveis locais, apenas a função MAIN pode utilizar
14.     x++;
15.     cout <<"Este programa foi iniciado "<<_iniciado<<" vez(es)";
16.     cout <<"\nO valor de X e "<<x;
17.     cout <<"\n\nDeseja reiniciar o programa?\n1.\t\tSIM\nOutro numero\
\tNAO\n";
18.     cin >> opcao;
19.     cin.ignore ();
20.     if (opcao==1)
21.         main (); // reiniciar o programa
22.     return 0;
23. }
```

FONTE: Xavier (2010)

Xavier (2010), apresenta a variável *_iniciado* que está sendo usada nas duas funções - *main* e *start*. Toda vez que a função *start* é chamada, é incrementada (somada) um a *_iniciado*.

RESUMO DO TÓPICO 2

Nesse tópico, você viu:

- Neste tópico de estudos nós falamos muito sobre as funções, sua conceituação, definição, características e suas funcionalidades, lembrando que uma característica muito interessante da função é que esta pode ser atribuída como um subprograma, ao qual a função recebe uma tarefa grande e se subdivide em sub-tarefas.
- Com a divisão do programa em sub-tarefas os códigos não se repetem, pois a função armazena as chamadas do programa na função e esta função pode ser chamada por diversas vezes.
- Uma definição muito importante sobre a função é que a mesma realiza o processo de agrupar conjuntos de comandos da linguagem, com isso as chamadas das funções retornam a resposta muito rápido, pois os parâmetros ficam armazenados, com isso podem ser chamados e consultados várias vezes. A característica mais relevante da função é que quando um código do programa se repete, este será transformado em uma nova função dentro do programa.
- A pilha é classificada como uma estrutura de dados, para que todo processo, como uma ação, chamada ou comando, possa especificamente passar pelo topo da estrutura da pilha. Outra característica da pilha de execução é: a programação segue uma lógica, em que cada código ou elemento é incluído em uma nova pilha, e esta por sua vez passa a fazer parte da estrutura de dados, lembrando que os elementos da pilha são retirados em ordem inversa, o último elemento, sempre é o primeiro a sair, essa lógica é conhecida como *Last In e First Out*.
- A principal funcionalidade dos ponteiros de variáveis é realizar o processo de armazenar endereços de memória, primeiro precisa-se declarar uma variável, para saber qual é o tamanho que esta variável vai ocupar na memória. Um exemplo desta funcionalidade é realizar a declaração da variável *int a*, estamos declarando que esta variável pode conter números inteiros.
- Recursividade, esta palavra já diz tudo, possui como finalidade executar chamadas de uma mesma função.
- A variável estática é definida por ser um valor que deve ser declarado dentro da função, este valor é armazenado na memória estática do programa e continua existindo, mesmo após a execução desta função, pois seu valor está armazenado na memória estática.



- 1 A pilha de execução possui a função de incluir cada elemento ao topo de uma pilha, são denominadas de estruturas de dados auxiliando a linguagem de programação, pois todo código precisa passar pelo topo da pilha. Com a inclusão dos elementos no topo da pilha, por qual método esses elementos podem ser removidos do topo?
 - () TOP.
 - () POP
 - () PUSH
 - () LAST.

- 2 Recursividade se define por executar chamadas de uma mesma função, esse processo ocorre dentro do corpo de uma função existente e declarada, a recursividade chama novamente a própria função. Toda vez que uma função é chamada de forma recursiva, estas são armazenadas na memória do parâmetro de cada chamada, isso ocorre para que os valores não se percam. Na recursividade pode-se executar a chamada da mesma função, através de suas formas:
 - () Contínua.
 - () Indireta.
 - () Direta.
 - () Instável.

VETORES E ALOCAÇÃO DINÂMICA

1 INTRODUÇÃO

Neste tópico de estudos vamos conhecer e entender melhor os conceitos sobre os vetores e alocação dinâmica, que também são elementos e funcionalidades da estrutura de dados e são partes fundamentais para estruturar uma linguagem de programação. Os vetores conhecidos como Arrays possuem como responsabilidade realizar a estruturação de um conjunto de dados, suas vantagens partem do princípio que podem armazenar mais de um valor em uma única variável. Os vetores (arrays) são considerados uma sequência de valores e são armazenados em uma sequência de dados na memória de um programa.

A alocação dinâmica por sua vez possui a característica por alocar um espaço na memória de um programa, isso ocorre para que se tenha um espaço determinado para uma variável ou função, para que o programador saiba de quanto espaço precisa e de quanto vai precisar para alocar espaços na memória. A alocação dinâmica é uma ferramenta muito utilizada para resolver problemas com estrutura de dados, a alocação é composta por quatro funções: *malloc()*, *Calloc()*, *realloc()* e a *free()*.

2 VETORES E ALOCAÇÃO DINÂMICA

2.1 VETORES (ARRAYS)

A principal funcionalidade de um vetor é realizar a estruturação de um conjunto de dados, para que isso ocorra precisa ser definição desde a estrutura, auxiliando o programador no desenvolvimento. Segundo Lorenzi, Mattos e Carvalho (2007, p. 51), “vetores permitem armazenar mais de um valor em uma mesma variável. O tamanho dessa variável é definido na sua declaração, e seu conteúdo é dividido em posições”.

Os vetores são considerados uma sequência de valores que possuem o mesmo tipo de dados, e são consequentemente armazenados, em sequência, na memória do programa que para acessar os valores armazenados faz uso utilizando o mesmo nome das variáveis para poder acessar esses valores já armazenados.

Os vetores podem ser compreendidos de outra forma, uma característica de ser entendido logicamente por ser uma lista de elementos, em que esses contenham os mesmos tipos de valor.

Os vetores (arranjos) são quase sempre implementados utilizando a memória consecutiva, no entanto nem sempre isso acontece. Intuitivamente um arranjo seria um conjunto de pares de índices e valores. Para cada índice definido existe um valor associado. Em termos matemáticos chamamos isso de correspondência ou mapeamento. Consequentemente que para os arranjos, estamos preocupados apenas com duas operações, aquelas que recuperam e a outra que armazena os valores. (HOROWITZ e SAHNI 1987, p. 47).

Como mencionado os vetores podem ser considerados como uma sequência lógica de elementos, podem estar disponíveis através de uma lista, para acessar esta lista existe a necessidade de seguir um índice, este índice por sua vez pode ser acessado individualmente, contendo dados constituídos por números inteiros. Para realizar a indexação desses elementos, é necessário que sejam do número 0, entre $n-1$, n é definido pela quantidade de elementos que compõem o vetor, o n pode ser considerado também como a dimensão ou o tamanho deste vetor. Após a declaração do vetor, seu tamanho torna-se fixo, não sendo possível alterar o seu tamanho posterior à execução do programa, pois seu tamanho está armazenado e gravado na memória.

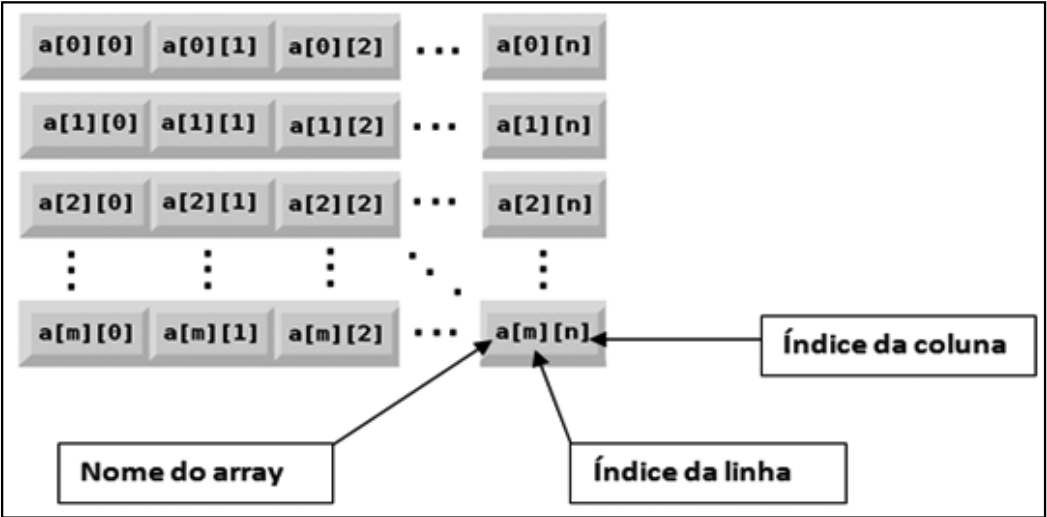
Segundo Aguilar (2011, p. 250), “um vetor (array) é uma sequência de objetos do mesmo tipo. Os objetos são chamados elementos do array e são numerados consecutivamente de 0, 1, 2, 3, esses valores são denominados **valores índices** ou **subíndice**, normalmente é utilizado para armazenar valores do tipo *char*, *int* ou *float*”. Perry (1999, p. 683), diz que para “monitorar e processar vários itens precisa colocar esses itens em um único array, com apenas um nome. A vantagem de um array é poder percorrer todas as variáveis com a instrução de *loop*, como um *loop for*”.

Conforme Aguilar (2011, p. 250), “um array pode conter a idade dos alunos de uma sala, as temperaturas de cada dia de um mês em uma determinada cidade ou o número de pessoas que vivem em cada estado do Brasil”. Será apresentado a seguir um exemplo de um array, onde este possui 6 (seis) elementos: $a[0] = 25,1$; $a[1] = 34,2$; $a[2] = 5,25$; $a[3] = 7,45$; $a[4] = 6,09$; e $a[5] = 7,54$.

Elementos (conteúdo)	25,1	34,2	5,25	7,45;	6,09	7,54
Índice (posição)	0	1	2	3	4	5

Pode-se perceber que a tabela acima está representando uma área da memória de um computador, pois o array sempre é armazenado com seus elementos em uma sequência de posições, como uma sequência contínua. Cada item no array é chamado de elemento, seguindo essa linha, cada elemento é acessado de forma independente, conforme a posição do índice, no exemplo acima fica muito claro, as posições são especificadas através dos números.

FIGURA 31 - EXEMPLO DE ARRAY



Fonte: Palmeira (2014)

2.2 MÉTODOS UTILIZADOS EM ARRAYS

Todos os array pertencem a uma classe denominada de `System.Array`, onde são demonstrados vários métodos para membros estáticos e instanciados.

TABELA 4: SYSTEM.ARRAY

Rank	Retorna o número de dimensões do array.
GetLength	Retorna o número de elementos da dimensão específica do array.
Length	Retorna o total de elementos do array.
GetLowerBound	Retorna limite baixo da dimensão especificada.
GetupperBound	Retorna o limite alto da dimensão especificada.
IsReadOnly	Indica se o array é apenas para leitura.
IsSynchronized	Indica se o acesso ao array é thread-safe.
SyncRoot	Retorna um objeto que pode ser usado para sincronizar o acesso para o array.
IsFixedSize	Indica se o array tem um tamanho fixo.
GetValue	Retorna a referência para o elemento localizado em uma posição específica no array.
GetEnumerator	Retorna o IEnumerator para o array, permitindo o uso da instrução foreach.
Sort	Ordena os elementos de um array.
IndexOf	Retorna o índice da primeira ocorrência do valor em um array unidimensional ou em uma parte dele.
LastIndexOf	Retorna o índice da última ocorrência do valor do array unidimensional ou em uma parte dele.
Reverse	Reverte a ordem dos elementos em um array unidimensional ou em uma parte dele.
Clone	Cria um novo array que é uma cópia superficial do array de origem.
CopyTo	Copia os elementos de um array para outro.
Copy	Copia a seção de um array para outro, executando qualquer modelagem requerida.
Clear	Configura a faixa de elementos no array para 0 ou null.

FONTE: Medeiros (2014)



Conheça mais sobre o conceito de Vetores (ARRAYS) acessando o Link: http://www.macoratti.net/10/05/c_arrays.htm

Estudo de: MACORATTI, José C. Conceitos - Apresentação Arrays.

Acadêmico(a) fique tranquilo, na próxima unidade de estudos, vamos explicar mais as funcionalidades e tipos de Arrays, aprofundar ainda mais nossos estudos acerca de elementos e seus valores dentro da programação.

3 ALOCAÇÃO DINÂMICA

Antes de explicar sobre o que se trata a alocação dinâmica, vamos falar um pouco sobre o uso da memória em um programa. Para que sejam armazenados dados e informações em um espaço de memória, precisamos proceder de três formas, utilizar as variáveis globais, variáveis locais e solicitar um espaço para o próprio sistema. Conforme Pereira (1996, p. 146), “sabemos que para executar um programa, o computador precisa carregar seu código executável, para a memória. Neste momento, uma parte da memória total disponível no sistema é reservada para o uso do programa e o restante fica livre”.

FIGURA 32 - ALOCAÇÃO DA MEMÓRIA



FONTE: Rocha (2010)

Conforme já visto anteriormente as variáveis globais são classificadas por continuar existindo enquanto o programa está sendo executado. Na variável local, o espaço é armazenado somente para funções declaradas, onde a variável está sendo executada. Já para armazenar espaço na memória ocorre um processo que exige que seja solicitado para o sistema, no entanto esse processo deve ocorrer enquanto o sistema está sendo executado, deve ser determinado também o espaço ao qual precisa ser armazenado, este espaço continua armazenado e registrado, enquanto não seja solicitado de forma explícita para o sistema, até que este espaço seja liberado.

Agora que sabemos sobre as definições das variáveis que realizam o armazenamento de memória, seja em um programa ou em um sistema operacional, vamos conhecer sobre a alocação dinâmica. A alocação dinâmica é definida por um processo de alocar um espaço na memória, este procedimento é utilizado, quando não se sabe exatamente o espaço correto que um elemento ou função vai ocupar, isso ocorre somente quando o programa ou sistema está em execução. Ainda segundo Pereira (1996, p. 146), “quem determina quanto de memória será usado para as instruções é o compilador. Alocar área para armazenamento de dados, entretanto, é responsabilidade do programador”.

Como a alocação de memória é responsabilidade do programador este consegue desenvolver variáveis durante o tempo de execução do programa, alocando desta forma memória para as novas variáveis que foram criadas. Com a criação das variáveis temos maior disponibilidade e melhor utilização da memória do programa, sem precisar utilizar a memória adicional.

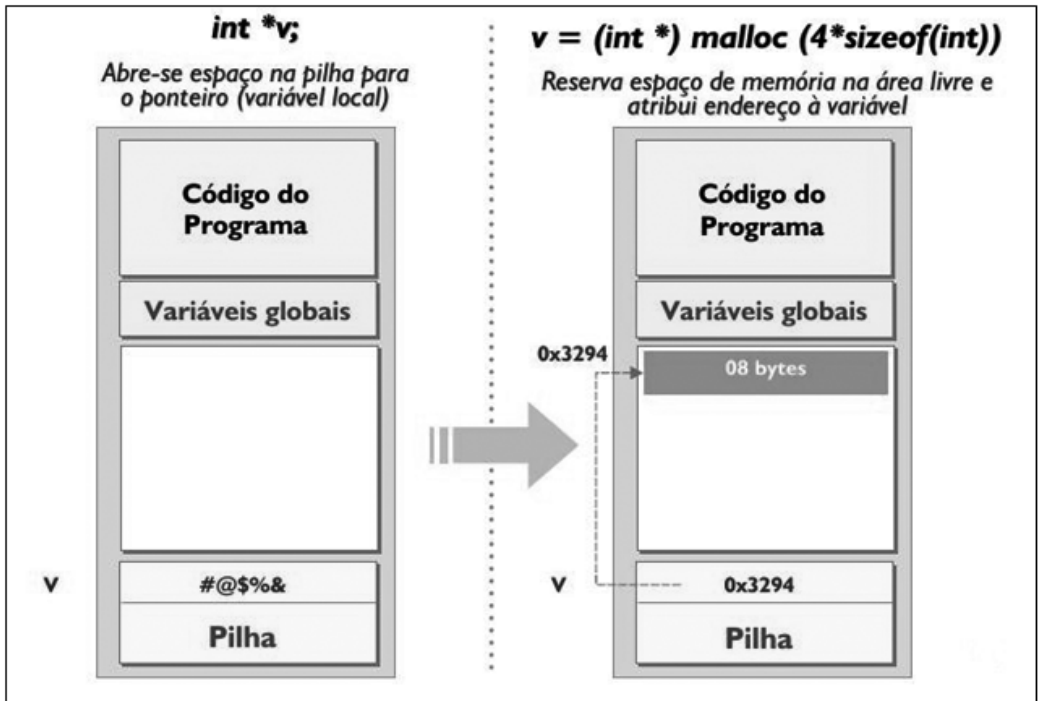
A alocação dinâmica é consideravelmente muito utilizada para resolver problemas com estruturas de dados, são denominadas quatro funções, estas as mais utilizadas para a alocação de memória, pertencem a biblioteca *Stdlib.h*: *malloc()*, *Calloc()*, *realloc()* e a *free()*. A funcionalidade de cada uma é muito importante para que o processo de alocar uma memória tenha êxito, as funções da *malloc()* e *Calloc()*, são iguais, elas realizam o processo de alocar espaço na memória, a *realloc()*, como sua denominação já menciona, tem por função realizar o espaço da memória e a função *free()* possui a responsabilidade de liberar as memórias que estão alocadas no programa.

A principal função da alocação dinâmica é a função *malloc()*, que está localizada dentro da biblioteca *stdlib.h*, a função *malloc()* recebe como parâmetros o número de bytes, que necessita alocar de memória, retornando com um endereço inicial em que foi alocada a memória. Quando não houver espaço suficiente para a solicitação de alocação a função *malloc()* retorna apontando um valor nulo (*null*), para verificar se existe espaço suficiente para a alocação é necessário o seguinte comando:

```
void *malloc(size_t tamanho);
```

Este exemplo acima é de um comando para obter certeza que existe espaço suficiente para alocar memória no programa. Vamos analisar na figura abaixo como ficaria disponibilizada e reservada a memória, utilizando a função *malloc()*.

FIGURA 33 - FUNÇÃO MALLOC()



FONTE: Rocha (2010)

É de extrema importância que seja verificado e testado se a alocação ocorreu de forma correta, esse processo deve ser realizado antes mesmo de utilizar o apontador. Vamos analisar como devem ser os comandos para essa verificação, segue exemplo da figura logo abaixo.

FIGURA 34 - TESTE DA FUNÇÃO MALLOC()

```
// Considere o código abaixo na alocação de um vetor de inteiros
// de tamanho 4
#include <stdlib.h>
int main(void) {
    int *v;
    v = (int*) malloc ( 4 * sizeof( int ) );
    if (v==NULL) {
        puts( "Memória insuficiente.\n" );
        return(1); // aborta o programa e retorna 1 para o SO
    }
    v[0] = 23; ...
}
```

FONTE: Rocha (2010)

Para realizar o procedimento de liberar espaço de memória alocada, precisa ser executado um comando utilizando a função *free()*, esta função recebe como parâmetro um ponteiro com um endereço da memória, a qual precisa ser liberada da memória do programa. Vamos analisar a seguir o comando que precisa ser executado para liberar o espaço:

```
void free( void *pt );
```

Para que o procedimento de liberação de espaço de uma memória alocada seja efetivo precisamos executar o comando que utiliza a função *free()*, deve ser observado que após a liberação do espaço na memória esta não poderá ser mais acessada. Vamos analisar a seguir o exemplo da função *free()* na execução de liberar espaço na memória alocada dinamicamente.

FIGURA 35 - FUNÇÃO *FREE()*

```
// Considere o código abaixo na alocação de um vetor de inteiros
// de tamanho 4
#include <stdlib.h>
int main(void) {
    int *v;
    v = (int*) malloc ( 4 * sizeof( int ) );
    if (v==NULL) {
        puts( "Memória insuficiente.\n" );
        return(1); // aborta o programa e retorna 1 para o SO
    }
    v[0] = 23; ...
    free( v );
}
```

FONTE: Rocha (2010)

A alocação dinâmica, nada mais é do que criar um espaço na memória de um programa ou sistema operacional, no entanto, esse procedimento de criar uma alocação, de armazenar um espaço da memória deve ser realizado enquanto o programa está sendo executado. Outra característica da alocação dinâmica é que para realizar o armazenamento de vários elementos só é possível alocar espaço para um elemento por vez, não é possível alocar espaço para vários ao mesmo tempo. Tanto para liberar espaço, como para desalocar espaço de um elemento a

função deve estar sendo executada pelo programa. Com relação aos elementos, para a liberação de espaço da memória alocada, precisa ser feito de um elemento de cada vez, como no processo de alocação de espaço da memória. A alocação de memória pode ocorrer em duas categorias, sendo Estática ou Dinâmica, como veremos a seguir na definição de cada uma dessas alocações de memória.

3.1 ALOCAÇÃO ESTÁTICA

Considerando que a definição de uma memória estática se dá através do processo em que uma variável pode ser uma área, um espaço de memória de um programa em que ainda não foi alocado nenhum espaço para a memória. Pereira (1996, p. 146), diz que “as variáveis de um programa têm alocação estática se a quantidade total da memória utilizada pelos dados é previamente conhecida e definida de modo mutável, no próprio código-fonte do programa”. Durante toda a execução, a quantidade de memória utilizada pelo programa não varia.

A alocação estática ocorre em tempo de compilação, ou seja, no momento que se define uma variável ou estrutura é necessário que se definam seu tipo e seu tamanho. Por quê? Porque esse tipo de alocação, ao se colocar o programa em execução, a memória necessária para se utilizar as variáveis e estruturas estáticas precisa ser reservada e deve ficar disponível até o término do programa. (LORENZI, MATTOS e CARVALHO 2007, p. 51).

3.2 ALOCAÇÃO DINÂMICA

Podemos considerar que a definição de uma alocação dinâmica ocorre quando a alocação de memória estava prevista no desenvolvimento do código do programa, em que o espaço já estava alocado, reservado para as variáveis, esse processo de predefinir um espaço para alocamento da memória é chamado de alocação dinâmica. Lorenzi, Mattos e Carvalho (2007, p. 51), definem a alocação dinâmica como “ocorre em tempo de execução, ou seja, as variáveis e estruturas são declaradas sem a necessidade de se definir seu tamanho, pois nenhuma memória será reservada ao colocar o programa em execução”.

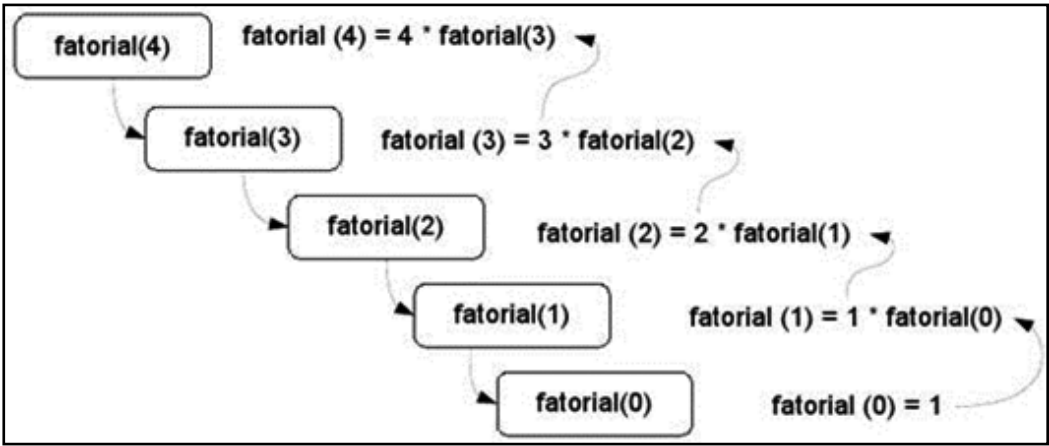
Pereira (1996, p. 146), diz “se o programa é capaz de criar novas variáveis enquanto executa, isto é, se as áreas de memória que não foram declaradas no programa passam a existir durante a sua execução, então dizemos que a alocação é dinâmica”.

LEITURA COMPLEMENTAR

RECURSIVIDADE EM JAVA

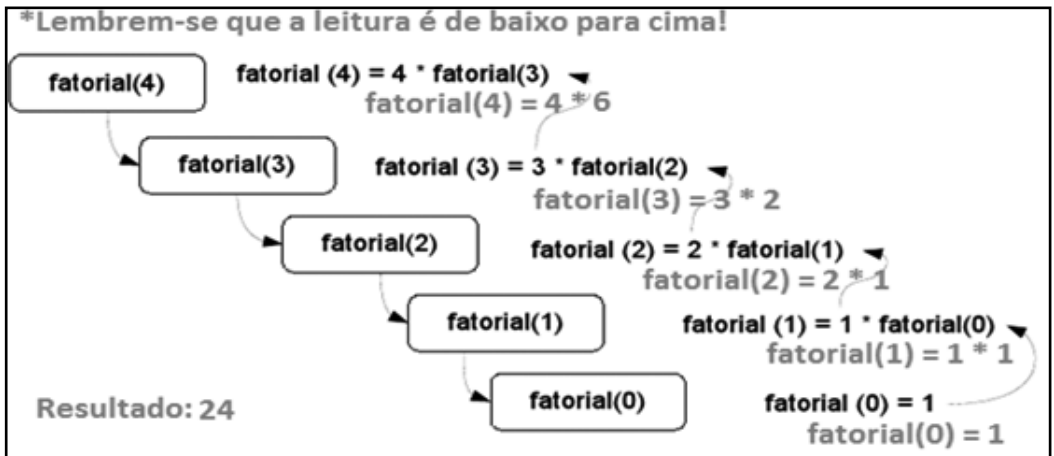
A recursividade é nada mais nada menos do que uma função dentro da outra e ela deve ser pensada como uma pilha (estrutura de dados onde o último a entrar deve ser o primeiro a sair). A estrutura dela consiste em descer até a base fazendo os cálculos ou rotinas de cada instrução, e então da base até o topo da pilha são empilhados os resultados de cada instrução e no final o topo contém o resultado que é retornado. Na figura a seguir, temos um exemplo que é frequentemente usado para explicar a recursividade, podemos encontrar em diversos livros didáticos, porque é um dos mais fáceis para se entender, estou falando do fatorial.

Ex.: desenvolver um método recursivo para descobrir o fatorial de um número “N”. Suponhamos que o N seja igual a 4.



Obs.: O “fatorial(4)” só pode ser descoberto depois que o “fatorial(3)” for descoberto, que por sua vez só poderá ser descoberto depois do fatorial(2) e assim por diante. Por isso vai do topo até a base, e depois vai empilhando os resultados da base até o topo.

Agora uma ilustração com os valores para fixar o conceito:



O conceito de recursividade, como podemos ver anteriormente, não é tão difícil, é claro que na hora da programação é sempre recomendável atenção, organização e fazer o tão “chato”, mas eficiente teste de mesa, ou seja, fazer o máximo para dar certo, pois a hora de errar é quando estiver treinando, fazendo exercícios e não deixar para errar quando estiver em um mercado de trabalho, onde um projeto pode ter mais de 10000 linhas. Agora vamos deixar de “Blábláblá”, e vamos ao que interessa, a seguir implementaremos o exemplo do fatorial no JAVA. Crie uma classe como a seguir:

```

1 public class Fatorial{
2     //Método recursivo para cálculo de fatorial
3     public int fatorialRecursivo(int num){
4         //se n é igual a 0, retorna 1
5         if (num == 0)
6             return 1;
7         //Caso contrário, o método recursivo é chamado:
8         return num*fatorialRecursivo(num-1);
9     }
10 }

```

Essa é a função recursiva do fatorial em JAVA. Agora vamos implementar um método principal para testar esta função, para facilitar coloque esse método dentro da classe que já foi criada. Como a seguir:

```
1 //Biblioteca java para caixas de texto
2 import javax.swing.JOptionPane;
3 //Classe ja criada anteriormente
4 public class Fatorial{
5     //Criando método principal
6     public static void main(String[] args) {
7         //Variável n recebendo o valor a ser fatorado
8         int n = Integer.parseInt(JOptionPane.showInputDialog("Digite um valor"));
9         //Objeto sendo instanciado
10        Fatorial b = new Fatorial();
11        //Chamando o método recursivo para fatorar a variável n, retornando
12        //o resultado na propria variavel n
13        n = b.fatorialRecursivo(n);
14        //Apresentando o resultado
15        JOptionPane.showMessageDialog(null,"O fatorial é: " + n);
16    }
```

Com isso eu encerro este artigo, espero que tenham gostado e entendido a tão temida recursiva. Qualquer dúvida podem entrar em contato.

Conclusão

Ao final deste artigo podemos ver que a **recursividade** não é aquele “bicho papão” que muitos pensam, é tudo questão de atenção e organização. Se você tiver paciência, atenção e organização vai se dar muito bem com a **recursividade**, não esquecendo é claro de fazer aquele bom e velho teste de mesa (pelo menos no começo), e é importante lembrar, também, que a **recursividade** trabalha com o conceito de pilha e pode até substituir laços de repetição.

FONTE: ALVES, Ricardo. Recursividade em Java. Disponível em: <<http://www.linhadecodigo.com.br/artigo/3316/recursividade-em-java.aspx#ixzz3EQu8W9rl>>. Acesso em: 08 Set. 2014.

RESUMO DO TÓPICO 3

Nesse tópico, você viu:

- Vetores (arrays) são classificados como um processo para estruturação de um conjunto de dados, possuem como característica realizar o armazenamento de mais de um valor em uma mesma variável. Para a definição do tamanho dessa variável, esse procedimento é realizado na sua própria declaração.
- Para realizar o armazenamento de dados e informações em um espaço de memória, seja de um programa ou sistema operacional, esta execução pode ser feita através de variáveis globais, variáveis locais e requisitando ao computador.
- A alocação dinâmica é definida especificamente por exercer a função de alocar um espaço da memória, isso para alocar um elemento ou uma função, este procedimento ocorre somente com a execução do programa.
- A alocação dinâmica possui a responsabilidade de resolver problemas com estruturas de dados, possui quatro funções para alocação de memória, *malloc()*, *Calloc()*, *realloc()* e *free()*, estas são pertencentes a biblioteca *Stdlib.h*. As funções da *malloc()* e *Calloc()*, realizam o processo de alocar espaço, a função *realloc()*, realocar espaço da memória e a função *free()*, libera espaço na memórias.
- A alocação estática se define por ser um espaço que ainda não foi alocado e não sabe-se o tamanho ao qual pode ocupar na memória. Já a alocação dinâmica ocorre ao contrário, quando se sabe o tamanho e o espaço da alocação que a mesma vai ocupar na memória do programa.

AUTOATIVIDADE



1 Os vetores (array) são definidos por serem uma sequência de objetos do mesmo tipo, estes objetos são denominados de elementos do array e são consequentemente numerados em sequência, uma vantagem muito interessante de um array é que estes podem percorrer todas as variáveis, utilizando a instrução *loop*. Os valores sequenciais dos arrays são denominados de:

- a) () Valores Índices.
- b) () Valores Programa.
- c) () Valores Subíndice.
- d) () Valores Subprograma.

2 Na alocação dinâmica são utilizados vários procedimentos para alocar espaço na memória de um programa, é um processo muito utilizado no desenvolvimento de sistemas, pois possuem como finalidade auxiliar no caso de problemas de estruturação de dados. Para realizar o procedimento de alocar espaço na memória, são utilizadas quais funções?

- a) () REALLOC()
- b) () FREE()
- c) () MALLOC()
- d) () CALLOC()

ESTRUTURAS DE DADOS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade você será capaz de:

- entender a cadeia de caracteres e seus dados primitivos;
- distinguir os tipos de caracteres;
- compreender vetores unidimensionais;
- conhecer os tipos de estruturas de dados;
- distinguir a diferença entre arranjos unidimensionais e multidimensionais;
- entender operadores e vetores;
- compreender os tipos e matrizes;
- entender os tipos abstratos de dados.

PLANO DE ESTUDOS

A Unidade 2 do Caderno de Estudos está dividida em três tópicos. No final de cada um deles, você encontrará autoatividades que servem de auxílio para fixar os conteúdos apresentados em toda a unidade e seus tópicos de estudos.

TÓPICO 1 – CADEIA DE CARACTERES

TÓPICO 2 – TIPOS DE ESTRUTURAS DE DADOS

TÓPICO 3 – MATRIZES

CADEIA DE CARACTERES

1 INTRODUÇÃO

Uma cadeia de caracteres é formada por uma sequência finita de caracteres. São formados por conjuntos de um ou vários caracteres. Usualmente são utilizados com os caracteres entre aspas (" ") e também podem aparecer entre apóstrofes (' '). Uma cadeia de caracteres é considerada como uma sequência finita de símbolos, estes símbolos estão presentes em um alfabeto específico, em que cada símbolo pertence a um tipo de alfabeto. Estes símbolos estão presentes no início da cadeia, como também no final da cadeia de caracteres.

Conforme os autores JUNIOR; et al. (2012, p. 389), “uma das formas mais básicas de composição das informações é a cadeia de caracteres, também conhecida como string. Conhecer esta estrutura permite a manipulação das informações que levam ao conhecimento”.

A cadeia de caracteres é formada por uma variedade de conceituações, sua definição pode ser descrita com o fator String que apresenta sua utilização na representação de valores textuais, em que uma string pode ser escrita seguindo uma sequência de caracteres. Entre os tipos de caracteres podemos encontrar as Constantes do Tipo Caracteres; Variáveis do tipo Caracteres; Cadeia de Caracteres e Operações; Vetor de Cadeia de Caracteres; Vetores Unidimensionais; Leitura de Vetores e Escrita de Vetores. Todos esses conceitos podem ser conferidos no conteúdo desta unidade de estudos.

2 CADEIA DE CARACTERES

2.1 CARACTERES

Os caracteres são definidos por serem dados primitivos, encontrados na maioria dos computadores. Esses dados primitivos estão relacionados com a linguagem máquina, onde o computador permite a manipulação desses dados primitivos. Uma cadeia é formada por uma sequência de caracteres que são ordenados para que ocorra a manipulação dos dados pela cadeia, caracteres são formados por um conjunto finito. Podemos citar como exemplo de um carácter uma representação lógica da memória, sendo esta uma sequência de *bits*, em que os *bits* são uma sequência de zeros(0) e uns(1), para cada carácter é atribuída uma

sequência distinta de *bits*, isso pode ser definido também como uma codificação de um conjunto de caracteres, em uma sequência de *bits* com tamanhos fixos.

Uma cadeia (*string*) de caracteres é um conjunto de caracteres, incluindo o branco, que é armazenado em uma área contínua da memória. Podem ser entradas ou saídas para um terminal. O comprimento de uma cadeia é o número de caracteres que ela contém. A cadeia que não contém nenhum carácter é denominada cadeia vazia ou nula, e seu comprimento é zero; não devemos confundir com uma cadeia composta somente de brancos, espaços em branco, já que esta terá como comprimento o número de brancos contidos. A representação das cadeias costuma ser com apóstrofes ou aspas. (AGUILAR 2011, p. 268).

Nos dias atuais, as cadeias de caracteres (*strings*) armazenam não somente dados, mas principalmente informações e, portanto, a maioria dos processamentos computacionais exige um tratamento mais complexo e apurado de *strings*.

Pode-se afirmar que a definição de cadeia de caracteres (*strings*) é uma sequência de letras, símbolos e também de números, em que sempre o último carácter é o carácter nulo '\0', no entanto essa *string* possui internamente uma coleção sequencial de dados que são armazenados somente como leitura de um *Char*. (JUNIOR; et al., 2011, p. 392).

Conforme a definição de Ramos; Neto; Vega (2009), caracteres podem ser utilizados para representar elementos de pontuação, dígitos, sinais, espaços e letras. Um valor da classe *string* é constituído por uma sequência de caracteres delimitada por um par de apóstrofes (') ou aspas (").

Goodrich e Tamassia (2002, p. 417), afirmam que “cadeia de caracteres podem surgir de uma variedade de origens, incluindo aplicações científicas, linguísticas e da internet”. Vamos analisar a seguir estes exemplos de cadeia de caracteres:

$P = \text{"GGTAAACTGCTTTAATCAAACGC"}$

$R = \text{"U.S. Men win soccer world cup!"}$

$S = \text{"http://www.wiley.com/college/goodrich/"}$

A primeira cadeia de caracteres, P , tem origem em aplicações de pesquisa sobre o DNA. A cadeia de caracteres R é uma manchete fictícia. A cadeia de caracteres S é um endereço (URL) do *site* da *web* que acompanha o livro do autor.

A string também é denominada “cadeia de caracteres”, pois é uma lista linear ou vetor, onde cada elemento desse vetor é um carácter, e o agrupamento deles irá representar uma informação. Vamos analisar o exemplo da seguinte frase: “Vendas crescem 28% ao mês!” (JUNIOR; et al. 2011, p. 393):

V e n d a s c r e s c e m 2 8 % a o m ê s !

No exemplo anterior pode ser analisada a estrutura de dados, esta estrutura comporta 26 caracteres, incluindo números e espaços em branco, todos esses caracteres da estrutura de dados formam uma informação.

O conceito de string pode ser construído através do uso de várias linguagens de programação, como, por exemplo a construção de algoritmos simples, vamos analisar a seguir esses modelos, vamos iniciar pela linguagem C. Segundo JUNIOR; et al. (2011, p. 402),

na linguagem de programação C não existe um tipo de dado string, pois nessa linguagem uma *string* é considerada um arranjo (ou vetor) de caracteres (tipo *char*), sendo que sua manipulação ocorre através do uso de diversas funções de manipulação de strings. Esse arranjo armazena uma sequência de caracteres do tipo *char*, cujo último elemento é o carácter: *NULL*, tipicamente, representado na forma do carácter '\0', ou simplesmente pelo seu valor 0.

Analise a seguir algumas sub-rotinas predefinidas em C, para tratamento de strings:

FIGURA 36 – SUB-ROTINAS PREDEFINIDAS EM C, PARA TRATAMENTO DE STRINGS

Sub-rotina - C
Funcionalidade
<code>int strlen(cadeia)</code>
Função que retorna o número de caracteres da armazenado na cadeia, não considerando o caractere NULL (/0).
<code>int strcmp(cadeia1,cadeia2)</code>
Função que retorna um valor 0 (zero) se as duas cadeias são iguais.
<code>charstrup(cadeia);</code>
Função que retorna cada caractere da cadeia convertido para maiúsculos.
<code>char strlwr(cadeia);</code>
Função que retorna cada caractere da cadeia convertido para minúsculos.
<code>char strcat(cadeia1,cadeia2);</code>
Função que retorna uma cadeia resultante da união entre duas cadeias passadas como parâmetro na função.
<code>char strcpy(cadeia1, cadeia2);</code>
Função que copia o conteúdo da cadeia2 para dentro da cadeia1. A cadeia 2 pode ser uma constante.
<code>char strncpy(cadeia1,cadeia2,int qtdcarac);</code>
Função que armazena na cadeia1 os primeiros caracteres da cadeia2, cuja quantidade está indicada em qtdcarac. Esse segundo parâmetro (qtdcarac) pode ser uma constante numérica inteira sem sinal. Atenção! O caractere NULL não é armazenado, devendo isto ser feito pelo programa.

FONTE: Juni; et al. (2011)

Agora que sabemos mais sobre as sub-rotinas que realizam o tratamento de *strings* na utilização da linguagem de programação C, sabemos como podem funcionar seus comandos e funções utilizando os caracteres. Vamos analisar a seguir algumas características dos dados que são caracteres.

Dados na forma de caracteres são armazenados nos computadores como codificações numéricas, onde a codificação mal-usada era o ASCII (Padrão de Codificação para Intercâmbio de Informação – *American Standard Code for Information Interchange*) dados de 8 bits, usam valores de 0 a 127 para codificar 128 caracteres diferentes. O padrão UCS-2, é um conjunto de caracteres de 16 bits. Essa codificação é geralmente chamada de Unicode. (SEBESTA 2010, p. 273).



Java foi a primeira linguagem de programação que utilizou amplamente conjunto de caracteres Unicode UCS-2, adotado posteriormente também pelas linguagens JavaScript, Python, Perl e em C#.

A utilização do padrão ASCII em vários tipos de equipamentos que utilizam códigos diferentes. Para utilizar códigos diferentes e para que máquinas diferentes consigam compilar esses códigos, podem ser utilizados alguns códigos do padrão ASCII. Vamos analisar esses códigos deste padrão que são associados aos caracteres:

TABELA 5: CARACTERES UTILIZANDO PADRÃO ASCII

	0	1	2	3	4	5	6	7	8	9
30			sp	!	“	#	\$	%	&	'
40	()	*	+	,	-	/	.	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_		a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

FONTE: Celes e Rangel (2008)

TABELA 6: CARACTERES DE CONTROLE PADRÃO ASCII

0	nul	<i>Null: nulo</i>
7	bel	<i>Bell: campainha</i>
8	bs	<i>Basckpace: voltar e apagar um caractere</i>
9	ht	<i>Tab ou tabulação horizontal</i>
10	nl	<i>Newline ou line feed: mudança de linha</i>
13	cr	<i>Carriage return: volta ao início da linha</i>
127	del	<i>Delete: apagar um caractere</i>

FONTE: Celes e Rangel (2008)

2.2 CADEIA DE CARACTERES (STRINGS)

Conforme mencionado anteriormente, uma cadeia de caracteres (*string*) é definida por ser composta por uma sequência ordenada de símbolos, letras e números, como também cada símbolo que é armazenado na memória de um programa, se transforma em um valor numérico e este número também é considerado um carácter. Esta sequência ordenada de números, letras e símbolos, sempre é representada pelo último carácter que identificado por '\0', como também são pertencentes da mesma família dos seguintes caracteres, '\n' e '\t'. Com relação aos caracteres representados por símbolos, estes podem ser tanto espaços em branco, dígitos, como também pontos de interrogação e de exclamação, bem como formados pelos símbolos da matemática.

No entanto essa aplicação em utilizar o carácter '\0', esse marcador é utilizado na linguagem C, para indicar o final de uma função string, os símbolos aos quais representam a cadeia de caracteres, como descrito anteriormente podem ser letras, números e símbolos, esses definem a cadeia de caracteres, também denominadas de strings, vamos analisar alguns exemplos desses símbolos, como: (__espaços, ?, !, &), logicamente são alguns exemplos, se analisarmos não só a matemática, mas também a quantidade existente de caracteres que formam um computador, podemos neste momento imaginar vários exemplos.

Analisando a linguagem C, com o uso de caracteres, pode-se perceber que o último elemento '\0' indica o fim de uma cadeia de caracteres, no entanto este não se torna exibível, por isso em uma situação composta por n elementos, apresentará uma tabela de n + 1 elemento do tipo *char*, em uma cadeia de caracteres. Como o exemplo a seguir, String “Boa noite”, será apresentado da seguinte forma na tabela de caracteres:

```
B   o   a   n   o   i   t   e   \0
```

Conforme Sebesta (2010, p. 274), “cadeia de caracteres é um tipo no qual os valores consistem em sequências de caracteres. Cadeia de caracteres são também um tipo essencial para todos os programas que realizam manipulação de caracteres”.

2.2.1 Constantes do tipo caracteres

Podemos definir as constantes como valores pertencentes a programas de computador que por sua vez não podem mudar seus valores durante a sua execução. Segundo Aguilar (2011, p. 84), “os programas de computador contêm certos valores que não devem mudar durante a execução do programa”. Tais

valores são chamados de constantes. Entretanto, existem outros valores que mudarão durante a execução do programa; esses valores são chamados de variáveis.

As constantes também são tipos de caracteres, utilizados para verificar os valores que uma função possui ao ser executada em um programa de computador, esses valores podem ser vistos com as entradas e saídas dos dados. Conforme Sebesta (2010), constantes do tipo de caracteres são usados para rotular a saída, e a entrada e saída de todos os tipos de dados é geralmente feita em termos de cadeia. As constantes podem ser declaradas, utilizando a palavra *constat*, esta constante recebe um valor no mesmo momento em que é declarada, essa modificação do valor pode ocorrer sem nenhum problema durante a execução do programa.

Segundo Aguilar (2011), as constantes não mudam durante a execução do programa, na linguagem C++ existem quatro tipos de constantes:

- Constantes literais.
- Constantes definidas.
- Constantes enumeradas.
- Constantes declaradas.

As constantes literais são as mais usuais; recebem valores, tais como 45.32564,222 ou “introduza seus dados” que não são escritos diretamente no texto do programa. As constantes definidas são identificadores que se associam a valores literais constantes e que recebem determinados nomes. As constantes declaradas são como variáveis: seus valores são armazenados na memória, mas não podem ser modificados. As constantes enumeradas permitem associar um identificador, tal como cor, a uma sequência de outros nomes, como azul, verde, vermelho e Amarelo. (AGUILAR 2011, p. 76).

As constantes podem ser utilizadas para a declaração de valores em várias linguagens de programação, como C++, Fortran 95, C#, bem como outras linguagens. Sua principal finalidade é definir e verificar quais são os valores das constantes dos tipos de caracteres dessa linguagem. Sebesta (2010, p. 259) apresenta a linguagem Java, “esta permite a vinculação dinâmica de valores a constantes nomeadas. Em Java, constantes nomeadas são definidas com a palavra reservada final”. O valor inicial pode ser dado na sentença de declaração ou em uma sentença de atribuição subsequente.

2.2.2 Variáveis do tipo caracteres

As variáveis do tipo caracteres possuem como finalidade armazenar letras e símbolos que podem existir em um texto. São as variáveis do tipo inteiros que armazenam números que podem ser associados a símbolos. Aguilar (2011) apresenta as variáveis como um tipo de carácter, essas variáveis devem ser declaradas diretamente no algoritmo, e conforme a linguagem de programação terão uma notação específica. Vamos analisar esse procedimento de como realizar esta declaração da variável, conforme exemplo a seguir:

```
var  
carácter : C, D  
cadeia : NOME, CIDADE
```

Segundo Rita (2009, p. 40), “variáveis do tipo carácter são aquelas que podem armazenar informações compostas por um simples carácter ou um conjunto de letras, dígitos e símbolos especiais”. Com a utilização das aspas simples não é possível realizar cálculos, pois o programa considera apenas como um texto, no entanto ao utilizar as aspas duplas o programa consegue ser atribuído para uma cadeia de caracteres.

As variáveis do tipo carácter possuem alguns pontos importantes que devem ser falados, conforme Almeida (2008, p. 27) cita:

- **A utilização de aspas simples (')**: é usada na hora em que se faz atribuições de um valor para uma variável do tipo carácter.
- **A utilização de aspas duplas (")**: é usada na hora em que se faz atribuições de um valor para uma variável do tipo de cadeia de caracteres.

Vamos analisar na imagem abaixo como ficaria um exemplo das variáveis do tipo carácter utilizando aspas simples e aspas duplas:

FIGURA 37 – VARIÁVEL DO TIPO CARACTERES

```

VAR:  Caractere A <guarda um caractere>
      Cadeia Endereço <guarda uma sequência de caracteres>
INÍCIO
LEIA A <A variável A recebe um caractere escrito pelo usu-
ário>
LEIA Endereço <A variável Endereço recebe uma palavra es-
crita pelo usuário>

IMPRIMA A <Mostrará no monitor o valor da variável A>
IMPRIMA 'A' <Mostrará no monitor o caractere 'A'>

IMPRIMA "Endereço" <Mostrará no monitor a palavra "Endere-
ço">
IMPRIMA Endereço <Mostrará no monitor o valor da variável
Endereço>
FIM

```

FONTE: Almeida (2008)

2.2.3 Cadeia de caracteres e operações

As cadeias de caracteres são formadas por várias operações, como: atribuição, concatenação, referência a subcadeias, comparação e casamentos de padrões.

- Iniciando pede referência a subcadeias, está é conhecida por ser chamada de fatia. As operações de atribuição e comparação de cadeia de caracteres são compiladas com a possibilidade de compilar operadores de diversos tamanhos.
 - O casamento de padrões é uma operação fundamental para as cadeias de caracteres, pois pode ser suportada diretamente, como pode ser fornecida por uma função ou uma biblioteca de classe para uma linguagem de programação.
- (SEBESTA 2010, p. 275):



Segundo Sebesta (2010), a linguagem SNOBOL, foi a primeira linguagem bastante conhecida a suportar o casamento de padrões.

Segundo Aguilar (2011, p. 269), “uma constante do tipo carácter é um carácter entre aspas e uma constante de tipo cadeia é um conjunto de caracteres válidos entre apóstrofes – para evitar confundir com nomes de variáveis,

operadores e inteiros”. As operações de caracteres é uma forma primitiva do computador executar uma determinada operação de caracteres.

Em vez de tentar determinar o tempo de execução específico de cada operação primitiva, vamos simplesmente contar quantas operações primitivas serão executadas e usar este número t como uma estimativa de alto nível do tempo de execução do algoritmo. Essa contagem de operações está relacionada com o tempo de execução em um *hardware* e *software* específicos, pois cada operação corresponde a uma instrução realizada em tempo constante e existe um número fixo de operação primitiva. (GOODRICH e TAMASSIA 2002, p. 25).

Em várias situações uma operação necessita identificar e reconhecer um determinado número de caracteres de uma cadeia, como por exemplo o seu comprimento, assim como a união, também conhecida como concatenação de cadeias. Conforme Aguilar (2011, p. 272), “o tratamento de cadeias é um tema importante, por causa da grande quantidade de informação armazenada nele”. Segundo o tipo de linguagem de programação escolhido, teremos maior ou menor facilidade para a realização de operações, várias linguagens, bem como as orientadas a objetos realizam o processo de manipular uma variedade de funções de caracteres. Em qualquer caso, as operações com cadeias mais usuais são:

- Cálculo do comprimento.
- Comparação.
- Concatenação.
- Extração de subcadeias.
- Busca de informação.

2.2.4 Strings

Strings são definidas por serem caracteres utilizadas para representar valores textuais, como, por exemplo, nomes e endereços, podem ser consideradas como arranjos unidimensionais, onde os elementos são caracteres, como podem ser consideradas um tipo de carácter básico da linguagem. Na linguagem de programação os caracteres que compõem uma *string*, compreendem os símbolos de uma tabela que esteja codificada, lembrando novamente que os valores de uma *string* geralmente são escritos em uma sequência de caracteres e esses estão identificados com a utilização de aspas duplas, como por exemplo “a” e 'a', este segundo exemplo representa um tipo de dado de carácter e o primeiro representa um tipo de dado *string*. A seguir vamos analisar como as operações são tratadas em relação ao tipo *string*:

TABELA 7: OPERAÇÕES SOBRE *STRINGS*

OPERAÇÃO	DESCRIÇÃO
Inversão	Retorna a string escrita na ordem inversa.
Comparação	Compara se duas strings são iguais, retornando 0 (zero) em caso positivo.
Concatenação	Retorna a junção das duas strings, com a segunda string começando imediatamente após o fim da primeira.

FONTE: Costa e Neta, 2014

Um tipo de dado *String* é utilizado para representar cadeia de caracteres, onde esses dados podem ser textos, nomes, sentenças; geralmente, esses tipos de dados *strings* são textos pequenos, formados por cadeias de caracteres. “Para efetuar operações com cadeia de caracteres em Java, precisa-se acionar métodos presentes na classe *String*, de quem a cadeia específica de caracteres será um objeto”. (FEIJÓ; SILVA; CLUA, 2010, p. 93).

Segundo Ramos; Neto; Vega (2009, p. 140), “a classe *java.lang.string* disponibiliza várias funcionalidades para manipular textos, como: comparação, contagem, pesquisa, eliminação e inserção de caracteres, conversão de caixa de texto e tratamento de conjuntos de caracteres”.

Vamos analisar na imagem a seguir os principais métodos fornecidos por *java.lang.string*:

FIGURA 38 – *java.lang.string*

public boolean equals(Object) – verifica se o conteúdo da <i>String</i> equivale ao de determinado objeto;
public int length() – retorna o tamanho da <i>String</i> ;
public int indexOf(char c) – retorna a posição em que determinado caractere se encontra;
public char charAt(int index) – retorna o caractere que ocupa determinada posição;
public String[] split(String cadeia) – retorna um array de <i>String</i> proveniente da divisão da <i>String</i> ;
public String trim() – retorna uma nova <i>String</i> , na qual são removidos os espaços em branco do início e do fim da cadeia;
public String toLowerCase() – converte os caracteres da <i>String</i> para minúsculas;

public String toUpperCase() – converte os caracteres da String para maiúsculas;

public String substring(int inicio, int fim) – retorna uma nova String contendo os caracteres do intervalo determinado (no caso, fim–inicio);

public String replace(char antigo, char novo) – retorna uma nova String, na qual determinado caractere (em todas as vezes que aparecer) é trocado por um novo.

FONTE: Ramos; Neto; Vega (2009)

Existem, dentro da classe `java.lang.string`, vários outros métodos, não apenas estes exemplificados acima.

2.3 VETOR DE CADEIA DE CARACTERES

Um vetor pode ser considerado como uma cadeia de caracteres, quando é representado em algumas aplicações de um programa, um vetor é considerado um tipo `char`. Quando o vetor é classificado como tipo `char`, pode representar elementos como uma cadeia de caracteres, e esse vetor é apontando para ponteiros `char`, esse processo de representar os elementos e apontar para os ponteiros do tipo `char` é conhecido como vetor bidimensional de `char`.

Uma cadeia pode ser implementada como um vetor, cada elemento do vetor contendo um ou mais caracteres. O vetor deverá ser dimensionado de modo a conter a cadeia; assim, se o comprimento da cadeia é m e cada elemento do vetor pode conter p caracteres, então o número n de elementos do vetor deverá ser tal $p \cdot n \geq m$. Como por exemplo: $n = \lceil m/p \rceil$ (menor inteiro maior ou igual a m/p). O armazenamento de mais de um carácter por elemento torna necessária a existência de operações que obtenham ou modifiquem o valor de cada carácter. Tais operações em geral utilizam algum mecanismo de deslocamento (“shift”) para trazer o carácter desejado para a primeira (ou última) posição dentro do espaço ocupado pelo elemento. (VELOSO, SANTOS e AZEREDO 1983, p. 78).

Para realizar a declaração da variável para um vetor, como por exemplo de uma turma de alunos, deve-se declarar a quantidade de caracteres, como o nome de cada aluno, deve ter no máximo até 70 caracteres, e o limite de alunos na turma deve ser de 40. Todo esse processo de identificar e declarar as variáveis é chamado de vetor bidimensional, é nele que todos os caracteres contendo os nomes ou números de alunos que formam a turma são armazenados. Conforme podemos analisar no exemplo a seguir, como declarar uma variável de um vetor bidimensional:

```
char alunos [40] [70];
```


Em uma situação, onde a representação de vetores de cadeia de caracteres, são declaradas, através de um ponteiro, e este ponteiro realiza a função de alocar de forma dinâmica cada elemento da cadeia de caracteres. Com este processo, o espaço da memória é otimizado e dessa forma reduzido, pois quando a quantidade de alunos for menor do que o declarado, a dimensão máxima da cadeia de vetor não é utilizada, lembrando que cada elemento do vetor da cadeia de caracteres é um ponteiro. No caso de existir a necessidade de armazenar algum nome de aluno na cadeia de caracteres, será prontamente alocado um espaço na memória do vetor.

```
char* lelinha (void)
{
    char linha[121];      /* variavel auxiliar para ler linha */
    scanf(" %120[^\n]", linha);
    return duplica(linha);
}
```

FONTE: Celes e Rangel (2008)

Vamos analisar alguns exemplos de funções que realizam a captura de funções, em um vetor de cadeia de caracteres. A função em específico irá capturar o nome dos alunos de uma determinada turma, onde a função irá iniciar realizando a leitura da quantidade de números que estão alocados na turma, como também esta função irá capturar o nome desses alunos que estão alocados nesta turma e apresentar estes nomes em uma sequência de linhas, realizando a alocação conforme ordem dos nomes que estão alocados. Esta função realizará a captura de uma linha de nomes, por exemplo, e posteriormente irá fornecer um retorno para a cadeia alocada de forma dinâmica, apresentando a linha que foi inserida.

Vamos analisar como a função para capturar os nomes dos alunos preenche o vetor de nomes e pode ter como valor de retorno o número de nomes lidos:

```
int lenomes (char** alunos)
{
    int i;
    int n;
    do {
        scanf("%d", &n);
    } while (n>MAX);

    for (i=0; i<n; i++)
        alunos[i] = lelinha();
    return n;
}
```

FONTE: Celes e Rangel (2008)

A função para liberar os nomes alocados na tabela pode ser implementada por:

```
void liberanomes (int n, char** alunos)
{
    int i;
    for (i=0; i<n; i++)
        free(alunos[i]);
}
```

FONTE: Celes e Rangel (2008)

Uma função para imprimir os nomes dos alunos pode ser dada por:

```
void imprimenomes (int n, char** alunos)
{
    int i;
    for (i=0; i<n; i++)
        printf("%s\n", alunos[i]);
}
```

FONTE: Celes e Rangel (2008)

Um programa que faz uso destas funções é mostrado a seguir:

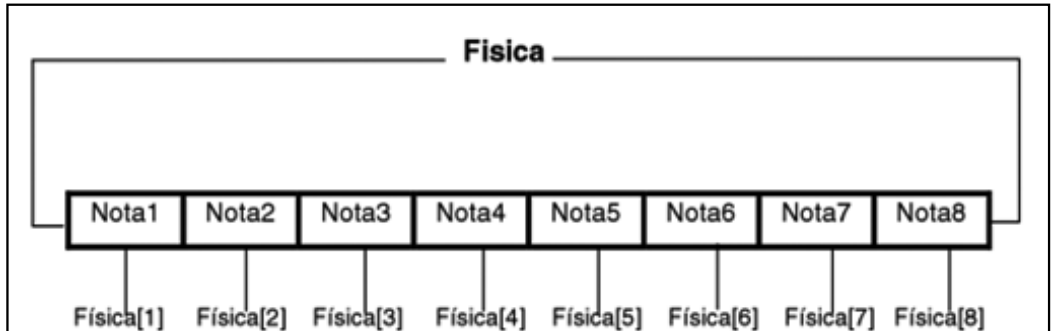
```
int main (void)
{
    char* alunos[MAX];
    int n = lenomes(alunos);
    imprimenomes(n,alunos);
    liberanomes(n,alunos);
    return 0;
}
```

FONTE: Celes e Rangel (2008)

2.3.1 Vetores unidimensionais

Leite (2006, p. 103) apresenta o vetor que possui apenas uma dimensão; “é como se seus elementos (notas mensais) pudessem ser expostos sobre uma linha”. Na realidade pode ser considerada como uma lista ordenada de elementos que são do mesmo tipo de dado, sempre iniciando pela sequência de um primeiro elemento, depois pelo segundo e assim consequentemente, como vamos apresentar na figura a seguir.

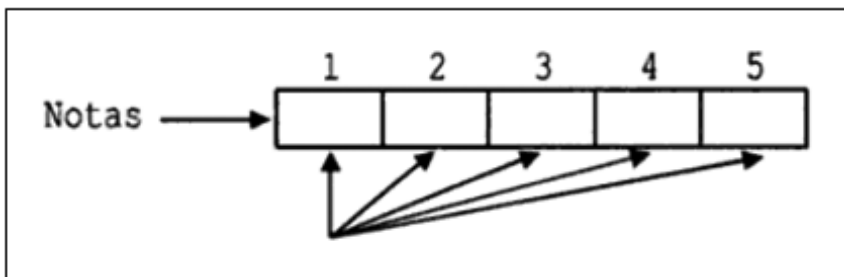
FIGURA 39 – VETORES



FONTE: Leite (2006)

“Pode-se definir um vetor unidimensional como uma lista ordenada de elementos do mesmo tipo. Por ordem entende-se que existe um primeiro elemento, um segundo, um terceiro e assim por diante”. (JUNIOR; et al. 2012). Já Lorenzi e Lopes (2000, p. 51), apresentam vetores como “estes permitem armazenar mais de um valor em uma mesma variável. Esta é dividida em posições, e cada posição pode receber um valor diferente, porém todos do mesmo tipo”. Conforme exemplo a seguir:

FIGURA 40 – VETORES UNIDIMENSIONAIS



FONTE: Lorenzi e Lopes (2000)

Observando que a cada posição apresentada, na figura anterior, deste vetor de Notas, podem receber valores, no entanto, apenas valores pré-definidos para cada tipo de vetor. Outro fator interessante é cada posição do vetor, ou cada

elemento do vetor pode ser acessado através de um índice, também desenvolvido para estes elementos do vetor. Conforme Lorenzi e Lopes (2000, p. 52), “um vetor pode ser declarado na seção de variáveis ou na seção tipos”. Quando definido um tipo de vetor na declaração de tipos, ele pode ser utilizado por qualquer variável declarada no programa, inclusive nas sub-rotinas. Vamos analisar o exemplo a seguir:

FIGURA 41 – DECLARAÇÃO DE TIPOS DE VETORES

```

Sintaxe: type
    Nome_tipo_vetor = array[1..tamanho] of tipo;
var
    Nome_variável : nome_tipo_vetor;

ou

var
    Nome_variável : array[1..tamanho] of tipo;

onde: Tamanho = número de posições do vetor;
        Tipo = qualquer tipo de dado.

Exemplo: type
    T_vetor = array[1..5] of Real;
var
    Notas : T_vetor;
  
```

FONTE: Lorenzi e Lopes (2000)

Como cada tipo de vetor possui estipuladas suas predefinições para os vários tipos de posições, este tipo pode ser manipulado, de formas diferentes dos outros tipos, como pode ser percebido na figura anterior, onde é declarado o tamanho, as posições e os tipos de dados dos vetores. Segundo Feijó, Silva e Clua (2010, p. 80), “um vetor unidimensional, em Java, caracteriza um conceito que admite diversos valores, com a restrição de que todos os valores em um vetor devem ter a mesma natureza”.

2.3.2 Leitura de vetores

“A leitura de um vetor pode ser feita com o emprego da estrutura de controle para...fim_para dentro da qual cada elemento é lido para a memória”. (LEITE 2006, p. 105).

Vamos analisar a seguir um pseudocódigo que realiza a leitura de um vetor denominado Números, de 15 elementos inteiros:

FIGURA 42 – LEITURA DE VETORES

```
programa LeVetor
var
  Numeros: array[1..15] de inteiro
  j: inteiro
início
  para j de 1 até 15 faça
    leia Numeros[j]
  fim_para
fim
```

FONTE: Leite (2006)

Tanto a leitura como a escrita de dados com vetores são denominadas como entrada e saída, são geralmente utilizadas em estruturas em que as operações são repetitivas, no entanto, as duas podem também ser utilizadas em estruturas seletivas. Pode-se permitir aos elementos de um vetor introduzir dados (escrever) nele ou visualizar seu conteúdo (ler). “A operação de efetuar uma ação geral sobre todos os elementos de um vetor é denominada varredura do vetor.” (AGUILAR 2011, p. 233).

2.3.3 Escrita de vetores

A escrita por sua vez recebe definições como argumentos de um apontador, isso ocorre quando é o início do espaço temporário, em que se necessita armazenar dados, como também são declarados os espaços que estes dados temporários necessitam. O processo de escrita dos elementos de um vetor segue o mesmo princípio da leitura. A única diferença é uma alteração a ser feita no comando, mudar de leia para escreva, dentro do laço para...fim_para.

Vamos analisar o exemplo do pseudocódigo a seguir:

FIGURA 43 – ESCRITA DE VETORES

```
programa EscreveVetor
var
  Numeros: array[1..15] de inteiro
  j: inteiro
início
  para j de 1 até 15 faça
    escreva Numeros[j]
  fim_para
fim
```

FONTE: Leite (2006)

“A escrita escreve sobre uma região contínua de memória para o descritor de arquivo. Todavia, um programa muitas vezes irá precisar escrever muitos itens de dados, cada um residindo em uma diferente parte da memória”. (MITCHELL; OLDHAM e SAMUEL 2001, p. 346). Isso quer dizer que cada elemento possui como responsabilidade especificar ou identificar uma região da memória que ainda precisa ser escrita, esse processo pode ser feito declarando uma variável como struct iovec, se possuir a certeza do espaçamento da região que irá precisar, caso contrário, se o número de regiões não for exata pode sofrer alterações de tamanho, é possível alocar um vetor de forma dinâmica.

RESUMO DO TÓPICO 1

Neste tópico vimos que:

- Os caracteres podem ser definidos como dados primitivos, são encontrados na maioria dos computadores, estão relacionados com a linguagem da máquina de um computador e essa linguagem máquina permite a manipulação desses dados primitivos.
- A definição de cadeia de caracteres é formada por uma sequência de caracteres, essas cadeias de caracteres são ordenadas para que os dados possam ser manipulados pela cadeia de caracteres, uma sequência de *bits* de tamanho fixo é considerada como um conjunto codificado de caracteres.
- Uma cadeia de carácter (string) é composta por uma sequência ordenada de símbolos, letras e números, como também são caracteres os espaços em branco, dígitos e pontos, essa sequência pode ser representada pelo último carácter identificado como '\0', são pertencentes dessa família de caracteres os símbolos '\n' e '\t'.
- As constantes do tipo caracteres são definidas como valores que pertencem a um programa de computadores. Esses valores não podem ser alterados durante a execução do programa e são chamados de constantes.
- As variáveis do tipo caracteres possuem a finalidade de realizar o armazenamento de letras e símbolos, que podem estar dentro de um texto. Essas variáveis são do tipo inteiros e armazenam números que podem ser associados aos símbolos.
- Uma cadeia de caracteres possui sua formação através de operações. Essas operações são constituídas através de Atribuição, Concatenação, Referência a Subcadeias e por Casamento de Padrões.
- O vetor é considerado uma cadeia de caracteres, este é considerado como um tipo de Char, o tipo char pode representar elementos de cadeia de caracteres e ser apontado para ponteiros também do tipo Char.
- Vetor unidimensional é definido por ser composto por uma linha ordenada de elementos e que são um tipo de dado. Para iniciar, a lista segue uma sequência crescente, como por exemplo, primeiro, segundo e terceiro elemento e assim por diante. Os vetores unidimensionais realizam a leitura e escrita de vetores.

A leitura de vetores possui como função realizar a leitura dos elementos da estrutura de dados para a memória. A escrita de vetores por sua vez possui a responsabilidade de armazenar espaço na memória para dados temporários. Os dois utilizam o comando *para...fim_para* de estrutura de controle.

AUTOATIVIDADE



1 As variáveis do tipo caracteres, possuem como finalidade armazenar letras e símbolos, estes estão acoplados dentro de um texto. As variáveis são do tipo inteiros e com isso armazenam os números, para que estes possam ser associados aos símbolos. As variáveis do tipo carácter possuem algumas características bem importantes, como realizar a atribuição de um valor para uma variável. Analise as sentenças a seguir e classifique V para as sentenças verdadeiras e F para as falsas.

- () Utilização de caracteres '`\n`' e '`\t`'.
- () Utilização de aspas simples (').
- () Utilização de aspas duplas (").
- () Utilização de caracteres '`\0`'.

Assinale a alternativa que apresenta a sequência correta:

- a) () V – V – F – F.
- b) () V – F – F – V.
- c) () F – V – V – F.
- d) () F – F – F – F.

2 Strings são caracteres e possuem como funcionalidade representar os valores textuais, como por exemplo nome e endereços. Em uma linguagem de programação a string é compreendida por símbolos que estão codificados, e os valores de uma string devem ser escritos em uma sequência lógica. Uma string pode realizar operações com tratamentos como: pode ser escrita em ordem inversa; compara duas strings e retorna a junção das duas strings comparadas, com relação a essas operações, classifique V para as sentenças verdadeiras e F para as falsas.

- () Comparação.
- () Inversão.
- () Concatenação.
- () Compilação.

Assinale a alternativa que apresenta a sequência correta:

- a) () V – F – F – F.
- b) () V – V – V – F.
- c) () V – V – F – V.
- d) () F – V – V – F.

TIPOS DE ESTRUTURA
DE DADOS

1 INTRODUÇÃO

Os tipos de estruturas de dados precisam ser definidos, específicos e eficientes, pois para cada linguagem de programação existe uma estrutura de dados apropriada. Isso tudo pode ser definido conforme a necessidade do programa a ser desenvolvido, onde existe a necessidade de definir as operações para cada estrutura e que este processo seja eficiente.

Uma característica muito interessante da estrutura de dados se deve ao seu objetivo de analisar a melhor a forma de como organizar os dados, métodos de armazenamento eficiente dos dados e quais códigos e algoritmos são mais propícios para realizar as operações de manipulação dos dados da estrutura. É lógico que não existe um processo predefinido, pois cada programa e cada linguagem de programação possuem suas especificidades e suas necessidades a serem atendidas.

As estruturas de dados podem conter vários tipos de aplicações. Estas suportam e conseguem auxiliar os códigos de programação, onde são fornecidos processos de estruturas como listas lineares, conjuntos de elementos e árvores, em que cada um desses elementos realiza um processo, seja ele de organizar, armazenar ou de acessar os dados que precisam ser utilizados em uma estrutura de dados.

O objetivo deste tópico de estudos é levar a uma compreensão sobre alguns pontos muito importantes e pertinentes aos estudos das estruturas de dados. Os estudos se darão com os seguintes temas: Variáveis do tipo Heterogênea; Variáveis do tipo Homogênea; Arranjos Unidimensionais; Arranjos Multidimensionais; Ponteiro para estruturas; Operadores; Definição de “novos” tipos; Comando Typedef; Vetores de estruturas; Vetores de ponteiros para estruturas; Tipo união e Tipo enumeração.

Vamos aos estudos!

2 TIPOS ESTRUTURADOS

Estruturar dados que são muitas vezes muito complexos exige que alguns processos sejam realizados. Como as informações são compostas por vários tipos de campos, devem ser analisados, contudo, como se processam os mecanismos que possibilitam o agrupamento de tipos distintos. A seguir vamos conhecer mais sobre como estruturar um tipo de dado. Segundo Edelweiss e Galante (2009, p. 36), “a representação dos dados manipulados por uma aplicação pode ser feita por diferentes estruturas de dados”. Um fator que determina o papel dessas estruturas no processo de programação de aplicações é a identificação de quão bem as estruturas de dados coincidem com o domínio do problema a ser tratado.

Conforme Pereira (1996, p. 5), os objetivos das estruturas de dados, podem ser identificados como teórico e prático:

- Teórico: identificar e desenvolver modelos matemáticos, determinando que classes de problemas podem ser resolvidas com o uso deles.
- Prático: criar representações concretas dos objetos e desenvolver rotinas capazes de atuar sobre estas representações.

2.1 TIPO ESTRUTURA

Um tipo estrutura de dados é definido por conter vários tipos de valores sua estrutura considerada muito básica é definida por ser um tipo de estrutura simples, tem como possibilidade otimizar e dinamizar utilização da memória de um programa de computador, disponibilizam estruturas mais compreensíveis para o desenvolvimento de programas mais complexos, isso ocorre através da codificação dos dados da estrutura. Conforme Pereira (1996, p. 4)

a maioria das linguagens de programação modernas oferecem um conjunto básico de tipos de dados primitivos (inteiro, real, caracteres e lógicos) que podem ser usados para solução de problemas em geral, bem como alguns mecanismos que permitem criar agrupamentos complexos de dados destes tipos primitivos (vetor, registro, união, ponteiro).

Na linguagem de programação C, um tipo de dado também é composto por diversos valores e estes são de tipo simples, para entender melhor pode-se dizer: para que um programa possa manipular dois pontos diferentes isso deve ocorrer na declaração dos pontos, como x e y, em que a linguagem C realiza o agrupamento de dados. Caso não se tenha essa opção de realizar o agrupamento dos dois componentes citados, os pontos teriam que ser representados em duas variáveis independentes. Vamos analisar no exemplo a seguir essa afirmação dos dois pontos de forma agrupada:

```
struct ponto {
    float x;
    float y;
};
```

Neste exemplo é demonstrado como é realizado o agrupamento dos dados, dos dois pontos *x* e *y*, os dois pontos são considerados como um tipo de variável simples. Após esse agrupamento é possível acessar os valores pertencentes aos dois pontos declarados na variável, como também é possível acessar todos seus dados, como nome e endereços, entre outros dados.

Um tipo de dado é definido por ser um conjunto de valores e formado por um conjunto de operações. Os dados podem ser números inteiros e estão presentes em várias linguagens de programação. Esses valores de tipo de dado que são números inteiros podem ser representados pelos seguintes números, -2, -1, 0, 1, 2 e assim consequentemente. Já os dados do tipo operações podem ser apresentados como uma sequência muito grande de valores, mas as mais comuns são soma, divisão, multiplicação, subtração entre outras operações, que podem ser vários tipos.

Resumidamente, um tipo estruturado de dados possui como finalidade permitir que dados complexos sejam estruturados, com objetivo de compor todos os pontos, declarar seus valores e suas variáveis, isso para diversos campos da estrutura. Como mencionado no exemplo dos pontos, anteriormente, o tipo estrutura realiza o agrupamento de diversas variáveis dentro de uma única estrutura. Analise a seguir um exemplo de uma declaração de uma variável do tipo ponto:

```
struct ponto p;
p. x = 10.0;
p. y = 5.0;
```

Os elementos podem ser acessados através do operador de acesso “ponto” (.), onde:

```
/* declara o ponto do tipo struct */;
/* declara o p como sendo uma variável do tipo struct ponto */;
/* este acessa os elementos de ponto */;
```

Segundo Edelweiss e Galante (2009, p. 36), “‘tipos de dados’ e ‘estruturas de dados’ são considerados como semelhantes, no entanto, os significados de

cada um são diferentes um do outro”, conforme seguem as explicações dessas diferenças:

- Um tipo de dado consiste da definição do conjunto de valores (denominado domínio) que uma variável pode assumir ao longo da execução de um programa e do conjunto de operações que podem ser aplicados sobre ele.
- Os tipos de dados estruturados permitem agregar mais do que um valor em uma variável, existindo uma relação estrutural entre seus elementos. Os principais tipos de dados estruturados fornecidos pelas linguagens de programação são: arranjos, registros, sequências e conjuntos.

Ainda conforme Edelweiss e Galante (2009, p. 36), existem os tipos de dados básicos e os tipos de dados definidos pelo usuário, vamos analisar os dois tipos a seguir:

- Os tipos de dados básicos, também denominados tipos primitivos, não possuem uma estrutura sobre seus valores, ou seja, não é possível decompor o tipo primitivo em partes menores. Os tipos básicos são, portanto, indivisíveis.
- Os tipos de dados definidos pelo usuário são também tipos de dados estruturados, construídos hierarquicamente através de componentes, os quais são de fato tipos de dados primitivos e/ou estruturados. Um tipo definido pelo usuário é constituído por um conjunto de componentes, que podem ser de tipos diferentes, agrupados sob um único nome.

2.1.1 Variáveis do tipo heterogênea

Variáveis do tipo heterogênea são do tipo estruturado, são definidas por serem elementos de tipos diferentes, os elementos que são do tipo estrutura de dados são heterogêneos, este conjunto de elementos que não são do mesmo tipo de dados, podem ser conhecidos como um registro. Vamos analisar o seguinte exemplo: uma pessoa precisa comprar uma passagem de ônibus e precisa saber algumas informações referente: qual será o número da passagem, origem e destino, data, horário de saída e chegada, número da poltrona, idade do viajante e nome do passageiro. Berg e Figueiró (2006, p. 149), apresentam “as variáveis do tipo heterogêneas são um conjunto de dados, onde os elementos não são do mesmo tipo de dados”. Segundo Aguilar (2011, p. 246), “um registro é uma estrutura de dados heterogênea, o que significa que cada um de seus componentes pode ser de tipos de dados diferentes. Consequentemente um arranjo de registros é uma estrutura de dados cujos elementos são do mesmo tipo heterogêneo”.

Podemos considerar cada um desses dados como Campos, os dados são os seguintes: número de passagem é um dado (inteiro), origem e destino são (caracteres), data e horário são (caracteres), número da poltrona e idade são do tipo (inteiro) e por último o nome do passageiro é um dado (carácter).

FIGURA 44 – REGISTRO DE VARIÁVEIS DO TIPO HETEROGÊNEA

PASSAGEM DE ÔNIBUS	
NÚMERO: 0001	
De:	Para:
Data: ____ / ____ / ____	Horário: ____ : ____
Poltrona: _____	Distância: _____

FONTE: Frozza (2014)

2.1.2 Variáveis do tipo homogênea

Variáveis do tipo homogênea também são do tipo estruturado, no entanto, sua definição em relação aos elementos é de que esses elementos podem ser do mesmo tipo, elementos como vetores e matrizes são do tipo homogêneas. As variáveis Compostas Homogêneas são alocadas em posições na memória de um programa, essas posições são identificadas através de um único nome do dado, podem ser definidas por serem individualizadas através de índices, seus dados são do mesmo tipo de elementos. Segundo Aguilar (2011, p. 246), “um arranjo é uma estrutura de dados homogênea, o que significa que cada um dos seus componentes deve ser do mesmo tipo de dado”. Berg e Figueiró (2006) apresentam que um conjunto homogêneo de dados é composto por variáveis do tipo primitivo.

Como o exemplo a seguir:

Notas:	6,0	5,0	4,5	9,0	10	7,0	2,0	8,0	3.5
Posição:	0	1	2	3	4	5	6	7	8

Pode ser percebido que a posição é o índice e as notas são os elementos que são individuais, para cada aluno, mas na estrutura de dados cada elemento está inserido em um índice, até mesmo para que o programa esteja organizado e que se tenha facilidade em uma determinada busca de dados ou de elementos.

2.1.3 Arranjos unidimensionais

Os arranjos unidimensionais são definidos por serem utilizados para armazenar vários conjuntos de dados, em que esses dados são elementos que estão diretamente endereçados para um único índice de elementos. Os arranjos unidimensionais também são conhecidos como vetores. Edelweiss e Galante (2014, p. 165) dizem “um vetor é um arranjo de uma só dimensão que, portanto, necessita um só índice para acessar seus elementos”.

As características de um vetor são:

- nome, comum a todos os elementos;
- índice que identificam, de modo único, a posição dos elementos dentro do vetor;
- tipo dos elementos (inteiros, reais, entre outros), comum a todos os elementos do vetor;
- conteúdo de cada elemento do vetor.

Este arranjo possui como finalidade armazenar elementos de dados em uma memória de computador, no entanto, esses elementos são armazenados de forma organizada, seguindo um armazenamento em sequência crescente, onde um elemento é organizado e armazenado após o outro. Conforme Edelweiss e Galante (2014, p. 165), “os arranjos podem ser unidimensionais (de uma só dimensão), quando precisam de um só índice para identificar seus elementos, os arranjos de uma única dimensão também são chamados de vetores”.

Outra característica deste arranjo é que o armazenamento é realizado através de um único identificador e possui a capacidade de guardar somente um valor, como também podem ser acessados seus elementos de forma aleatória. Ele possui outras características que o definem como:

- Seus valores são do mesmo tipo de dado.
- Possui vários tipos de valores definidos.
- É identificado por um único nome como uma variável.
- Os valores podem ser acessados de forma aleatória.
- Seus elementos e dados podem ser armazenados e acessados de forma aleatória.

Vamos analisar o exemplo a seguir que demonstra um único identificador, para uma estrutura de dados. Neste exemplo será trabalhada a altura de pessoas:

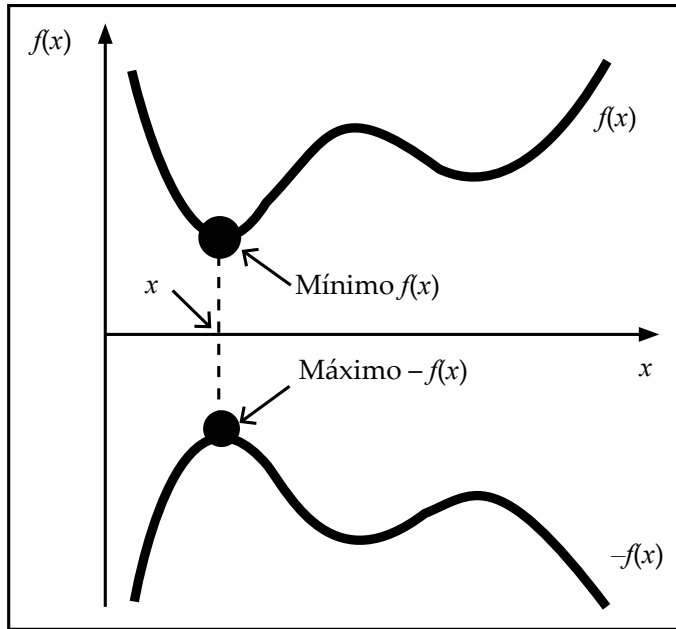
Valores:	1,60	1,80	1,65	2,00	2,05	1,76
Índices:	0	1	2	3	4	5

Podemos perceber que os valores correspondem aos dados de altura das 5 pessoas, são conteúdos lidos do usuário. No item índices é apresentada a ordem crescente de forma organizada, como os valores estão armazenados e como podem ser acessados e manipulados pelo índice.

Para otimizar e solucionar problemas de cálculos de raízes, vários métodos numéricos diretos estão disponíveis para auxiliar arranjos unidimensionais, Chapra (2012), apresenta como os problemas unidimensionais envolvem funções que dependem de uma única variável dependente. Vamos analisar a figura a

seguir que demonstra um arranjo unidimensional otimizando os problemas das funções:

FIGURA 45 – OTIMIZAÇÃO DO ARRANJO UNIDIMENSIONAL



FONTE: Chapa (2012)

Pode ser percebido na imagem a cima que para resolver os problemas de otimização das funções unidimensionais, a otimização oscila entre picos, subindo e descendo maximizando a eficiência dos arranjos e minimizando os problemas das funções.



VOCÊ SABIA QUE:

"Os arranjos em Java são unidimensionais: usa-se um único índice para acessar cada célula do arranjo. Apesar disso, existe uma maneira de definir arranjos de duas dimensões em Java – pode-se criar um arranjo de duas dimensões como um arranjo de arranjos. Isto é, pode-se definir um arranjo bidimensional como sendo um arranjo em que cada uma de suas células é outro arranjo. Tal arranjo bidimensional é por vezes chamado de matriz". (GOODRICH e TAMASSIA, 2002, p. 116).

2.1.4 Arranjos multidimensionais

Os arranjos multidimensionais também possuem a funcionalidade de armazenar vários tipos de conjuntos de dados, com a diferença que esses conjuntos de dados precisam que os elementos sejam endereçados para mais de um índice de elementos. Conforme Edelweiss e Galante (2014, p. 165), “os arranjos podem ser multidimensionais (de duas ou mais dimensões), também são chamados de matrizes”.

As matrizes podem ter duas ou mais dimensões, embora a grande maioria dos problemas não envolva mais do que três ou quatro. O número de dimensões de uma matriz deverá ser definido em função das necessidades do problema que está sendo analisado e das limitações eventuais da linguagem em uso. Isso porque, embora teoricamente não exista limitação para o número de dimensões de uma matriz, uma implementação particular de uma linguagem pode definir um limite para esse número. (EDELWEISS e GALANTE 2014, p. 165).

Uma característica muito interessante deste arranjo e que seus dados podem ser agrupados em diferentes direções, como sua variável possui um único nome denominado para o conjunto de dados, precisa-se realizar a diferenciação de um elemento do outro, esse procedimento pode ser realizado através de um índice que seja associado ao conjunto de dados desses elementos.

FIGURA 46 – ARRANJOS MULTIDIMENSIONAL

	0	1	2	3	4
0	0.0	0.1	0.2	0.3	0.4
1	1.0	1.1	1.2	1.3	1.4
2	2.0	2.1	2.2	2.3	2.4
3	3.0	3.1	3.2	3.3	3.4
4	4.0	4.1	4.2	4.3	4.4

FONTE: A autora

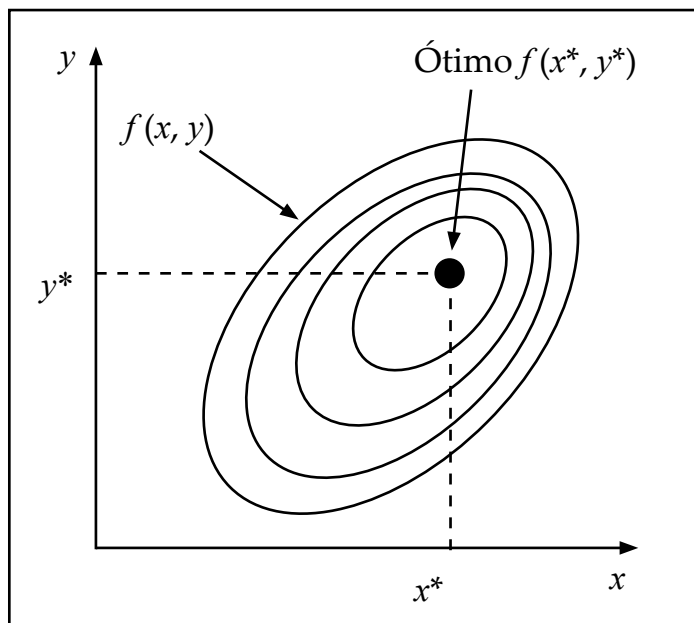
A diferenciação dos arranjos multidimensionais pode ser percebida no exemplo anterior, onde cada elemento está disposto em linhas ou colunas, ordenados de forma organizada e sequencial.

No arranjo multidimensional também podem ser encontrados problemas de otimização.

Os problemas multidimensionais envolvem funções que dependem de duas ou mais variáveis dependentes, da mesma forma que o arranjo unidimensional, este possui a mesma lógica para solucionar os problemas de otimização, seguindo o fluxo que oscila entre o mínimo e máximo, fazendo com que os problemas de otimização sejam diminuídos. (CHAPRA 2012, p. 185).

Segundo o referido autor, do mesmo modo como uma trilha real não se limita a andar em uma única direção; em vez disso, a topografia deve ser examinada a fim de se alcançar a meta de forma eficiente. Vamos analisar a seguir o exemplo da otimização multidimensional:

FIGURA 47 – OTIMIZAÇÃO DO ARRANJO MULTIDIMENSIONAL



FONTE: Chapa (2012)

Como já mencionado anteriormente, para solucionar problemas de funções a lógica segue a mesma do arranjo unidimensional. O arranjo multidimensional realiza o mesmo processo de otimização oscilando entre picos, subindo e descendo, maximizando a eficiência dos arranjos e minimizando os problemas das funções. Chapra deixa claro, na figura anterior, como pode ser observada a otimização e demonstra como pode ser utilizada para representar a maximização (os contornos aumentam em elevação até o máximo, como em uma montanha) ou a minimização (os contornos decrescem em elevação até o mínimo, como em um vale).

2. 2 PONTEIRO PARA ESTRUTURAS

Ponteiros são definidos como variáveis e possuem a responsabilidade de armazenar endereços na memória de um programa de computador. Ao realizar o processo de armazenamento de endereços na memória do programa é possível que sejam incluídos valores, bem como, também, podem ser utilizados. No entanto esse processo não é apenas armazenar ou gravar um endereço na memória, para que isso ocorra existe a necessidade de se fazer a declaração de um ponteiro, para que realize este armazenamento do endereço. Brookshear (2013, p. 313) apresenta:

um ponteiro como uma área de armazenamento que contém um desses endereços codificados. No caso de estruturas de dados os ponteiros são usados para gravar a posição na qual os itens de dados estão armazenados. Por exemplo, se precisássemos mover repetidamente um item de dados de uma posição para outra, poderíamos designar uma posição fixa para servir como um ponteiro. Então, cada vez que movêssemos um item, poderíamos atualizar o ponteiro para refletir o novo endereço dos dados. Posteriormente, quando precisássemos acessar o item de dados, poderíamos encontrá-lo por meio do ponteiro. Na verdade, o ponteiro sempre “apontará” para os dados.

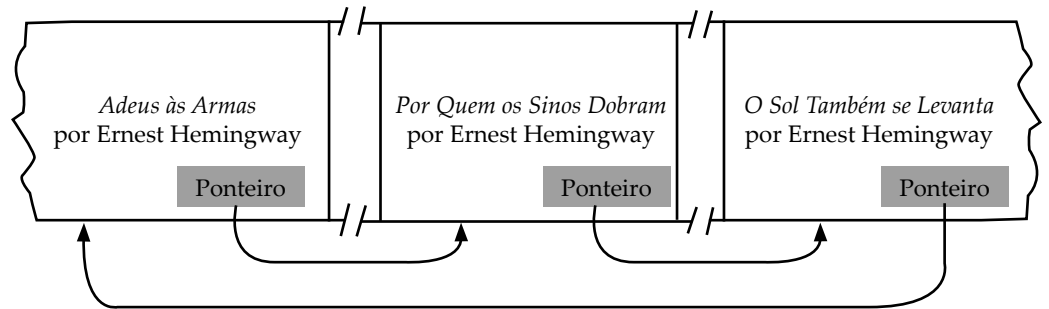
O ponteiro, também conhecido como apontador, é definido por ser uma variável que possui como uma de suas principais finalidades armazenar endereço na memória de um programa, tendo como características em suas essenciais funcionalidades apontar e acessar a posição, o endereço que a variável está armazenada, apontar para as variáveis que estão alocadas e declaradas, sua associação é sempre em relação ao tipo da variável que se está apontando.

Para que dados e informações possam ser encontrados em uma lista enorme de dados, é possível, segundo Brookshear (2013, p. 313),

reservar uma célula de memória adicional, dentro de cada bloco de células como ponteiros, onde esses ponteiros serão apontados para outros blocos, estes representam um livro do mesmo autor. Com isso os livros com os mesmos autores, mas com temas diferentes em comum podem tornar-se interligados com setas de identificação.

Vamos analisar a figura a seguir, que representa esta funcionalidade do ponteiro.

FIGURA 48 – PONTEIROS INTERLIGADOS



FONTE: Brookshear (2013)

O ponteiro pode apontar para uma estrutura. “Pode-se declarar um ponteiro para uma estrutura tal como se declara um ponteiro para qualquer outro objeto. Declara-se um ponteiro: especificando um ponteiro em lugar do nome da variável estrutura”. (AGUILAR, 2011, p. 330). Ao realizar a declaração do ponteiro, também são indicados ao compilador a quantidade de espaço da memória que se necessita fazer o armazenamento dos valores. Após realizar esses procedimentos de declarar o ponteiro e indicar para o compilador o espaço necessário na memória, uma variável identificada como tipo ponteiro, será apontada para uma variável que está identificada como tipo (char, int, float e double).

Uma característica muito interessante que deve ser observada ao realizar a declaração de um ponteiro é: precisa-se especificar para qual localização e qual tipo de variável que este ponteiro será apontado. Geralmente, os ponteiros são declarados através do seguinte operador (*), utilizado antes do nome da variável ao qual deverá ser apontado. Podemos citar como exemplo deste comando de declaração de ponteiro:

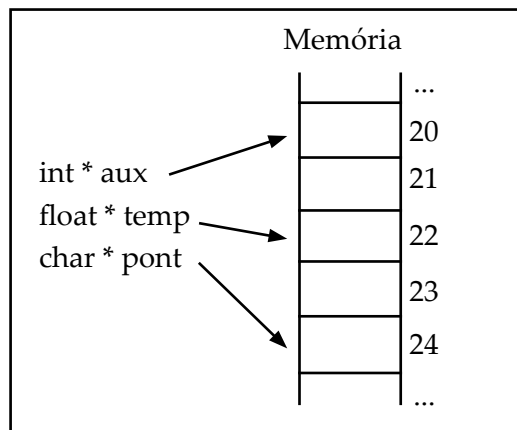
```
Tipo *nomeVariavel;
```

Outro exemplo:

```
Int *aux;  
Float *temp;  
Char *pont;
```

Essas variáveis, como aux, temp e pont, realizam o procedimento de armazenar os endereços na memória do programa, pois estão utilizando o operador (*) para apontar para a variável. Vamos analisar a figura a seguir que trata essas variáveis:

FIGURA 49 – PONTEIROS



Fonte: Almeida (2013)

Analisando a figura é possível verificar que as variáveis do tipo `aux`, `temp` e `pont`, realizam o armazenamento dos endereços na memória do programa, lembrando que isso é necessário, pois ao declarar as variáveis é possível identificar o espaço necessário que essas variáveis vão ocupar no programa, é um fator muito importante para o desenvolvimento de programas. O operador (`*`) é utilizado para identificar e acessar a posição de memória da variável, observando que é a posição da variável que deve ser acessada e não o endereço da memória.

Os ponteiros possuem algumas vantagens muito interessantes e importantes, como possibilitam que se tenha passagem para os parâmetros de funções por referência, além de alocar endereço na memória, também possui a finalidade de liberar espaço na memória do programa e este processo ocorre de forma dinâmica, durante a execução do programa que está sendo utilizado, outra vantagem que é muito importante no processo de um ponteiro é que o mesmo permite que sejam implementadas de forma eficaz e dinâmica novas estruturas de dados.

2.2.1 Operadores

“O significado de um operador – é uma operação que realiza e tipo de resultado – depende dos tipos dos seus operandos. Até que se conheça o tipo de operando(s), não se pode conhecer o significado da expressão”. (AGUILAR 2008, p. 12). Os ponteiros possuem os seguintes operadores utilizados para realizar a declaração das variáveis, são: (`*` , `&` e `Null`). O operador (`*`) possui como finalidade apontar a variável para um ponteiro, bem como realizar a recuperação de conteúdo das variáveis declaradas. O operador (`&`) tem a funcionalidade de indicar que um ponteiro poderá apenas receber endereços de memória, esta é sua única e principal função para um operador. O operador Nulo (`Null`) possui como funcionalidade receber valores dele mesmo, valores nulos, no entanto, esses valores não serão apontados para nenhuma variável na memória.

Especificamente o operador (`*`) realiza as seguintes funções:

- preferenciamento da variável;
- armazenar através de;
- apontar uma variável para um ponteiro.

O ponteiro (`&`) possui as seguintes funcionalidades:

- identificar o endereço da memória;
- armazenar um endereço na memória.

A seguir a figura demonstra esses dois operadores, que possuem grande importância para a definição dos ponteiros e para a declaração das variáveis:

FIGURA 50 – OPERADORES DE PONTEIROS



FONTE: Almeida (2013)

Conforme Pinheiro (2012, p. 112), “os operadores podem ser classificados quanto ao número de operadores em unários, binários e ternários. Os operadores unários requerem um operador, os binários, dois e os ternários, três números. Podem existir operadores que realizam operações com quatro, cinco ou mais operações”. Uma característica interessante do operador é que ele é avaliado pelas operações, pode resultar em um valor específico para a função. Ainda segundo o autor:

a notação utilizada para representar uma operação pode ser classificada como:

- prefixada, se o operador ocorre antes dos operandos;
- pós-fixada, se o operador ocorre após os operandos;
- infixada, se o operador ocorre entre os operandos.

Um operador nada mais é do que expressões, caracteres e vários jargões da linguagem de programação, que são considerados como valores e podem ser utilizados para realizar a alimentação dos operadores. Podemos encontrar vários tipos de operadores e cada um possui sua finalidade e especificidade. A seguir vamos listar cada um desses operadores, para que tenha uma noção de quais são eles:

- Operadores Aritméticos.
- Operadores Bit – a – Bit.
- Operadores de Comparação.
- Operadores de Controle de erro.
- Operadores de Execução.
- Operadores de Incremento/Decremento.
- Operadores Lógicos.
- Operadores de String.
- Operadores de Arrays.
- Operadores de Tipo.



Acesse o *Link* a seguir e saiba mais sobre cada um desses operadores: [<http://php.net/manual/pt_BR/language.operators.php>](http://php.net/manual/pt_BR/language.operators.php).

2.3 DEFINIÇÃO DE “NOVOS” TIPOS

Na linguagem de programação C é possível realizar a soma de dois números inteiros, somando as duas frações seguindo as funções com parâmetros, utilizando os novos tipos de dados, que são a base do tipo de dados primitivos, como o *char*, *int*, *float* e *double*, esses tipos de dados primitivos também são conhecidos como tipos estruturados. Lembrando que um tipo estruturado de dado é uma variável que foi declarada e definida pelo usuário, levando em consideração que as variáveis fazem parte da estrutura de dados.

Um benefício dos objetos é que eles oferecem ao programador uma forma conveniente de construir novos tipos de dados. Suponha que você necessite trabalhar com coordenadas *x* e *y* (par de quantidades numéricas independentes) no seu programa. Seria interessante expressar operações sobre esses valores simplesmente usando operações aritméticas normais, como:

$$\text{posição1} = \text{posição2} + \text{origem}$$

Onde essas variáveis *posição1*, *posição2* e *origem* representam as coordenadas.

FONTE: Filho (2011, p. 11)

É perceptível que ao criarmos essa classe, como no exemplo do autor anteriormente mencionado, esta classe realiza a função de incorporar esses dois valores que estão declarados, como *posição1*, *posição2* e *origem* e os torne como objetos dessa classe. Com isso pode-se dizer que estamos criando um novo tipo de dado.

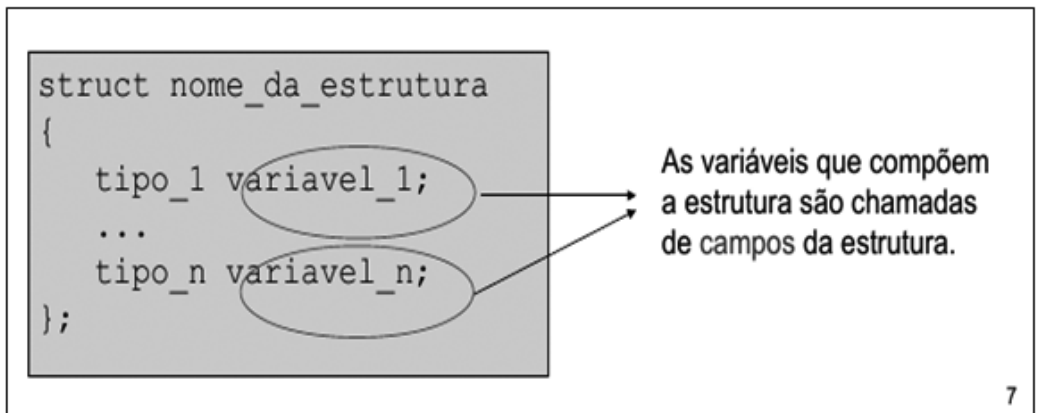
Para ser desenvolvida uma estrutura de dados, formada por vários tipos de variáveis é necessário que seja utilizado o comando *Struct*, utilizando essa palavra (comando) é possível declarar um novo tipo de dado. O comando *Struct* é definido por ser um tipo de dado, formado por números *Int* e *Float*, onde cada membro é declarado entre ([]), como também essa declaração é terminada em (;).

Quando é realizada a declaração de uma estrutura, os campos dessa estrutura não podem em situação alguma ser inicializados, pois para que isso aconteça precisa ser alocado espaço na memória do programa. Após realizar o procedimento de declarar um novo tipo de dado, podem ser criadas novas variáveis, que sirvam para armazenar valores e conjuntos de dados.

Um struct é um tipo de dado composto por um conjunto de outros tipos e dados. Struct possui um tamanho fixo. Os atributos de um struct são acessados usando a sintaxe <struct>.<nome do atributo> (também conhecido como notação de pontos). Atributos struct são armazenados na memória na mesma ordem em que aparecem no código. Você pode alinhar structs. Se utilizar typedef com um struct, não precisa dar um nome ao struct. (GRIFFITHS e GRIFFITHS, 2013, p. 235).

Como o objetivo da estrutura é agrupar vários tipos de variáveis em uma única variável, esse processo possibilita otimizar toda a estrutura de dados existente. Vamos analisar na figura a seguir o procedimento para se criar uma estrutura utilizando o comando *Struct*:

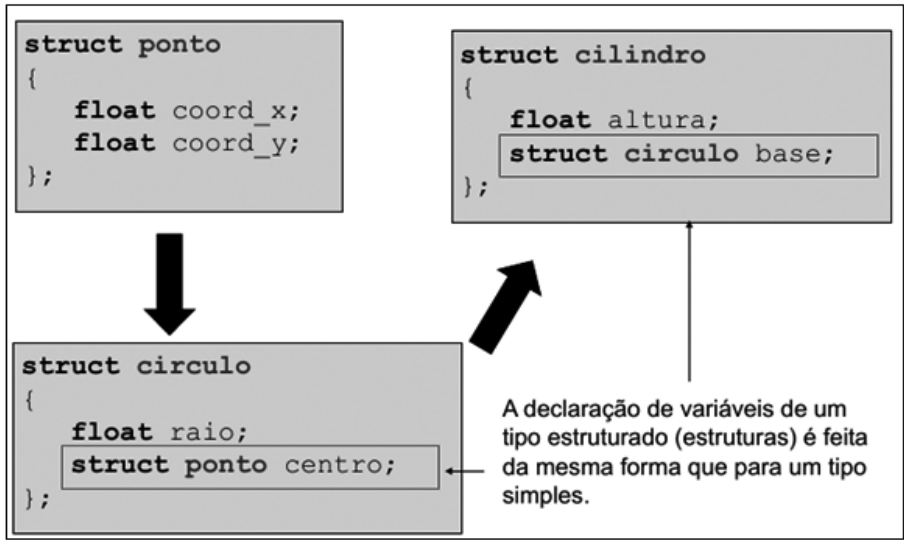
FIGURA 51 – COMANDO STRUCT



FONTE: Melo (2014)

Vamos analisar nas figuras a seguir como são definidos os novos tipos de dados:

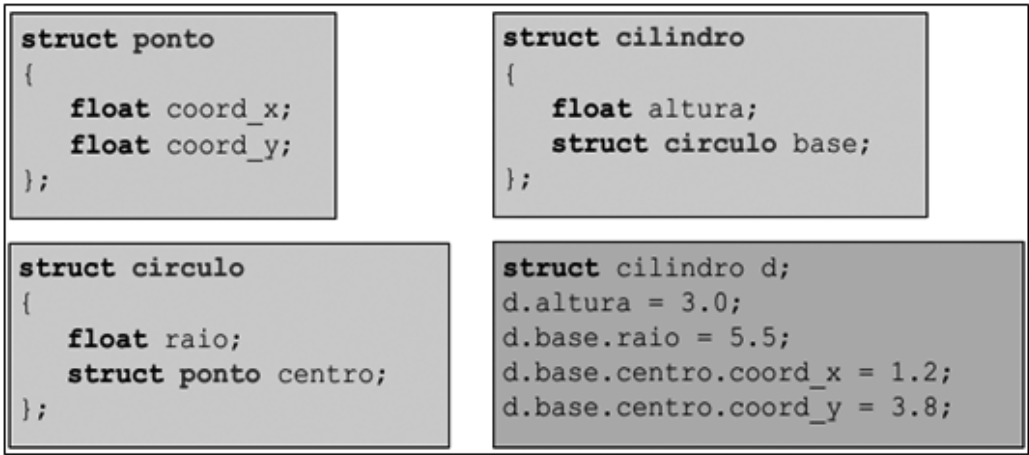
FIGURA 52 – DEFININDO OS NOVOS TIPOS DE DADOS



FONTE: Melo (2014)

Agora, vamos analisar o procedimento de como acessar os campos de uma estrutura de dados, para que isso ocorra existe a necessidade de adicionar o símbolo (.), para assim separar o nome da variável, conforme exemplo a seguir:

FIGURA 53 – ACESSANDO OS NOVOS TIPOS DE DADOS



FONTE: Melo (2014)

Para que um novo tipo de dado seja definido é preciso informar para o compilador do programa os seguintes dados: nome; tamanho do dado; procedimento de como deve ser armazenado e procedimento de como recuperar os dados na memória do programa. Com este processo de definir o novo tipo de dado, pode-se afirmar que este novo tipo agora existe e pode ser utilizado para criar variáveis que são denominadas como similares, como tipos de variáveis do tipo simples.

2.3.1 Comando typedef

Antes de iniciarmos a definição sobre o comando C, vamos analisar também a definição de uma declaração typedef. Segundo Cocian (2004, p. 202):

Uma declaração typedef é uma declaração que usa keyword typedef como classe de armazenamento. O declarador aparecerá como sendo um novo tipo de dado. Pode utilizar as declarações typedef para construir tipos com nome menores ou mais significativos, para tipos previamente definidos pela linguagem ou para novos tipos que se desejam criar. Os nomes typedef permitem encapsular detalhes de implementação que podem mudar. É importante notar, que uma declaração typedef não cria tipos. Ele cria sinônimos para tipos existentes, ou nomes para tipos que possam ser especificados de maneiras diferentes. Quando um nome typedef é utilizado como um especificador de tipo, ele pode ser combinado como um certo tipo de especificadores, mas não com todos.

Este comando denominado de typedef possui a finalidade de definir que um novo nome seja atribuído para um determinado tipo de dado, como mencionado, sua função não é produzir ou criar novos tipos de dados, mas sim definir novos nomes, sinônimos e até mesmo pode-se dizer apelidos para os tipos de dados já existentes e pré-definidos em uma estrutura. Geralmente podemos utilizar seu formato geral, conforme exemplo a seguir:

FIGURA 54 – COMANDO typedef

```
typedef nome_antigo nome_novo;
```

FONTE: A autora

Segundo Cocian (2004), o comando typedef permite ao programador definir um novo nome para um determinado tipo de dado. O comando typedef pode ser utilizado para dar nome a tipos complexos, como as estruturas.

2.4 VETORES DE ESTRUTURAS

Os vetores de estruturas podem ser definidos, ou até mesmo de forma mais específica, podemos afirmar que podem ser identificados como um vetor onde seus elementos apontam para dados do tipo struct, esses elementos também podem ser definidos como índices, pois os vetores, como já estudados anteriormente, são dados armazenados e organizados em uma sequência ou índice de dados. Voltando ao contexto dos vetores de estruturas, seu acesso ocorre através do índice. Este índice encontra-se entre colchetes, como após os colchetes pode ser encontrado o nome do vetor, posteriormente o mesmo é seguido pelo operador (.), como também o nome do campo da estrutura de dados.

Para iniciar o acesso ao índice, onde pode ser realizado com a delimitação do conteúdo do vetor, com a utilização do par de chaves { }, esse procedimento ocorre da mesma forma com o conteúdo que será atribuído para cada posição do vetor. Com relação aos valores do vetor, cada um dos membros precisa seguir e respeitar a ordem ou os índices que já estão declarados e separados por vírgula, dentro da estrutura de dados. Vamos analisar a seguir um exemplo de um vetor de estrutura:

```
struct aluno  
{  
    char nome [80];  
    int matricula;  
} faculdade [10];
```

2.5 VETORES DE PONTEIROS PARA ESTRUTURAS

É possível utilizar vetores de ponteiros para que seja possível tratar vários conjuntos de elementos e estes possuem como especificidade serem complexos, lembrando que vetores são considerados uma sequência de valores e estes valores são do mesmo tipo de dado, como são alocados e armazenados em uma sequência organizada na memória do programa.

Como já mencionado anteriormente os vetores de ponteiros para estrutura são muito úteis quando existe a necessidade de tratar conjuntos de elementos que são considerados complexos. Um exemplo complexo é uma tabela de alunos, onde:

```
Matricula: número inteiro ;  
Nome: cadeia com 80 caracteres ;  
Endereço: cadeia com 120 caracteres ;  
Telefone: cadeia com 20 caracteres ;  
m C podemos representar aluno:
```

FONTE: Garcia (2010)

Para representar esses conjuntos de elementos na linguagem C, o exemplo dessa tabela alunos, ficaria da seguinte forma:

FIGURA 55 – VETOR DE CADEIA DE CARACTERES

```

struct aluno {
    int mat ;
    char nome [81] ;
    char end [121] ;
    char tel [ 21 ] ; } ;
typedef struct aluno ALUNO ;

```

FONTE: Garcia (2010)

Lima (2014) sugere duas soluções para a tabela de alunos, na primeira:

FIGURA 56 – SOLUÇÃO 1 – TABELA ALUNO

```

#define MAX 100

struct aluno
{
    int mat;
    char nome[81];
    char end[121];
    char tel[21];
};
typedef struct aluno Aluno;

Aluno tab[MAX];

```

a estrutura ocupando pelo menos $4 + 81 + 121 + 21 = 227$ bytes

vetor de Aluno: representa um desperdício significativo de memória, se o número de alunos for bem inferior ao máximo estimado

FONTE: Lima (2014)

No segundo exemplo, podemos observar a seguinte solução:

FIGURA 57 – SOLUÇÃO 2 – TABELA ALUNO

```
#define MAX 100

struct aluno
{
    int mat;
    char nome[81];
    char end[121];
    char tel[21];
};
typedef struct aluno Aluno;

Aluno* tab[MAX]; ←
```

Vetor de ponteiros para Aluno:

- um elemento do vetor ocupa espaço de um ponteiro;
- alocação dos dados de um aluno no vetor:
 - nova cópia da estrutura Aluno é alocada dinamicamente
 - endereço da cópia é armazenada no vetor de ponteiros
- posição vazia do vetor: valor é o ponteiro nulo

FONTE: Lima (2014)

Esses exemplos apresentam a necessidade de definir o número máximo de alunos para o vetor, esse procedimento auxilia para se saber e até mesmo para alocar o espaço necessário na memória desses dados no vetor, para que a memória seja utilizada de forma dinâmica e eficiente.

2.5.1 Funções de vetores de ponteiros para estruturas

Podemos afirmar que os vetores de ponteiros para estruturas possuem várias funções, além de alocar memória para um vetor, vamos apresentar a seguir algumas funções que fazem parte dos vetores de ponteiros, como:

- Função Inicializa.
- Função Preenche.
- Função Retira.
- Função Imprime.

Vamos apresentar primeiramente a **Função Inicializa**. Ela possui a função de iniciar a tabela, como no exemplo anterior, tabela de alunos. Esta função irá realizar os seguintes procedimentos para executar a função inicializar, que recebe um vetor de ponteiros. Outra funcionalidade é atribuir um valor Null para todos os elementos da tabela. Vamos analisar o exemplo.

FIGURA 58 – FUNÇÃO INICIALIZA

```
void inicializa(int n, Aluno** tab)
{
    int i;
    for (i=0; i<n; i++)
        tab[i] = NULL;
}
```

FONTE: Lima (2014)

A **Função Preencher** possui a principal funcionalidade de iniciar a tabela, como já foi dito, podemos citar como exemplo a tabela de alunos. Esta função ocorrerá através do recebimento da posição do local em que os dados serão armazenados, a função irá analisar se a posição na tabela está vazia, a função irá criar a alocação de uma nova estrutura, caso contrário, a própria função atualiza a estrutura que está apontada pelo ponteiro. Vamos analisar o exemplo a seguir, com relação ao funcionamento desta função:

FIGURA 59 – FUNÇÃO PREENCHE

```
void preenche(int n, Aluno** tab, int i)
{
    if (i < 0 || i >= n)
    {
        printf("Indice fora do limite do vetor\n");
        exit(1);      /* aborta o programa */
    }
    if (tab[i] == NULL)
        tab[i] = (Aluno*)malloc(sizeof(Aluno));
    printf("Entre com a matricula:");
    scanf("%d", &tab[i]->mat);
    ...
}
```

FONTE: Lima (2014)

Na **Função Retira**, a principal funcionalidade é remover os dados dos vetores, como no exemplo anterior, falamos da tabela de alunos, esta função irá realizar o procedimento de preencher, como também armazenar dados de um novo aluno na tabela.

FIGURA 60 – FUNÇÃO RETIRA

```

void retira(int n, Aluno** tab, int i)
{
    if (i<0 || i>=n)
    {
        printf("Indice fora do limite do vetor\n");
        exit(1);      /* aborta o programa */
    }
    if (tab[i] != NULL)
    {
        free(tab[i]);
        tab[i] = NULL; /* indica que na posição não mais existe dado */
    }
}

```

FONTE: Lima (2014)

E por último, mas não menos importante, a **Função Imprime** possui como principal função imprimir os dados de um aluno que está alocado na tabela, onde a função recebe qual é a posição do aluno na tabela e depois realiza a impressão dos dados desse aluno. A seguir vamos analisar o exemplo desta função:

FIGURA 61 – FUNÇÃO IMPRIME

```

void imprime(int n, Aluno** tab, int i)
{
    if (i<0 || i>=n)
    {
        printf("Indice fora do limite do vetor\n");
        exit(1);      /* aborta o programa */
    }
    if (tab[i] != NULL)
    {
        printf("Matrícula: %d\n", tab[i]->mat);
        printf("Nome: %s\n", tab[i]->nome);
        printf("Endereço: %s\n", tab[i]->end);
        printf("Telefone: %s\n", tab[i]->tel);
    }
}

```

FONTE: Lima (2014)

A **função Imprimi**, também, possui a disponibilidade de realizar a impressão de toda a tabela, com a **Função Imprime Tudo**, em que a função recebe o tamanho dos dados existentes na tabela e realiza a impressão de todos os alunos e seus dados.

FIGURA 62 – FUNÇÃO IMPRIME TUDO

```

void imprime_tudo(int n, Aluno** tab)
{
    int i;
    for (i=0; i<n; i++)
    {
        imprime(n, tab, i);
    }
}

```

FONTE: Lima (2014)

2.6 TIPO UNIÃO

Um tipo estruturado de união possui como funcionalidade armazenar valores que são considerados como valores heterogêneos, realiza o armazenamento desses valores em um único espaço na memória do programa. Conforme Aguilar (2011, p. 302), “o tipo de dado união é similar a *struct*, de maneira que é uma coleção de membros dados de tipos diferentes ou semelhantes”. Uma observação deve ser levada em consideração quando for realizar o armazenamento, pois apenas um único elemento de união pode estar armazenado na memória, pois cada vez que um processo de atribuição é realizado para um campo da união, este por sua vez subscreve os valores que já estavam armazenados anteriormente e atribuídos para os campos já armazenados. A definição de um tipo de estrutura de união pode ser dada através do seguinte comando:

```

union exemplo {
    int i;
    char c;
}

```

A declaração de uma variável do tipo de estrutura União pode ser declarada com o seguinte comando:

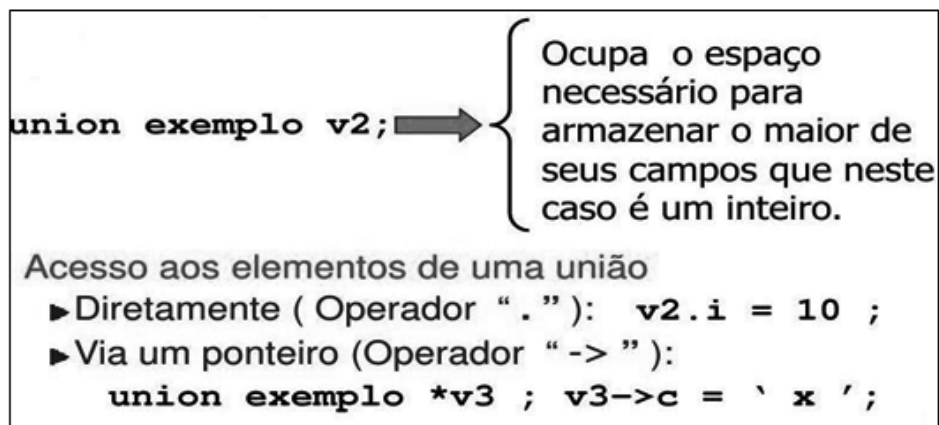
```

union exemplo
v ;

```

Observando que esta estrutura de código não possui a função de declarar as variáveis da estrutura e dados, mas sim possui a função de definir o tipo de estrutura de união.

FIGURA 63 – FUNÇÃO UNIÃO



FONTE: Garcia (2010)

Segundo Aguilar (2011, p. 298), “as uniões declaradas com a palavra reservada *union* armazenam também membros múltiplos em um pacote; não obstante, em vez de localizar seus membros um atrás do outro em uma união”.

2.7 TIPO ENUMERAÇÃO

O tipo estruturado de enumeração possui como finalidade especificar os valores do qual uma variável do seu tipo pode possuir, é formado por um conjunto de constantes inteiras, com nomes que identificam as variáveis. Segundo Cocian (2004, p. 385), “em uma enumeração pode-se indicar ao compilador quais os valores que uma determinada variável pode assumir”. O tipo estrutura de enumeração é um procedimento que organiza todos os valores de uma constante. Quando são criados valores do tipo *booleano*, as variáveis deste tipo podem receber os valores da seguinte forma: 0 (*FALSE*) ou 1 (*TRUE*). Vamos analisar a seguir um exemplo de duas constantes que estão simbolicamente dissociadas e utilizam um valor inteiro, em que o mesmo poderá representar o tipo *booleano*, com os valores *FALSE* e *TRUE*.

```
#define FALSE 0
#define TRUE 1
typedef int Bool;
/* pode armazenar qualquer inteiro*/
```

Com relação ao exemplo anterior pode ser percebido que com as definições de *FALSE* e *TRUE*, é possível utilizar esses símbolos no código do programa, que tornam mais claro o processo de armazenar os números inteiros, observando que o tipo *booleano* pode ser equivalente a qualquer número, não apenas *FALSE* e *TRUE*.

Vamos analisar o exemplo a seguir, com relação à criação de um tipo *booleano*, utilizando o tipo estrutura de enumeração:

FIGURA 64 – TIPO *BOOLEANO*

```
enum    bool {
    FALSE, /*O valor 0 é atribuído a FALSE */
    TRUE  /*O valor 1 é atribuído a TRUE */
} ;
typedef enum bool    BOOL ;
```

FONTE: Garcia (2010)

O exemplo anterior representa o comando de criação do tipo *booleano*, usando o tipo estrutura de enumeração, em que os valores são atribuídos, seguindo uma ordem em que os elementos foram definidos para o tipo enumeração.

LEITURA COMPLEMENTAR

Garantindo o limite dos vetores

Extrapolar os limites de um vetor é causa de inúmeros erros.

Recomendação 1 Declarar vetores com tamanho fixo, usando macros para definir o tamanho.

```
#define TAM (31);
long int vetor[TAM];
```

Razão. Favorece a consistência do programa. Quando o tamanho do vetor for alterado, basta modificar a definição da macro para que o resto do código fique adaptado ao novo tamanho.

Cadeias de caracteres. Um tipo especial de vetor, as cadeias de caracteres devem ser terminadas com o caractere nulo. A maior parte das funções usa esse caractere para determinar o tamanho da cadeia e não invadir outras áreas da memória. Uma simples impressão, com o comando `printf`, de uma cadeia não terminada com ‘\0’ pode gerar resultados imprevisíveis.

Recomendação 2 Evitar o uso de funções que não permitem determinar o tamanho da cadeia, como, por exemplo, `gets`. Deve-se optar pelas funções que permitem determinar um limite para os caracteres lidos, assegurando-se que o vetor onde eles serão armazenados possui um tamanho apropriado, como, por exemplo, `fgets`.

Razão. Com essas funções não ocorre acesso indevido à memória, caso a entrada contenha um número maior de caracteres que o vetor utilizado para armazená-los.

Recomendação 3 Criar cadeias delimitadas como caractere nulo. Por exemplo, nas operações de leitura, deve-se usar a função `fgets` ou diretivas do tipo `%30s` e `%30[^\n]`, que incluem o caractere nulo automaticamente, evitando-se diretivas como `%30c`.

Razão. As cadeias que não possuem o caractere nulo ao final não podem ser usadas nas funções da biblioteca-padrão que assumem a existência desse caractere.

RESUMO DO TÓPICO 2

Neste tópico vimos:

- Os tipos de estruturas de dados possuem a responsabilidade de estruturar o processo de programação de aplicações, para que essas estruturas contenham um domínio dos dados que necessitam ser tratados, identificados, declarados e armazenados.
- O tipo de estrutura é formado por vários tipos de valores, sua estrutura é definida como um tipo simples, com isso objetiva otimizar e dinamizar espaços na memória de um programa de computador. Ao realizar a codificação de dados, a estrutura torna compreensível o desenvolvimento de programas que são considerados mais complexos.
- Os elementos do tipo estrutura de dados são denominados variáveis do tipo heterogênea, as variáveis do tipo heterogênea possuem seus elementos de tipo diferente. As variáveis do tipo homogênea são consideradas como um tipo estruturado e ao contrário do anterior, seus elementos podem ser definidos como elementos do mesmo tipo. Os elementos como os vetores e matrizes são do tipo homogênea.
- Os arranjos unidimensionais são utilizados para armazenar conjuntos de dados, em que esses dados são identificados como elementos e estão diretamente endereçados para um único índice de elementos. Os arranjos multidimensionais possuem como objetivo armazenar também vários tipos de conjuntos de dados, no entanto devem estar endereçados para mais de um índice de elementos.
- Um operador é identificado pelo elemento (—). Possui como objetivo realizar a operação de um tipo de resultado, enquanto não se conhece o tipo do operador, não se pode conhecer o significado da expressão.
- Vetores de ponteiros para estruturas são utilizados para tratar vários tipos de conjunto de elementos, principalmente conjunto de elementos que possuem sua estrutura muito complexa de ser especificada.
- Funções de vetores de ponteiros para estruturas possuem como funcionalidade alocar memória de um vetor para uma estrutura de ponteiros, realiza outros processos como: função inicializa, função preenche, função retira, função imprime.

- Tipo de estrutura de união realiza o armazenamento de valores e estes são considerados como valores heterogêneos, o armazenamento dos valores são alocados em um único espaço na memória do programa. Tipo estruturado de enumeração realiza a especificação dos valores, que uma variável do tipo deve obter, é formado por um conjunto de constantes inteiras, são identificados com nomes nas variáveis, organizam a enumeração para organizar os valores de uma constante.



- 1 Estruturas de dados possuem como finalidade realizar a manipulação de estruturas de dados e suas aplicações, pois realiza o processo de organizar os dados para o desenvolvimento de uma aplicação. Segundo Pereira (1996), as estruturas de dados possuem dois objetivos principais, quais são?

 - () Arranjos e Operador.
 - () Teórico e Prático.
 - () União e Enumeração.
 - () Ponteiros e Vetores.

- 2 Os tipos de dados definidos pelo usuário são tipos de dados estruturados, são construídos hierarquicamente através de componentes, os quais são de fato tipos de dados primitivos ou tipo de dados estruturados. Segundo Edelweiss e Galante (2009) existem dois tipos de dados estruturados. Assinale a alternativa que corresponde a esses dados:

 - () Tipo de dados Inteiro e Definidos por Funções.
 - () Tipo de dados Vetores e Definido pelo Arranjo.
 - () Tipo de dados Básico e Definido pelo Usuário.
 - () Tipo de dado Booleano e Definido por Variáveis.

MATRIZES

1 INTRODUÇÃO

Uma matriz é definida por ser uma estrutura de dados que possui a funcionalidade de armazenar vários conjuntos de elementos. O armazenamento, como o acesso aos dados, é realizado de forma organizada e sequencial. Os elementos podem ser acessados seguindo um índice. O índice é organizado através das linhas e colunas pertencentes à matriz e sua estrutura de dados. As matrizes podem ser entendidas como matrizes dimensionais, bidimensionais e multidimensionais, são formadas por várias dimensões.

Neste tópico de estudos veremos as matrizes. Serão abordados os seguintes conteúdos que definem a estrutura e os tipos de matrizes: Matrizes Especiais; Vetores bidimensionais – Matrizes; Matrizes dinâmicas; Representação de matrizes e Representação de matrizes simétricas.

Os tipos abstratos de dados possuem como finalidade agrupar toda a estrutura de dados, isso em conjunto com as operações, consequentemente, realiza o processo de encapsular a estrutura de dados, toda a implementação da estrutura é específica dos tipos abstratos de dados. O processo de abstração dos dados ocorre através das classes *structs*, objetos, encapsulamentos, como por exemplo, listas, filas, árvores, entre outros.

Seguindo com os estudos dessa unidade de estudos, daremos ênfase também à descrição dos conceitos dos tipos abstratos de dados, utilizando: os tipos de dados abstratos; ponto e tipo de dado abstrato; matriz.

2 MATRIZES

As matrizes são consideradas arranjos ordenados, sua definição também específica que podem ser formadas por n dimensões, observando que essas dimensões são denominadas de dimensional. As matrizes são formadas por várias dimensões, elas são chamadas de bidimensional, quando a matriz possui duas dimensões, chama-se tridimensional, quando é formada por três dimensões e assim consequentemente. Conforme Laureano (2008, p. 5), a matriz é uma estrutura de dados que necessita de um índice para referenciar a linha e outro para referenciar a coluna para que seus elementos sejam endereçados.

A matriz possui algumas funcionalidades de um vetor, como por exemplo, realizar a declaração de variáveis em um arranjo bidimensional, organizando esses arranjos em linhas e colunas, dentro dessas linhas e colunas são criadas várias células, e estas possuem a capacidade de armazenar dados do mesmo tipo.

Uma matriz é um arranjo bidimensional ou multidimensional de alocação estática e sequencial. Todos os valores da matriz são do mesmo tipo e tamanho, e a posição de cada elemento é dada pelos índices, um para cada dimensão. Os elementos ocupam posições contíguas na memória. A alocação dos elementos da matriz na memória pode ser feita colocando os elementos linha-por-linha ou coluna-por-coluna. (LOREZI, MATTOS e CARVALHO 2007, p. 8).

Como dito anteriormente, uma matriz segue muitos parâmetros dos vetores, como organizar uma sequência de células em uma estrutura, a matriz é definida por ser uma variável composta de dados homogêneos bidimensionais, com dados do mesmo tipo, com nome do mesmo identificador da variável, e realiza o processo de alocar de forma sequencial espaço na memória do programa. A matriz utiliza identificadores de valores para mapear as linhas e colunas, esse procedimento é feito através do índice, que está armazenado e alocado de forma organizada e em sequência, dentro da sua estrutura de dados. A matriz realiza o procedimento de mapear os valores utilizando as linhas e colunas, usando o indicador entre ([] colchetes), para que seja possível identificar esses valores.

Uma das características de declaração de uma matriz é que ela precisa realizar alguns processos como, informar o tipo de dados que será armazenado, informar a quantidade de células, linhas e colunas que serão disponibilizadas para o arranjo. A matriz possui dimensões de linhas e colunas que devem ser especificadas e ter uma delimitação, isso pode ser feito utilizando o delimitador [], após o nome referido da matriz, são reconhecidos como tipos de dados para a matriz os seguintes valores: int, float, double, char, bool, string. Para realizar a declaração de uma matriz pode ser utilizada a seguinte sintaxe:

```
tipo_de_dados nome_da_variavel [quantidade_de_linhas][quantidade_de_colunas];
```

A matriz também é conhecida como Arrays, possui a mesma definição já mencionada anteriormente de que é formada por uma coleção de dados que estão armazenados na memória, estes dados podem ser acessados, seja em conjunto ou de forma individual, isso é definido de acordo com a posição espacial dentro da estrutura, que foram dimensionadas e alocadas para a matriz. Vamos analisar a seguir uma matriz com 1 dimensão, contendo no máximo 15 elementos:

FIGURA 62 – MATRIZ DIMENSIONAL

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

FONTE: Cenapad-SP (2014)

Como foi mencionado anteriormente, a matriz é composta por dois tipos, matriz unidimensional e matriz multidimensional. Sobre a matriz unidimensional podemos afirmar que é o modelo da figura acima, composta apenas por uma única dimensão, em que os dados e números são organizados de forma contínua. Conforme Aguilar (2011, p. 237), “são denominados arrays (matriz) unidimensionais e nelas cada elemento é definido ou é feita referência por um índice ou subíndice”. Esses vetores são elementos de dados escritos em uma sequência.

A matriz multidimensional é formada por duas ou mais dimensões, como por exemplo uma tabela contendo linhas e colunas. Aguilar (2011, p. 240) apresenta um array (matriz), que pode ser definido em três dimensões, quatro e até em n dimensões. Em geral, um array de n dimensões requer que os valores dos n subíndices possam ser especificados para se identificar um elemento individual do array.

“Java não suporta criar arrays unidimensionais cujos elementos também sejam arrays unidimensionais, por isso arrays de arrays. Eles costumam ser usados para representar tabelas de valores, que consistem em informações dispostas em linhas e colunas”. (SERSON 2007, p. 166), Nos dois formatos, a matriz se comporta da mesma forma em que os dados são armazenados e consultados de maneira contínua na memória do programa. Vamos analisar a seguir um exemplo de uma matriz multidimensional:

FIGURA 65 – MATRIZ MULTIDIMENSIONAL

		Colunas				
		0	1	2	3	4
Linhas	0	65	10	39	34	20
	1	55	69	29	33	65
	2	43	34	77	40	35
	3	91	24	40	97	11
	4	88	61	21	70	100

25 Espaços

FONTE: Disponível em: <http://cae.ucb.br/conteudo/programar/labor1/new_matriz.html>. Acesso em: 3 out. 2014

2.1 MATRIZES ESPECIAIS

Podemos considerar que uma matriz é especial quando na sua maioria seus elementos são iguais a zero ou considerados um valor constante. Atenção muito especial se dá neste modelo de matriz, pois existe um desperdício de espaço na alocação da memória, devido aos elementos possuírem o mesmo valor. Para que esse processo seja evitado, existe a necessidade de serem apresentados apenas valores que são considerados representativos e significativos para a matriz, como valores que sejam diferentes de zero e valores constantes.

VELOSO; SANTOS; AZEREDO e FURTADO (1983) apresentam as matrizes especiais como representações que guardam todos os elementos da matriz. Frequentemente ocorrem matrizes de números representando uma boa parte de elementos nulos. Com isso os espaços que podem ser inutilizados podem diminuir, para que todos os espaços tenham utilização e tornem o processo mais dinâmico.

Compõem as matrizes especiais as matrizes diagonais, matrizes triangulares, matrizes simétricas e antissimétricas e cada uma possui uma finalidade dentro das matrizes especiais. Como, por exemplo, as matrizes diagonais são formadas por valores significativos da diagonal principal ou secundária. As matrizes triangulares possuem a finalidade de armazenar valores

significativos em um vetor e possuem valores diferentes de constantes e zero. As matrizes simétricas e antissimétricas utilizam a matriz para representar apenas os elementos do triângulo superior e inferior.

2.2 VETORES BIDIMENSIONAIS – MATRIZES

Segundo Horstmann (2008, p. 341), “vetores e arrays podem armazenar sequências lineares de números”. Acontece com frequência querermos armazenar coleções de números que tem um *layout* bidimensional. Os vetores bidimensionais são considerados como matrizes de um vetor bidimensional realizam a função de armazenar informações de forma organizada, através de linhas e colunas, lembrando que uma matriz é uma estrutura de dados lógicos e organiza um conjunto de valores na estrutura formados por linhas e colunas, em que a organização se dá através de um índice, o acesso sempre se dará pelo índice que indicará primeiramente a linha e como segundo acesso a coluna dos vetores bidimensionais. Vamos analisar a seguir uma matriz bidimensional, contendo uma tabela com linhas e colunas, em que estes possuem os pesos [3] [5], a matriz pode ser pensada da seguinte forma:

TABELA 8: TABELA DE VETORES BIDIMENSIONAIS – LINHAS E COLUNAS

FONTE: Adaptado de: Neto (2014)

Observe a tabela anterior, onde o peso [3], representa as linhas da tabela e o segundo peso está representando [5] as colunas pertencentes da tabela. Analisando que o peso [3] é composto por valores de zero até o número 2 e o peso [5] é composto por valores de zero ao número 4, sabendo a composição e as limitações desses dois pesos, podemos a partir dessa premissa determinar os índices de cada posição dos valores na matriz. No exemplo a seguir vamos representar a tabela contendo os valores:

TABELA 9: TABELA DE LINHAS E COLUNAS E OS PESOS

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

FONTE: Adaptado de: Neto (2014)

Perceba que na tabela o que determina os valores são os números da primeira coluna, depois elas continuam a sequência organizada dos valores. Na próxima tabela vamos analisar a posição de cada índice e vamos dar valores para os pesos, como por exemplo:

TABELA 10: VALORES E PESOS

10	30	45	70	36
86	44	63	82	80
70	61	52	63	74

FONTE: Adaptado de: Neto (2014)

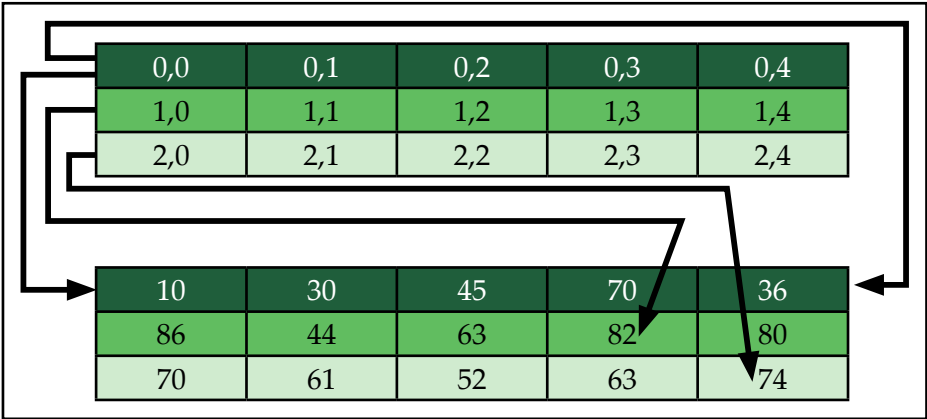
Agora que temos os valores para cada peso e para os índices, seguindo a metodologia dos vetores bidimensionais, podemos analisar a tabela anterior da seguinte forma:

TABELA 11: VALORES E PESOS DE VETORES BIDIMENSIONAIS.

```
pesos[1][3] = 82
pesos[0][4] = 36
pesos[0][0] = 10
Peso [2][4] = 74
```

FONTE: Adaptado de: Neto (2014)

FIGURA 66 – RESULTADO DOS VALORES E PESOS



FONTE: A autora

Vamos analisar a seguir como pode ser representado o comando da matriz bidimensionais desses valores e pesos, que estão expostos no exemplo da figura 63.

Para preencher nossa matriz com os valores mostrados na tabela anterior podemos usar uma declaração como:

```
int pesos [3] [5] = {{10, 30, 45, 70, 36},
                    {86, 44, 63, 82, 80},
                    {70, 61, 52, 63, 74}};
```

Podemos manipular os elementos de nossa matriz bidimensional usando duas variáveis e um laço sendo da mesma maneira que fizemos com as matrizes comuns. Observe o código a seguir:

```
/* manipulando uma matriz bidimensional */
#include <stdio.h>

int main()
{
    int pesos [3] [5] = {{10, 30, 45, 70, 36},
                        {86, 44, 63, 82, 80},
                        {70, 61, 52, 63, 74}};

    int linha,coluna;

    for (linha = 0; linha < 3; linha++)
        for (coluna = 0; coluna < 5; coluna++)
            printf ("elemento [%d] [%d] = %d\n", linha, coluna, pesos [linha]
[coluna]);
    return (0);
}
```

FONTE: Neto (2014)

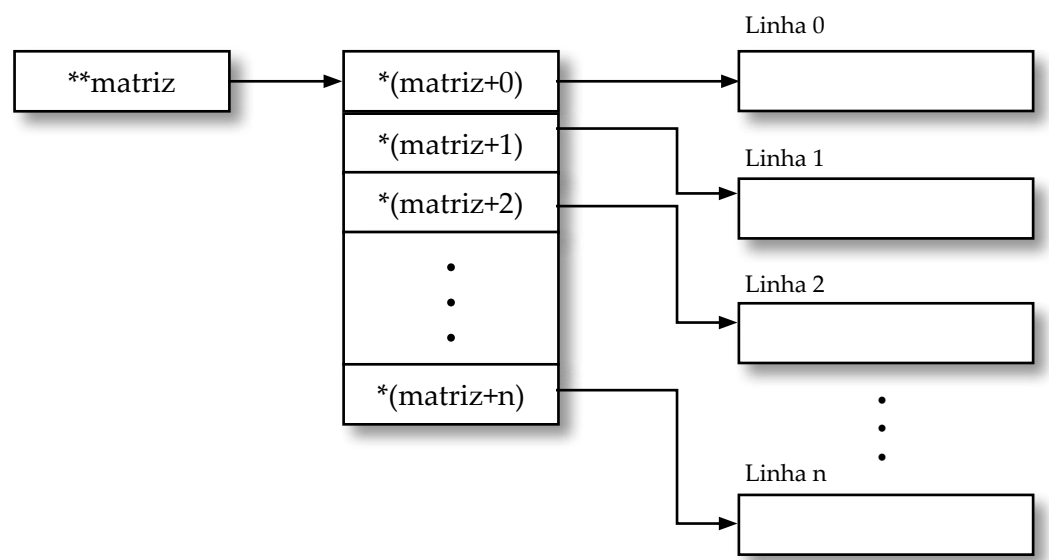
Feijó, Silva e Clua (2010) colocam que a manipulação de matrizes bidimensionais exige o controle cuidadoso das variáveis de índices, para que seja possível apontar exatamente para cada valor desejado.

2.3 MATRIZES DINÂMICAS

As matrizes dinâmicas exigem que sejam criadas abstrações conceituais com vetores e assim representar de forma bidimensional os conjuntos dessa estrutura e representar quais matrizes estão alocadas dinamicamente. A matriz dinâmica é representada por vetores do tipo simples, também é representada por vetores do tipo unidimensional. A matriz dinâmica possui como finalidade armazenar elementos do vetor na primeira linha da matriz. Cria abstrações conceituais com os vetores, para que possam ser representadas as matrizes que estão alocadas dinamicamente. Rita (2009, p. 62), coloca que “uma matriz poderá ser formada de uma única linha (unidimensional) ou com mais de uma linha e colunas; nesse caso, será tridimensional”.

A matriz dinâmica possui seu conjunto bidimensional que é representado por um vetor unidimensional. Para realizar o acesso aos seus elementos precisa-se acessar utilizando algumas regras e procedimentos. Com relação ao acesso dos elementos que estão armazenados isso ocorre em uma sequência contínua. O primeiro acesso é na primeira linha, depois o acesso aos elementos da segunda linha e assim consequentemente. Ainda segunda Rita (2009, p. 62), “há matrizes retangulares e quadradas. Quando o número de linhas é diferente do número de colunas, uma matriz é considerada como uma matriz retangular. Caso o número de linhas e colunas seja idêntico, a matriz é quadrada”.

FIGURA 67 – MATRIZ DINÂMICA



FONTE: Cruz (1999)

A matriz dinâmica possui algumas limitações que podem ser observadas como, sua alocação é estática, suas dimensões devem ser analisadas antes de utilizar, armazenar e acessar seus elementos, a alocação é realizada diretamente na matriz, para que seja alocada de forma dinâmica se dará apenas para

conjuntos de elementos unidimensionais, a representação das matrizes alocadas dinamicamente ocorre através de abstrações conceituais e vetores que realizam o papel de alocação. Rita (2009, p. 63) afirma que “podemos declarar uma matriz com tamanho fixo ou uma onde seu tamanho possa ser alterado durante a execução do programa, nesse caso, ela será considerada uma matriz dinâmica”.

A declaração de uma matriz dinâmica pode ser feita de modo em que as dimensões da matriz possam sofrer alterações, elas são aumentadas ou diminuídas, isso pode ocorrer ainda enquanto a macro da matriz for executada. Um problema que pode ocorrer quando estiver realizando a declaração da matriz dinâmica, onde os conteúdos podem ser pedidos ao redimensionar a matriz. Para realizar a declaração de uma matriz dinâmica, pode ser utilizado o seguinte comando:

```
dim < array name > ( ) as < data type >
```

Para criar uma matriz dinâmica com tamanho variável, de forma estática, pode-se utilizar **n** dimensões de forma simples, vamos analisar a seguir um exemplo de como dimensionar essa matriz, será utilizado o seguinte comando:

```
int array [ 5 ] [ 10 ] [ 20 ];
```

Para que a matriz se torne dinâmica, é preciso criar um ponteiro para esse tipo de dados que vai ser utilizado na matriz, nesse dado deve conter a mesma quantidade de indireções que as dimensões da matriz, para isso vamos utilizar uma matriz bidimensional de valores inteiros e posterior a isso deve-se criar um ponteiro que esteja apontado para outro ponteiro. Vamos analisar o exemplo a seguir, de como seria a criação dessa matriz dinâmica:

```
int **array;
```

Posterior a isso precisa-se alocar os ponteiros das linhas e depois alocar cada linha da matriz. Vamos analisar o exemplo a seguir de como seriam os comandos para alocar os ponteiros das linhas:

```
array = (int **)malloc(sizeof(int *) * 10);
for(c = 0; c < 10; c++)
    array[c] = (int *)malloc(sizeof(int) * 20);
```

Também existe o processo contrário. Para realizar o deslocamento de um ponteiro, os comandos serão feitos de forma contrária. Primeiro deve-se desalocar as linhas da matriz e em sequência desalocar os ponteiros de linhas. Para que isso possa ocorrer, vamos utilizar o seguinte comando como exemplo:

```
for(c = 0; c < 10; c++)  
    if(array[c])  
        free(array[c]);  
if(array)  
    free(array);
```

2.4 REPRESENTAÇÃO DE MATRIZES

Para realizar a representação de matrizes precisa-se levar em consideração alguns aspectos, como procedimentos e funções que possam ser realizadas para representar as tarefas das matrizes. Essas operações computacionais devem seguir as seguintes funções, como: realizar a leitura de uma matriz, tanto as dimensões como todos os dados da matriz; desenvolver uma matriz que seja identificada como a identidade matriz; realizar a comparação de duas matrizes; fazer a somatória dessas duas matrizes, fazer a multiplicação dessas duas matrizes e para finalizar deve-se imprimir uma dessas matrizes. Esses são alguns pontos que devem ser levados em consideração para realizar o procedimento de representar uma matriz, sempre lembrando que uma matriz é formada por linhas e colunas e podem ser do tipo quadrada e de tipo de identidade.

Uma função muito interessante da representação da matriz é que é possível utilizar as linhas e colunas da matriz para associar os vértices, como também é possível utilizar os elementos da matriz para identificar se existem na estrutura da matriz arestas, algum problema com algum ângulo ou algo do tipo. Sempre se deve observar que as matrizes servem para representar dados das dimensões da matriz e como essas dimensões estão ordenadas e organizadas, pode ser utilizada para vários tipos de operações, seja para cálculos de adição, multiplicação e inversão.

Para representar um tipo de matriz, pode ser escolhido um conjunto de operações que pode ser implementado sobre o tipo de matriz escolhido, utilizando esses cálculos anteriormente citados, como adição, multiplicação e a inversão das matrizes. Para que isso ocorra vamos utilizar a implementação das seguintes operações básicas da matriz, como:

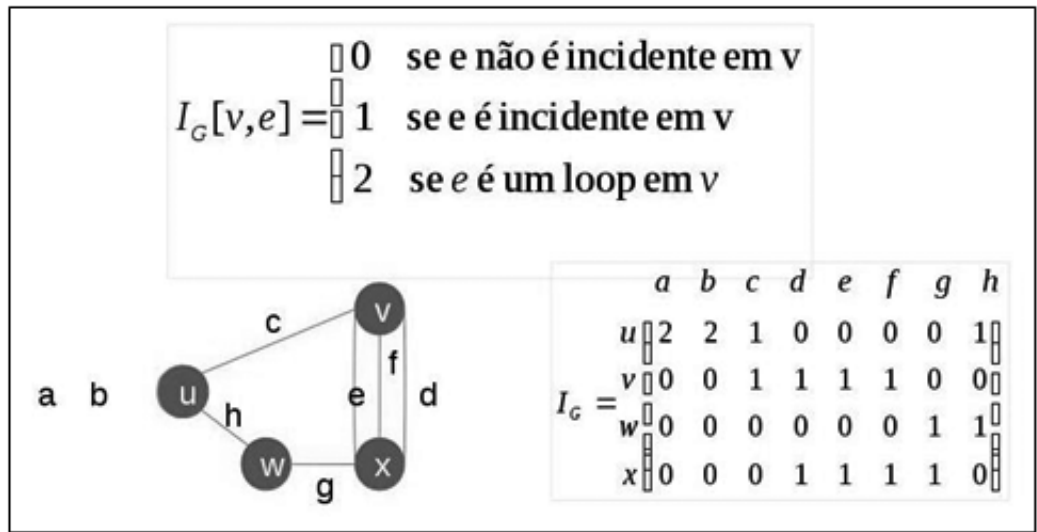
TABELA 12: OPERAÇÕES BÁSICAS DA MATRIZ

OPERAÇÕES	RESULTADO
CRIA	operação que cria uma matriz de dimensão m por n;
LIBERA	operação que libera a memória alocada para a matriz;
ACESSA	operação que acessa o elemento da linha i e da coluna j da matriz;
ATRIBUI	operação que atribui o elemento da linha i e da coluna j da matriz.

FONTE: Adaptado de: Celes e Rangel (2008)

Lopes (1997) apresenta que a matriz de adjacência e a lista de adjacência são duas formas encontradas para a representação de grafos dirigidos. As representações de matrizes podem ser representadas por matriz adjacência e pela matriz de incidência. A matriz de incidência pode ser definida como uma representação de um grafo G , esta é uma matriz $m \times n$, em que as linhas e colunas podem ser indexadas, seguindo a ordenação dos vértices e das arestas do grafo G . Vamos analisar a seguir a figura que representa esses dados da matriz de incidência.

FIGURA 68 – MATRIZ DE INCIDÊNCIA

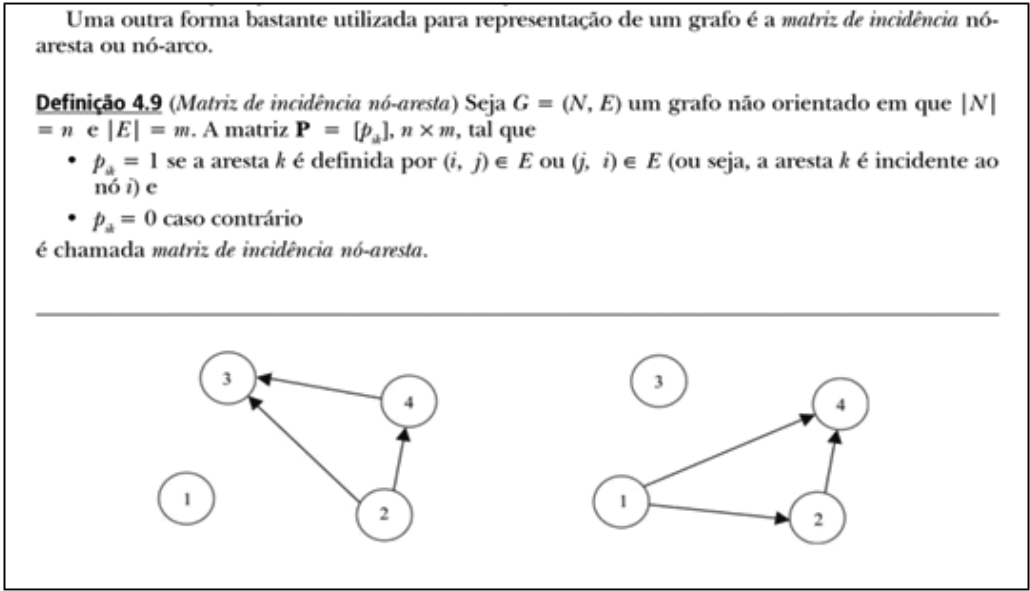


FONTE: Fontes (2009)

Para representar um grafo por uma matriz de adjacência, utiliza-se uma matriz quadrada booleana $M(1..N, 1..N)$, onde N indica o número de vértices do grafo a ser representado, sendo preenchido com valores verdade (v) sempre que um vértice j é adjacente a um vértice i , i e j variando de 1 a n , com notação $M(i, j) = v$. $M(i, j) = \text{falso (f)}$ quando um vértice j não é adjacente a um vértice i . Para matrizes onde N é grande, essa representação deve ser evitada, devido ao desperdício de memória. (LOPES 1997, p. 429).

A lista da **matriz de incidência** é formada por uma tabela, em que sua funcionalidade é listar todos os vértices V , como também listar todas as arestas incidentes em V . Essa matriz de incidência geralmente é composta, em sua maior parte, por valores em zero. A matriz de incidência possui como vantagem a possibilidade de representar todos os grafos gerais da matriz. Também pode ser construída uma representação mais eficiente em relação ao uso eficiente do espaço da memória. A matriz incidência está em desvantagem nas representações, pois suas funcionalidades precisam rotular todas as arestas existentes do grafo. Perceba:

FIGURA 69 – MATRIZ DE INCIDÊNCIA



FONTE: Arenales; et al. (2011)

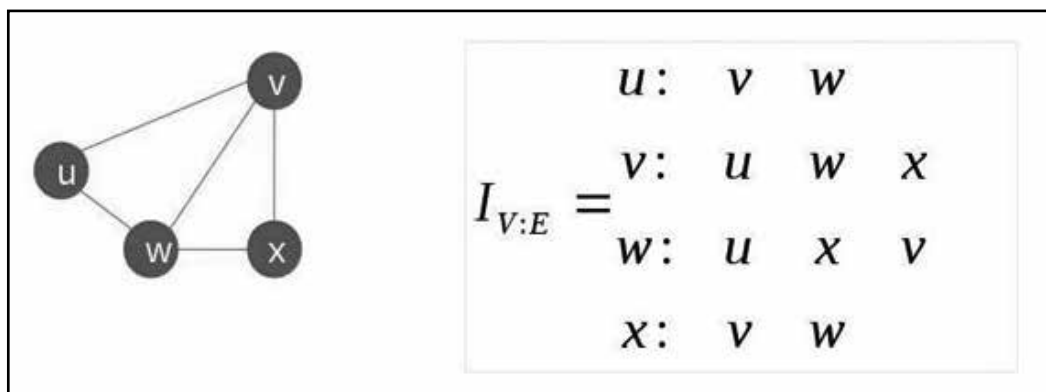
A **matriz adjacência** possui a finalidade de representar através de grafos simples suas listas, com isso as representações se tornam mais eficientes, é constituída em sua grande parte por suas adjacências por valores zero. A matriz adjacência de um grafo $G = (V, A)$, formada por n vértices, como se pode afirmar que é uma matriz $n \times n$ contendo n bits, em que $A[i, j]$ é 1 (ou verdadeiro) possui um arco do vértice i apontado para o vértice j . A existência de grafos $A[i, j]$, onde esses possuem o rótulo ou peso que está associado com a aresta e, com isso se pode afirmar que a matriz não é formada por *bits*. Acaso ocorra de não possuir uma aresta de i para j existe a necessidade de utilizar um valor que não possa ser usado como rótulo ou peso. Logo adiante vamos apresentar a figura representando a matriz adjacência.

Lipson e Lipschutz (1997, p. 205), afirma que “quaisquer duas matizes de adjacência estão intimamente relacionados, podendo uma ser obtida a partir da outra pela simples troca de posição de linhas e colunas”, no entanto, LIPSON e LIPSCHUTZ (1997, p. 171) diz que:

a matriz de adjacência A de um grafo G depende da ordenação dos vértices de G , isto é, uma ordem diferente dos vértices conduz a uma matriz de adjacência diferente. Porém, duas matrizes de adjacência do mesmo grafo estão fortemente relacionadas entre si, de modo que uma pode ser obtida a partir da outra, simplesmente permutadas as linhas e colunas. Por outro lado, a matriz de adjacência não depende da ordem na qual as arestas são inseridas no computador.

Vamos analisar a figura a seguir que trata sobre as representações da matriz de adjacência:

FIGURA 70 – MATRIZ ADJACÊNCIA



FONTE: Fontes (2009)

Em relação às arestas existe uma diferença fundamental entre as duas primeiras estruturas e a terceira. A estrutura de lista de arestas e a estrutura de lista de adjacência armazenam apenas as arestas realmente presentes no grafo, enquanto a matriz adjacente armazena uma posição para cada par de vértices. (GOODRICH e TAMASSIA 2002, p. 603).



Saiba mais sobre o que são os GRAFOS acessando: <http://www.obmep.org.br/docs/Apostila5-Grafos.pdf>.

2.5 REPRESENTAÇÃO DE MATRIZES SIMÉTRICAS

O significado de matriz simétrica, seguindo conceitos de matemática, esse resultado vem da matriz transposta, em que essa matriz é o resultado da troca de linhas por colunas, isso pode ocorrer em uma determinada matriz, por isso toda matriz simétrica é chamada desta forma por ser igual a sua matriz transposta.

Para entendermos melhor o que significa a matriz simétrica, vamos analisar a definição desta matriz.

Para que possamos compreender a definição de uma matriz simétrica, é necessário compreendermos algumas notações usadas no estudo de matrizes.

a_{ij} - elemento da linha i , coluna j

matriz $A = (a_{ij})_{n \times n} \rightarrow$ Formação da matriz com n linhas e n colunas

Conhecendo estas notações, vejamos a definição para uma matriz simétrica.

Uma matriz simétrica é uma matriz quadrada de ordem n , que satisfaz:

$$A^t = A$$

Outra forma para enunciar esta definição é fazendo as igualdades dos elementos da matriz. Dizemos que uma matriz é simétrica quando,

$$a_{ij} = a_{ji} \text{ para todo } i, j$$

Vejamos alguns exemplos de matrizes simétricas.

$$A = \begin{pmatrix} 5 & 8 \\ 8 & 1 \end{pmatrix}, \text{ note que a transposta é igual e os elementos } a_{12} = a_{21}$$

$$B = \begin{pmatrix} 3 & 1 & 2 \\ 1 & 4 & 3 \\ 2 & 3 & 5 \end{pmatrix}, a_{12} = a_{21}; a_{13} = a_{31}; a_{23} = a_{32} \leftarrow \text{Veja que}$$

Vejamos um exemplo geral, com elementos quaisquer, simétricos.

$$C = \begin{pmatrix} a & b & c & d & e \\ b & f & g & h & i \\ c & g & j & m & n \\ d & h & m & l & o \\ e & i & n & o & k \end{pmatrix} \text{ Veja que os elementos } a_{ij} \text{ são iguais aos } a_{ji}$$

Você parou para pensar por que uma matriz simétrica é uma matriz quadrada? Façamos a seguinte reflexão: o que devemos fazer para obter a matriz transposta de uma determinada matriz?

Devemos inverter as linhas com as colunas, ou seja, uma matriz:

$$A = (a_{ij})_{m \times n} \rightarrow A^t = (a'_{ji})_{n \times m}$$

Veja que trocamos a quantidade de linhas pela quantidade de colunas. Para que uma matriz seja simétrica devemos ter a igualdade desta matriz com a sua transposta.

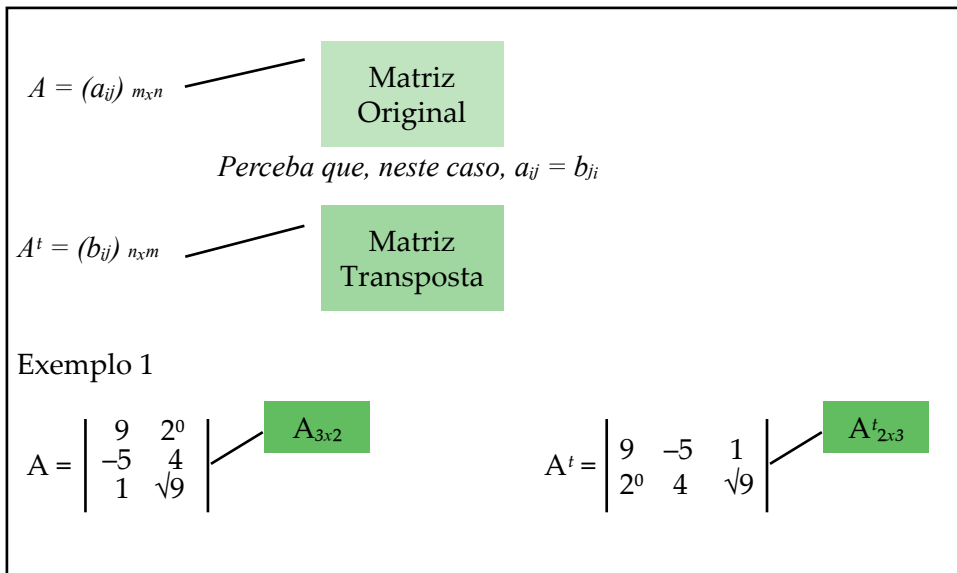
$$\text{Ou seja, } A = (a_{ij})_{m \times n} = A^t = (a'_{ji})_{n \times m}$$

Isto só será possível caso, $m = n$, e quando isso ocorre dizemos que a matriz é quadrada.

FONTE: OLIVEIRA, Gabriel A. Matriz Simétrica. Disponível em: <<http://www.mundoeducacao.com/matematica/matriz-simetrica.htm>>. Acesso em: 30 set. 2014.

A matriz transposta é formada pela base de uma matriz A, seguindo a ordem $m \times n$, com isso teremos uma matriz transposta, indicando diretamente para A^t , em que serão invertidas as posições de m e n , isso significa que será $n \times m$, vamos analisar como serão expostos esses valores:

FIGURA 71 – MATRIZ SIMÉTRICA



FONTE: Sá (2014)

Analisando o exemplo anterior podemos perceber que a primeira linha de A tornou-se a primeira coluna de A^t ; a segunda linha de A tornou-se a segunda coluna de A^t ; a terceira linha de A tornou-se a terceira coluna de A^t . Vamos analisar no exemplo a seguir como fica a linha e coluna de B:

FIGURA 72 – MATRIZ SIMÉTRICA TRANSPOSTA

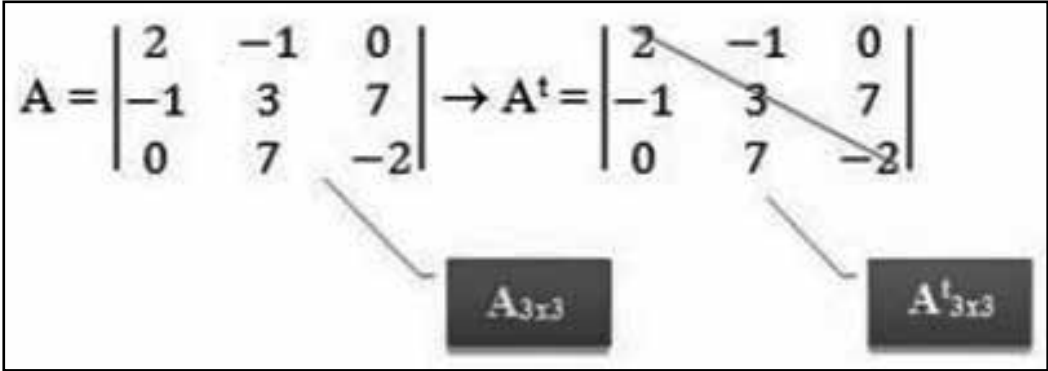


FONTE: Sá (2014)

Seguindo o exemplo da figura, pode-se perceber que a primeira linha de B tornou-se a primeira coluna de B^t; a segunda linha de B tornou-se a segunda coluna de B^t.

Analisando os exemplos anteriores, precisa-se observar que a matriz transposta A^t, onde a mesma for igual à matriz quadrada de A, pode-se dizer que esta matriz de A é uma matriz simétrica, isso significa que se A = A^t, A sempre será uma matriz simétrica. Em uma matriz simétrica, os elementos sempre serão simétricos, isso em relação à diagonal principal da matriz quadrada, onde podemos ter A = (a_{ij}), esta é simétrica, temos a_{ij} = a_{ji}. Perceba no exemplo a seguir:

FIGURA 73 – MATRIZ SIMÉTRICA



FONTE: Sá (2014)

Agora que conseguimos observar como são dispostos os elementos da matriz simétrica, vamos analisar no exemplo a seguir como pode ser aplicado o conceito da representação da matriz simétrica.

FIGURA 74 – APLICANDO A MATRIZ SIMÉTRICA

1. Dada a matriz $A = \begin{bmatrix} 3 & 4 \\ 2 & 5 \end{bmatrix}$, encontre:

a) A^t

Resposta $\rightarrow A^t = \begin{bmatrix} 3 & 2 \\ 4 & 5 \end{bmatrix}$

b) $(A^t)^t$

Resposta $\rightarrow (A^t)^t = \begin{bmatrix} 3 & 4 \\ 2 & 5 \end{bmatrix}$

c) O que se pode dizer das matrizes A e de $(A^t)^t$?

Resposta $\rightarrow (A^t)^t = A$

2. Escreva a matriz quadrada de ordem 2, com $A = (a_{ij}) \mid a_{ij} = i - j$, e determine a sua transposta A^t .

Resposta \rightarrow

$$\begin{array}{ll} a_{11} = 1 - 1 = 2 & a_{12} = 1 - 2 = -1 \\ a_{21} = 2 - 1 = 1 & a_{22} = 2 - 2 = 0 \end{array}$$

Temos, $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}$

Portanto, $A^t = \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix}$

FONTE: Sá (2014)

Agora que temos noção do que é uma matriz simétrica, vamos aprofundar ainda mais nossos estudos, analisando que uma matriz simétrica é considerada como $n \times n$, e não existe a necessidade de armazenar os seguintes elementos $\text{mat}[i][j]$ e $\text{mat}[j][i]$, pois tanto esses valores quanto $i \neq j$, possuem o mesmo valor. Com isso existe a necessidade apenas de armazenar os valores dos elementos que estão na diagonal, bem como da metade dos elementos restantes, segundo o exemplo dos elementos da diagonal, com os quais $i > j$, isso pode ajudar no processo e obter mais espaço na memória, aumentando a eficiência com a alocação da matriz. Vamos analisar a seguir o exemplo de como realizar esse processo, ao invés de n^2 valores, vamos armazenar apenas s como elementos, sendo este s dado, com o seguinte comando:

$$s = n + \frac{(n^2 - n)}{2} = \frac{n(n+1)}{2}$$

Para que possamos determinar o s como um elemento da soma de uma progressão aritmética, precisa-se armazenar um elemento na primeira linha, depois dois elementos na segunda linha, posterior três elementos na terceira linha e assim consequentemente. Vamos analisar como seriam estes valores:

$$s = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

2.6 TIPOS ABSTRATOS DE DADOS

Quando um módulo define um novo tipo de dado, como define o conjunto de operações que irão manipular os dados desse novo tipo de dado? Esse procedimento está representando um tipo abstrato de dado, conhecido pela sigla TAD. O tipo de dado abstrato é definido por ter sua finalidade do tipo e de suas operações, não como são implementadas. Para criar uma TAD para que esta possa representar matrizes que sejam alocadas dinamicamente, é preciso criar um tipo de matriz, como também uma série de funções para manipular essas matrizes. No caso de criar formas de acessar a matriz, podem ser utilizadas funções que possam acessar e manipular os elementos dessa matriz, para que isso ocorra podemos criar um tipo abstrato de dado, onde é possível esconder a forma como são implementadas essas funções.

Os tipos abstratos de dados (TADs) são estruturas de dados capazes de representar os tipos de dados que não foram previstos no núcleo das linguagens de programação e que, normalmente, são necessários para as aplicações específicas. Essas estruturas são divididas em duas partes: os dados e as operações. Um TAD é, portanto, uma forma de definir um novo tipo de dado juntamente com as operações que manipulam esse novo tipo de dado. As aplicações que utilizam esses TADs são denominadas clientes do tipo de dado. (EDELWEISS e GALANTE, 2009, p. 37).

Lembrando que o uso de tipos abstratos de dados, para serem utilizados, precisam apenas conhecer para qual finalidade se está criando essa funcionalidade, não a forma de como pode ser implementada, com este processo de saber suas funcionalidades, é possível reutilizar os códigos, bem como facilitar a manutenção dessa matriz. TAD é definido por ser formado por uma coleção de dados definidos a serem armazenados, é também formado por um grupo de operadores que possuem a finalidade de serem aplicados para a manipulação dos dados. O TAD possui como suas principais características, que são denominadas de fundamentais para a formação dos dados abstratos, onde os operadores de TAD realizam a implementação de regras e essas precisam estar bem definidas para a manipulação dos valores que estão armazenados, outra característica é os valores que estão armazenados, precisam ser manipulados de forma exclusiva pelos operadores do TAD.


O TAD é um tipo de dado que estabelece sua forma de representar esses dados de forma separada, sua representação é definida pelo modelo da matemática, em que esta definição é dada pelo par de $(V \text{ e } O)$, onde o V é representado por um conjunto de valores e o O é formado por um conjunto de operações sobre esses valores de V . Vamos analisar a seguir a representação do tipo de dado abstrato real:

$$V = \mathbb{R}$$
$$O = \{+, -, *, /, =, <, >, \leq, \geq\}$$

A definição de um TAD serve para especificar e descrever quais os dados podem ser armazenados, como é possível utilizar esses dados através dos operadores dos dados TAD, no entanto, essas definições não descrevem como devem ser implementados o TAD em um programa. Sua aplicação possui apenas a definição de ser um TAD modelo, em que este modelo define e especifica como manipular e armazenar os dados.

Vamos analisar um exemplo de TAD, em que é possível armazenar e manipular a idade das pessoas. São componentes (1) os dados armazenados e (2) os operadores. O operador *Nasce* e operador *Aniversário*, uma característica fundamental dos TADs é que os dados armazenados devem ser manipulados EXCLUSIVAMENTE através dos operadores do TAD. (BARBOSA; MIYOSHI e GOMES 2008).

FIGURA 75 – TIPO ABSTRATO DE DADOS

Mundo Real	Coleção Bem Definida de Dados	Grupo de Operadores
 pessoa	<ul style="list-style-type: none">a idade da pessoa	<ul style="list-style-type: none">nasce (idade recebe o valor zero)aniversário (idade aumenta em 1)
 fila de espera	<ul style="list-style-type: none">nome de cada pessoa e sua posição na fila	<ul style="list-style-type: none">sai da fila (o primeiro)entra na fila (no fim)
 cadastro de funcionários	<ul style="list-style-type: none">o nome, cargo e o salário de cada funcionário	<ul style="list-style-type: none">entra no cadastrosai do cadastroaltera o cargoaltera o salário
 Pilha de Cartas	<ul style="list-style-type: none">informações que identificam a carta (nipe, valor), e sua posição na pilha de cartas	<ul style="list-style-type: none">põe uma carta na pilha (no topo da pilha)retira uma carta da pilha (a do topo)

FONTE: Barbosa; Miyoshi e Gomes (2008)

2.6.1 Tipo de dado abstrato Ponto

Para a criação de um tipo de dado, em que seja possível representar um ponto no R2, deve-se primeiramente definir um tipo de dado abstrato, que pode ser chamado de Ponto, como o conjunto de funções que pode operar sobre esse tipo de dado. Esse ponto realiza as seguintes operações, de acordo com Lima (2014):

- cria: operação que cria um ponto com coordenadas x e y;
- libera: operação que libera a memória alocada por um ponto;
- acessa: operação que devolve as coordenadas de um ponto;
- atribui: operação que atribui novos valores às coordenadas de um ponto;
- distancia: operação que calcula a distância entre dois pontos.

Segundo Celes e Rangel (2008), a interface deste módulo, contendo as operações de cria, libera, acessa, atribui e distancia, pode ser dada através do seguinte código:

FIGURA 76 – ARQUIVO PONTO.H.

```
/* TAD: Ponto (x,y) */
/* Tipo exportado */
typedef struct ponto Ponto;

/* Funções exportadas */

/* Função cria
** Aloca e retorna um ponto com coordenadas (x,y)
*/
Ponto* cria (float x, float y);

/* Função libera
** Libera a memória de um ponto previamente criado.
*/
void libera (Ponto* p);

/* Função acessa
** Devolve os valores das coordenadas de um ponto
*/
void acessa (Ponto* p, float* x, float* y);

/* Função atribui
** Atribui novos valores às coordenadas de um ponto
*/
void atribui (Ponto* p, float x, float y);

/* Função distancia
** Retorna a distância entre dois pontos
*/
float distancia (Ponto* p1, Ponto* p2);
```

FONTE: Celes e Rangel (2008)

Podemos citar alguns exemplos de características do TAD Ponto, como por exemplo, definir o nome do tipo e definir os nomes das funções que serão exportadas. Este TAD Ponto possui também algumas limitações, como a estrutura

Ponto não contempla as partes da interface, não fazem parte dessa estrutura do Ponto, pois não são exportadas pelo módulo, não pertence à interface do módulo, como também não está disponível visivelmente para os outros módulos.

Podemos encontrar outras limitações em relação aos módulos que utilizam o TAD Ponto, essas limitações do módulo são: não é possível acessar diretamente os campos da estrutura do Ponto e só podem acessar aos dados que foram obtidos através das funções exportadas. Para realizar a implementação do TAD Ponto são seguidos os passos como incluir os arquivos de interface de Ponto, definir qual a composição da estrutura Ponto e incluir a implementação das funções externas. Vamos analisar a seguir um exemplo de como pode ser utilizado o TAD Ponto:

FIGURA 77 – COMANDO TAD PONTO

```
#include <stdio.h>
#include "ponto.h"

int main(void)
{
    float x, y;
    Ponto* p = pto_cria(2.0,1.0);
    Ponto* q = pto_cria(3.4,2.1);
    float d = pto_distancia(p,q);
    printf("Distancia entre pontos: %f\n",d);
    pto_libera(q);
    pto_libera(p);
    return 0;
}
```

FONTE: Lima (2014)

2.6.2 Tipo de dado abstrato Matriz

Como um TAD é implementado de forma que fica escondido dentro do seu próprio módulo, dessa forma fica mais fácil de experimentar formas que sejam diferentes para implementar um mesmo tipo de dado TAD. Esse processo de utilizar o mesmo tipo de dado TAD, torna-se mais fácil de experimentar diversas formas de implementar o mesmo TAD, sem mesmo afetar os seus clientes. Com esta dependência da implementação de dados TAD, em que se pode criar vários tipos de dados abstratos, em que esse representa matrizes de valores reais, e esses valores reais estão alocados de forma dinâmica, onde as dimensões de $m \times n$, são fornecidos no mesmo tempo de execução. Especificamente existe a necessidade de definir um tipo de dado abstrato, que será denominado de matriz, como também será definido o conjunto de funções que irão operar sobre esses tipos de dados.

Segundo Lima (2014) neste dado do TAD matriz serão consideradas as seguintes operações:

- cria: operação que cria uma matriz de dimensão m por n;
- libera: operação que libera a memória alocada para a matriz;
- acessa: operação que acessa o elemento da linha i e da coluna j da matriz;
- atribui: operação que atribui o elemento da linha i e da coluna j da matriz;
- linhas: operação que devolve o número de linhas da matriz;
- colunas: operação que devolve o número de colunas da matriz.

Conforme Celes e Rangel (2008), a interface deste módulo, contendo as operações de cria, libera, acessa, atribui e distancia, pode ser dada através do tipo de dado TAD matriz, com o seguinte código:

FIGURA 78 – ARQUIVO MATRIZ.H.

```
/* TAD: matriz m por n */

/* Tipo exportado */
typedef struct matriz Matriz;

/* Funções exportadas */

/* Função cria
** Aloca e retorna uma matriz de dimensão m por n
*/
Matriz* cria (int m, int n);

/* Função libera
** Libera a memória de uma matriz previamente criada.
*/
void libera (Matriz* mat);

/* Função acessa
** Retorna o valor do elemento da linha i e coluna j da matriz
*/
float acessa (Matriz* mat, int i, int j);

/* Função atribui
** Atribui o valor dado ao elemento da linha i e coluna j da matriz
*/
void atribui (Matriz* mat, int i, int j, float v);

/* Função linhas
** Retorna o número de linhas da matriz
*/
int linhas (Matriz* mat);
```

FONTE: Celes e Rangel (2008)

Para que possamos analisar e verificar se uma matriz é realmente simétrica, precisa-se observar se os valores retornam como: 1 se verdadeiro e 0 se falso,

sempre lembrando que só pode ser uma matriz simétrica se ela for igual a sua transposta e a matriz transposta é o retorno do resultado da troca entre linhas por colunas, isso ocorre da matriz original, sendo transposta para uma matriz simétrica. A seguir vamos analisar como pode ser apresentado o TAD Matriz:

FIGURA 79 – TAD MATRIZ

```
int simetrica(Matriz *mat)
{
    int i, j;
    for (i=0; i<mat_linhas(mat); i++)
    {
        for (j=0; j<mat_colunas(mat); j++)
        {
            if (mat_acessa(mat,i,j) != mat_acessa(mat,j,i))
                return 0;
        }
    }
    return 1;
}
```

FONTE: Lima (2014)

RESUMO DO TÓPICO 3

Neste tópico vimos:

- As matrizes são consideradas arranjos ordenados, podem ser formadas por n dimensões, essas são denominadas de dimensional. As matrizes possuem várias dimensões, em que uma é chamada de dimensional, quando são duas é bidimensional, três é tridimensional e assim consequentemente.
- Quando a grande parte dos elementos é igual a zero ou é considerada como um valor constante, é chamado de matriz especial. Esta matriz exige muita atenção devido ao seu desperdício de espaço ao realizar a alocação da memória. Isso se deve à matriz especial possuir elementos que tem o mesmo valor, para que o espaço não seja desperdiçado precisa-se apresentar apenas valores que são representativos e significativos para a matriz.
- Os vetores bidimensionais de matrizes possuem como responsabilidade armazenar informações de forma organizada, esse processo é realizado através de linhas e colunas, em que a matriz é formada por uma estrutura de dados que são lógicas e por um conjunto de valores da estrutura.
- A matriz dinâmica é representada por vetores que são do tipo simples, é representada também por vetores do tipo unidimensional. As principais funções da matriz dinâmica é armazenar os elementos do vetor na primeira linha da matriz, criar abstrações conceituais com os vetores.
- A representação de matriz ocorre através de dois processos, como por exemplo, procedimentos e funções, em que estes realizam a funcionalidade de representar as tarefas das matrizes. As representações dessas operações ocorrem da seguinte forma: realizar a leitura da matriz; desenvolver uma matriz; fazer a comparação somatória e manipular as duas matrizes.
- A representação de matrizes simétricas ocorre através de um processo em que existe um resultado na troca de linhas e colunas, esse procedimento é conhecido como matriz transposta, então a matriz simétrica é igual a sua matriz transposta e isso a define como a matriz simétrica.
- O tipo abstrato de dados – TAD é definido por ser formado de um novo tipo de dado, em que o módulo define um conjunto de operações que serão manipuladas por esse novo tipo de dado, sua principal característica é a finalidade de suas operações. A utilização do TAD é para saber qual é a sua finalidade e sua funcionalidade de ser implementada, formado por um grupo de operadores, em que esses manipulam os dados. O TAD é representado através de dois tipos de dados: dado Abstrato Ponto e o dado Abstrato Matriz.



- 1 As matrizes são formadas por dimensões que podem ser chamadas de dimensional, bidimensional, tridimensional e assim consequentemente, é uma estrutura de dados que necessita de um índice. Os vetores bidimensionais de matrizes também são referenciados por índices, esses referenciam:
 - () Linha e Coluna.
 - () Vetores e Arrays.
 - () Alocação e arranjo.
 - () Memória e Valores.

- 2 A definição de tipos abstratos de dados se dá através da sua forma de definir para qual finalidade pode ser utilizada e como pode utilizar as suas operações. Suas estruturas são divididas em duas partes de dados e operações. O tipo de dado abstrato Ponto é formado por cinco operações, analise as alternativas a seguir e assinale a correta.
 - () Linha, Colunas, Cria, Libera e Acessa.
 - () Cria, Libera, Acessa, Atribui e Distancia.
 - () Atribui, Distancia, Aloca, Retorna e Liga.
 - () Aloca, Retorna, Liga, Linha e Colunas.

ESTRUTURAS DE DADOS AVANÇADAS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade você será capaz de:

- formar estruturas de dados encadeadas com referências, classes autorreferenciais e recursão;
- criar e manipular estrutura de dados dinâmicas como listas encadeadas, duplamente encadeadas, pilhas e filas;
- entender várias aplicações importantes de estruturas de dados encadeadas e como criar estruturas de dados reutilizáveis com classes, herança e composição.

PLANO DE ESTUDOS

Esta unidade está dividida em quatro tópicos. Ao final de cada um deles, você encontrará atividades que auxiliarão a fixar as estruturas e praticar problemas práticos em uma linguagem de programação.

TÓPICO 1 – LISTAS ENCADEADAS

TÓPICO 2 – LISTAS DUPLAMENTE ENCADEADAS

TÓPICO 3 – PILHAS

TÓPICO 4 – FILAS

LISTAS ENCADEADAS

1 INTRODUÇÃO

Conforme já estudamos, a representação de um grupo de dados pode se dar através do uso de vetor. E é a forma mais primitiva de representar diversos elementos agrupados. Com o objetivo de sintetizar a discussão dos conceitos que serão apresentados a seguir, iremos tomar por base o desenvolvimento de uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos, conforme se verifica na figura a seguir.

FIGURA 81 – DECLARAÇÃO DE VETOR DE INTEIROS

```
int n = 10; // tamanho do vetor
int vet[] = new int[n]; // declaração e alocação de espaço para o vetor "vet"
int i; // índice ou posição
```

FONTE: O autor

Ao declarar um vetor, reserva-se um espaço contíguo de memória para armazenar seus elementos, sendo que seu índice iniciará na posição 0, conforme ilustra a figura a seguir.

FIGURA 82 – VETOR PREENCHIDO COM VALORES INTEIROS

vet[0]	vet[1]	vet[2]	vet[3]	vet[4]	vet[5]	vet[6]	vet[7]	vet[8]	vet[9]
0	1	2	3	4	5	6	7	8	9

FONTE: O autor

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. Efetivamente, o símbolo `vet`, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de `vet` é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação `vet[i]`. Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

Entretanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

FONTE: Disponível em: <<http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap10.pdf>>. Acesso em: 24 nov. 2014

Além disso, o uso inadequado de vetores pode afetar diretamente o desempenho do nosso programa, por exemplo, ao adicionar um elemento na primeira posição de um Vetor teremos que deslocar todos os outros elementos uma posição para frente. Esta operação, à medida que a quantidade de elementos do nosso vetor cresce, consumirá tempo linear em relação ao número de elementos. Da mesma forma, ao remover um elemento da primeira posição implicará deslocar todos os outros elementos que estão na sua frente para trás.

Com o objetivo de minimizar estes problemas, a solução é utilizar estruturas de dados que possibilitem o seu dimensionamento de forma automática, acrescentando ou retirando elementos conforme a necessidade de armazenamento. Tais estruturas são chamadas dinâmicas e armazenam cada um dos seus elementos usando alocação dinâmica.

Nas seções a seguir, discutiremos a estrutura de dados conhecida como lista encadeada. As listas encadeadas são amplamente utilizadas na implementação de diversas outras estruturas de dados com semânticas próprias, que serão tratadas nos tópicos seguintes.

2 A LISTA ENCADEADA

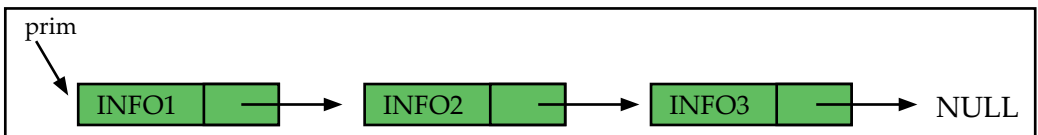
Ao utilizar uma lista encadeada, cada novo elemento inserido na estrutura será alocado de forma dinâmica na memória. Consequentemente, o espaço total de memória utilizado pela estrutura será proporcional ao número de elementos nela armazenado.

De acordo com Deitel e Deitel (2003, p. 62):

Criar e manter estruturas de dados dinâmicas exige *alocação dinâmica de memória* – a capacidade de um programa obter mais espaço de memória durante a execução para armazenar novos nodos e liberar espaço não mais necessário. [...] O limite para alocação dinâmica de memória pode ser tão grande quanto à quantidade de memória física disponível no computador ou a quantidade de espaço disponível em disco em um sistema de memória virtual. Frequentemente, os limites são muito menores porque a memória disponível do computador deve ser compartilhada entre muitos aplicativos.

Outra característica interessante do processo de alocação dinâmica é que não podemos afirmar que os elementos armazenados na lista ocuparão um espaço de memória contíguo, isto quer dizer, um ao lado do outro. Logo, não há possibilidade de acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente armazenar o encadeamento dos elementos. Desta forma, armazena-se a informação de cada elemento juntamente com um ponteiro para o próximo elemento da lista. A figura a seguir ilustra o arranjo da memória de uma lista encadeada.

FIGURA 83 – ARRANJO DA MEMÓRIA DE UMA LISTA ENCADEADA



FONTE: O autor

A estrutura consiste numa sequência encadeada de elementos, comumente denominada de nós da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó), a partir do qual se pode ascender ao segundo, seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, indicando que não há mais elementos na lista.

Para exemplificar a implementação de listas encadeadas em Java, vamos considerar um exemplo simples em que queremos armazenar o nome e o peso de uma pessoa numa lista encadeada. O nó da lista pode ser representado pela estrutura a seguir:

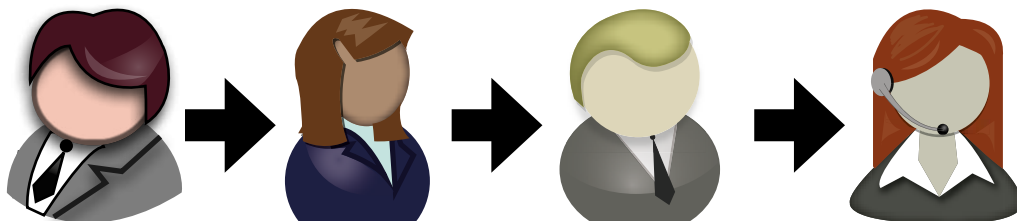
FIGURA 84 – ESTRUTURA LISTA EM JAVA

```
1 package exemplo;  
2  
3 public class PessoaLista {  
4     private String nome;  
5     private Double peso;  
6     private PessoaLista prox;
```

FONTE: O autor

A classe PessoaLista trata-se de uma estrutura autorreferenciada, pois, além dos campos que armazenam as informações (nome e peso), há o atributo (prox) que é um ponteiro para uma próxima estrutura do mesmo tipo. Normalmente, o acadêmico fica confuso em relação a isso, como se esse código fosse gerar um laço infinito, ou uma recursão infinita. No entanto, não é o caso, já que a declaração do atributo prox é apenas uma referência, que não cria uma instância de PessoaLista na memória, apenas criando uma referência a referida classe já instanciada!

FIGURA 85 – REPRESENTAÇÃO DA LISTA DE PESSOAS



FONTE: O autor

Importante destacar que, embora a Figura 85 represente as pessoas uma ao lado da outra, na memória provavelmente elas estejam em regiões bem distintas, contudo, cada uma delas sabe dizer em que local se encontra a próxima pessoa já que há a referência ao próximo pelo atributo prox.

O código exibido na Figura 84 não é a melhor forma de implementar uma lista, tendo em vista que há uma confusão de responsabilidades, onde a classe PessoaLista além de armazenar informações sobre uma pessoa, armazena também uma lista de pessoas. Logo, manipulá-la, pode ficar estranho e muito trabalhoso, já que há a necessidade de manter constantemente uma referência para a primeira pessoa na memória em algum outro lugar, o que também pode ser confuso, além de violar o conceito de coesão.

Ademais, sendo necessário criar uma lista encadeada de produtos, por exemplo, ao invés de reaproveitar o código existente, prática comum

na programação orientada a objetos, será necessário criar uma nova classe denominada `ProdutoLista` semelhante com a `PessoaLista`.

Neste sentido, para criar uma estrutura mais funcional, utiliza-se uma classe separada como “nós”, evitando mesclar a classe de dados (`Pessoa`) da nossa estrutura.

2.1 NÓS E LISTA ENCADEADA

Para isolar a manipulação da estrutura de dados será imprescindível a criação de uma classe para representar o nó, a qual deverá possuir uma referência para o elemento ao qual ela se refere, e uma referência para o próximo nó, que pode ser null, caso esse seja o último nó da Lista.



Ao trabalhar com lista duplamente encadeada, além de armazenar a referência para o próximo nó, será necessário armazenar a referência para o nó anterior, conforme veremos no tópico a seguir.

Através do ambiente integrado de desenvolvimento Eclipse Luna, disponível para *download* no link <<http://www.eclipse.org/downloads/>>, daremos início ao projeto de desenvolvimento da Lista Encadeada, criando inicialmente a classe “`No`”, conforme figura a seguir, a qual irá possuir apenas dois atributos getters e setters, além dos construtores para facilitar nosso trabalho.

FIGURA 86 – CÓDIGO DA CLASSE NO

```

1 package exemplo;
2
3 public class No {
4     private No proximo;
5     private Object elemento;
6     public No(No proximo, Object elemento) {
7         this.proximo = proximo;
8         this.elemento = elemento;
9     }
10
11     public No(Object elemento) {
12         this.elemento = elemento;
13     }
14
15     public void setProximo(No proximo) {
16         this.proximo = proximo;
17     }
18
19     public No getProximo() {
20         return proximo;
21     }
22
23     public Object getElemento() {
24         return elemento;
25     }
26 }
27

```

FONTE: O autor

Importante ressaltar que a classe “No” nunca será acessada diretamente por quem for desenvolver uma classe de Lista Encadeada. Desta forma, estaremos escondendo funcionamento da nossa classe e, conseqüentemente, garantindo que ninguém altere o funcionamento da sua estrutura interna.

Concluída a classe “No”, crie uma nova classe denominada “**ListaEncadeada**” que possuirá inicialmente apenas uma referência para o primeiro nó e outra para o último. Através do primeiro nó ocorrerá a iteração para chegar a qualquer posição necessária, enquanto que a referência ao último irá facilitar a inserção no fim da Lista.

FIGURA 87 – CÓDIGO DA CLASSE LISTAENCADEADA

```
public class ListaEncadeada {
    private No primeiro;
    private No ultimo;
```

FONTE: O autor

2.2 DEFININDO A INTERFACE

A nossa interface deverá possibilitar ao usuário as seguintes operações:

- Adicionar uma pessoa no fim da lista.
- Adicionar uma pessoa em uma posição específica.
- Pegar uma pessoa em uma posição específica.
- Remover uma pessoa de uma posição específica.
- Verificar se determinada pessoa está armazenada.
- Informar o número de pessoas armazenadas.

Definidas as operações que deverão constar em nossa interface, podemos esboçar os métodos que serão implementados na classe “**ListaEncadeada**”.

FIGURA 88 – MÉTODOS DA CLASSE LISTAENCADEADA

```
3 public class ListaEncadeada {
4     private No primeiro;
5     private No ultimo;
6
7     public void adiciona(Object elemento) {}
17
18     public void adicionaPosicao(int posicao, Object elemento) {}
19
20     public Object pega(int posicao) {return null;}
21
22     public void remove(int posicao){}
23
24     public int tamanho() {return 0;}
25
26     public boolean contem(Object o) {return false;}
```

FONTE: O autor

Através da utilização de uma lista encadeada, as operações de adicionar e remover da primeira e da última posição serão computacionalmente eficientes, diferentemente do que ocorria quando da utilização de vetores. Contudo, para que isso seja possível, precisaremos desenvolver mais três métodos na classe “**ListaEncadeada**”, que observamos na figura a seguir.

FIGURA 89 – MÉTODOS ESSENCIAIS DA CLASSE LISTAENCADEADA

```

public void adicionaNoComeco(Object elemento) {}

public void removeDoComeco() {}

public void removeDoFim() {}

```

FONTE: O autor

Neste momento, você deve se perguntar: por que não foi definido o método `adicionaNoFim()`? A explicação é simples: O método “`adiciona()`” já definido anteriormente (Figura 88), exercerá a função de inserir no fim da Lista, não havendo desta forma, a necessidade de criar um novo método para realizar esta tarefa.

2.3 IMPLEMENTAÇÃO DOS MÉTODOS

Aqui vamos começar a aumentar o código da nossa classe **ListaEncadeada** baseado nas operações que precisamos realizar com ela. Para facilitar a contagem de elementos (pessoas) e algumas operações, logo a seguir dos atributos primeiro e último, adicione o atributo **int totalDeElementos**, perceba.

FIGURA 90 – CRIAÇÃO DO ATRIBUTO PARA CONTAR OS ELEMENTOS

```

3 public class ListaEncadeada {
4     private No primeiro;
5     private No ultimo;
6
7     private int totalDeElementos;

```

FONTE: O autor

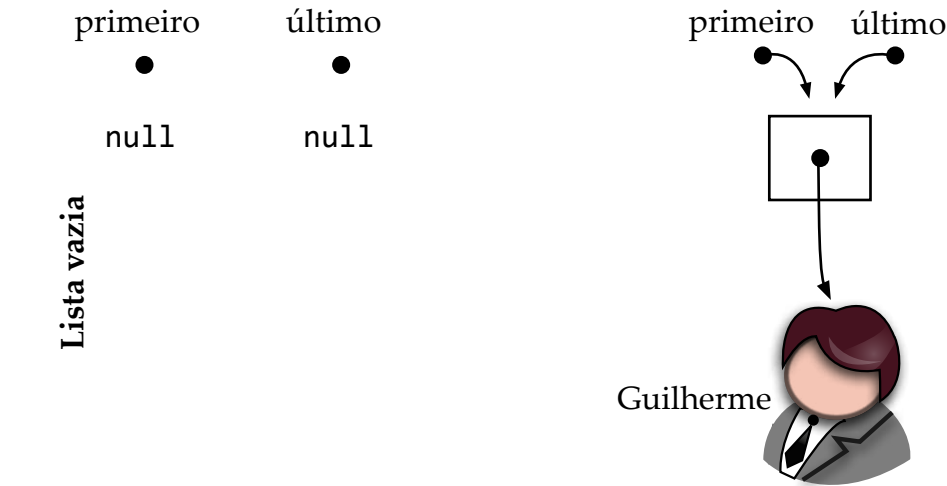
2.3.1 Método `adicionaNoComeco()`

Inserir no começo da Lista é muito simples, basta criarmos um novo nó, que se dará através da utilização do operador **new**, que segundo Deitel e Deitel (2003) é essencial para alocação dinâmica na memória, já que recebe como operando o tipo do objeto que está sendo dinamicamente alocado e devolve uma referência para um objeto desse tipo recém-criado. Além disso, este objeto recém-criado terá a referência “proximo” apontando para o atual “primeiro” da lista, sendo que posteriormente atualizamos o atributo primeiro para se referenciar a este novo objeto.

No entanto, é importante verificar se a Lista encontra-se vazia. Nestas situações devemos atualizar a referência que aponta para o último nó também. Para

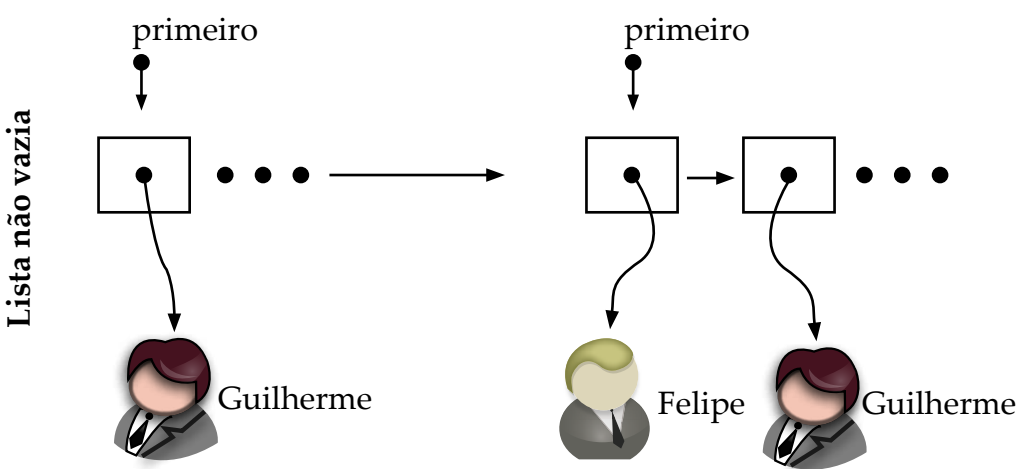
tanto, precisaremos de um novo atributo, que chamaremos de **int totalDePessoas**, e será responsável por retornar o total de pessoas que se encontram armazenadas nesta lista.

FIGURA 91 – ADICIONA NO COMEÇO COM A LISTA VAZIA



FONTE: O autor

FIGURA 92 – ADICIONA NO COMEÇO COM A LISTA NÃO VAZIA



FONTE: O Autor

Compreendido o conceito de como deve ocorrer o processo de inserção das pessoas na lista encadeada, vamos ao desenvolvimento do código-fonte no ambiente de desenvolvimento:

FIGURA 93 – CÓDIGO PARA ADICIONAR NO COMEÇO

```
9 public void adicionaNoComeco(Object elemento) {
10     No nova = new No (this.primeiro, elemento);
11     this.primeiro = nova;
12
13     if (this.totalDeElementos == 0) {
14         //caso especial da lista vazia
15         this.ultimo = this.primeiro;
16     }
17     this.totalDeElementos++;
18 }
```

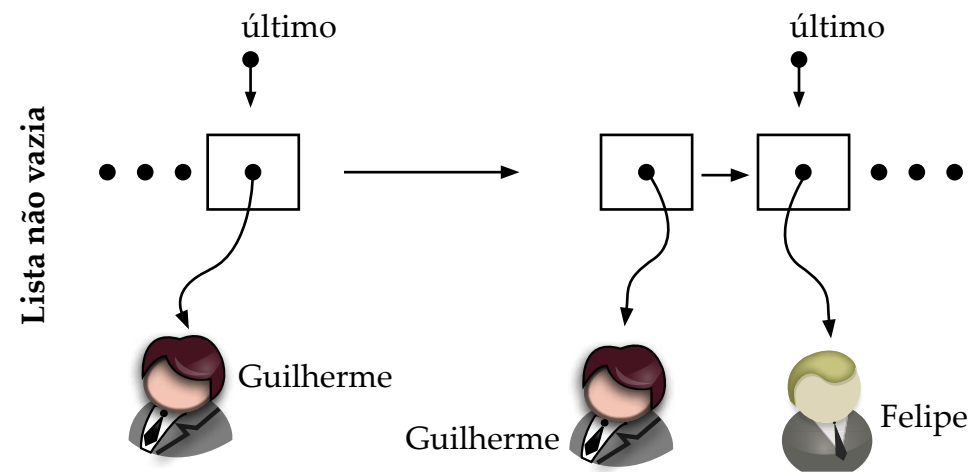
FONTE: O autor

2.3.2 Método adiciona()

O método adiciona() permitirá adicionar a pessoa no último nó da Lista, no entanto, se não tivéssemos guardado a referência para o último nó precisaríamos percorrer nó a nó até o fim da Lista para alterar a referência **proximo** do último nó, reduzindo consideravelmente o desempenho do nosso programa, já que havendo um grande número de elementos e o processo tornar-se-ia lento.

No caso especial da Lista estar vazia, adicionar no começo ou no fim dessa lista dá o mesmo efeito. Então, se a Lista estiver vazia, chamaremos o método já definido anteriormente adicionaNoComeco(Object), (CAELUM, 2014). Conforme veremos no código-fonte a seguir.

FIGURA 94 – ADICIONA NO ÚLTIMO NÓ COM A LISTA NÃO VAZIA



FONTE: O autor

FIGURA 95 – CÓDIGO PARA ADICIONAR NO ÚLTIMO NÓ DA LISTA

```

20 public void adiciona(Object elemento) {
21     if (this.totalDeElementos == 0) {
22         this.adicionaNoComeco(elemento);
23     } else {
24         No nova = new No(elemento);
25         this.ultimo.setProximo(nova);
26         this.ultimo = nova;
27         this.totalDeElementos++;
28     }
29 }

```

FONTE: O autor

Perceba que aqui estamos fazendo que haja n nós para n elementos, isto é, um nó para cada elemento. Outra implementação clássica de lista encadeada é usar um nó sentinela a mais para indicar o começo da Lista, e outro para o fim, assim poderíamos simplificar um pouco alguns desses métodos, como o caso particular de inserir um elemento quando não há ainda elemento algum. Vale sempre lembrar que aqui estamos estudando uma implementação de estrutura de dados, e que há sempre outras formas de codificá-las que podem ser mais ou menos elegantes.

FONTE: Caelum (2014)

2.3.3 Método adicionaPosicao()

A inserção no começo e no fim da Lista já foram devidamente tratadas nos itens anteriores. Aqui vamos nos preocupar em inserir em uma posição no interior da Lista, ou seja, qualquer posição que não seja a primeira e nem a última. Para inserir um elemento em qualquer posição precisamos pegar o nó anterior ao da posição desejada, porque precisamos mexer na sua referência **proximo**. (CAELUM, 2014). Para isso vamos criar um método auxiliar responsável por pegar determinado nó, ao qual atribuiremos o nome de **pegaNo**. Ao utilizar este método devemos tomar cuidado no caso da posição não existir. Para tanto, iremos inicialmente desenvolver o método **posicaoOcupada**, que verificará se a posição existe ou não.

FIGURA 96 – CÓDIGO PARA VERIFICAR POSIÇÃO OCUPADA

```
108 private boolean posicaoOcupada(int posicao){
109     return posicao >= 0 && posicao < this.totalDeElementos;
110 }
111
112 private No pegaNo(int posicao) {
113     if(!this.posicaoOcupada(posicao)){
114         throw new IllegalArgumentException("Posição não existe");
115     }
116     No atual = primeiro;
117     for (int i = 0; i < posicao; i++) {
118         atual = atual.getProximo();
119     }
120     return atual;
121 }
```

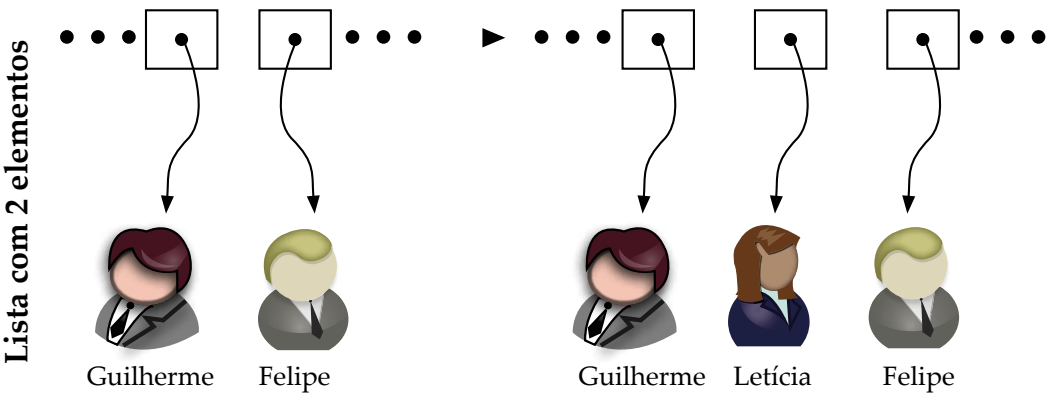
FONTE: O autor

Os métodos são privados (private), pois não queremos que ninguém de fora tenha acesso ao funcionamento interno da nossa estrutura de dados. É importante notar que o método **pegaNo** consome tempo linear.

Desenvolvido os métodos auxiliares, torna-se mais fácil a implementação do método **adicionaPosicao(int, Object)**. Basta pegar o nó anterior, a posição onde a inserção será feita e atualizar as referências. O **anterior** deve apontar para um novo nó e o novo nó deve apontar para o antigo **proximo** do **anterior**. (CAELUM, 2014).

Devemos tomar cuidado com os casos particulares nos quais a posição para inserir é o começo ou o fim da Lista.

FIGURA 97 – ADICIONA EM UMA POSIÇÃO DA LISTA



FONTE: O autor

FIGURA 98 – CÓDIGO PARA ADICIONAR EM UMA POSIÇÃO DA LISTA

```

33 public void adicionaPosicao(int posicao, Object elemento) {
34     if(posicao == 0){ // No começo.
35         this.adicionaNoComeco(elemento);
36     } else if(posicao == this.totalDeElementos){ // No fim.
37         this.adiciona(elemento);
38     } else {
39         No anterior = this.pegarNo(posicao - 1);
40         No novo = new No(anterior.getProximo(), elemento);
41         anterior.setProximo(novo);
42         this.totalDeElementos++;
43     }
44 }

```

FONTE: O autor

2.3.4 Método pegar()

Para pegar um elemento é muito simples: basta pegarmos o nó em que aquele elemento se encontra e acessar o elemento que se encontra dentro dele. Podemos utilizar o método **pegarNo(int)** previamente criado:

FIGURA 99 – CÓDIGO PARA PEGAR UM ELEMENTO NA LISTA

```

44 public Object pegar(int posicao) {
45     return this.pegarNo(posicao).getElemento();
46 }

```

FONTE: O autor

Perceba que este método consome tempo linear. Esta é uma grande desvantagem da Lista Encadeada em relação aos Vetores. Vetores possuem o chamado acesso aleatório aos elementos, ou seja, qualquer posição pode ser acessada em tempo constante. Apesar dessa grande desvantagem, diversas vezes utilizamos uma Lista e não é necessário ficar acessando posições aleatórias: comumente percorremos a lista por completa, que veremos como fazer mais adiante.

FONTE: O autor

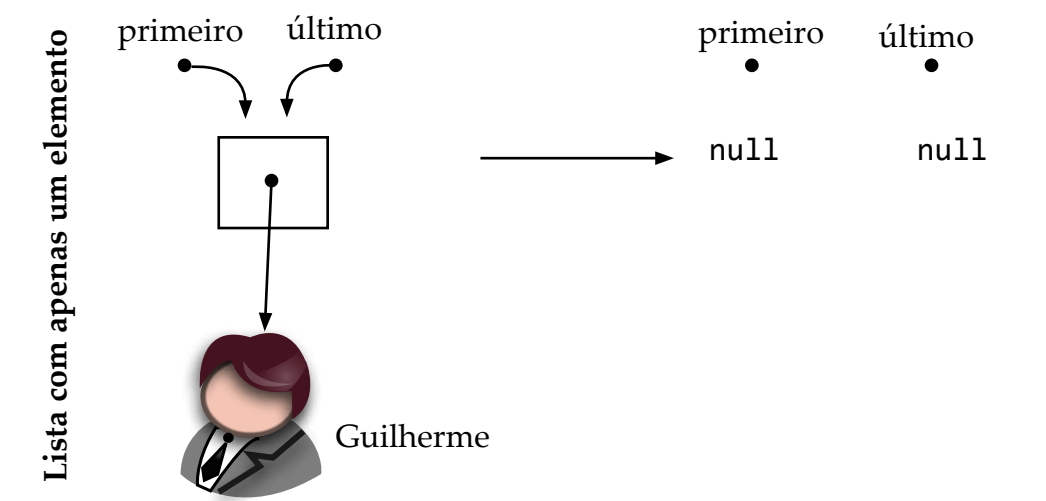
2.3.5 Método removeDoComeco()

Antes de tentar remover devemos verificar se a posição está ocupada. Não faz sentido remover algo que não existe. Depois, basta “avançar” a referência que aponta para o primeiro nó.

Por fim, é importante perceber que a Lista pode ficar vazia. Neste caso, devemos colocar null na referência que aponta para o último nó.

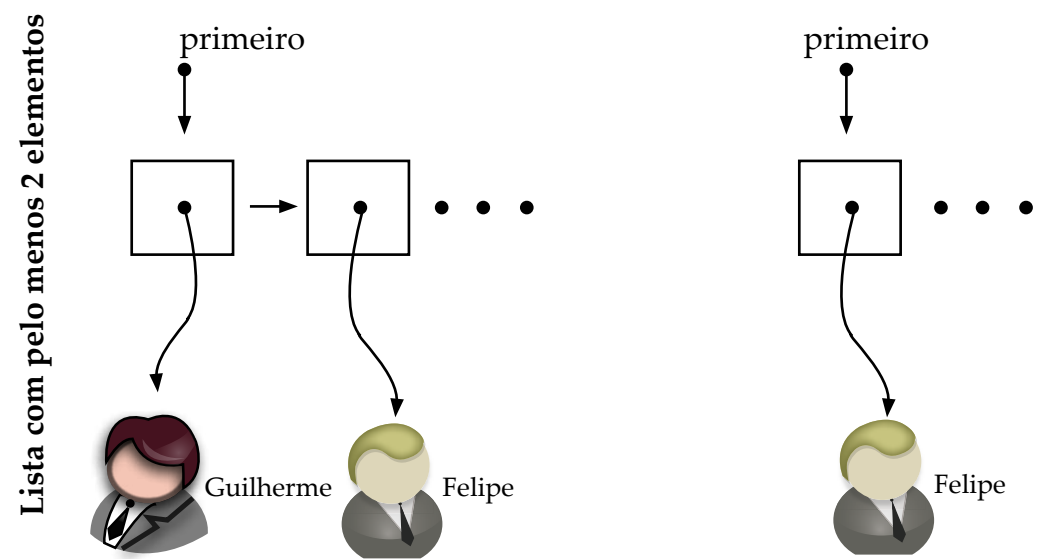
Se não fizermos isso ficaríamos em um estado inconsistente, em que o atributo primeiro é null e o último não, ou seja, tem um último, mas não tem um primeiro, algo que não faria sentido.

FIGURA 100 – REMOVE DO COMEÇO – LISTA COM APENAS UM ELEMENTO



FONTE: O autor

FIGURA 101 – REMOVE DO COMEÇO – LISTA COM PELO MENOS DOIS (2) ELEMENTOS



FONTE: O autor

FIGURA 102 – CÓDIGO PARA REMOVER DO COMEÇO DA LISTA

```

54 public void removeDoComeco() {
55     if (!this.posicaoOcupada(0)) {
56         throw new IllegalArgumentException("Posição não existe");
57     }
58     this.primeiro = this.primeiro.getProximo();
59     this.totalDeElementos--;
60     if (this.totalDeElementos == 0) {
61         this.ultimo = null;
62     }
63 }

```

FONTE: O autor

2.3.6 Método removeDoFim()

A primeira verificação a ser realizada tem por objetivo constatar se a última posição existe, que poderá ser realizada através do método anteriormente criado **posicaoOcupada(int)**.

Ao constatar que a Lista possui apenas um elemento o processo de remoção do fim será idêntico a remover do começo, possibilitando desta forma a reutilização do método **removeDoComeco()**, nestes casos.

No entanto, se a Lista possui mais que um elemento, devemos pegar o penúltimo nó, fazer o próximo do penúltimo ser null e fazer o atributo ultimo apontar para o penúltimo.

A questão neste momento é: como pegar o penúltimo nó? Podemos fazer isso usando o método **pegaNo(int)**, mas isso consumiria tempo linear, reduzindo consideravelmente o desempenho de nosso programa. Desta forma, como queremos consumo constate teremos que achar outra solução.

Então, em vez de fazer uma Lista Encadeada simples, vamos fazer uma Lista Duplamente Encadeada. Ou seja, cada nó aponta para o seu anterior além de apontar para a próxima. (CAELUM, 2014). Para tanto, devemos reabrir a classe No e adicionar o código a seguir.

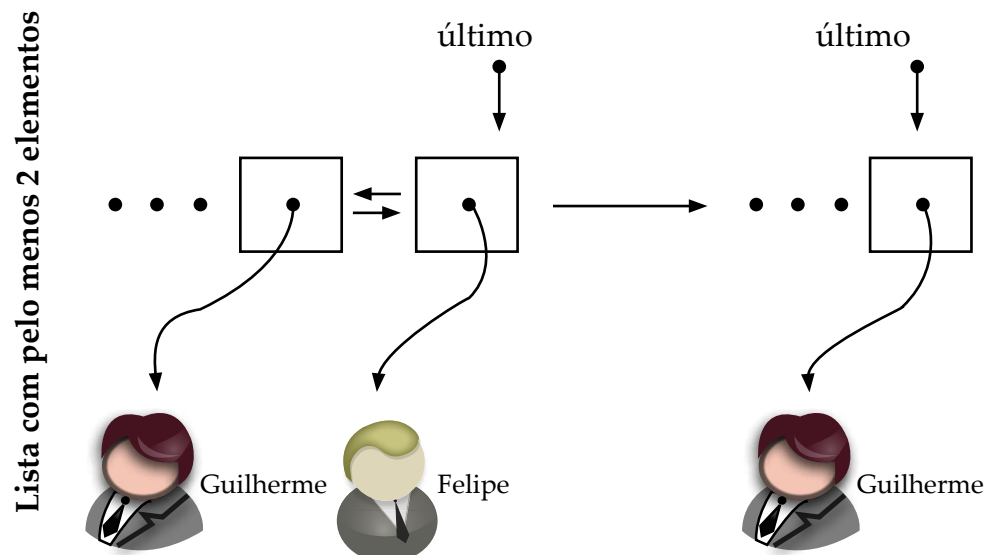
FIGURA 103 – CÓDIGO PARA IMPLEMENTAR A LISTA DUPLAMENTE ENCADEADA

```
3 public class No {
4     private No proxima;
5     private Object elemento;
6     private No anterior;
7
8     public No getAnterior() {
9         return anterior;
10    }
11
12    public void setAnterior(No anterior) {
13        this.anterior = anterior;
14    }
```

FONTE: O autor

Com cada nó sabendo quem é o seu anterior, fica fácil escrever o método **removeDoFim()**.

FIGURA 104 – REMOVE DO FIM – LISTA COM PELO MENOS DOIS (2) ELEMENTOS



FONTE: O autor

FIGURA 105 – CÓDIGO PARA REMOVER DO FIM DA LISTA

```

94 public void removeDoFim() {
95     if (!this.posicaoOcupada(this.totalDeElementos - 1)) {
96         throw new IllegalArgumentException("Posição não existe");
97     }
98     if (this.totalDeElementos == 1) {
99         this.removeDoComeco();
100     } else {
101         No novo = new No(elemento);
102         this.ultimo.setProximo(novo);
103         novo.setAnterior(this.ultimo);
104         this.ultimo = novo;
105         No penultima = this.ultimo.getAnterior();
106         penultima.setProximo(null);
107         this.ultimo = penultima;
108         this.totalDeElementos--;
109     }
110 }

```

FONTE: O autor

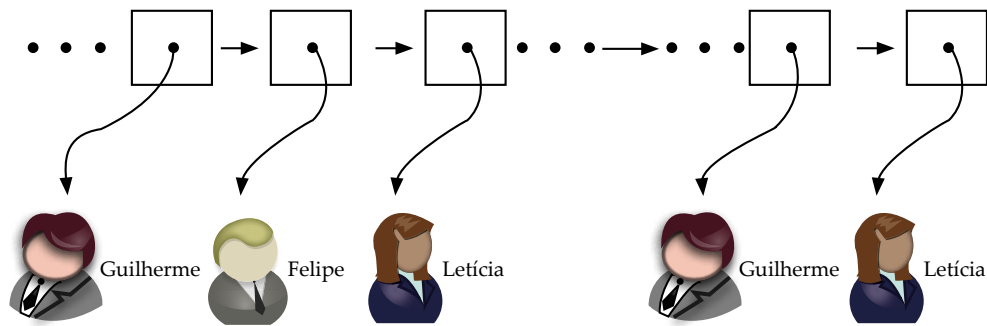
A modificação para Lista Duplamente Encadeada implicará pequenas modificações nos outros métodos que já tínhamos implementado, que serão apresentadas no Tópico a seguir.

2.3.7 Método remove()

O respectivo método possibilitará a remoção do elemento de qualquer posição da Lista. Contudo, inicialmente devemos verificar se a posição está ou não ocupada. Se não estiver ocupada, devemos lançar uma exceção, caso contrário, devemos verificar se a remoção é do começo ou do fim da Lista, haja vista que se for um destes casos, basta chamarmos os métodos já implementados **removeDoComeco** ou **removeDoFim**.

Por fim, se a remoção é no interior da Lista, devemos atualizar as referências dos nós relacionados ao nó que vamos remover (anterior e próximo). O próximo do **anterior** deve ser o **proximo** e o anterior do **proximo** deve ser o **anterior**.

FIGURA 106 – REMOVE DO INTERIOR DA LISTA



Lista com pelo menos 3 elementos

FONTE: O autor

FIGURA 107 – CÓDIGO PARA REMOVER DO INTERIOR DA LISTA

```
50 public void remove(int posicao){
51     if (!this.posicaoOcupada(posicao)) {
52         throw new IllegalArgumentException("Posição não existe");
53     }
54     if (posicao == 0) {
55         this.removeDoComeco();
56     } else if (posicao == this.totalDeElementos - 1) {
57         this.removeDoFim();
58     } else {
59         No anterior = this.pegarNo(posicao - 1);
60         No atual = anterior.getProximo();
61         No proximo = atual.getProximo();
62         anterior.setProximo(proximo);
63         proximo.setAnterior(anterior);
64         this.totalDeElementos--;
65     }
66 }
```

FONTE: O autor

2.3.8 Método contém()

Esta operação deve percorrer a Lista e comparar com o método **equals(Object)** o elemento procurado contra todos os elementos da Lista.

FIGURA 108 – CÓDIGO PARA VERIFICAR SE UM ELEMENTO ESTÁ NA LISTA

```

72 public boolean contem(Object elemento) {
73     No atual = this.primeiro;
74     while (atual != null) {
75         if (atual.getElemento().equals(elemento)) {
76             return true;
77         }
78         atual = atual.getProximo();
79     }
80     return false;
81 }

```

FONTE: O autor

2.3.9 Método tamanho()

Está operação não tem segredo, pois já definimos um atributo que possui esta informação, qual seja **totalDeElementos**.

FIGURA 109 – CÓDIGO PARA RETORNAR O TAMANHO DA LISTA

```

67 public int tamanho() {
68     return this.totalDeElementos;
69 }

```

FONTE: O autor

2.4 VERIFICANDO O FUNCIONAMENTO DOS MÉTODOS

Para que possamos verificar se os métodos anteriormente definidos estão funcionando corretamente, iremos implementar algumas classes que nos permitirão fazer a inclusão e exclusão de pessoas em nossa Lista Encadeada.

Todavia, para que esses métodos consigam imprimir o conteúdo da nossa lista, devemos primeiramente reescrever o método `toString()`, o qual permitirá visualizar facilmente o conteúdo da Lista. Utilizamos a classe `StringBuilder` para construir a `String` que mostrará os elementos da Lista.

FIGURA 110 – CÓDIGO PARA RETORNAR UMA STRING

```

149 @Override
150 public String toString() {
151     StringBuilder sb = new StringBuilder();
152     No temp = primeiro;
153
154     for(int i = 0; i < this.totalDeElementos; i++){
155         sb.append(temp.getElemento()+" ");
156         temp = temp.getProximo();
157     }
158     return sb.toString();
159 }

```

FONTE: O autor

2.4.1 Classe TesteAdicionaNoFim

Crie a classe TesteAdicionaNoFim, conforme a figura a seguir, a qual possibilitará adicionarmos pessoas em nossa Lista Encadeada.

FIGURA 111 – IMPLEMENTAÇÃO DA CLASSE TesteAdicionaNoFim

```

3 public class TesteAdicionaNoFim {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         System.out.println(lista);
9     }
10 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Guilherme, Felipe]

2.4.2 Classe TesteAdicionaPorPosicao

Crie a classe TesteAdicionaPorPosicao, conforme a figura a seguir, a qual possibilitará adicionarmos pessoas em nossa Lista Encadeada, conforme a posição informada.

FIGURA 112 – IMPLEMENTAÇÃO DA CLASSE TesteAdicionaPorPosicao

```

3 public class TesteAdicionaPorPosicao {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adicionaPorPosicao(0, "Felipe");
8         lista.adicionaPorPosicao(1, "Letícia");
9         System.out.println(lista);
10    }
11 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Felipe, Letícia, Guilherme]

2.4.3 Classe TestePegaPorPosicao

Crie a classe TestePegaPorPosicao, conforme a figura a seguir, a qual possibilitará pegarmos uma pessoa em nossa Lista Encadeada, de acordo com a posição informada.

FIGURA 113 – IMPLEMENTAÇÃO DA CLASSE TestePegaPorPosicao

```

3 public class TestePegaPorPosicao {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         lista.adiciona("Letícia");
9         System.out.println(lista.pegar(1));
10    }
11 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Felipe]

2.4.4 Classe TesteRemovePorPosicao

Crie a classe TesteRemovePorPosicao, conforme a figura a seguir, a qual possibilitará removermos uma pessoa em nossa Lista Encadeada, de acordo a posição informada.

FIGURA 114 – IMPLEMENTAÇÃO DA CLASSE TesteRemovePorPosicao

```

3 public class TesteRemovePorPosicao {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         lista.adiciona("Letícia");
9         lista.remove(1);
10        System.out.println(lista);
11    }
12 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Guilherme, Letícia]

2.4.5 Classe TesteTamanho

Crie a classe TesteTamanho, conforme a figura a seguir, a qual possibilitará visualizarmos a quantidade de elementos que compõem nossa Lista Encadeada.

FIGURA 115 – IMPLEMENTAÇÃO DA CLASSE TesteTamanho

```

3 public class TesteTamanho {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         System.out.println(lista.tamanho());
9         lista.adiciona("Letícia");
10        System.out.println(lista.tamanho());
11    }
12 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:

2
3

2.4.6 Classe TesteContemElemento

Crie a classe TesteContemElemento, conforme a figura a seguir, a qual possibilitará pesquisarmos se determinado elemento encontra-se armazenado em nossa Lista Encadeada.

FIGURA 116 – IMPLEMENTAÇÃO DA CLASSE TesteContemElemento

```

3 public class TesteContemElemento {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         System.out.println(lista.contem("Guilherme"));
9         System.out.println(lista.contem("Letícia"));
10    }
11 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:

true
false

2.4.7 Classe TesteAdicionaNoComeço

Crie a classe TesteAdicionaNoComeço, conforme a figura a seguir, a qual permitirá inserir um novo elemento no começo da nossa Lista Encadeada.

FIGURA 117 – IMPLEMENTAÇÃO DA CLASSE TesteAdicionaNoComeco

```

3 public class TesteAdicionaNoComeco {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adicionaNoComeco("Guilherme");
7         lista.adicionaNoComeco("Felipe");
8         System.out.println(lista);
9     }
10 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Felipe, Guilherme]

2.4.8 Classe TesteRemoveDoComeco

Crie a classe TesteRemoveDoComeco, conforme a figura a seguir, a qual permitirá remover um elemento no começo da nossa Lista Encadeada.

FIGURA 118 – IMPLEMENTAÇÃO DA CLASSE TesteRemoveDoComeco

```

3 public class TesteRemoveDoComeco {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         lista.removeDoComeco();
9         System.out.println(lista);
10    }
11 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Felipe]

2.4.9 Classe TesteRemoveDoFim

Crie a classe TesteRemoveDoFim, conforme a figura a seguir, a qual permitirá remover o último elemento da nossa Lista Encadeada.

FIGURA 119 – IMPLEMENTAÇÃO DA CLASSE TesteRemoveDoFim

```

3 public class TesteRemoveDoFim {
4     public static void main(String[] args) {
5         ListaEncadeada lista = new ListaEncadeada();
6         lista.adiciona("Guilherme");
7         lista.adiciona("Felipe");
8         lista.adiciona("Letícia");
9         lista.removeDoFim();
10        System.out.println(lista);
11    }
12 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:
[Guilherme, Felipe]

2.5 CÓDIGO COMPLETO DA CLASSE LISTA ENCADEADA

Disponibilizamos a seguir o código completo da classe Lista Encadeada.

FIGURA 120 - COMPLETO DA CLASSE LISTA ENCADEADA

```

package exemplo;

public class ListaEncadeada {
    private No primeiro;
    private No ultimo;

    private int totalDeElementos;

    private Object elemento;

    public void adicionaNoComeco(Object elemento) {
        No novo = new No (this.primeiro, elemento);
        this.primeiro = novo;

        if (this.totalDeElementos == 0) {
            //caso especial da lista vazia
            this.ultimo = this.primeiro;
        }
        this.totalDeElementos++;
    }
}

```

```

public void adiciona(Object elemento) {
    if (this.totalDeElementos == 0) {
        this.adicionaNoComeco(elemento);
    } else {
        No novo = new No(elemento);
        this.ultimo.setProximo(novo);
        this.ultimo = novo;
        this.totalDeElementos++;
    }
}

public void adicionaPosicao(int posicao, Object elemento) {
    if(posicao == 0){ // No começo.
        this.adicionaNoComeco(elemento);
    } else if(posicao == this.totalDeElementos){ // No fim.
        this.adiciona(elemento);
    } else {
        No anterior = this.pegaNo(posicao - 1);
        No novo = new No(anterior.getProximo(),
elemento);

        anterior.setProximo(novo);
        this.totalDeElementos++;
    }
}

public Object pega(int posicao) {
    return this.pegaNo(posicao).getElemento();
}

public void remove(int posicao){
    if (!this.posicaoOcupada(posicao)) {
        throw new IllegalArgumentException("Posição não
existe");
    }
    if (posicao == 0) {
        this.removeDoComeco();
    } else if (posicao == this.totalDeElementos - 1) {
        this.removeDoFim();
    } else {
        No anterior = this.pegaNo(posicao - 1);
        No atual = anterior.getProximo();
        No proximo = atual.getProximo();
        anterior.setProximo(proximo);
        proximo.setAnterior(anterior);
        this.totalDeElementos--;
    }
}

```

```

    }
}

public int tamanho() {
    return this.totalDeElementos;
}

public boolean contem(Object elemento) {
    No atual = this.primeiro;
    while (atual != null) {
        if (atual.getElemento().equals(elemento)) {
            return true;
        }
        atual = atual.getProximo();
    }
    return false;
}

public void removeDoComeco() {
    if (!this.posicaoOcupada(0)) {
        throw new IllegalArgumentException("Posição não
existe");
    }
    this.primeiro = this.primeiro.getProximo();
    this.totalDeElementos--;
    if (this.totalDeElementos == 0) {
        this.ultimo = null;
    }
}

public void removeDoFim() {
    if (!this.posicaoOcupada(this.totalDeElementos - 1)) {
        throw new IllegalArgumentException("Posição não
existe");
    }
    if (this.totalDeElementos == 1) {
        this.removeDoComeco();
    } else {
        No novo = new No(elemento);
        this.ultimo.setProximo(novo);
        novo.setAnterior(this.ultimo);
        this.ultimo = novo;
        No penultima = this.ultimo.getAnterior();
        penultima.setProximo(null);
        this.ultimo = penultima;
    }
}

```

```

        this.totalDeElementos--;
    }
}

private boolean posicaoOcupada(int posicao){
    return posicao >= 0 && posicao < this.totalDeElementos;
}

private No pegaNo(int posicao) {
    if(!this.posicaoOcupada(posicao)){
        throw new IllegalArgumentException("Posição não
existe");
    }
    No atual = primeiro;
    for (int i = 0; i < posicao; i++) {
        atual = atual.getProximo();
    }
    return atual;
}

public No getPrimeiro() {
    return primeiro;
}

public void setPrimeiro(No primeiro) {
    this.primeiro = primeiro;
}

public No getUltimo() {
    return ultimo;
}

public void setUltimo(No ultimo) {
    this.ultimo = ultimo;
}

public int getTotalDeElementos() {
    return totalDeElementos;
}

public void setTotalDeElementos(int totalDeElementos) {
    this.totalDeElementos = totalDeElementos;
}

```

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    No temp = primeiro;

    for(int i = 0; i < this.totalDeElementos; i++){
        sb.append(temp.getElemento()+" ");
        temp = temp.getProximo();
    }
    return sb.toString();
}

public Object getElemento() {
    return elemento;
}

public void setElemento(Object elemento) {
    this.elemento = elemento;
}

}
```

FONTE: O autor

RESUMO DO TÓPICO 1

Neste tópico vimos que:

- As estruturas de dados dinâmicas podem crescer e encolher durante a execução.
- As listas encadeadas são coleções de itens de dados “colocados em fila”, sendo que as inserções e exclusões podem ser feitas em qualquer lugar da lista em uma lista encadeada.
- A classe autorreferencial contém uma referência para outro objeto do mesmo tipo de classe. Os objetos autorreferenciais podem ser encadeados entre si para formar estruturas de dados úteis, como listas, filas e pilhas.
- Criar e manter estruturas de dados dinâmicas exige alocação dinâmica de memória – a capacidade de um programa obter mais espaço de memória durante a execução para armazenar novos nós e liberar espaço não mais necessário.
- O limite para alocação dinâmica pode ser tão grande quanto à memória física disponível no computador ou a quantidade de espaço em disco disponível em um sistema de memória virtual. Frequentemente os limites são muito menores, porque a memória disponível no computador deve ser compartilhada entre muitos usuários.
- O operador **new** recebe como operando o tipo do objeto que está sendo dinamicamente alocado e devolve uma referência a um objeto desse tipo recém-criado.
- A lista encadeada é uma coleção linear (isto é, uma sequência) de objetos de classe autorreferencial, denominados nós, conectados por ponteiros de referência.
- A lista encadeada é acessada através de uma referência ao primeiro nó da lista. Cada nó subsequente é acessado através do membro de referência de ponteiro armazenado no nó anterior.
- Por convenção, a referência de ponteiro do último nó de uma lista é configurada como **null** para marcar o fim da lista.
- O nó pode conter dados de qualquer tipo, incluindo objetos de outras classes.
- A lista encadeada é apropriada quando o número de elementos de dados a representar na estrutura de dados é imprevisível.

AUTOATIVIDADE



- 1 Desenvolva um programa que possibilite a inserção de 25 valores do tipo inteiro aleatórios de 0 a 100 em ordem em um objeto de lista encadeada. Ao final o programa deverá calcular a soma dos elementos e a média em ponto flutuante dos elementos.
- 2 Desenvolva um programa que crie um objeto de lista encadeada de 10 caracteres e depois cria um segundo objeto de lista que contém uma cópia da primeira lista, mas na ordem inversa e exiba os dados ao usuário.

LISTA DUPLAMENTE
ENCADEADA

1 INTRODUÇÃO

Para o método responsável por remover o elemento do fim da Lista ter consumo de tempo constante, o último nó deve se referir ao penúltimo. Isso é possível se utilizarmos o esquema de Lista Duplamente Encadeada, no qual os nós possuem referências tanto para a próxima, quanto para o nó anterior (CAELUM, 2014).

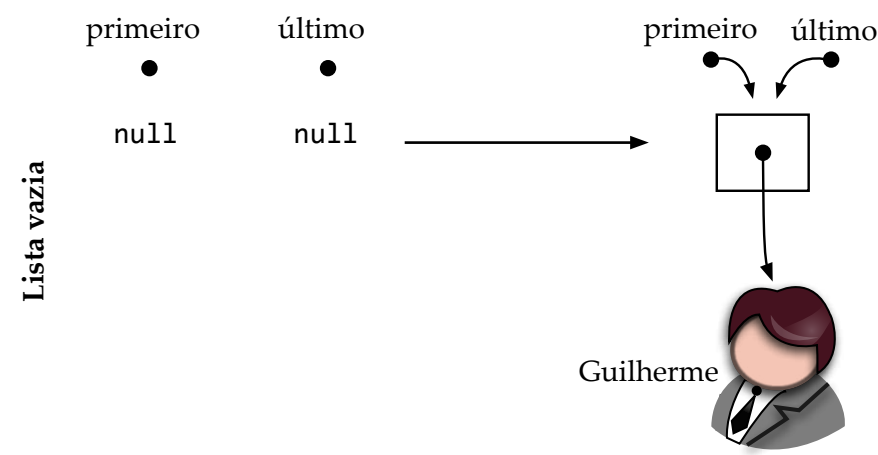
Está modificação implica alterações na implementação dos outros métodos, como já falamos anteriormente, as quais serão vistas a seguir. Para realizar estas mudanças, iremos criar uma nova classe, a qual chamaremos de `ListaDuplamenteEncadeada` e posteriormente copiaremos todos os métodos já definidos na classe `ListaEncadeada` a fim de agilizar nosso trabalho.

2 MÉTODO `ADICIONANOCOMEÇO()`

Para este método devemos considerar dois casos: Lista Vazia e Lista não Vazia, sendo que para o primeiro caso o novo nó será o primeiro e o último. Além disso, ele não terá próximo nem anterior, pois será o único nó.

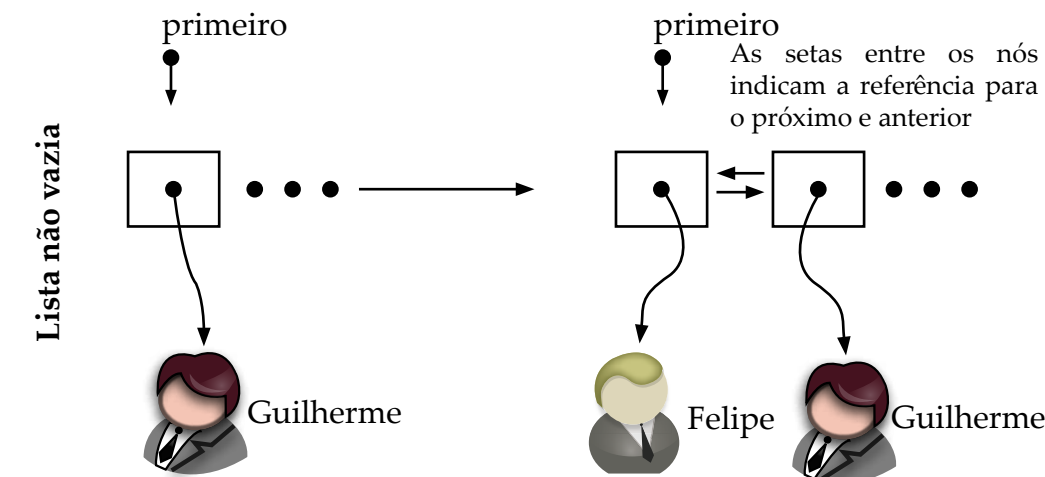
Já no caso da Lista não encontrar-se vazia, será necessário ajustar os ponteiros para o novo segundo (antiga referência primeiro) apontar para o novo primeiro e vice-versa.

FIGURA 121 – ADICIONA NO COMEÇO – LISTA DUPLAMENTE ENCADEADA VAZIA



FONTE: O autor

FIGURA 122 – ADICIONA NO COMEÇO – LISTA DUPLAMENTE ENCADEADA NÃO VAZIA



FONTE: O autor

FIGURA 123 – CÓDIGO PARA ADICIONAR NO COMEÇO DA LISTA DUPLAMENTE ENCADEADA

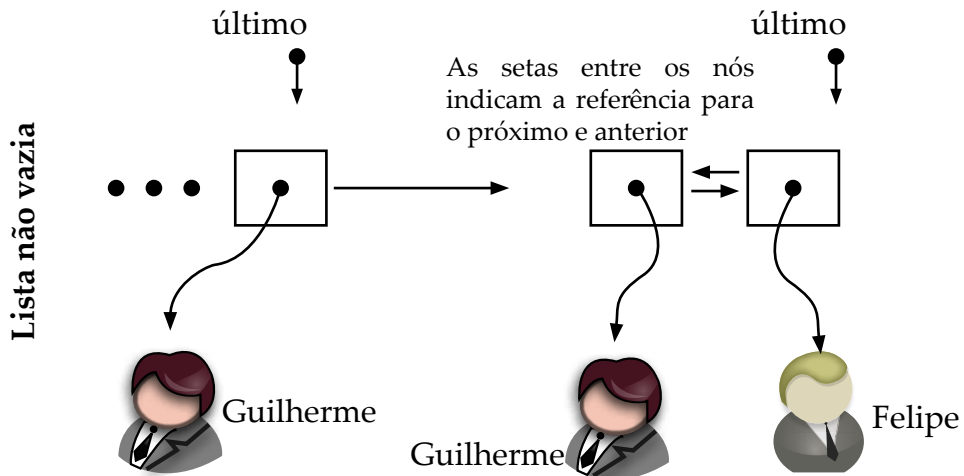
```
11 public void adicionaNoComeco(Object elemento) {
12     if(this.totalDeElementos == 0){
13         No novo = new No(elemento);
14         this.primeiro = novo;
15         this.ultimo = novo;
16     } else {
17         No novo = new No(this.primeiro, elemento);
18         this.primeiro.setAnterior(novo);
19         this.primeiro = novo;
20     }
21     this.totalDeElementos++;
22 }
```

FONTE: O autor

3 MÉTODO ADICIONA()

Ao utilizar o método adiciona(), a pessoa será adicionada no último nó da lista. No caso em que a Lista está vazia, adicionar no fim é a mesma coisa que adicionar no começo. Agora, caso a Lista não esteja vazia então devemos ajustar as referências de tal forma que o novo último nó aponte para o novo penúltimo (antigo último) e vice-versa.

FIGURA 124 – ADICIONA NO FIM – LISTA DUPLAMENTE ENCADEADA NÃO VAZIA



FONTE: O autor

FIGURA 125 – CÓDIGO PARA ADICIONAR NO FIM DA LISTA DUPLAMENTE ENCADEADA

```

24 public void adiciona(int posicao, Object elemento) {
25     if(posicao == 0){ // No começo.
26         this.adicionaNoComeco(elemento);
27     } else if(posicao == this.totalDeElementos){ // No fim.
28         this.adiciona(posicao, elemento);
29     } else {
30         No anterior = this.pegarNo(posicao - 1);
31         No proximo = anterior.getProximo();
32         No novo = new No(anterior.getProximo(), elemento);
33         novo.setAnterior(anterior);
34         anterior.setProximo(novo);
35         proximo.setAnterior(novo);
36         this.totalDeElementos++;
37     }
38 }

```

FONTE: O autor

4 MÉTODO ADICIONAPOSICAO()

Separamos os casos em que a inserção é no começo ou no fim porque podemos reaproveitar os métodos já implementados.

Resta a situação em que a inserção é no meio da Lista, ou seja, entre dois nós existentes. Neste caso, devemos ajustar as referências para o novo nó ser apontado corretamente pelos dois nós relacionados a ele (o anterior e o próximo). E também fazer o novo nó apontar para o anterior e o próximo (Caelum, 2014).

Para tanto, precisaremos utilizar os métodos pegaNo e posicaoOcupada, os quais já foram definidos na classe ListaEncadeada e serão reaproveitados em sua íntegra, sem sofrer qualquer alteração.

FIGURA 126 – CÓDIGO PARA ADICIONAR EM UMA POSIÇÃO DA LISTA DUPLAMENTE ENCADEADA

```

40 public void adicionaPosicao(int posicao, Object elemento) {
41     if(posicao == 0){ // No começo.
42         this.adicionaNoComeco(elemento);
43     } else if(posicao == this.totalDeElementos){ // No fim.
44         this.adiciona(posicao, elemento);
45     } else {
46         No anterior = this.pegaNo(posicao - 1);
47         No novo = new No(anterior.getProximo(), elemento);
48         anterior.setProximo(novo);
49         this.totalDeElementos++;
50     }
51 }

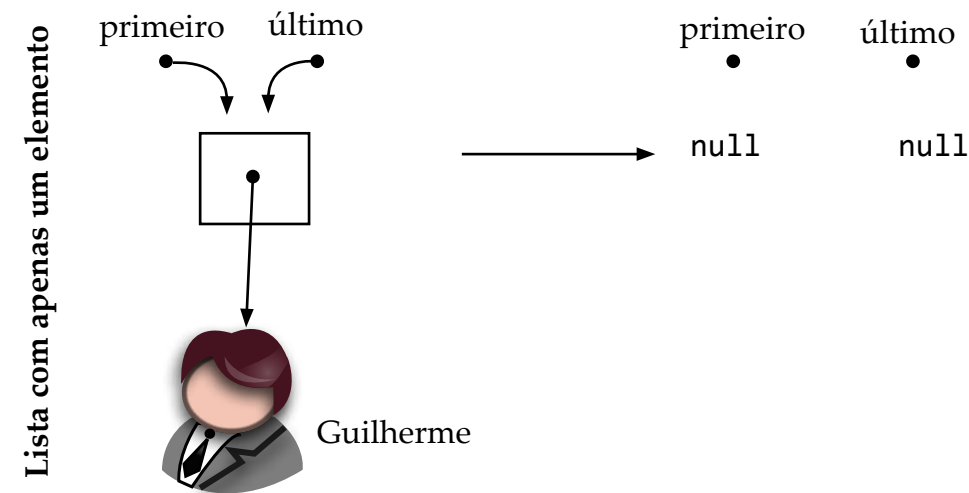
```

FONTE: O autor

5 MÉTODO REMOVEDOCOMEÇO()

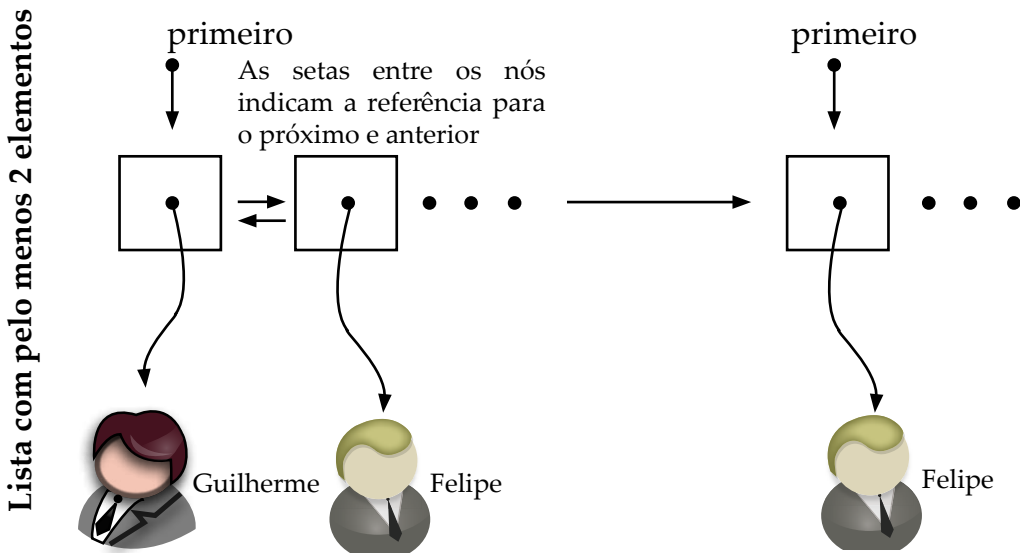
“Esta operação é idêntica em ambos os tipos de Lista Encadeada (simples ou dupla). Ela apenas deve avançar a referência primeira para o segundo nó e tomar cuidado com o caso da Lista ficar vazia, pois, neste caso, a referência último deve ser atualizada também” (CAELUM, 2014, p. 34).

FIGURA 127 – REMOVE DO COMEÇO – LISTA DUPLAMENTE ENCADEADA COM APENAS UM ELEMENTO



FONTE: O autor (2014)

FIGURA 128 – REMOVE DO COMEÇO – LISTA DUPLAMENTE ENCADEADA COM PELO MENOS DOIS ELEMENTOS

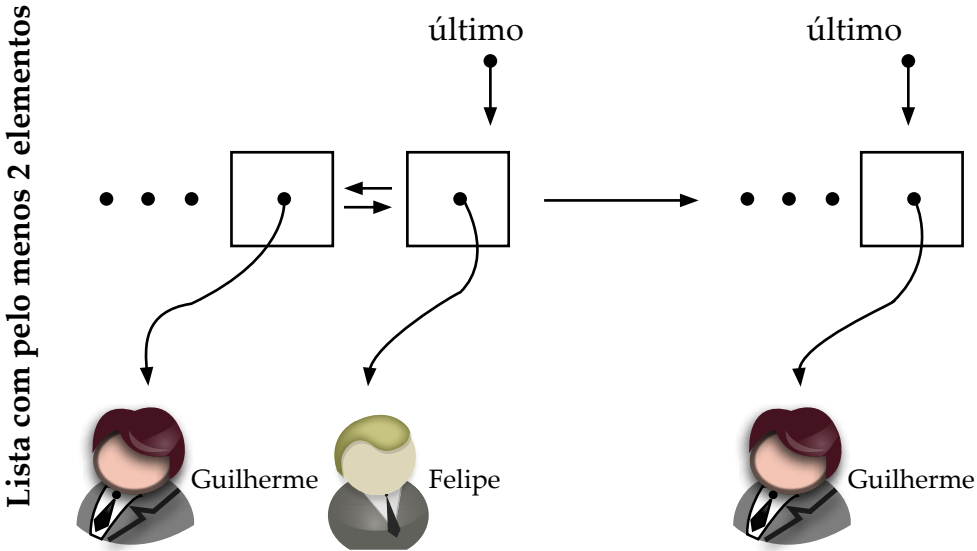


FONTE: O autor

6 MÉTODOS REMOVEDOFIM() E REMOVE()

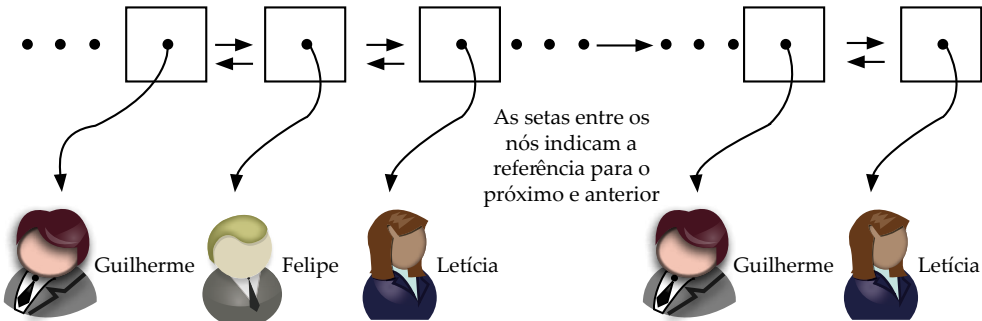
Estes dois métodos já foram tratados e implementados anteriormente usando o esquema de Lista Duplamente Encadeada.

FIGURA 49 – REMOVE DO FIM – LISTA DUPLAMENTE ENCADEADA



FONTE: O autor

FIGURA 130 – REMOVE DO INTERIOR DA LISTA DUPLAMENTE ENCADEADA



Lista com pelo menos 3 elementos

FONTE: O autor



Dentro da biblioteca da linguagem Java dispomos da classe `LinkedList`, a qual faz o papel da nossa Lista Encadeada, possuindo os mesmos métodos que a `ArrayList`, e adiciona alguns outros, como o `addFirst(Object)`, `removeFirst()`, `addLast(Object)`, `removeLast()`, que operam no começo e no fim da Lista em tempo constante.



RESUMO DO TÓPICO 2

Neste tópico vimos que:

- A lista duplamente encadeada visa suprir a ineficiência de algumas operações da lista simplesmente encadeada, como, por exemplo, a remoção do último elemento ou um elemento intermediário, já que é necessário percorrer toda lista para encontrar o elemento anterior.
- Na lista duplamente encadeada, cada nó armazena o elemento, o ponteiro para o próximo nó e para o anterior.

AUTOATIVIDADE



- 1 Desenvolva uma função que possibilite a concatenação de duas listas duplamente encadeadas.
- 2 Desenvolva uma função que possibilite a cópia dos dados de um vetor para uma lista duplamente encadeada.

1 INTRODUÇÃO

Um determinado produto é composto por diversas peças (digamos p_1, p_2, \dots, p_n). O processo de montagem deste produto é automático (executado por uma máquina) e exige que as peças sejam colocadas em uma ordem específica (primeiro a p_1 , depois a p_2 , depois a p_3 e assim por diante).

As peças são empilhadas na ordem adequada e a máquina de montagem vai retirando peça por peça do topo desta pilha para poder montar o produto final.

A mesma máquina que faz a montagem é capaz de trocar uma peça quebrada de um produto já montado.

O que a máquina faz é desmontar o produto até chegar à peça defeituosa, trocá-la e então depois recolocar as peças que foram retiradas. Isso também é feito com o uso da pilha de peças. Veja a seguir o algoritmo que a máquina montadora implementa para fazer a manutenção de um produto com defeito.

- 1) Retirar e empilhar peça por peça do produto até chegar na peça defeituosa.
- 2) Retirar a peça defeituosa.
- 3) Colocar uma peça nova sem defeitos.
- 4) Desempilhar e montar peça por peça do topo da pilha até a pilha ficar vazia.

FONTE: Caelum (2014, p. 1)

2 O CONCEITO DE PILHAS

A estrutura de dados denominada pilha admite a remoção e inserção de novos elementos de forma dinâmica na memória, sujeitando-se a seguinte regra de operação: o elemento a ser removido será sempre o que está na estrutura há menos tempo.

A pilha é uma versão limitada de uma lista encadeada – novos nodos podem ser adicionados a uma pilha e removidos de uma pilha apenas na parte superior (topo). Por essa razão, a pilha é conhecida como uma estrutura de dados último a entrar, primeiro a sair (*last-in, first-out* - LIFO). O membro de *link* do nodo inferior (isto é, o último) da pilha é configurado como nulo para indicar a base da pilha. (DEITEL E DEITEL, 2003, p. 56).

Neste sentido, as pilhas têm muitas aplicações interessantes, principalmente na análise de expressões e sintaxe, como no caso das calculadoras que utilizam a Notação Polonesa Reversa, que implementam a estrutura de dados de pilha para expressar seus valores, podendo ser representadas de forma prefixa, posfixa ou infixa, ou ainda, os compiladores de muitas linguagens de programação ao realizar a análise sintática de expressões e blocos de programas.

3 A IMPLEMENTAÇÃO DA CLASSE PEÇA

De alguma forma uma peça precisa ser representada em nosso programa. Como estamos usando orientação a objetos as peças serão representadas por objetos. Uma classe Java será criada somente para modelar as peças, algo similar ao código a seguir:

FIGURA 131 – CÓDIGO PARA IMPLEMENTAR A CLASSE PEÇA

```

3  public class Peca {
4
5      private String nome;
6
7      public String getNome() {
8          return nome;
9      }
10
11     public void setNome(String nome) {
12         this.nome = nome;
13     }
14 }

```

FONTE: O autor

Com a classe Peca já é possível criar objetos para representar as peças que a máquina montadora utiliza. Porém, o sistema deve definir como armazenar estes objetos, ou seja, ele precisa escolher uma estrutura de dados. Esta estrutura de dados deve manter os dados seguindo alguma lógica e deve fornecer algumas operações para a manipulação destes e outras operações para informar sobre seu próprio estado. (CAELUM, 2014, p. 3).

3.1 SOLUÇÃO DOS PROBLEMAS DAS PEÇAS

Para implementar o algoritmo de manutenção do carro é necessário criar uma estrutura de dados que se comporte como a pilha de peças. Vamos chamar esta estrutura de dados de Pilha.

Primeiro, definimos a interface da Pilha, ou seja, o conjunto de operações que queremos utilizar em uma Pilha.

- 1) Insere uma peça (coloca uma peça no topo da Pilha).
- 2) Remove uma peça (retira a peça que está no topo da Pilha).
- 3) Informa se a Pilha está vazia.

Podemos criar uma classe Pilha para implementar a esta estrutura de dados. Os métodos públicos desta classe serão a implementação das operações.

FONTE: Caelum (2014)

Figura 132 – CÓDIGO PARA IMPLEMENTAR A CLASSE pilha

```

3 public class Pilha {
4     public void insere(Peca peca) {
5         // implementação
6     }
7     public Peca remove() {
8         return null;
9         // implementação
10    }
11    public boolean vazia() {
12        return false;
13        // implementação
14    }
15 }

```

FONTE: O autor

O primeiro fato importante que devemos observar é que uma vez que a interface da Pilha foi definida, já saberíamos usar a classe Pilha. Vamos criar uma classe de teste bem simples para exemplificar o uso de uma Pilha (CAELUM, 2014).

FIGURA 133 – CÓDIGO PARA IMPLEMENTAR A CLASSE DE TESTE DA PILHA

```

3 public class TestePilha {
4     public static void main(String[] args) {
5         Pilha pilha = new Pilha();
6         Peca pecaInsere = new Peca();
7         pilha.insere(pecaInsere);
8         @SuppressWarnings("unused")
9         Peca pecaRemove = pilha.remove();
10        if (pilha.vazia()) {
11            System.out.println("A pilha está vazia");
12        }
13    }
14 }

```

FONTE: O autor

O segundo fato importante é que a estrutura que queremos aqui é muito similar as Listas que vimos anteriormente. A semelhança fundamental entre as Listas e as Pilhas é que ambas devem armazenar os elementos de maneira sequencial. Este fato é o ponto-chave deste Tópico.

Qual é a diferença entre uma Lista e uma Pilha? A diferença está nas operações destas duas estruturas de dados. As operações de uma Pilha são mais restritas do que as de uma Lista. Por exemplo, você pode adicionar ou remover um elemento em qualquer posição de uma Lista, mas em uma Pilha você só pode adicionar ou remover do topo.

Então, uma Lista é uma estrutura mais poderosa e mais genérica do que uma Pilha. A Pilha possui apenas um subconjunto de operações da Lista. Então o interessante é que para implementar uma Pilha podemos usar uma Lista. Isso mesmo! Vamos criar restrições sobre as operações da Lista e obteremos uma Pilha.

Nós implementamos dois tipos de Listas: Vetores e Listas Encadeadas. Vimos, também que, na biblioteca do Java, há implementações prontas para estes dois tipos de Listas. Neste capítulo, vamos utilizar a classe `LinkedList` para armazenar as peças que serão guardadas na Pilha.

FONTE: Caelum (2014)

O segundo fato importante é que a estrutura que queremos aqui é muito similar as Listas que vimos anteriormente. A semelhança fundamental entre as Listas e as Pilhas é que ambas devem armazenar os elementos de maneira sequencial. Este fato é o ponto-chave deste Tópico.

Qual é a diferença entre uma Lista e uma Pilha? A diferença está nas operações destas duas estruturas de dados. As operações de uma Pilha são mais restritas do que as de uma Lista. Por exemplo, você pode adicionar ou remover um elemento em qualquer posição de uma Lista, mas em uma Pilha você só pode adicionar ou remover do topo.

Então, uma Lista é uma estrutura mais poderosa e mais genérica do que uma Pilha. A Pilha possui apenas um subconjunto de operações da Lista. Então o interessante é que para implementar uma Pilha podemos usar uma Lista. Isso mesmo! Vamos criar restrições sobre as operações da Lista e obteremos uma Pilha.

Nós implementamos dois tipos de Listas: Vetores e Listas Encadeadas. Vimos, também que, na biblioteca do Java, há implementações prontas para estes dois tipos de Listas. Neste capítulo, vamos utilizar a classe `LinkedList` para armazenar as peças que serão guardadas na Pilha.

FONTE: Caelum (2014)

Para tanto, será indispensável que a classe Pilha faça a importação das referidas bibliotecas do Java, conforme demonstra a figura a seguir:

FIGURA 134 – CÓDIGO PARA IMPLEMENTAR A CLASSE PILHA UTILIZANDO AS BIBLIOTECAS DO JAVA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Pilha {
7     @SuppressWarnings("unused")
8     private List<Peca> pecas = new LinkedList<Peca>();
9 }

```

FONTE: O autor

Dentro de nossa Pilha teremos uma `LinkedList` encapsulada, que vai simplificar bastante o nosso trabalho: delegaremos uma série de operações para essa Lista Ligada, porém sempre pensando nas diferenças essenciais entre uma Pilha e uma Lista.

Devemos ter um `getPecas()` que devolve uma referência para essa nossa `LinkedList`? Nesse caso, a resposta é não, pois estaríamos expondo detalhes de nossa implementação, e o usuário dessa classe poderia mexer no funcionamento interno da nossa pilha, desrespeitando as regras de

sua interface. É sempre uma boa prática expor o mínimo possível do funcionamento interno de uma classe, gera um menor acoplamento entre as classes.

FONTE: Caelum (2014)

3.2 OPERAÇÕES EM PILHAS

Agora vamos implementar as operações da Pilha.

3.2.1 Inserir uma peça

As peças são sempre inseridas no topo da Pilha. Ainda não definimos onde fica o topo da Pilha. Como estamos utilizando uma Lista para armazenar os elementos então o topo da Pilha poderia ser tanto o começo ou o fim da Lista. Aqui escolheremos o fim da Lista. Então, inserir na Pilha é simplesmente adicionar no fim da Lista. (CAELUM, 2014, p. 11).

FIGURA 135 – CÓDIGO PARA IMPLEMENTAR O MÉTODO ADICIONA NA PILHA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Pilha {
7     private List<Peca> pecas = new LinkedList<Peca>();
8
9     public void insere(Peca peca) {
10         this.pecas.add(peca);
11     }
12 }

```

FONTE: O autor

Recordando que o método `add(Object)` adiciona no fim da Lista.

3.2.2 Remover uma peça

A remoção também é bem simples, basta retirar o último elemento da Lista.

FIGURA 136 – CÓDIGO PARA IMPLEMENTAR O MÉTODO REMOVE NA PILHA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Pilha {
7     private List<Peca> pecas = new LinkedList<Peca>();
8
9     public void insere(Peca peca) {}
10
11
12
13     public Peca remove() {
14         return this.pecas.remove(this.pecas.size() - 1);
15     }
16 }

```

FONTE: O autor

“É bom observar que se o método remove() for usado com a Pilha vazia então uma exceção será lançada, pois o método remove(int) da List lança IndexOutOfBoundsException quando não existir elemento para remover”. (CAELUM, 2014, p. 12).

3.2.3 Informar se a Pilha está vazia

Para implementar esta operação basta verificar se o tamanho da Lista é zero. Contudo, o método isEmpty() já implementado e disponível na classe LinkedList é a forma mais convenientemente para verificar se a pilha encontra-se vazia.

FIGURA 137 – CÓDIGO PARA IMPLEMENTAR O MÉTODO PARA VERIFICAR SE A PILHA ESTÁ VAZIA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Pilha {
7     private List<Peca> pecas = new LinkedList<Peca>();
8
9     public void insere(Peca peca) {}
10
11
12
13     public Peca remove() {}
14
15
16
17     public boolean vazia() {
18         return this.pecas.isEmpty();
19     }
20 }

```

FONTE: O autor

3.2.4 Generalização

Nossa Pilha só funciona para armazenar objetos da classe Peca. Vamos generalizar a Pilha para poder armazenar qualquer tipo de objeto. Isso será feito utilizando a classe Object da qual todas as classes derivam direta ou indiretamente. Criaremos uma LinkedList de Object em vez de uma LinkedList de Peca. (CAELUM, 2014).

FIGURA 138 – CÓDIGO PARA IMPLEMENTAR A CLASSE PILHA GENÉRICA

```
3+ import java.util.LinkedList;
5
6 public class Pilha {
7     private List<Object> objetos = new LinkedList<Object>();
8
9     public void insere(Object objeto) {
10         this.objetos.add(objeto);
11     }
12
13     public Object remove() {
14         return this.objetos.remove(this.objetos.size() - 1);
15     }
16
17     public boolean vazia() {
18         return this.objetos.isEmpty();
19     }
20 }
```

FONTE: O autor

Agora, podemos armazenar qualquer tipo de objeto na Pilha. Isso é uma grande vantagem, pois a classe Pilha poderá ser reaproveitada em diversas ocasiões. Mas, há uma desvantagem, quando removemos um elemento da Pilha não podemos garantir qual é o tipo de objeto que virá.

No Java 5 poderíamos usar Generics para solucionar este problema. A nossa classe Pilha poderia ser uma classe parametrizada. Na criação de um objeto de uma classe parametrizada é possível dizer com qual tipo de objeto que queremos trabalhar.

FIGURA 139 – CÓDIGO PARA IMPLEMENTAR A CLASSE PILHA PARAMETRIZADA

```
3 import java.util.LinkedList;
4
5 public class Pilha<T> {
6
7     private LinkedList<T> objetos = new LinkedList<T>();
8
9     public void insere(T t) {
10         this.objetos.add(t);
11     }
12
13     public T remove() {
14         return this.objetos.remove(this.objetos.size() - 1);
15     }
16
17     public boolean vazia() {
18         return this.objetos.isEmpty();
19     }
20 }
```

FONTE: O autor

Poderíamos também adicionar outros métodos, provavelmente úteis a manipulação de uma pilha, como saber o tamanho da pilha, espiar um elemento em determinada posição, entre outros. Vamos testar a classe Pilha que usa Generics.

FIGURA 140 – CÓDIGO PARA IMPLEMENTAR A CLASSE TESTEPILHA

```

3 public class TestePilha {
4
5     public static void main(String[] args) {
6         Pilha<Peca> pilha = new Pilha<Peca>();
7         Peca peca = new Peca();
8         pilha.insere(peca);
9         @SuppressWarnings("unused")
10        Peca pecaRemove = pilha.remove();
11        if (pilha.vazia()) {
12            System.out.println("A pilha está vazia");
13        }
14
15        Pilha<String> pilha2 = new Pilha<String>();
16        pilha2.insere("Guilherme");
17        pilha2.insere("Leticia");
18        String guilherme = pilha2.remove();
19        String leticia = pilha2.remove();
20        System.out.println(guilherme);
21        System.out.println(leticia);
22    }
23 }

```

FONTE: O autor

Neste exemplo, criamos duas Pilhas. A primeira vai armazenar só objetos do tipo Peca e a segunda só String. Se você tentar adicionar um tipo de objeto que não corresponde ao que as Pilhas estão guardando então um erro de compilação será gerado para evitar que o programador cometa um erro lógico. (CAELUM, 2014, p. 16).



Na biblioteca do Java existe uma classe que implementa a estrutura de dados que foi vista neste Tópico, esta classe chama-se Stack e será testada pelo código a seguir.

FIGURA 141 – CLASSE STACK

```
3 import java.util.Stack;
4
5 public class Teste {
6     public static void main(String[] args) {
7         Stack<Peca> pilha = new Stack<Peca>();
8         Peca pecaInsere = new Peca();
9         pilha.push(pecaInsere);
10        if (pilha.contains(pecaInsere)){
11            System.out.println(pilha.contains(pecaInsere));
12            @SuppressWarnings("unused")
13            Peca pecaRemove = pilha.pop();
14            if (!pilha.contains(pecaInsere)){
15                System.out.println("A pilha está vazia");
16            }
17        }
18    }
19 }
```

FONTE: O autor

RESUMO DO TÓPICO 3

Neste tópico vimos que:

- As pilhas demonstram-se importantes em compiladores e sistemas operacionais – as inserções e exclusões são feitas apenas em uma extremidade de uma pilha, ou seja, seu topo.
- A pilha é uma versão limitada de uma lista encadeada – novos nós podem ser adicionados e removidos apenas da parte superior de uma pilha. A pilha é conhecida como uma estrutura de dados “último a entrar, primeiro a sair” (LIFO).
- Os métodos primários utilizados para manipular uma pilha são **push** e **pop** (**classe Stack**). O método **push** adiciona um novo nó ao topo da pilha, enquanto o método **pop** remove um nó do topo da pilha e devolve o objeto do nó retirado.
- As pilhas têm muitas aplicações interessantes. Quando é feita uma chamada de método, o método chamado deve saber retornar para seu chamador, assim o endereço de retorno é adicionado à pilha de execução do programa. Se ocorrer uma série de chamadas de métodos, os sucessivos valores de retorno são adicionados à pilha, na ordem último a entrar, primeiro a sair, de modo que cada método possa retornar para seu chamador.
- A técnica de implementar cada método de pilha como uma chamada para um método **List** é chamada de delegação, o método de pilha invocado delega a chamada para o método de **List** adequado.

AUTOATIVIDADE



- 1 Desenvolva um programa que permita a leitura de uma linha de texto e utiliza uma pilha para imprimir as palavras da linha na ordem inversa.
- 2 Uma estrutura de dados caracteriza-se como sendo um objeto do tipo LIFO, assim, é correto afirmar que:
 - a) Inclusões ocorrem no final e exclusões no início.
 - b) Exclusões ocorrem no final e inclusões no início.
 - c) Inclusão de novos elementos, assim como a exclusão se processa sempre no final do objeto.
 - d) Exclusões e inclusões ocorrem no início (primeiro elemento).
 - e) Podem ocorrer exclusões e inclusões em qualquer extremidade do objeto.

1 INTRODUÇÃO

No dia a dia, estamos acostumados com as filas em diversos lugares: nos bancos, nos mercados, nos hospitais, nos cinemas entre outros. As filas são importantes, pois elas determinam a ordem de atendimento das pessoas.

As pessoas são atendidas conforme a posição delas na fila. O próximo a ser atendido é o primeiro da fila. Quando o primeiro da fila é chamado para ser atendido a fila “anda”, ou seja, o segundo passa a ser o primeiro, o terceiro passa a ser o segundo e assim por diante até a última pessoa.

Normalmente, para entrar em uma fila, uma pessoa deve se colocar na última posição, ou seja, no fim da fila. Desta forma, quem chega primeiro tem prioridade.

Neste tópico estamos interessados em desenvolver estrutura de dados com o comportamento das filas. Assim como Listas e Pilhas, as Filas são estruturas de dados que armazenam os elementos de maneira sequencial.

Assim como as Pilhas, as Filas têm operações mais restritas do que as operações das Listas. Nas Filas, os elementos são adicionados na última posição e removidos da primeira posição. Nas Listas, os elementos são adicionados e removidos de qualquer posição.

Então, podemos implementar uma Fila simplesmente colocando as restrições adequadas nas operações de adicionar e remover elementos de uma Lista. Isso é bem parecido ao que fizemos com as Pilhas.

FONTE: Caelum (2014, p. 2)

2 O CONCEITO DE FILAS

As estruturas de dados denominadas Filas, são estruturas do tipo FIFO (*first-in, first-out*), ou seja, o primeiro elemento a ser inserido, será o primeiro a ser retirado, assim sendo, adiciona-se itens no fim e remove-se do início.

A fila é semelhante a uma fila de caixa em um supermercado – a primeira pessoa na fila é atendida primeiro e os outros clientes entram na fila apenas no final e esperam ser atendidos. Os nodos da fila são removidos apenas do início (ou cabeça) da fila e são inseridos somente no final (ou cauda) da fila. (DEITEL E DEITEL, 2003, p. 72).

Segundo Deitel e Deitel (2003), as filas podem ser aplicadas em diversas aplicações computacionais, haja vista que a maioria dos computadores tem apenas um único processador, logo, apenas um aplicativo pode ser atendido por vez. Desta forma, os pedidos dos outros aplicativos são colocados em uma fila, onde cada pedido avança gradualmente para o início da fila à medida que os aplicativos vão sendo atendidos.

São exemplos de uso de fila em um sistema:

- Controle de documentos para impressão (*spool* de impressão).
- Troca de mensagem entre computadores numa rede (pacotes de informações).
- Solicitações de acesso a arquivos em uma rede.

3 INTERFACE DE USO

Em nosso estudo de caso realizaremos uma fila de Pessoas, para tanto, precisaremos criar inicialmente a classe Pessoa, responsável por permitir a criação de objetos para representar as pessoas a serem cadastradas.

FIGURA 142 – CÓDIGO PARA IMPLEMENTAR A CLASSE PESSOA

```

3 public class Pessoa {
4     private String nome;
5
6     public String getNome() {
7         return nome;
8     }
9
10    public void setNome(String nome) {
11        this.nome = nome;
12    }
13
14    public String toString() {
15        return this.nome;
16    }
17
18    public boolean equals(Object o) {
19        Pessoa outra = (Pessoa)o;
20        return this.nome.equals(outra.nome);
21    }
22 }

```

FONTE: O autor

As operações que formam a interface de uso da Fila de pessoas são:

- 1) Insere uma Pessoa (coloca uma pessoa no fim da Fila).
- 2) Remove uma Pessoa (retira a pessoa que está no começo da Fila).
- 3) Informa se a Fila está vazia.

O esboço da classe Fila seria mais ou menos assim:

FIGURA 143 – CÓDIGO PARA IMPLEMENTAR A CLASSE FILA

```

3 public class Fila {
4     public void insere(Pessoa pessoa) {
5         // implementação
6     }
7
8     public Pessoa remove() {
9         return null;
10        // implementação
11    }
12
13    public boolean vazia() {
14        return false;
15        // implementação
16    }
17 }

```

FONTE: O autor

Agora que já temos a interface de uso da Fila definida vamos escrever a classe `TesteFila`, a fim de verificar como ela deveria se comportar.

FIGURA 144 – CÓDIGO PARA IMPLEMENTAR A CLASSE TESTEFILA

```

3 public class TesteFila {
4     public static void main(String[] args) {
5         Fila fila = new Fila();
6         Pessoa pessoa = new Pessoa();
7         fila.insere(pessoa);
8         @SuppressWarnings("unused")
9         Pessoa pessoaRemovida = fila.remove();
10        if (fila.vazia()) {
11            System.out.println("A fila está vazia");
12        }
13    }
14 }

```

FONTE: O autor

Importante ressaltar que a classe `TesteFila` somente funcionará efetivamente, após a realização da implementação de cada método definido na classe `Fila` (Figura 143). Contudo, neste primeiro momento, para que possamos compreender o funcionamento da estrutura de dados de uma Fila, utilizaremos a classe `LinkedList` do pacote `java.util`, para armazenar as pessoas da Fila, a qual já traz os referidos métodos implementados.

FIGURA 145 – IMPLEMENTAÇÃO DA CLASSE FILA A PARTIR DA CLASSE LINKEDLIST

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila {
7     @SuppressWarnings("unused")
8     private List<Pessoa> pessoas = new LinkedList<Pessoa>();
9 }

```

FONTE: O autor

3.1 OPERAÇÕES EM FILA

Na sequência, implementaremos as operações da Fila de pessoa.

3.1.1 Inserir uma pessoa

As pessoas que entram na Fila devem sempre se colocar no fim da mesma. Vamos definir que o fim da Fila é o fim da Lista que estamos utilizando para implementar. Então, entrar na Fila e adicionar no fim da Lista.

FIGURA 146 – CÓDIGO PARA ADICIONAR UMA PESSOA NA FILA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila {
7     private List<Pessoa> pessoas = new LinkedList<Pessoa>();
8
9     public void insere(Pessoa pessoa) {
10         this.pessoas.add(pessoa);
11     }
12 }

```

FONTE: O autor

3.1.2 Remover uma pessoa

A próxima pessoa a ser atendida é sempre a que está no início da Fila. No nosso caso, quem está no início da Fila é a pessoa que está no início da Lista.

Então, basta remover a primeira pessoa.

FIGURA 147 – CÓDIGO PARA REMOVER UMA PESSOA DA FILA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila {
7     private List<Pessoa> pessoas = new LinkedList<Pessoa>();
8
9     public void insere(Pessoa pessoa) {}
10
11     public Pessoa remove() {
12         return this.pessoas.remove(0);
13     }
14 }

```

FONTE: O autor

“É bom observar que se o método `remove()` for usado com a Fila vazia então uma exceção será lançada, pois o método `removeFirst()` lança `IndexOutOfBoundsException` quando não existir elemento para remover”. (CAELUM, 2014, p. 5).

3.1.3 Informar se a Fila está vazia

Para implementar esta operação basta verificar se o tamanho da Lista é zero.

FIGURA 148 – CÓDIGO PARA VERIFICAR SE A FILA ESTÁ VAZIA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila {
7     private List<Pessoa> pessoas = new LinkedList<Pessoa>();
8
9     public void insere(Pessoa pessoa) {}
12
13     public Pessoa remove() {}
16
17     public boolean vazia() {
18         return this.pessoas.size() == 0;
19     }
20 }

```

FONTE: O autor

Ou ainda, poderá utilizar a função **isEmpty**, a qual retornará um valor *booleano*, indicando se a Fila está ou não vazia.

FIGURA 149 – CÓDIGO PARA VERIFICAR SE A FILA ESTÁ VAZIA COM ISEMPY

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila {
7     private List<Pessoa> pessoas = new LinkedList<Pessoa>();
8
9     public void insere(Pessoa pessoa) {}
12
13     public Pessoa remove() {}
16
17     public boolean vazia() {
18         return this.pessoas.isEmpty();
19     }
20 }

```

FONTE: O autor

3.2 GENERALIZAÇÃO

Nossa Fila só funciona para armazenar objetos da classe Pessoa. Vamos generalizá-la para poder armazenar qualquer tipo de objeto. Isso será feito utilizando a classe Object da qual todas as classes derivam direta ou indiretamente. Criaremos uma LinkedList de Object em vez de uma LinkedList de Pessoa. (CAELUM, 2014).

FIGURA 150 – CÓDIGO DA CLASSE FILA GENERALIZADA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila {
7     private List<Object> objetos = new LinkedList<Object>();
8
9     public void insere(Object objeto) {
10         this.objetos.add(objeto);
11     }
12
13     public Object remove() {
14         return this.objetos.remove(0);
15     }
16
17     public boolean vazia() {
18         return this.objetos.isEmpty();
19     }
20 }

```

FONTE: O autor

Agora podemos armazenar qualquer tipo de objeto na Fila. Isso é uma grande vantagem, pois a classe Fila poderá ser reaproveitada em diversas ocasiões. Mas, há uma desvantagem, quando removemos um elemento da Fila não podemos garantir qual é o tipo de objeto que virá. A solução para este problema é utilizar o recurso do Generics. Assim, quando criarmos uma Fila, poderemos definir com qual tipo de objetos ela deve trabalhar. (CAELUM, 2014, p. 16).

Conforme veremos na figura a seguir.

FIGURA 151 - CÓDIGO DA CLASSE FILA GENERALIZADA E PARAMETRIZADA

```

3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Fila<T> {
7     private List<T> objetos = new LinkedList<T>();
8
9     public void insere(T t) {
10         this.objetos.add(t);
11     }
12
13     public T remove() {
14         return this.objetos.remove(0);
15     }
16
17     public boolean vazia() {
18         return this.objetos.isEmpty();
19     }
20 }

```

FONTE: O autor

Vamos criar duas Filas, uma para Pessoa e outra para String.

FIGURA 152 – CLASSE TESTEFILA COM DUAS FILAS DISTINTAS

```

3 public class TesteFila {
4     public static void main(String[] args) {
5         Fila<Pessoa> fila = new Fila<Pessoa>();
6
7         Pessoa pessoa = new Pessoa();
8         fila.insere(pessoa);
9
10        @SuppressWarnings("unused")
11        Pessoa pessoaRemovida = fila.remove();
12
13        if (fila.vazia()) {
14            System.out.println("A fila está vazia");
15        }
16
17        Fila<String> filaDeString = new Fila<String>();
18
19        filaDeString.insere("Guilherme");
20        filaDeString.insere("Letícia");
21
22        String guilherme = filaDeString.remove();
23        String leticia = filaDeString.remove();
24
25        System.out.println(guilherme);
26        System.out.println(leticia);
27    }
28 }

```

FONTE: O autor

Ao executar a classe acima, teremos como saída esperada:

```

[
A fila está vazia
Guilherme
Letícia
]

```

“Em tempo de compilação é verificado o tipo de objetos que estão sendo adicionados na Fila. Portanto, se você tentar inserir um objeto do tipo Pessoa em uma Fila de String um erro de compilação será gerado”. (CAELUM, 2014, p. 17).

FIGURA 153 – INSERÇÃO DE UM OBJETO PESSOA EM UMA FILA DE STRING

```

3 public class TesteFila {
4     public static void main(String[] args) {
5         Pessoa pessoa = new Pessoa();
6         Fila<String> filaDeString = new Fila<String>();
7         // este código não compila
8         filaDeString.insere(Pessoa);
9     }
10 }

```

FONTE: O autor



Na biblioteca do Java, existe uma interface que define a estrutura de dados Fila. Essa interface chama-se Queue, umas das classes que implementam Queue é a LinkedList. O funcionamento fica extremamente semelhante com a implementação que fizemos neste Tópico.

FIGURA 154 – IMPLEMENTAÇÃO QUEUE

```

3 import java.util.LinkedList;
4 import java.util.Queue;
5
6 public class TesteFila {
7     public static void main(String[] args) {
8
9         Queue<Pessoa> fila = new LinkedList<Pessoa>();
10
11         Pessoa pessoa = new Pessoa();
12
13         fila.offer(pessoa);
14
15         @SuppressWarnings("unused")
16         Pessoa pessoaRemovida = (Pessoa)fila.poll();
17
18         if(fila.isEmpty()){
19             System.out.println("A fila está vazia");
20         }
21     }
22 }

```

FONTE: O autor

LEITURA COMPLEMENTAR

O COLLECTIONS FRAMEWORK

“Desde a versão 1.2 do JDK (depois que o renomearam para Java 2 SDK), a plataforma J2SE inclui um framework de coleções (a Collections Framework). Uma coleção é um objeto que representa um grupo de objetos. Um framework de coleções é uma arquitetura unificada para representação e manipulação de coleções, permitindo que elas sejam manipuladas independentemente dos detalhes de sua implementação.

AS PRINCIPAIS VANTAGENS DO COLLECTIONS FRAMEWORK SÃO:

- Redução do esforço de programação, fornecendo estruturas de dados e algoritmos úteis, para que não seja necessário reescrevê-los.
- Aumento da performance, fornecendo implementações de alta performance. Como as várias implementações de cada interface são substituíveis, os programas podem ser facilmente refinados trocando-se as implementações.
- Interoperabilidade entre APIs não relacionadas, estabelecendo uma linguagem comum para passagem de coleções.
- Redução do esforço de aprendizado de APIs, eliminando a necessidade de se aprender várias APIs de coleções diferentes.
- Redução do esforço de projetar e implementar APIs, eliminando a necessidade de se criar APIs de coleções próprias.
- Promover o reuso de software, fornecendo uma interface padrão para coleções e algoritmos para manipulá-los.

O COLLECTIONS FRAMEWORK CONSISTE EM:

- Interfaces de coleções - representam diferentes tipos de coleção, como conjuntos, listas e arrays associativos. Estas interfaces formam a base do *framework*.
- Implementação de uso geral - implementações básicas das interfaces de coleções.
- Implementações legadas - as classes de coleções de versões anteriores, Vector e Hashtable, foram remodeladas para implementar as interfaces de coleções (mantendo a compatibilidade).
- Implementações de wrappers - adicionam funcionalidades, como sincronização, a outras implementações.

- Implementações convenientes - “mini-implementações” de alta performance das interfaces de coleções.
- Implementações abstratas - implementações parciais das interfaces de coleções, para facilitar a criação de implementações personalizadas.
- Algoritmos - métodos estáticos que performam funções úteis sobre coleções, como a ordenação de uma lista.
- Utilitários de Arrays - funções utilitárias para arrays de primitivas e objetos. Estritamente falando, não pertence ao Collections Framework, porém, esta funcionalidade foi adicionada à plataforma Java ao mesmo tempo, e utiliza parte da mesma infraestrutura.

COLEÇÕES

Collection

Interface base para todos os tipos de coleção. Ela define as operações mais básicas para coleções de objetos, como adição (add) e remoção (remove) abstratos (sem informações quanto à ordenação dos elementos), esvaziamento (clear), tamanho (size), conversão para array (toArray), objeto de iteração (iterator), e verificações de existência (contains e isEmpty).

List

Interface que estende Collection, e que define coleções ordenadas (sequências), onde se tem o controle total sobre a posição de cada elemento, identificado por um índice numérico. Na maioria dos casos, pode ser encarado como um “array de tamanho variável”, pois como os arrays primitivos, é acessível por índices, além disso, possui métodos de inserção e remoção.

ArrayList

Implementação de List que utiliza internamente um array de objetos. Em uma inserção onde o tamanho do array interno não é suficiente, um novo array é alocado (de tamanho igual a 1.5 vezes o array original), e todo o conteúdo é copiado para o novo array. Em uma inserção no meio da lista (índice < tamanho), o conteúdo posterior ao índice é deslocado em uma posição. Esta implementação é a recomendada quando o tamanho da lista é previsível (evitando realocações) e as operações de inserção e remoção são feitas, em sua maioria, no fim da lista (evitando deslocamentos), ou quando a lista é mais lida do que modificada (otimizado para leitura aleatória).

LinkedList

Implementação de List que utiliza internamente uma lista encadeada. A localização de um elemento na n -ésima posição é feita percorrendo-se a lista da ponta mais próxima até o índice desejado. A inserção é feita pela adição de novos nós, entre os nós adjacentes, sendo que antes é necessária a localização desta posição. Esta implementação é recomendada quando as modificações são feitas em sua maioria tanto no início quanto no final da lista, e o percorrimeto é feito de forma sequencial (via Iterator) ou nas extremidades, e não aleatória (por índices). Um exemplo de uso é como um fila (FIFO - First-In-First-Out), onde os elementos são retirados da lista na mesma sequência em que são adicionados.

Vector

Implementação de List com o mesmo comportamento da ArrayList, porém, totalmente sincronizada. Por ter seus métodos sincronizados, tem performance inferior ao de uma ArrayList, mas pode ser utilizado em um ambiente multitarefa (acessado por vários threads) sem perigo de perda da consistência de sua estrutura interna.

Em sistemas reais, essa sincronização acaba adicionando um overhead desnecessário, pois mesmo quando há acesso multitarefa à lista (o que não acontece na maioria das vezes), quase sempre é preciso fazer uma nova sincronização nos métodos de negócio, para 'atomizarem' operações seguidas sobre o Vector. Por exemplo, uma chamada ao método `contains()`, seguida de uma chamada do método `add()` caso o elemento não exista, podem causar condições de corrida, se há acessos de vários threads ao mesmo tempo. Assim, acaba sendo necessária uma sincronização adicional, englobando estas duas operações e tornando desnecessária a sincronização inerente da classe.

Portanto, a não ser que hajam acessos simultâneos de vários threads à lista, e a sincronização simples, provida pela classe Vector, seja o bastante, prefira outras implementações como ArrayList e LinkedList, que oferecem performance superior.

Stack

Implementação de List que oferece métodos de acesso para uso da lista como uma pilha (LIFO - Last-In-First-Out), como `push()`, `pop()` e `peek()`. Estende Vector, portanto herda as vantagens e desvantagens da sincronização deste. Pode ser usado para se aproveitar as implementações das operações específicas de pilha.

Set

Interface que define uma coleção, ou conjunto, que não contém duplicatas de objetos. Isto é, são ignoradas as adições caso o objeto ou um objeto equivalente já exista na coleção. Por objetos equivalentes, entenda-se objetos que tenham o mesmo código hash (retornado pelo método `hashCode()`) e que retornem verdadeiro na comparação feita pelo método `equals()`.

Não é garantida a ordenação dos objetos, isto é, a ordem de iteração dos objetos não necessariamente tem qualquer relação com a ordem de inserção dos objetos. Por isso, não é possível indexar os elementos por índices numéricos, como em uma `List`.

HashSet

Implementação de `Set` que utiliza uma tabela hash (a implementação da Sun utiliza a classe `HashMap` internamente) para armazenar seus elementos. Não garante a ordem de iteração, nem que a ordem permanecerá constante com o tempo (uma modificação da coleção pode alterar a ordenação geral dos elementos). Por utilizar o algoritmo de tabela hash, o acesso é rápido, tanto para leitura quanto para modificação.

LinkedHashSet

Implementação de `Set` que estende `HashSet`, mas adiciona previsibilidade à ordem de iteração sobre os elementos, isto é, uma iteração sobre seus elementos (utilizando o `Iterator`) mantém a ordem de inserção (a inserção de elementos duplicados não altera a ordem anterior). Internamente, é mantida uma lista duplamente encadeada que mantém esta ordem. Por ter que manter uma lista paralelamente à tabela hash, a modificação deste tipo de coleção acarreta em uma leve queda na performance em relação à `HashSet`, mas ainda é mais rápida que uma `TreeSet`, que utiliza comparações para determinar a ordem dos elementos.

SortedSet

Interface que estende `Set`, adicionando a semântica de ordenação natural dos elementos. A posição dos elementos no percorrido da coleção é determinado pelo retorno do método `compareTo(o)`, caso os elementos implementem a interface `Comparable`, ou do método `compare(o1, o2)` de um objeto auxiliar que implemente a interface `Comparator`.

TreeSet

Implementação de `SortedSet` que utiliza internamente uma `TreeMap`, que por sua vez utiliza o algoritmo Red-Black para a ordenação da árvore de

elementos. Isto garante a ordenação ascendente da coleção, de acordo com a ordem natural dos elementos, definida pela implementação da interface `Comparable` ou `Comparator`. Use esta classe quando precisar de um conjunto (de elementos únicos) que deve estar sempre ordenado, mesmo sofrendo modificações. Para casos onde a escrita é feita de uma só vez, antes da leitura dos elementos, talvez seja mais vantajoso fazer a ordenação em uma `List`, seguida de uma cópia para uma `LinkedHashSet` (dependendo do tamanho da coleção e do número de repetições de elementos).

Map

Interface que define um array associativo, isto é, ao invés de números, objetos são usados como chaves para se recuperar os elementos. As chaves não podem se repetir (seguindo o mesmo princípio da interface `Set`), mas os valores podem ser repetidos para chaves diferentes. Um `Map` também não possui necessariamente uma ordem definida para o percorrimento.

HashMap

Implementação de `Map` que utiliza uma tabela hash para armazenar seus elementos. O tempo de acesso aos elementos (leitura e modificação) é constante (muito bom) se a função de hash for bem distribuída, isto é, a chance de dois objetos diferentes retornarem o mesmo valor pelo método `hashCode()` é pequena.

LinkedHashMap

Implementação de `Map` que estende `HashMap`, mas adiciona previsibilidade à ordem de iteração sobre os elementos, isto é, uma iteração sobre seus elementos (utilizando o `Iterator`) mantém a ordem de inserção (a inserção de elementos duplicados não altera a ordem anterior). Internamente, é mantida uma lista duplamente encadeada que mantém esta ordem. Por ter que manter uma lista paralelamente à tabela hash, a modificação deste tipo de coleção acarreta em uma leve queda na performance em relação à `HashMap`, mas ainda é mais rápida que uma `TreeMap`, que utiliza comparações para determinar a ordem dos elementos.

Hashtable

Assim como o `Vector`, a `Hashtable` é um legado das primeiras versões do JDK, igualmente sincronizado em cada uma de suas operações. Pelos mesmos motivos da classe `Vector`, dê preferência a outras implementações, como a `HashMap`, `LinkedHashMap` e `TreeMap`, pelo ganho na performance.

Properties

Classe que estende Hashtable, especializada para trabalhar com Strings. Não é propriamente uma coleção (já que se restringe basicamente a Strings), mas serve como exemplo de uma implementação especializada da interface Map.

IdentityHashMap

Implementação de Map que intencionalmente viola o contrato da interface, utilizando a identidade (operador ==) para definir equivalência das chaves. Esta implementação deve ser usada apenas em casos raros onde a semântica de identidade referencial deve ser usada para equivalência.

WeakHashMap

Implementação de Map que utiliza referências fracas (WeakReference) como chaves, permitindo a desalocação pelo Garbage Collector dos objetos nela contidos. Esta classe pode ser usada como base para a implementação de caches temporários, com dados automaticamente desalocados em caso de pouca utilização.

SortedMap

Interface que estende Map, adicionando a semântica de ordenação natural dos elementos, análogo à SortedSet. Também adiciona operações de partição da coleção, com os métodos headMap(k) - que retorna um SortedMap com os elementos de chaves anteriores a k -, subMap(k1,k2) - que retorna um SortedMap com os elementos de chaves compreendidas entre k1 e k2 - e tailMap(k) - que retorna um SortedMap com os elementos de chaves posteriores a k.

TreeMap

Implementação de SortedMap que utiliza o algoritmo Red-Black para a ordenação da árvore de elementos. Isto garante a ordenação ascendente da coleção, de acordo com a ordem natural dos elementos, definida pela implementação da interface Comparable ou Comparator. Use esta classe quando precisar de um conjunto (de elementos únicos) que deve estar sempre ordenado, mesmo sofrendo modificações. Análogo ao TreeSet, para casos onde a escrita é feita de uma só vez, antes da leitura dos elementos, talvez seja mais vantajoso fazer a ordenação em uma List, seguida de uma cópia para uma LinkedHashSet (dependendo do tamanho da coleção e do número de repetições de chaves).

Interfaces auxiliares

O framework Collections define ainda uma série de interfaces auxiliares, que definem operações de objetos retornados por métodos das interfaces de coleções.

Iterator

Interface que define as operações básicas para o percorrimto dos elementos da coleção. Utiliza o pattern de mesmo nome (Iterator, GoF), desacoplando o código que utiliza as coleções de suas estruturas internas. É possível remover elementos da coleção original utilizando o método `remove()`, que remove o elemento atual da iteração, mas esta operação é de implementação opcional, e uma `UnsupportedOperationException` será lançada caso esta não esteja disponível.

ListIterator

Interface que estende `Iterator`, adicionando funções específicas para coleções do tipo `List`. É obtida através do método `listIterator()` de uma `List`, e possui operações de percorrimto em ambos os sentidos (permitido pela indexação dos elementos feita pela `List`).

Comparable

Interface que define a operação de comparação do próprio objeto com outro, usado para definir a ordem natural dos elementos de uma coleção. Pode ser usado caso os objetos que serão adicionados na coleção já implementem a interface (`Integer`, `Double`, `String` etc.), ou serão implementados pelo programador, que tem a chance de embutir esta interface em seu código.

Comparator

Interface que define a operação de comparação entre dois objetos por um objeto externo. É utilizado quando os objetos a serem adicionados não podem ser modificados para aceitarem a interface `Comparable` (são de uma biblioteca de terceiros, por exemplo), ou é necessária a troca da estratégia de ordenação em tempo de execução (ordenação das linhas de uma tabela, por exemplo). Esta interface provê maior flexibilidade, sem custo adicional significativo. Prefira seu uso, ao invés da interface `Comparable`, a não ser que esta seja necessária (isto é, você usa uma classe que espera objetos que a implementem) ou se a própria semântica dos objetos exija essa ordem natural (o que é bem subjetivo e específico do domínio/sistema).

Enumeration

\ 'Ancestral\ ' da interface `Iterator`, que provê as mesmas funções, a não ser a retirada de elementos da coleção. Os nomes de seus métodos são bem mais longos - `hasMoreElements()` e `nextElement()` contra `hasNext()` e `next()` - o que dificulta a digitação e a leitura do código. É utilizada basicamente apenas nas classes `Vector` e `Hashtable`, podendo ser praticamente ignorada e substituída pela interface `Iterator`.

RandomAccess

Interface marcadora (marker interface), que apenas assinala que determinadas implementações de List são otimizadas para acesso aleatório (por índice numérico, ao invés de iterators). Esta informação pode ser usada por algumas classes, para que estas possam alterar suas estratégias internas para ganho de performance.

Classes Utilitárias

Classes utilitárias, como assumidas aqui, são classes que possuem apenas métodos estáticos, provendo algoritmos comuns.

Collections

Oferece métodos que efetuam operações sobre coleções. Algumas operações disponíveis são: busca binária em listas; substituição dos elementos de uma lista por um determinado objeto; busca pelo menor ou maior elemento (de acordo com a ordem natural definida pelas interfaces Comparable ou Comparator); inversão de uma lista; embaralhamento; deslocamento dos elementos; obtenção de coleções sincronizadas ou imutáveis, baseadas em coleções já existentes (através do pattern Decorator).

Arrays

Classe utilitária que oferece operações sobre arrays, incluindo: ordenação, procura binária, comparação entre arrays, preenchimento, cálculo do hashCode baseados nos elementos de um array, transformação em String e o invólucro de um array por uma implementação de tamanho fixo de List.

PROGRAMANDO POR INTERFACES

O Collections framework do Java nos fornece toda uma infraestrutura para estruturas de dados. E, por usar interfaces básicas bem definidas (Collection, List, Set, SortedSet, Map, SortedMap), cria também abstrações que aumentam a flexibilidade e a facilidade de modificação.

Quando programamos usando estas classes, devemos sempre referenciá-las pelas interfaces. Alguém pode dizer: “Mas não dá para instanciar interfaces!” Claro que não. Quando precisamos instanciar uma coleção, utilizamos a implementação concreta mais apropriada, mas os tipos de parâmetros e retornos dos métodos devem sempre ser uma destas interfaces básicas.

Por exemplo, no código seguinte:

1. **private** Vector listaDeCompras;
2. **public void** setListaDeCompras(Vector lista) {

```

3. this.listaDeCompras = lista;
4. }
5. public Vector getListaDeCompras() {
6. return this.listaDeCompras;
7. }

```

Suponha que tenhamos notado que não há acesso concorrente a este objeto, e a sincronização inerente do Vector está impactando na performance do sistema. Para alterarmos o tipo de Vector para ArrayList, eliminando o overhead desnecessário da sincronização, temos que alterar todas as classes que utilizam este método, pois elas passam e esperam um Vector, não um ArrayList. Ou pior: suponha que estas classes que utilizam nossa lista de compras (criadas por outras equipes, portanto, mais difíceis de serem alteradas) também têm suas próprias estruturas internas de dados, utilizando a classe LinkedList. O inferno está mais próximo do que você imagina! Para passar parâmetros ou receber valores é preciso fazer conversões entre coleções diferentes, copiando os elementos de uma para outra, aumentando ainda mais o problema de performance, além de dificultar a leitura e a manutenção do código.

Nestes casos, se no lugar de Vector tivéssemos usado a interface List, bastaria alterar o local de instanciação da classe, que todo o resto do código teria automaticamente a nova implementação, sem modificações. Uma coleção criada em um componente poderia facilmente ser utilizada por outro, pois tudo o que ele espera é uma implementação de List, e não uma classe concreta.

Portanto, utilize interfaces sempre que puder, principalmente em assinaturas de métodos públicos, e limite ao máximo referências a classes concretas, mesmo internamente na classe.

E QUE INTERFACE USAR?

Temos agora pelo menos seis interfaces para escolher: Collection, List, Set, SortedSet, Map e SortedMap. Se acrescentarmos as auxiliares de acesso à coleção, temos ainda a Iterator e a ListIterator. Qual delas usar?

A Collection é a mais genérica, portanto é a mais indicada sempre, certo?

Errado! Por ser tão genérica, suas operações são muito restritas, dificultando sua manipulação. Por exemplo, ela não permite o acesso direto a um elemento, toda a leitura deve ser feita através de seu Iterator. Se a coleção será sempre lida como uma sequência de elementos, e o acesso aleatório (por índice) não faz sentido, isto não é problema. Mas se este tipo de acesso é necessário (ou mesmo conveniente), faz mais sentido usar uma interface List.

Este é um problema sem uma resposta definitiva, pois depende muito de cada caso. Mas posso listar algumas regras básicas, obtidas da minha própria experiência. Não que isso seja grande coisa, mas vale a tentativa.

- A interface `Collection` normalmente é usada para conjuntos descartáveis, isto é, que vão ser lidos apenas uma vez, e não guardados internamente e manipulados (no máximo copiados para uma outra estrutura).
- A interface `Set` serve mais como um marcador da semântica esperada, pois ela não acrescenta nenhuma operação à `Collection`.
- As interfaces `SortedSet` e `SortedMap` servem para forçar um contrato, de que os elementos da coleção virão ordenados. Aparecem, principalmente, como parâmetros de métodos que esperam por esta ordenação. Porém, na maioria das vezes, as classes `TreeSet` e `TreeMap` serão usadas apenas localmente, como containers temporários de ordenação.

EXEMPLOS DE USO:

Loop típico em uma `Collection`, usando o `Iterator`:

```
1. Iterator it = colecao.iterator();
2. while (it.hasNext()) {
3. String item = (String) it.next();
4. System.out.println(item);
5. }
```

Loop típico em um `Map`, usando o `Iterator`:

```
1. Map mapa = new HashMap();
2. Iterator itChaves = mapa.keySet().iterator();
3. while (itChaves.hasNext()) {
4. Object chave = itChaves.next();
5. Object valor = mapa.get(chave);
6. System.out.println(chave + " = " + valor);
7. }
```

Usando um `iterator` para retirar elementos de uma coleção:

```
1. Iterator it = listaCores.iterator();
2. while (it.hasNext()) {
3. String item = (String) it.next();
4. if (item.charAt(0) == '\v') { //retirando as cores que começam com '\v'
5. it.remove();
6. }
7. }
```

Utilizando um `TreeSet` para listar o conteúdo de um `Map/Properties` de maneira ordenada:

```
1. Properties p = new Properties();
2. //...
3. Iterator it = new TreeSet(p.keySet()).iterator(); //ordena as chaves antes de criar o iterator
4. while (it.hasNext()) {
```

```

5. String key = (String) it.next();
6. System.out.println(key + "=" + p.getProperty(key));
7. }

```

Inicialização de uma Collection:

```

1. // Versão normal:
2. List listaCores = new ArrayList();
3. listaCores.add("vermelho");
4. listaCores.add("verde");
5. listaCores.add("amarelo");
6. listaCores.add("branco");
7.
8. // Versão abreviada utilizando um array
9. List listaCores = new ArrayList(Arrays.asList(new String[] {
10. "vermelho", "verde", "amarelo", "branco" }));

```

Criando e ordenando uma Collection:

```

1. List listaCores = new ArrayList(Arrays.asList(new String[] {
2. "vermelho", "verde", "amarelo", "branco" }));
3.
4. Collections.sort(listaCores);

```

Criando e ordenando uma Collection:

```

1. List listaCores = new ArrayList(Arrays.asList(new String[] {
2. "vermelho", "verde", "amarelo", "branco" }));
3.
4. Collections.sort(listaCores);

```

Intersecção de conjuntos utilizando Set:

```

1. Set coresQuentes = new HashSet(Arrays.asList(new String[] {
2. "vermelho", "laranja", "amarelo" }));
3.
4. Set conjuntoCores = new HashSet(Arrays.asList(new String[] {
5. "vermelho", "verde", "amarelo", "branco" }));
6.
7. conjuntoCores.retainAll(coresQuentes); // conjuntoCores == {"vermelho",
    "amarelo"} O

```

RESUMO DO TÓPICO 4

Neste tópico vimos que:

- As filas representam filas de espera; as inserções são feitas na parte superior de uma fila e as exclusões são feitas na parte da frente.
- A fila é uma versão limitada de uma lista, sendo semelhante a uma fila de caixa de supermercado, onde a primeira pessoa a ser atendida é a primeira da fila, enquanto os outros clientes entram na fila apenas no fim e esperam ser atendidos.
- Os nós da fila são removidos apenas no início da fila e são removidos somente no final da fila. Por essa razão, a fila é conhecida como uma estrutura de dados primeiro a entrar, primeiro a sair (first-in, first-out - FIFO).
- As filas têm muitas aplicações nos sistemas de computador. A maioria dos computadores tem apenas um único processador, portanto apenas um usuário por vez pode ser atendido. Os pedidos para os outros usuários são colocados em uma fila. O pedido no início da fila é o próximo a ser atendido. Cada entrada avança gradualmente para o início da fila à medida que os usuários são atendidos.
- As filas também são muito utilizadas para suportar *spooling* de impressão. Um ambiente multiusuário pode ter só uma impressora. Muitos usuários podem estar gerando saídas para impressão. Se a impressora estiver ocupada, outras saídas ainda podem ser geradas. Estas são colocadas no *spool* em disco, onde esperam na fila até a impressora ficar disponível.



1 Considerando os seguintes dados de entrada: 08, 20, 31, 13, 28, 39, 09, 19, 21, 11. Insira os dados de entrada numa fila. Em seguida retire cada dado da fila e insira numa pilha. Exiba a pilha. Depois retire os dados da pilha e insira na fila. Exiba a fila. Diante disso, analise as afirmativas a seguir identificando como as estruturas serão exibidas ao usuário.

I. Pilha: (topo) 11 - 21 - 19 - 09 - 39 - 28 - 13 - 31 - 20 - 08

II. Fila: (começo) 08 - 20 - 31 - 13 - 28 - 39 - 09 - 19 - 21 - 11 (fim)

III. Fila: (começo) 11 - 21 - 19 - 09 - 39 - 28 - 13 - 31 - 20 - 08 (fim)

IV. Pilha: (topo) 08 - 20 - 31 - 13 - 28 - 39 - 09 - 19 - 21 - 11

V. A fila mostrada fica com os elementos em ordem invertida dos dados de entrada.

Agora assinale a alternativa correta:

- a) () III e IV.
- b) () II e IV.
- c) () I, II e III.
- d) () I, III e V.
- e) () I, IV e V.

2 Considere uma estrutura de Fila composta por números inteiros e que possui duas operações básicas: Inserir(x) e Excluir(). Além disso, a representação do estado da fila em determinado momento é realizada listando os elementos, de forma que o primeiro elemento, da esquerda para a direita, é o mais antigo presente na fila. Diante disso, analise a sequência de comandos a seguir, considerando que a lista começa vazia.

Inserir (2) > Inserir (3) > Excluir () > Inserir (1) > Excluir () > Inserir (4) > Inserir (5) Excluir ()

O estado final da fila será:

- a) () 1 2 3 4 5
- b) () 2 3 1 4 5
- c) () 3 1 4
- d) () 4 5
- e) () 5

REFERÊNCIAS

AGUILAR, Luis J. **Fundamentos de programação** – 3 ed.: Algoritmos, estruturas de dados e objetos. Tradução: VALLE, Paulo H. C. Revisão Técnica: SILVA, Flavio S. C. Dados Eletrônicos. Porto Alegre: AMGH, 2011.

_____. **Programação em C ++**. Algoritmos, estruturas de dados e objetos. Tradução. ALONSO, Maria C.; FELICE, Marines P.: 2 ed.: Dados Eletrônicos. Porto Alegre: AMGH, 2011.

AKITA, Fabio. **Repensando a web com rails**. Rio de Janeiro: Ed. Brasport, 2006.

ALMEIDA, Marilane. **Curso essencial de lógica de programação**. São Paulo: Ed. Digerati Books, 2008.

ALVES, Ricardo. **Recursividade em Java**. Disponível em: <<http://www.linhadecodigo.com.br/artigo/3316/recursividade-em-java.aspx#ixzz3EQu8W9rl>>. Acesso em: 08 set. 2014.

ANCIBE. **Conceitos básicos de linguagem de programação C**. Disponível em: <<http://www.ancibe.com.br/Apostila%20de%20Algoritmos/apostila%20de%20linguagem%20C%20otima.pdf>>. Acesso em: 08 set. 2014.

ARENALES, Marcos; et al. Pesquisa operacional. Rio de Janeiro: Ed. Elsevier, 2011.

BARBOSA, Eduardo; MIYOSHI, Ronaldo A.; GOMES, Marco D. G. **Tipos abstratos de dados**. 2008. Disponível em: <<http://www2.dc.ufscar.br/~bsi/materiais/ed/u2.html>>. Acesso em: 3 out. 2014.

BERG, Alexandre; FIGUEIRÓ, Joice P. **Lógica de programação**. Canoas: Ed. Ulbra, 2006.

BORAKS, Silvio. Programação com Java: Uma Introdução Abrangente. Rio de Janeiro. Ed. Bookman, 2013.

BRAZ. Maria H. **Conceitos básicos de programação**. Disponível em: <<https://fenix.tecnico.ulisboa.pt/downloadFile/3779571246306/aula78911.pdf>>. Acesso em: 10 Set. 2014.

BROOKSHEAR, Glenn J. **Ciência da computação, uma visão abrangente**. Porto Alegre: Ed. Bookman, 2013.

C Progressivo. Funções recursivas: pra aprender recursividade, tem que saber recursividade. Disponível em: <<http://www.cprogressivo.net/2013/03/O-que-sao-e-como-usar-funcoes-recursivas-em-linguagem-C.html>>

CAELUM. **Algoritmos e estruturas de dados em java**. Disponível em: <<http://www.ime.usp.br/~cosen/verao/alg.pdf>>. Acesso em: 10 out. 2014.

CELES, Waldemar; RANGEL, José Luis. **Cadeia de caracteres**. Disponível em: <<http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap06.pdf>>. Acesso em: 1 out. 2014.

_____. **Conceitos Fundamentais**. 2002. Disponível em: <<http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap01.pdf>>. Acesso em: 08 set. 2014.

CENAPAD-SP. **Introdução ao Fortran 90**. Disponível em: <http://www.fis.ufba.br/~edmar/fortran/fortran_apostila.pdf>. Acesso em: 3 out. 2014.

CHAPRA, Steven C. **Métodos numéricos aplicados com MATLAB para engenheiros e cientistas**. Porto Alegre: AMGH, 2012.

COCIAN, Luis Fernando Espinosa. **Manual da linguagem C**. Canoas. Ed. ULBRA, 2004, 500p.

CODEPAD. Disponível em: <<http://codepad.org/>>. Acesso em: 08 Set. 2014.

COSTA, Daniel G. **Administração de redes com scripts: Bash Script, Python e VBScript**. 2 ed. Rio de Janeiro: Ed. Brasport, 2010.

COSTA, Umberto S.; NETA, Natália, S. L. **Matemática aplicada**. Disponível em: <http://www.metroloedigital.ufrn.br/aulas/disciplinas/mat_aplicada/aula_07.html>. Acesso em: 1 out. 2014.

CRUZ, Adriano. Ponteiros. 1999. Disponível em: <<http://equipe.nce.ufrj.br/adriano/c/apostila/ponte.htm#matrizes>>. Acesso em: 3 out. 2014.

DEITEL. Harvey M.; DEITEL. Paul J. **Java, como programar**. Tradução Carlos Arthur Lang Lisboa, 4. ed. – Porto Alegre: Bookman, 2003.

DORNELLES, Adalberto A. F. **Fundamentos Linguagem de C**. 1997. Disponível em: <<http://www.ebah.com.br/content/ABAAAAMX8AL/fundamentos-linguagem-c>>. Acesso em: 08 set. 2014.

EDELWEISS, Nina; LIVI, Maria A. C. **Algoritmos e programação com exemplos em Pascal e C**. Porto Alegre: Ed. Bookman, 2014.

EDELWEISS, Nina; GALANTE, Renata. **Estrutura de dados**. Porto Alegre: Ed. Bookman, 2009.

_____. **Estruturas de dados**: Volume 18. Porto Alegre: Bookman, 2009.

FARIAS, Ricardo. **Estrutura de dados e algoritmos**. Disponível em: <<http://www.cos.ufrj.br/~rfarias/cos121/pilhas.html>>. Acesso em: 10 set. 2014.

FEIJÓ, Bruno; SILVA, Flávio S. C.; CLUA, Esteban. **Introdução à ciência da computação Com Jogos**. Rio de Janeiro: Ed. Elsevier, 2010.

FEOFIOFF, Paulo. **Algoritmos em linguagem C**. Rio de Janeiro: Ed. Campus/Elsevier, 2009.

FIALHO, Rodrigo. **Operadores aritméticos com variáveis, MsgBox e Chr – VBA**. Disponível em: <<http://doutorexcel.wordpress.com/2011/03/11/operadores-aritmeticos-com-variaveis-no-vba/>>. Acesso em: 10 set. 2014.

FILHO, Antônio, M. S. **Introdução à programação orientada a objetos com C++**. Rio de Janeiro: Ed. Elsevier, 2011.

FIORESE, Virgilio. **Wireless** - Introdução às Redes de Telecomunicação Móveis Celulares. Rio de Janeiro: Ed. Brasport, 2005.

FONTES, Fábio F. C. **Estrutura e representação**. 2009. Disponível em: <<http://www2.ufersa.edu.br/portal/view/uploads/setores/146/arquivos/aula%205%20-%20Estrutura%20e%20Representa%C3%A7%C3%A3o.ppt>>. Acesso em: 3 out. 2014.

FROZZA, Angelo Augusto. **Estrutura de dados variáveis compostas**. Disponível em: <<http://www.ifc-camboriu.edu.br/~frozza/2011.1/IX10/IX10-EDD-Aula004-VariaveisCompostas-parcial.pdf>>. Acesso em: 1 out. 2014.

GARCIA, Vinícius. **Métodos computacionais. Tipos estruturados**. 2010. Disponível em: <<http://viniciusgarcia.files.wordpress.com/2010/08/aulatiposestruturados.pdf>>. Acesso em: 1 out. 2014.

GAÚCHO, Felipe. **Básico e intermediário**. Disponível em: <<http://www.milfont.org/blog/wp-content/upload/Manual.pdf>>. Acesso em: 10 set. 2014.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Projeto de algoritmos, fundamentos, análise e exemplos da internet**. Porto Alegre: Ed. Artmed, 2002.

_____. **Estrutura de dados e algoritmos em java.** Tradução Bernardo Copstein e João Batista Oliveira, 2. ed. – Porto Alegre: Bookman, 2002.

GRIFFITHS, David e GRIFFITHS, Dawn. **O guia amigo do seu cérebro. Use a cabeça!** (Head First) Rio de Janeiro: Ed. Alta Books, 2013.

GUIMARÃES, Ângelo M.; LAGES, Newton, A. C. **Algoritmo e estrutura de dados.** Rio de Janeiro: Ed: LCT, 2013.

HORSTMANN, Cay. **Conceitos de computação com Java.** Porto Alegre: Ed. Bookman, 2008.

HOROWITZ, Ellis; SAHNI, Sartaj. **Fundamentos de estrutura de dados.** Tradução RAWICKI, Thomaz R. Rio de Janeiro: Ed. Campus, 1987.

JAVA. **O que é a tecnologia Java e porque preciso dela.** Disponível em: <http://www.java.com/pt_BR/download/faq/whatis_java.xml>. Acesso em: 08 set. 2014.

Javafree.org. **O collections Framework.** Disponível em: <<http://javafree.uol.com.br/artigo/847654/Collections-Framework.html>>. Acesso em: 18 nov. 2014.

JPROFESSOR. **Estruturas de pilhas em Java.** Disponível em: <<http://jprof.pro.br/exemplos.html#arquivob1>>. Acesso em: 10 set. 2014.

JUNIOR, Dilermando P.; et al. **Algoritmos e programação de computadores.** Rio de Janeiro: Ed. Elsevier, 2012.

Laboratório de Programação I. Estruturas de Dados Homogênea. Disponível em: <http://cae.ucb.br/conteudo/programar/labor1/new_matriz.html>. Acesso em: 3 out. 2014.

LAUREANO, Marcos. **Estrutura de dados com algoritmos.** Rio de Janeiro. Brasport. 2008.

LEITE, Mario. **Técnicas de programação. Uma abordagem Moderna.** Rio de Janeiro: Ed. Brasport, 2006.

LIMA, Edirlei S. INF 1007 – Programação II. Aula 12 – Tipos Abstratos de Dados. Disponível em: <http://edirlei.3dgb.com.br/aulas/prog2_2014_1/Prog2_Aula_12_TAD.pdf>. Acesso em: 3 out. 2014.

_____. **Programação II.** Disponível em: <http://edirlei.3dgb.com.br/aulas/prog2_2014_1/Prog2_Aula_08_Vetor_Ponteiros_2014.pdf>. Acesso em: 1 out. 2014.

LIMA, Isaias. **Introdução à estrutura de dados**. Disponível em: <http://www.isaias.unifei.edu.br/Introducao_ED.pdf>. Acesso em: 1 out. 2014.

LIPSON, Marc e LIPSCHUTZ, Seymour. **Matemática discreta**. Porto Alegre: Ed. Bookman, 1997.

LOPES, Arthur V. **Introdução à programação com ADA 95**. Canoas: Ed. Ulbra, 1997.

LOPES, Gills V. **Minitutorial compilando e rodando programa C no GNU/Linuxmore**. Disponível em: <http://www.academia.edu/4297165/Minitutorial_Compilando_e_rodando_programas_C_no_GNU_Linux>. Acesso em: 08 set. 2014.

LORENZI, Fabiano; LOPES, Arthur V. **Linguagem de programação Pascal**. Canoas: Ed. Ulbra, 2000.

LORENZI, MATTOS e CARVALHO 2007. Disponível em: <<file:///C:/Users/05524964992/Downloads/85-221-0556-1.pdf>>. Acesso em: 10 set. 2014.

MACEDO, Diego. **Conversões de Linguagens: Tradução, Montagem, Compilação, Ligação e Interpretação**. Disponível em: <<http://www.diegomacedo.com.br/conversoes-de-linguagens-traducao-montagem-compilacao-ligacao-e-interpretacao/>>. Acesso em: 08 set. 2014.

MARTIN, John H. **Programação com objective C**. 5 ed. Porto Alegre: Ed. Bookman, 2014.

MATTOS, Érico T. **Programação Java para Wireless**. São Paulo: Ed. Digerati Books, 2005.

MEDEIROS, Higor. **Trabalhando com Arrays**. Disponível em: <<http://www.devmedia.com.br/trabalhando-com-arrays/5212#ixzz3EeKpuuUh>>. Acesso em: 10 set. 2014.

MELO, Pedro O. S. Vaz. **Algoritmos e estruturas de dados I**. Disponível em: <<http://homepages.dcc.ufmg.br/~olmo/aula9-estruturas.pdf>>. Acesso em: 1 out. 2014.

MENDES, Antonio S. F. **Introdução à programação orientada a objetos com C++**. Rio de Janeiro: Ed. Elsevier, 2011.

MITCHELL, Mark; OLDHAM, Jeffrey e SAMUEL, Alex. **Programação linux avançada**. Ed. First, 2001.

MOKARZEL, Fábio; SOMA, Nei Y. **Introdução à Ciência da Computação**. Rio de Janeiro: Ed. Elsevier, 2008.

NETO, Samuel D. **Linguagem C – intermediário**. Disponível em: <http://homepages.dcc.ufmg.br/~joaoreis/Site%20de%20tutoriais/c_int/index.htm#matrizes>. Acesso em: 3 out. 2014.

OLIVEIRA, Gabriel A. **Matriz simétrica**. Disponível em: <<http://www.mundoeducacao.com/matematica/matriz-simetrica.htm>>. Acesso em: 3 out. 2014.

PALMEIRA, Thiago V. V. **Trabalhando com arrays em Java**. Disponível em: <<http://www.devmedia.com.br/trabalhando-com-arrays-em-java/25530>>. Acesso em: 10 set. 2014.

PEREIRA, Silvio L. **Estruturas de dados fundamentais**. Conceitos e Aplicações. São Paulo: Ed. Érica, 1996.

PERRY, Greg. **Aprenda em 21 dias Visual Basic 6**. Tradução FURMANKIEWICZ, Edson. Rio de Janeiro: Ed. Elsevier, 1999.

PINHEIRO, Francisco A. C. P. **Elementos de programação em C**. Dados Eletrônicos. Porto Alegre: Bookman, 2012.

RAMOS, Vinícius M; NETO, João N.; VEGA, Ítalo S. **Linguagens formais**. Porto Alegre: Ed. Artmed, 2009.

REBOLLO, Carlos. **Introdução à linguagem C**. 2013. Disponível em: <http://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_C.pdf>. Acesso em: 09 out. 2014.

RITA, Sandra. **Treinamento em lógica de programação**. São Paulo: Ed. Digerati Books, 2009.

ROCHA, César. **Linguagem C: Estruturas e Alocação Dinâmica**. 2010. Disponível em: <http://www.linuxemais.com/cesar/_media/disciplinas/pged/pged_02.pdf>. Acesso em: 10 Set. 2014.

RODRIGUES, Diego M. **Linguagem C – (Algoritmos)**. Disponível em: <<http://www.ebah.com.br/content/ABAAABacgAB/linguagem-c-algoritmo>>. Acesso em: 08 set. 2014.

SÁ, Robison. **Transposição de matrizes**. Disponível em: <<http://www.infoescola.com/matematica/transposicao-de-matrizes/>>. Acesso em: 3 out. 2014.

SCHEPP, Luís G. **5 dia de Treinamento SAP**. 2011. Disponível em: <<http://luisgustavosap.blogspot.com.br/>>. Acesso em: 10 set. 2014.

SCHUTZER, Waldeck; MASSAGO, Sardão. **Programação Java**. Disponível em: <<http://www.dm.ufscar.br/~waldeck/curso/java/>>. Acesso em: 10 set. 2014.

SEBESTA, Roberto W. **Conceitos de linguagem de programação**. Porto Alegre: Ed. Bookman, 2010.

SERSON, Roberto R. **Programação orientada a objetos com Java 6: Curso Universitário**. Rio de Janeiro: Ed. Brasport, 2007.

SOMERA, Guilherme. **Treinamento Profissional em Java**. São Paulo: Ed. Digerati Books, 2006.

SOUZA, Jackson G. **Paradigmas de linguagem de programação**. 2009. Disponível em: <<http://plp2009c.wordpress.com/>>. Acesso em: 10 set. 2014.

VEGA, Italo S. **Ciclo de programação**. 2004. Disponível em: <<http://www.pucsp.br/~dcc-lp/2004/c22/pucsp-lp-c22-2004-02.pdf>>. Acesso em: 10 set. 2014.

VELOSO, Paulo; SANTOS, Clesio; AZEREDO, Antonio. **Estrutura de dados**. Rio de Janeiro: Ed. Elsevier, 1983.

WordPress. CEULP. **Paradigmas de Linguagem de Programação 2009/1** – Ceulp. Disponível em: <<http://plp2009c.wordpress.com/>>. Acesso em: 08 set. 2014.

XAVIER, Denys W. **Funções**. 2010. Disponível em: <<http://www.tiexpert.net/programacao/java/funcoes.php>>. Acesso em: 10 set. 2014.

ANOTAÇÕES

[illegible]

[illegible]