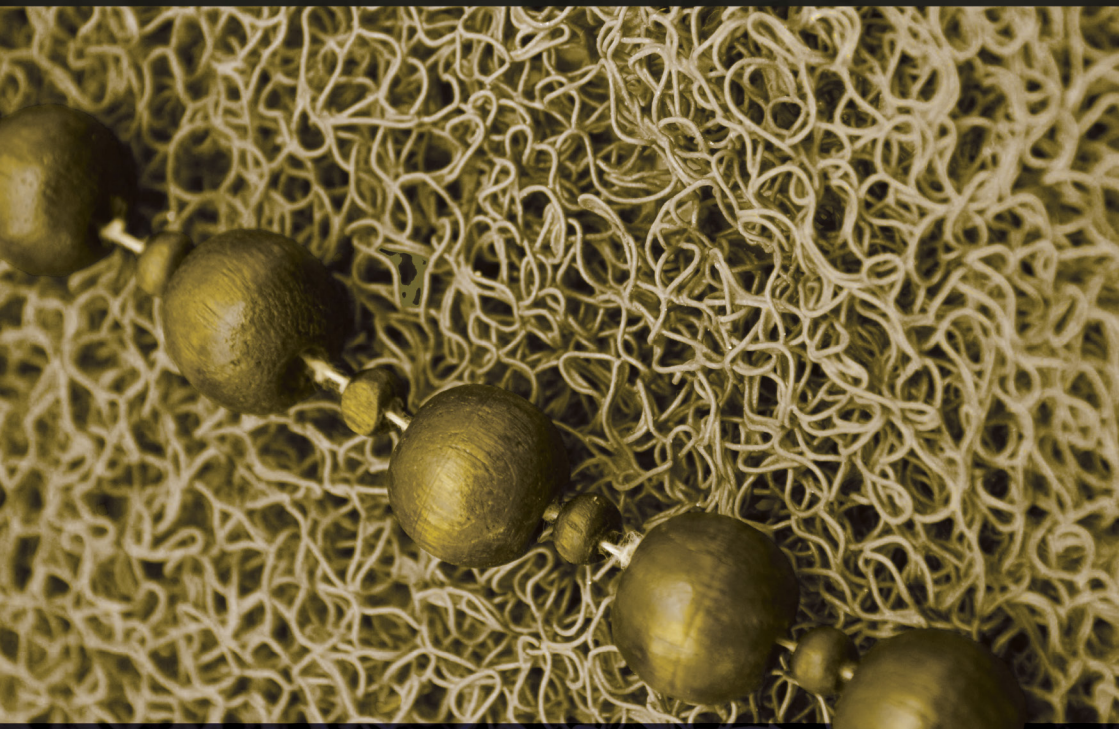
Estruturas de dados lineares básicas

Abordagem prática, com implementações em C e Java.



Valéria Maria Bezerra Cavalcanti
Nadja da Nóbrega Rodrigues

editoraIFPB

Estruturas de dados lineares básica

Abordagem prática, com implementações em C e Java.

Valéria Maria Bezerra Cavalcanti

Nadja da Nóbrega Rodrigues

editoraIFPB

JOÃO PESSOA, 2015

As informações contidas neste livro são de inteira responsabilidade dos autores.

Dados Internacionais de Catalogação na Publicação (CIP)
Biblioteca Nilo Peçanha – IFPB, *Campus* João Pessoa

C376e Cavalcanti, Valéria Maria Bezerra.

Estruturas de dados lineares básicas / Valéria Maria Bezerra Cavalcanti, Nadja da Nóbrega Rodrigues. – João Pessoa: IFPB, 2015.

296 p. : il.

Inclui referências e apêndices.
ISBN 978-85-63406-61-3

1. Ciência da computação. 2. Estrutura de dados. 3. Programação de computador. 4. Linguagem de programação. I. Cavalcanti, Valéria Maria Bezerra. II. Título.

CDU 004.422.63

Dedicamos este trabalho às pessoas que representam
a essência das nossas vidas pessoais e profissionais.
Dedicamos este livro às *nossas pessoas*.

Copyright © 2015 por Valéria Maria Bezerra Cavalcanti e Nadja da Nóbrega Rodrigues.

PRESIDENTE DA REPÚBLICA
Dilma Rousseff

MINISTRO DA EDUCAÇÃO
Aloizio Mercadante

SECRETÁRIO DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
Marcelo Machado Feres

REITOR DO INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Cícero Nicácio do Nascimento Lopes

PRÓ-REITORA DE PESQUISA, INOVAÇÃO E PÓS-GRADUAÇÃO
Francilda Araújo Inácio

PRÓ-REITOR DE ADMINISTRAÇÃO E PLANEJAMENTO
Marcos Vicente dos Santos

PRÓ-REITOR DE DESENVOLVIMENTO INSTITUCIONAL E INTERIORIZAÇÃO
Ricardo Lima e Silva

PRÓ-REITORA DE ENSINO
Mary Roberta Meira Marinho

PRÓ-REITORA DE EXTENSÃO
Vânia Maria de Medeiros

DOCUMENTAÇÃO
Taize Araújo da Silva

DIRETOR EXECUTIVO
Carlos Danilo Miranda Regis

REVISÃO
Elaine Cristina Juvino de Araújo

CAPA E DIAGRAMAÇÃO
Adino Bandeira

IMPRESSÃO
F&A Gráfica

Agradecimentos

Agradecemos a todas as pessoas envolvidas diretamente em nossas conquistas profissionais, ao longo de nossas vidas: familiares, amigos e parceiros de trabalho, dentre estes, especialmente, colegas de academia e indústria, e alunos.

Sumário

Prefácio

A estrutura de dados é a disciplina que estuda a solução de problemas por meio do uso de algoritmos computacionais. Existem diversos tipos de estruturas de dados e a aplicação de cada uma delas está diretamente relacionada a cenários indicados de aplicação, ou seja, a padrões de representação de dados e de manipulação sobre estes dados.

Uma forma de abordar os estudos sobre estruturas de dados é dividi-las de acordo com a maturidade dos alunos em programação: inicialmente, são estudadas as representações mais simples e mais facilmente aplicadas em sistemas em geral, chamadas estruturas fundamentais (geralmente vistas em uma disciplina, como Estruturas de Dados I); a partir do amadurecimento de vários conceitos de programação e do conhecimento das estruturas de dados fundamentais, os alunos estão prontos para conhecer estruturas mais complexas, geralmente utilizadas em aplicações específicas, e que possuem necessidade de representação e manipulação de dados mais elaboradas (esses conceitos podem ser vistos em outras disciplinas, como Estruturas de Dados II).

Inicialmente, este livro visa a atender às demandas de cursos introdutórios de estruturas de dados, especialmente para alunos de cursos na área de Informática, mas também para alunos de outras áreas que utilizam linguagens de programação para a construção de aplicações que manipulam dados. Nesse contexto, o pré-requisito para os usuários deste livro é que tenham algum conhecimento prévio de programação.

O livro ainda se destina aos profissionais de software que desejam aprender ou relembrar o conteúdo referente às estruturas de dados, a nível conceitual (abstrato) ou técnico (a partir das implementações apresentadas nas linguagens C e Java, além das dicas sobre a biblioteca de estruturas Java – *Java Collections Framework*).

O material que compõe o livro reflete a experiência acadêmica das autoras no ensino de programação e estruturas de dados em diversas instituições de ensino superior e na prática em indústria, enquanto desenvolvedoras de software. Nesse sentido, as autoras agradecem aos seus alunos, que auxiliaram no refinamento do material, e aos colegas de indústria, por compartilharem conhecimentos, lições aprendidas e experiências profissionais.

Algumas linguagens possuem bibliotecas padrões para o uso de estruturas de dados fundamentais. Ou seja, essas linguagens facilitam a aplicação das estruturas no desenvolvimento dos softwares, trazendo estas estruturas prontas para que sejam utilizadas pelos desenvolvedores. O estudo das estruturas de dados pelos alunos de cursos de Informática pode ser dividido em dois momentos: inicialmente, é muito importante que esse aluno conheça as estruturas e consiga construir as rotinas básicas de manipulação dessas estruturas; em um segundo momento, ao utilizar linguagens de programação que já implementam as estruturas, é importante que esse aluno saiba aplicá-las corretamente para solucionar problemas práticos, para que ele não tenha que ficar “reinventando a roda”.

Este livro pretende abordar os dois momentos de contato dos alunos com as estruturas de dados. O primeiro momento se dá na apresentação conceitual da estrutura, do seu contexto de aplicação nos softwares e de como esta pode ser implementada nas linguagens C e Java. O ponto-chave dessa etapa é o entendimento das estruturas, sendo sua implementação a forma mais eficiente

para fixação dos seus conceitos e práticas de programação. O segundo momento se dá na apresentação de como utilizar a biblioteca padrão de Java para empregar as estruturas de dados no desenvolvimento de aplicações. Nesse momento, o aluno já conhece as estruturas, sendo apresentado a aspectos que agilizam a sua implementação em Java.

Pensando nas especificidades dos currículos dos cursos que tratam aspectos de programação e estruturas de dados e na forma como esses conteúdos são apresentados por meio de paradigmas e linguagens de programação diferentes, este livro apresenta algumas características: visando o entendimento das estruturas, o livro apresenta seus conceitos base por meio de linguagens estruturadas e orientadas a objetos, trazendo implementações em C e em Java (embora o livro não tenha o objetivo de abordar conceitos específicos desses paradigmas, mas apenas a ideia de aumentar sua abrangência, uma vez que dependendo do currículo do curso, podem ser utilizadas diferentes linguagens de programação para apresentação das estruturas); visando o uso das estruturas a partir da biblioteca padrão de Java, este livro traz dicas de uso de recursos dessa biblioteca para programadores que já entendem o uso das estruturas e conhecem as suas implementações; visando permitir o seu uso em diferentes momentos, pelos leitores, respeitando a sua maturidade e incentivando a prospecção do seu desenvolvimento individual, o livro traz diversos exemplos e exercícios práticos, com diferentes graus de dificuldade; visando aproximar a academia das demandas da indústria, o livro traz uma abordagem predominantemente prática.

Este livro contém as estruturas de dados lineares fundamentais para programação. Nenhuma estrutura de dados única funciona bem para todos os propósitos, e assim é importante conhecer os pontos fortes e as limitações de algumas delas.

O conteúdo do livro está dividido de forma a serem entendidos diversos aspectos relacionados às estruturas de dados. Inicialmente, apresenta-se uma introdução contendo uma explanação sobre o contexto de importância e aplicação das estruturas de dados e ainda uma apresentação sobre tipos de dados. Em um segundo momento, apresentam-se algumas estruturas lineares básicas, como lista genérica, pilha, fila e deque, explicadas de forma simples e com exemplos que ilustram o passo a passo para criá-las e manipulá-las. Para estas estruturas, são apresentadas implementações estáticas (mais simples, baseadas em elementos definidos nas linguagens de programação, como vetores e tipos estruturados) e dinâmicas (mais complexas, porém mais flexíveis, por exemplo, quanto ao tamanho/número de elementos, que pode ser ajustado dinamicamente). O livro traz uma breve explanação sobre árvores, apenas para que o leitor entenda a diferença básica entre a representação das estruturas lineares e não lineares (representada, nesse caso, pela estrutura árvore).

1 Introdução

Moraes (2001) acredita que o computador é uma máquina fantástica, e explica que com sua ajuda, foi possível ao homem pousar na lua, iniciar expedições exploratórias interplanetárias, mapear o genoma humano e criar a Internet.

Na indústria de software, a demanda por produtos sofisticados e que produzam os resultados esperados é representada por indicadores em constante ascensão. Devido à pressão do ambiente e às mudanças nos cenários organizacionais, as Tecnologias de Informação e Comunicação vêm sendo utilizadas não apenas como ferramentas para automatizar ou informatizar empresas e instituições em geral - por meio de sistemas de informação tradicionais, ou simples – mas também como base para seus negócios e competitividade.

Com o surgimento de novos paradigmas e recursos, e o consequente aumento do poder proporcionado por informações e conhecimentos, houve uma transferência de valores entre os instrumentos organizacionais, e a tecnologia passou a ser vista como elemento imprescindível para elaboração de estratégias e formação de inteligência de negócios. O resultado da percepção dessa constante evolução das tecnologias e do aumento de poder proporcionado por elas pode ser associado ao aumento no nível de exigência dos clientes de tecnologia. Para acompanhar essas transformações, os perfis das aplicações de software sofreram mudanças, passando a apresentar características que representam maior complexidade para a sua construção.

Assim como qualquer processo produtivo, o desenvolvimento de software necessita de sistematização. Sua produção é considerada uma atividade mais particular, quando comparada às outras áreas de desenvolvimento de produtos, devido às características dos softwares. Segundo Sommerville (2011), os sistemas de software são abstratos e intangíveis. Eles não são restringidos pelas propriedades dos materiais, nem governados pelas leis da física ou pelos processos de manufatura.

Segundo Moraes (2001), o nosso mundo, cheio de sutilezas e de ricos detalhes, não dá para ser representado em um computador conforme nós o vemos e sentimos, devido às limitações da máquina, sendo necessário o uso de abstração. Abstraindo a nossa realidade, podemos capturar o que existe de mais relevante em uma situação real, tornando possível a construção de modelos que podem ser implementados nos computadores por meio de uma linguagem de programação.

Devido a essa singularidade, o desenvolvimento de sistemas requer o uso de abordagens, técnicas e instrumentos muitas vezes também abstratos para sua construção, avaliação e uso. Nesse cenário, muitos são os fatores que podem fazer um projeto de software falhar, levando-o ao fracasso total ou a enfrentar problemas, gerando uma profunda insatisfação nos clientes. Tomar decisões corretas e fazer escolhas precisas sobre elementos que envolvem tanto a etapa de projeto quanto as demais etapas do ciclo de vida de um software é algo desejado por profissionais da área.

Em qualquer subárea de atuação, é necessário que os profissionais de desenvolvimento de software criem a cultura de conhecer os recursos disponíveis para a execução de suas atividades, de amadurecer as propostas de uso desses recursos (por meio do entendimento de seus respectivos cenários de aplicação) e ainda de incorporar em seus projetos as boas práticas da área. Acredita-

se que essas ações podem contribuir para o êxito e o aumento da qualidade nos projetos. Para Celes *et al.* (2004), o conhecimento de técnicas de programação adequadas para a elaboração de programas de computador tornou-se indispensável para profissionais que atuam nas áreas técnico-científicas.

Falando especificamente sobre etapas como projeto e codificação, pode-se dizer que elas enxergam o software especialmente como um conjunto de dados que serão processados e outro conjunto de regras de processamento sobre estes dados (algoritmos). Conforme comentado anteriormente, em virtude dos diversos recursos técnicos que podem ser utilizados para construir os softwares, os desenvolvedores têm que fazer escolhas. Essas escolhas devem ser baseadas no contexto de aplicação (inclusive nos algoritmos) e têm como resultado o sucesso da aplicação e a satisfação dos seus usuários, ou poderão trazer problemas ao software, tornando-o lento, insuficiente ou inseguro, por exemplo. Veloso *et al.* (1993) explicam que a escolha das estruturas de dados pode ser um fator decisivo na eficiência do programa.

Em qualquer paradigma de programação (programação estruturada ou programação orientada a objetos, por exemplo), os desenvolvedores devem analisar e projetar a forma de tratamento dos dados, respondendo perguntas como: Como está sendo feito o "tratamento" dos dados? Os dados estão sendo "bem tratados"? Como posso conseguir eficiência no momento da persistência dos dados? Essa persistência é realmente importante? Como o banco de dados consegue obter um desempenho impressionante comparado com o arquivo simples?

A resposta esperada deve estar associada à constatação de que o software faz uso de algoritmos e soluções inteligentes para persistência, acesso e manipulação desses dados. Celes *et al.* (2004) acreditam que o conhecimento de uma linguagem de programação por si só não capacita programadores, pois é

necessário saber usar os recursos de programação de maneira adequada. Para esses autores, a elaboração de um programa envolve diversas etapas, incluindo a identificação das propriedades dos dados e suas características funcionais. Para que possamos fazer um programa atender de maneira eficiente às funcionalidades para as quais ele foi projetado, precisamos conhecer técnicas para organizar de maneira estruturada os dados a serem manipulados. Ou seja, precisamos conhecer as principais técnicas de estruturação de dados, além de uma linguagem de programação.

Um conceito muito forte em Informática é o do reúso. Reusar ideias, soluções, técnicas, modelos, códigos, entre outros, ajudam os desenvolvedores tanto a otimizar as suas tarefas, aproveitando algo que já foi feito (ao invés de ter que "reinventar a roda") e assim melhorando a produtividade, como trazem qualidade ao projeto, uma vez que componentes reusados já podem ter sido massivamente testados e, portanto, a probabilidade de erros de construção ou de lógica tende a diminuir com o reúso.

Nesse cenário de reúso, algumas técnicas auxiliam os desenvolvedores na resolução de problemas, usando estruturas previamente definidas. Cormen *et al.* (2002) descrevem uma estrutura de dados como um meio para armazenar e organizar dados com o objetivo de facilitar o acesso e as modificações.

Estruturas de dados podem ser vistas como o "coração" de qualquer software mais sofisticado. A escolha do tipo de estrutura que será utilizada para persistência e tratamento dos dados faz uma enorme diferença na complexidade da implementação do software. Enquanto a escolha da representação dos dados correta facilita em muito a construção do software, a escolha errada, por outro lado, pode custar um tempo enorme de codificação (inclusive pela dificuldade para manipulação dos dados), além de aumentar a dificuldade de compreensão do código. Segundo Silva

(2007) a estrutura de dados é uma disciplina fundamental para Informática, uma vez que se ocupa com o desenvolvimento de algoritmos, estrutura dos dados, suas representações e transformações para solução de problemas.

Ao trabalhar na sua estrutura de dados é importante que o desenvolvedor não confunda dado e estrutura. Um dado é uma informação armazenada e a estrutura de dados é quem administra os dados. O ideal é que a estrutura de dados seja o mais independente possível dos dados que ela vai armazenar. Dessa forma, pode-se aproveitar a mesma estrutura de dados para diversos tipos de dados.

Para exemplificar o uso das estruturas de dados, será utilizado um contexto bem familiar para os dias de hoje. Caso o leitor deste livro ainda esteja se perguntando em que aplicar estruturas de dados, a resposta pode estar mais próxima do que ele imagina: em seu bolso, por exemplo. Isso mesmo, a agenda do seu celular pode ser vista como uma estrutura de dados. A grande maioria das pessoas utiliza a agenda do celular para armazenar seus contatos. Nesse contexto, a agenda de contatos inicialmente terá que definir como as informações ou dados dos contatos serão armazenados e ainda disponibilizar operações para criar, recuperar, ordenar, atualizar e remover contatos, entre outras.

Estruturas de dados trazem inúmeras vantagens ao projeto e construção de software, podendo ser citadas: apresentam algoritmos e soluções prontas para problemas recorrentes; seus conceitos são independentes de linguagem de programação; são aplicados em diversos cenários de aplicações diferentes; facilitam o entendimento da arquitetura de um banco de dados; agilizam o desenvolvimento de novas aplicações; seus conceitos, uma vez aprendidos, serão utilizados durante toda a vida dos desenvolvedores.

2 Conceitos Básicos

Softwares são desenvolvidos basicamente para obter dados do usuário e processar esses dados com a intenção de fornecer informação. Esses dados precisam ser armazenados na memória e, para isso, a linguagem de programação que está sendo usada precisa, necessariamente, "entender" o tipo de informação que está sendo manipulado. Os bytes correspondentes às informações precisam ser manipulados com precisão na memória, caso contrário, a consistência da informação estaria seriamente comprometida.

2.1. Tipo de Dado

Puga e Rissetti (2003) definem dados como valores que serão utilizados para a resolução de um problema. Em uma aplicação, os dados podem ser fornecidos pelo usuário do programa ou ser originados a partir de processamentos (cálculos) ou de arquivos, bancos de dados ou outros programas.

Antes de conhecer as estruturas de dados, é necessário discutir melhor como os dados são "tratados". Villas *et al.* (1993) comparam a forma como a matemática e a computação lidam com dados: a matemática classifica suas variáveis de acordo com as características, distinguindo constantes, equações, conjuntos e funções; em computação, precisamos identificar os tipos de dados que o computador, a linguagem de programação ou mesmo um algoritmo são capazes de entender.

Para Tenembaum *et al.* (1995) um tipo de dado é um conceito abstrato, definido por um conjunto de propriedades lógicas. Os

tipos de dados podem ser diferenciados pelo conjunto de valores que podem assumir, pelo conjunto de operações que podem ser efetuadas sobre esses dados, pela quantidade de bytes ocupada por eles e pela maneira como os bits serão interpretados e representados na memória. Para Puga e Rissetti (2003), definir o tipo de dado mais adequado é uma questão de grande importância para garantir a resolução de um problema. Por exemplo, a linguagem C apresenta os seguintes tipos primitivos: int, char, float, long int, entre outros.

Moraes (2001) explica que durante a execução de qualquer tipo de programa, os computadores estão manipulando informações representadas pelos diferentes tipos de dados, armazenadas em sua memória principal, em posições de memória denominadas variáveis. Toda variável possui um nome, um tipo de dados e uma informação por ela guardada. Cada variável só pode armazenar um determinado tipo de dado.

2.1.1. Classificação dos Tipos de Dados

Independentemente da linguagem de programação, os tipos de dados podem ser classificados de acordo com as seguintes categorias: tipos primitivos ou tipos derivados; tipos estáticos ou tipos dinâmicos.

Tipos Primitivos

O tipo da informação manipulada pelo computador ou tipo de dado entendido nativamente pela linguagem de programação é chamado de "tipo primitivo". Villas *et al.* (1993) e Veloso *et al.* (1993) apresentam como tipos primitivos: números inteiros, números reais, caracteres, lógicos e ponteiros. Para os autores citados, esses tipos de dados são os mais frequentes nas linguagens de programação.

Cada um dos tipos primitivos tem um conjunto de valores e operações:

a) Inteiro: representa uma quantidade contável de objetos. Algumas operações: soma, subtração, multiplicação, divisão, resto. Veloso *et al.* (1993) ainda apresentam as operações de comparação entre inteiros: igualdade, diferença, maior do que, menor do que, maior ou igual, menor ou igual. Exemplos: -45; 2; 1037.
b) Real: representa um valor que pode ser fracionado. Algumas operações: soma, subtração, multiplicação, divisão. Assim como fizeram para os inteiros, Veloso *et al.* (1993) ainda apresentam as operações de comparação entre reais: igualdade, diferença, maior do que, menor do que, maior ou igual, menor ou igual. Exemplos: -4,78; 1,25; 2,333...
c) Lógico: representa dois estados. Algumas operações: E (conjunção); OU (disjunção); NÃO (negação). Exemplos: [Verdadeiro, Falso]; [V;F]; [0,1].
d) Caracter: representa um símbolo, que pode ser dígito, letra ou sinal. Algumas operações: igualdade, diferença, concatenação. Exemplos: "A", "X", "1", "+".
e) Ponteiro: representa o endereço de um dado na memória.

Ao definir que o tipo de dado escolhido é inteiro, o programador tem ao seu dispor uma estrutura de armazenamento que permite manipular na memória informações contidas no universo dos números inteiros (bit por bit).

!	O conjunto dos números inteiros é infinito e, considerando que a memória da máquina é finita, é compreensível ser impossível representar esse conjunto na sua totalidade. Por causa disso, apenas uma "faixa de valores" é permitida. Cada linguagem especifica o valor mínimo e o valor máximo que pode ser armazenado.

#	É um erro comum tentar atribuir um valor que não está compreendido entre a faixa de valores permitida pela linguagem de programação. O programador deve ter bastante atenção, pois esse é um erro de execução difícil de encontrar.

Recaptulando, os tipos primitivos equivalem aos tipos básicos predefinidos na linguagem, podendo ser utilizados tanto para a declaração de dados dos programas como para a definição de novos tipos definidos pelo usuário. Villas *et al.* (1993) explicam que tipos primitivos são aqueles a partir dos quais podemos definir os demais tipos, não importando como são implementados e manipulados. Essencialmente, são os números, valores lógicos e caracteres que são identificados na maioria das linguagens.

Na linguagem C, podemos citar como exemplos de tipos primitivos: int, float, char, double, entre outros. Assim, aplicações em C podem manipular dados do tipo int (inteiro), como a idade de uma pessoa, float (real), como uma quantia em dinheiro, e da mesma forma é feito para os demais tipos primitivos.

Tipos Derivados

Linguagens de programação de alto nível (por exemplo: Pascal, C e Java) permitem a construção de novos tipos de dados, chamados de “tipos derivados”. Esses novos tipos podem ser construídos a partir de tipos primitivos e/ou derivados. Como exemplo, Puga e Rissetti (2003) citam o tipo registro de paciente, composto por nome (conjunto de caracteres), idade (inteiro) e peso (real).

Para o programador essa é uma prática muito útil, pois permite organizar conjuntos de variáveis que se combinam para formar determinado conceito no programa. Por meio dessa prática, fica mais simples gerenciar o código, e ainda pode ser evitada uma superpopulação de variáveis.

Por exemplo, vamos imaginar um sistema que precisa armazenar a descrição de um produto e suas datas de fabricação e de validade. Considerando que a data é composta por dia, mês e ano, um programador inexperiente declara 7 (sete) variáveis, uma para cada informação. O excesso de variáveis pode representar um futuro problema.

Um programador mais experiente percebe que existe o conceito "data" e este é utilizado em duas informações do produto, sendo elas: fabricação e validade. Neste contexto, é mais interessante definir o tipo de dado derivado "data" e utilizá-lo sempre que necessário.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 2.1) e Java (ver Código Java 2.1) para o problema descrito.

Código C 2.1 – Definição do tipo derivado de dados

Em C:

```
typedef struct{
   int dia, mes, ano;
}Data;
```

Construído com tipo primitivo.

```
typedef struct{
   char nome[40];
   Data fabricacao, validade;
}Produto;
```

Construído com tipos primitivo e derivado.

Código Java 2.1 – Definição do tipo derivado de dados

Em Java:

```
public class Data{
   private int dia, mes,
           ano;
}
```

Construído com tipo primitivo.

```
public class Produto{
  private String nome;
  private Data fabricacao;
  private Data validade;
}
```

Construído com tipos primitivo e derivado.

Percebe-se que essa abordagem (com tipos derivados) permite ao programador segmentar o conjunto de informações, organizá-las e manipulá-las de forma mais simples, além de permitir a reutilização, que é uma excelente prática de programação. Então, qualquer situação que precise manipular uma data pode utilizar a solução proposta "tipo data".

Definido o tipo de dado, primitivo ou derivado, uma variável pode ser declarada com objetivo de armazenar informações na memória e, consequentemente, conseguir realizar operações. Uma variável recebe um identificador e a ela é associado um tipo de dado. Cada tipo de dado aceita determinadas operações: com o tipo inteiro, por exemplo, podem ser realizadas as 4 (quatro) operações básicas da aritmética (soma, subtração, divisão e multiplicação).

Analisando o tipo data (que contém dia, mês e ano), algumas operações específicas poderiam ser planejadas, como a soma de duas datas e a ordenação entre datas.

Com objetivo de promover ainda mais a independência do tipo de dado derivado definido, um conjunto de operações podem ser associadas ao tipo de dados, determinando o que pode ser manipulado nos dados armazenados. Essa abordagem é conhecida como tipo abstrato de dado.

2.2. Tipo Abstrato de Dados

O tipo abstrato de dados (TAD) define uma coleção de valores e um conjunto de operações que podem ser realizadas sobre esses valores. O TAD pode ser definido como um modelo matemático constituído por um conjunto de valores e uma coleção de operações que podem ser realizadas sobre esses valores.

Matematicamente é um par (V,O), onde:

- V: é um conjunto de valores;
- O: Conjunto de operações sobre esses valores.

O objetivo aqui não é resolver todo o sistema, mas apenas uma pequena fatia, bem específica, que está interessada em manipular um conjunto de valores. Por exemplo, vamos pensar em uma data. Para manipular a data, é necessário conhecer os valores que a data pode assumir e as operações que podem ser realizadas sobre ela. Assim sendo, temos:

- V: dia, mês e ano;
- O: {somar, subtrair, verificar igualdade, validar, ordenar}.

Nesse momento não interessa como essas operações são realizadas, mas apenas o que elas permitem fazer.
Segundo Tenembaum *et al.* (1995), um novo tipo de dados é composto por objetos dos tipos de dados já existentes, além de uma especificação de como esse objeto será manipulado em conformidade com as operações definidas para ele.

Assim como na Matemática as constantes, as funções e as demais elementos são diferenciados, na Informática os dados são identificados de acordo com as suas características. Segundo Villas *et al.* (1993) por exemplo, poderíamos definir que fruta é um tipo de dado que poderia assumir valores como maçã, pera, banana, abacaxi etc. As operações sobre esse tipo dizem respeito às suas ações de manipulação, como por exemplo, comparar, comprar, comer e servir. Usando um contexto mais comum na área, pode-se pensar no tipo inteiro. Quando se usa um tipo inteiro, não se está interessado em saber como são manipulados os detalhes da sua implementação, ou seja, os bits (baixo nível).

Os TADs se preocupam especialmente com os dados e com o seu tratamento (ou seja, as operações sobre esses dados, o que pode ser feito com eles), sem se preocupar especificamente com a

forma como eles são manipulados (implementação). A ideia principal do TAD é possibilitar ao programador separar "o que fazer" de "como fazer". Existe uma evidente separação entre os cenários "o que" e "como". Horowitz e Sahni (1987) explicam que é importante distinguir entre a especificação da estrutura de dados e a sua realização dentro de uma linguagem de programação.

Nesse contexto, há uma separação clara entre conceito e implementação. O conceito de abstração dos dados está relacionado ao fato de que para o usuário do TAD não importa como estão sendo feitas as operações (como estão implementadas) e sim para que serve aquele "tipo abstrato de dados". Segundo Preiss (2000), a abstração pode ser pensada como um mecanismo de supressão de detalhes irrelevantes e, ao mesmo tempo, de ênfase em detalhes relevantes. Um benefício importante da abstração é que por meio dela torna-se fácil para o programador pensar sobre o problema a ser resolvido. Ou seja, o TAD é independente da sua implementação.

A ideia central do TAD é encapsular (esconder) de quem usa um determinado tipo, a forma concreta com que ele foi implementado. Segundo Preiss (2000), o encapsulamento auxilia o projetista por meio da omissão da informação. A omissão da implementação de um objeto resulta na independência conceitual, permitindo que sejam feitas alterações na implementação sem que seja necessário modificar o código do usuário.

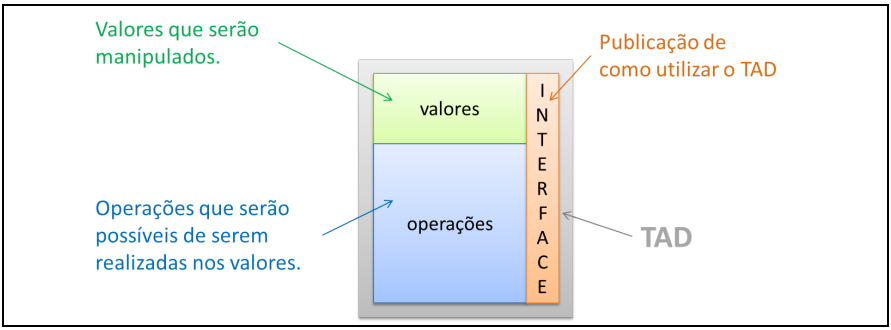
Celes *et al.* (2004) exemplificam o TAD dizendo que se criamos um tipo para representar um ponto no espaço, um cliente desse tipo usa-o de forma abstrata, com base apenas nas funcionalidades por ele oferecidas. A forma pela qual esse tipo foi efetivamente implementado (armazenando cada coordenada em um campo ou agrupando todas em um vetor) passa a ser um detalhe de implementação, que não deve afetar o uso do tipo nos mais

diversos contextos. Com o TAD, desacoplamos a implementação do uso, facilitamos a manutenção e aumentamos o potencial de reutilização do tipo criado. Através do TAD, a implementação do tipo pode ser alterada sem afetar seu uso.

Para trabalhar com o TAD é necessário apenas conhecer qual é o tipo de informação (valores) e a interface de acesso (conjunto de operações que podem ser realizadas), conforme observado na Figura 2.1.

Analogamente, o TAD pode ser entendido com uma "caixa preta", que permite armazenar e manipular informações.

Figura 2.1 – Estrutura do TAD.



Para os desenvolvedores que trabalham com orientação a objetos, pode-se fazer uma analogia e dizer que a essência do TAD é semelhante ao conceito "classe" de Java (no sentido de que a classe possui atributos, métodos e interface de acesso).

!	Apenas as operações definidas no TAD (interface) podem ser realizadas. Quanto maior o número de operações, maior utilidade terá o TAD. Por meio da interface, o programador consegue garantir que nenhuma operação ilegal será realizada e apenas as que foram definidas serão permitidas.

O TAD também pode ser entendido como um módulo independente do sistema. Essa definição permite a percepção de 4 (quatro) características motivadoras, apresentadas a seguir:

- ✓ Modelagem de solução simples e objetiva: uma vez que não é necessário ter o domínio geral sobre o sistema que está em desenvolvimento, e sim sobre apenas uma parte muito específica, com operações determinadas, propor e desenvolver uma solução é muito menos complexo;

- ✓ Reutilização: uma vez desenvolvido e testado, o TAD pode ser utilizado em qualquer outro sistema que precise de solução semelhante, trazendo agilidade à atividade de desenvolvimento;

- ✓ Abstração: programadores podem fazer uso de um TAD definido por terceiros sem a necessidade de entendimento do seu funcionamento, sendo apenas necessário saber o que pode ser feito com o TAD;

- ✓ Atualização Independente: novas operações podem ser adicionadas ao TAD, sendo este compilado de forma independente e atualizado no sistema.

Para Tenembaum *et al.* (1995), especificando-se as propriedades matemáticas e lógicas de um tipo de dado, o TAD será uma diretriz útil para implementadores e uma ferramenta de apoio para os programadores que queiram usar o tipo de dado corretamente.

Um TAD possui formas particulares de implementação, denominadas “estrutura de dados”.

2.3. Estruturas de Dados

A partir do entendimento do TAD, pode-se melhor definir o que vem a ser uma estrutura de dados: uma estrutura de dados é uma forma particular de implementar um TAD; a implementação de um TAD escolhe uma estrutura de dados para representá-lo; as estruturas de dados são compostas por tipos de dados mais as operações a serem realizadas sobre eles. Para Veloso *et al.* (1993), uma estrutura de dados retrata as relações lógicas entre os dados de modo análogo ao uso de um modelo matemático, para espelhar alguns aspectos de uma realidade física.

!	Uma estrutura de dados é a implementação de um TAD. Ela determina o(s) tipo(s) de dado(s) (primitivo e/ou derivado) do(s) valor(es) que será(ão) manipulado(s) e implementa operações definidas sobre esse(s) dado(s).

O programa que fará uso da estrutura de dados não possui acesso direto aos valores armazenados. Só é permitido o acesso a estrutura por meio de duas operações. Dessa forma, o programador garante que, considerando o exemplo da data, na alteração do mês não poderá ser atribuído valor nulo, negativo ou superior a 12 (doze).

Vamos explicar com um pouco mais de detalhes o exemplo da agenda de contatos como sendo uma estrutura de dados. A grande maioria das pessoas utiliza a agenda do celular para armazenar seus contatos. Desse modo, a agenda de contatos inicialmente terá que definir como as informações ou dados dos contatos serão armazenados e ainda disponibilizar operações para criar, recuperar, ordenar, atualizar e remover contatos. Lembrando que uma estrutura de dados tem como principais objetivos armazenar e organizar dados e facilitar seu acesso e suas modificações, a agenda de celular mantém os dados dos contatos telefônicos dos usuários organizados seguindo alguma lógica (geralmente

armazenados em ordem alfabética, sendo composto por dados como nome, telefone fixo, telefone celular e email, por exemplo) e disponibiliza operações para o usuário manipular os dados (como acrescentar um novo contato, alterar, consultar ou remover contatos existentes).

Tecnicamente falando, definir como os dados da estrutura serão armazenados é uma tarefa crucial para o desempenho do software. Se a estrutura não armazena os dados de forma adequada então será muito mais complicado manipulá-los de forma eficiente. A escolha de como guardar as informações deve levar em consideração as operações que serão disponibilizadas pela agenda. No caso da agenda de contatos, seria interessante manter os contatos em ordem alfabética para facilitar a busca. Outro aspecto técnico considerado no projeto e na construção da agenda é a definição dos dados que serão apresentados ao usuário. Essa definição está ligada ao conceito de abstração, que se preocupa em apresentar dados essenciais e esconder dados que não são necessários para o usuário. Nesse caso, a organização interna da agenda não precisa e não deve ser exposta ao usuário, uma vez que os detalhes de como ela foi construída não são necessários para o contato do usuário com a sua agenda.

É importante lembrar que o objetivo desse usuário é utilizar a agenda por meio das suas operações, de forma mais eficiente possível. Para manipulação da agenda, o usuário precisa apenas conhecer as suas operações (inserir, recuperar, atualizar, remover contato, saber quantos contatos estão na agenda, por exemplo), por isso seus detalhes técnicos podem ser abstraídos. Esse conjunto de operações disponíveis para que o usuário possa usar a agenda é chamado de interface da agenda.

Uma vantagem de trabalhar os conceitos de interface e abstração é que estes favorecem a flexibilidade para implementação dos detalhes da agenda, de forma independente, para cada celular. Ou

seja, cada celular pode implementar a sua agenda de contatos de forma totalmente diferente dos outros e de acordo com os seus requisitos, que podem ser obter mais performance, ser mais confiável ou gastar menos memória, por exemplo. Independentemente dos requisitos e da forma como a agenda foi implementada, o conjunto básico de operações oferecidas por ela é sempre o mesmo, o que facilita a vida do usuário, que pode reusar o seu conhecimento sobre a manipulação das agendas. Assim, se o usuário tiver que trocar de celular não vai ter que aprender novamente como usar uma agenda de contatos. Mantida a interface, o usuário troca uma agenda que já não atende mais as suas necessidades por outra mais adequada, a partir do princípio de troca da implementação, mas sem mudança na interface.

Indo um pouco mais além da troca de aparelho celular em si, ainda podemos dizer que as implementações diferentes da agenda podem ser usadas pelo mesmo celular. Ou seja, caso se deseje modificar a implementação de uma agenda, devido a mudanças de requisitos, por exemplo, pode-se mudar a implementação sem que seja alterada a interface. Desse modo, o mesmo celular poderia escolher um tipo de implementação para sua agenda.

Por fim, diante de tudo o que foi exposto, pode-se dizer que se a agenda e o celular assinarem a mesma interface, eles poderão ter independência de implementação: uma mesma agenda poderá ser utilizada por vários celulares, como mostrado na Figura 2.2 ; várias implementações de agendas diferentes estarão disponíveis para uso por um mesmo celular, como mostrado na Figura 2.3.

Relembrando as inúmeras vantagens que as estruturas trazem ao projeto e construção de software, podemos citar: apresentam algoritmos e soluções prontas para problemas recorrentes; seus conceitos são independentes de linguagem de programação; são aplicados em diversos cenários de aplicações diferentes; facilitam o entendimento da arquitetura de um banco de dados; agilizam o

desenvolvimento de novas aplicações; seus conceitos, uma vez aprendidos, serão utilizados durante toda a vida dos desenvolvedores.

Figura 2.2 – Uma mesma implementação de agenda pode ser utilizada por celulares diferentes

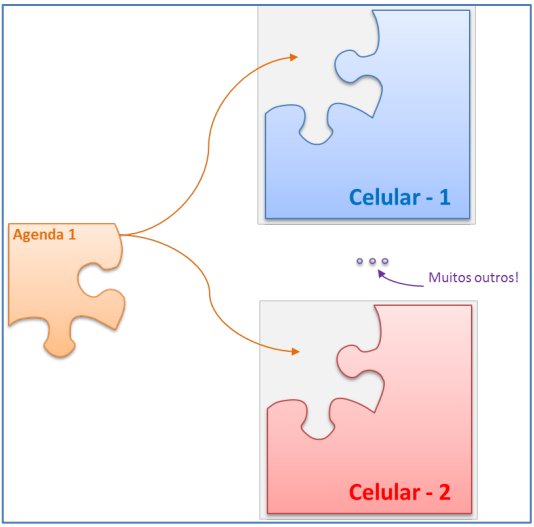
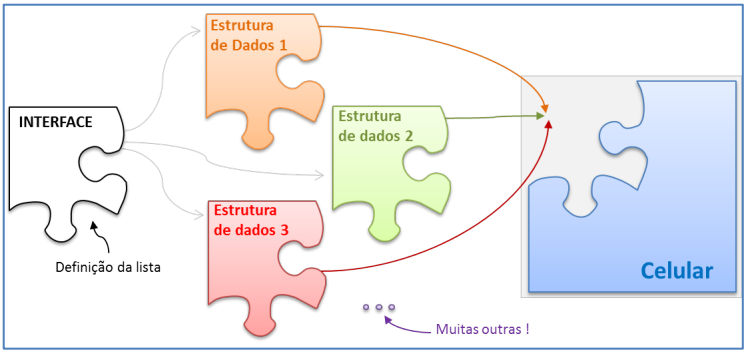


Figura 2.3 – Implementações diferentes de agendas estão disponíveis para uso pelo mesmo celular



Existe muita confusão nos conceitos de tipo de dado, TAD e estrutura de dado. Uma vez esclarecidas as diferenças entre esses conceitos, passaremos a analisar como podem ser feitas as suas implementações.

2.4. Formas de Implementação

Uma estrutura de dados pode ser implementada de diferentes formas. A principal preocupação é com relação ao gerenciamento da memória para armazenamento das informações. Quem vai indicar o critério para escolha é a aplicação, e o programador precisa estar bastante atento aos requisitos do sistema.

Basicamente, a decisão mais crítica é a flexibilidade quanto ao tamanho da estrutura, que pode ser estática ou dinâmica.

2.4.1. Estruturas Estáticas

Os tipos estáticos possuem estruturas definidas previamente, na aplicação, antes de serem realizadas as operações sobre eles. Villas *et al.* (1993) explicam que definido um tipo estático de dado, ele não poderá conter mais elementos do que o previsto inicialmente. Para esses autores, não só a limitação de elementos torna um tipo de dados estático, como também seus sucessores e antecessores não se modificam – o conteúdo pode ser modificado, mas não as suas posições na memória. A partir dessas limitações, a estrutura permanece estática. Villas *et al.* (1993) citam como exemplo de tipo estático o vetor (ou array), que é um agregado homogêneo de dados. Segundo Celes (2004), estruturas de dados estáticas são baseadas na utilização de formas primitivas de estruturação de dados disponibilizadas pela linguagem de programação, como vetores e tipos estruturados.

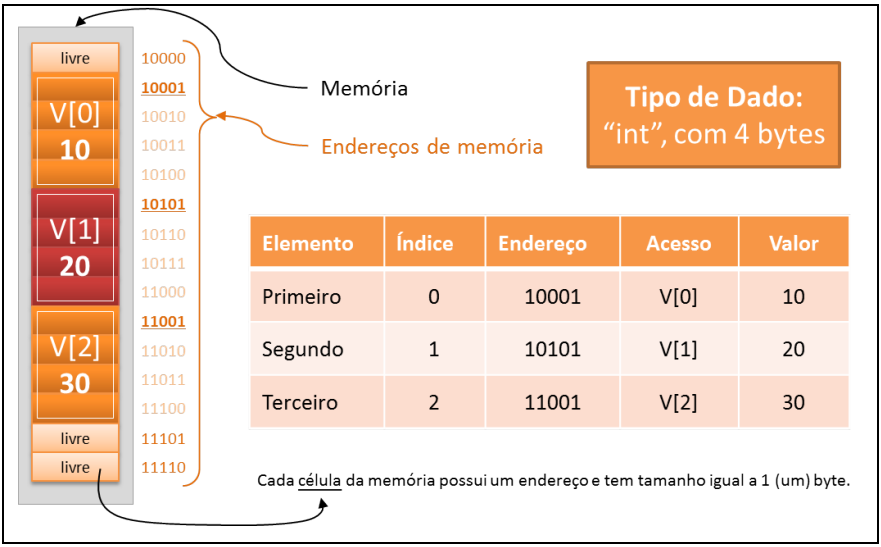
A principal limitação da estrutura implementada na forma estática é o seu tamanho "engessado", ou seja, uma vez definido não pode

mais ser alterado. Em contrapartida, apresenta desenvolvimento simples e possibilida acesso direto aos elementos contidos na estrutura.

Conforme comentado, a estrutura usada para armazenamento é o vetor. Segundo Damas (2007), um vetor é um conjunto de elementos consecutivos, todos do mesmo tipo, que podem ser acessados individualmente.

Na Figura 2.4 está graficamente representado um vetor de inteiro (considerando que cada elemento ocupa quatro bytes), com tamanho 3 (três), ou seja, capacidade para armazenar até 3 (três) números. Os elementos são posicionados consecutivamente na memória, considerando que o primeiro elemento está posicionado no byte "10001".

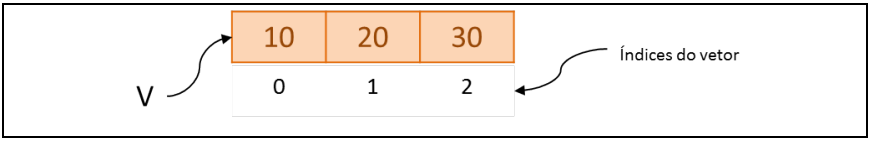
Figura 2.4 – Estrutura estática vetor



Elemento	Índice	Endereço	Acesso	Valor
Primeiro	0	10001	V[0]	10
Segundo	1	10101	V[1]	20
Terceiro	2	11001	V[2]	30

A Figura 2.5 apresenta uma outra representação gráfica para o mesmo vetor. Essa é a representação mais adotada pela comunidade de programadores.

Figura 2.5 – Representação gráfica de um vetor



2.4.2. Estruturas Dinâmicas

Diferente do que ocorre com a estrutura estática (vetor como estrutura de armazenamento), a estrutura dinâmica consegue alterar sua capacidade de armazenamento em tempo de execução, de acordo com a necessidade do usuário.

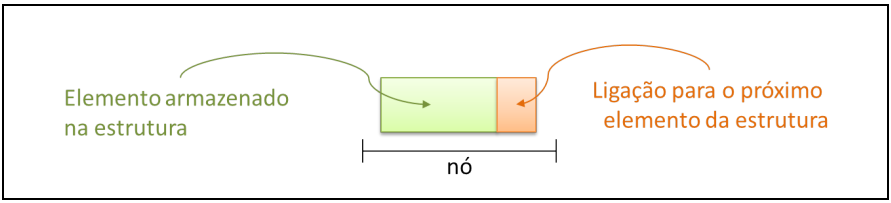
O princípio da estrutura dinâmica é espalhar os elementos na memória, não existindo a necessidade de estes elementos estarem agrupados (consecutivos). É possível estabelecer a união da estrutura fazendo com que os elementos possuam ligações entre eles.

Essa ligação entre os elementos é chamada de "encadeamento". Cada elemento do encadeamento é denominado "nó". Nesse caso, a estrutura dinâmica pode ser entendida como um encadeamento de nó.

Segundo Celes (2004), estruturas dinâmicas oferecem suporte adequado para inserção e remoção de elementos. Para cada elemento, essas estruturas alocam dinamicamente memória para seu armazenamento, portanto não são estruturas pré-dimensionadas. O número de elementos que podemos armazenar nessas estruturas é arbitrário.

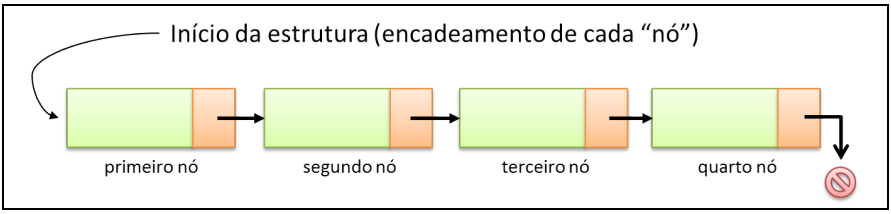
Na Figura 2.6 está representado graficamente o nó, constituído do elemento que está sendo armazenado, e uma ligação (encadeamento) para o próximo elemento da estrutura. Esse nó apresentado no exemplo é classificado como simples, por possuir um único encadeamento.

Figura 2.6 – Nó, elemento da estrutura encadeada



Na Figura 2.7 está representada uma estrutura encadeada. Cada elemento (nó) possui um valor armazenado e uma conexão (ligação) para o próximo elemento da estrutura. Quanto ao último elemento (quarto nó), a sua conexão para o próximo elemento precisa indicar que não existe próximo.

Figura 2.7 – Estrutura encadeada simples



Cada elemento (nó) da estrutura é alocado dinamicamente (de acordo com a demanda da aplicação), o que permite a flexibilidade impossível de ser obtida através da implementação estática.

Com o encadeamento consecutivo dos elementos, é possível obter essa estrutura dinâmica. Por outro lado, não é possível acessar

diretamente um elemento, uma vez que apenas é conhecido o primeiro nó e este é quem vai dar acesso aos demais elementos.

Analisando o gerenciamento da memória, enquanto na estrutura estática todos os elementos precisam estar alocados consecutivamente, em um único bloco, na estrutura dinâmica os elementos podem estar espalhados, não existindo a necessidade de buscar um bloco de memória disponível para alocá-los. Considerando uma coleção de tamanho expressivo e o dinamismo das operações de inserção e remoção de elementos, essa decisão pode ser fundamental.

O acesso aos elementos também é uma questão que merece atenção. Na estrutura estática os elementos pode ser acessados diretamente (por meio do índice do vetor), enquanto nas estruturas dinâmicas esse acesso direto não está disponível. Para as estruturas dinâmicas, o acesso é feito partindo-se do primeiro elemento, e sequencialmente a estrutura é percorrida, até que se chegue ao elemento buscado.

Por fazer uso de vetores, a implementação das estruturas estáticas é mais simples do que as estrutura dinâmicas, que utilizam essencialmente ponteiros (apontadores).

Embora exista um pouco de receio sobre o uso de ponteiros, por parte de alguns desenvolvedores, especialmente para iniciantes em programação, de fato o seu uso não representa um problema. Este livro apresenta implementações com o uso de ponteiros, de forma simples e objetiva.

O Quadro 2.1, apresenta um resumo dessas características discutidas.

Quadro 2.1 – Comparativo entre as estruturas estáticas e dinâmicas.

Característica	Estática	Dinâmica
Tamanho	Fixo	Flexível
Estrutura	Vetor	Encadeamento de "nó"
Acesso	Direto	Sequencial
Armazenamento	Sequencial	Aleatório
Complexidade	Simples	Elaborada

O ponto-chave a ser analisado pelo programador para escolha da implementação é a disponibilidade de memória da máquina e a necessidade da aplicação.

2.4.3. Aspectos Técnicos de Desenvolvimento

As implementações deste livro utilizaram como base as linguagens de programação C (Padrão) e Java versão 1.6.

2.5. Exercícios Propostos

1) O que é um tipo de dado? Cite 03 (três) exemplos.

2) Existe diferença entre os tipos primitivo e derivado? Cite 04 (quatro) características de cada tipo.

3) É possível representar na memória qualquer número pertencente ao conjunto dos números inteiros? Por quê?

4) É possível construir um tipo de derivado a partir de outro tipo derivado? Cite um exemplo.

5) O que é um Tipo Abstrato de Dados (TAD)?

6) Cite 04 vantagens conquistadas com o uso do TAD.

7) Cite dois exemplos de Tipos Abstrados de Dados (TAD).

8) O que é Estrutura de Dados?

9) Qual é a diferença entre tipo de dado, tipo abstrato de dado e estrutura de dados?

10) Desenvolver software que faz uso de estrutura de dados é mais ágil/simples? Cite 4 (quatro) vantagens e 2 (duas) desvantagens.

11) Explique o funcionamento da forma estática de gerenciamento da memória. Cite 04 características.

12) Explique o funcionamento da forma dinâmica de gerenciamento da memória. Cite 04 características.

3 Tipos de Estruturas

No desenvolvimento de estruturas de dados, uma grande preocupação do programador é como será a manipulação dos dados. Escolhas equivocadas podem resultar, por exemplo, em uma complexidade desnecessária ao longo da construção do software, ou até mesmo no não atendimento aos requisitos propostos. Nesse ponto é fundamental a escolha de uma estrutura que permita facilidade e flexibilidade, além de atender aos requisitos do sistema.

Na literatura existem diversas estruturas conhecidas, que vão desde as mais simples até as mais complexas. O fato de existirem estruturas prontas não impede que outras estruturas possam ser construídas.

Uma forma de classificar as estruturas refere-se à disposição dos seus elementos. Quanto a esta classificação, as estruturas de dados podem ser lineares ou hierárquicas (não lineares). As lineares são aquelas em que os dados estão representados no mesmo nível de acesso, não existindo dependência entre eles. São exemplos dessas estruturas: lista, pilha, fila, deque, entre outras. Estruturas hierárquicas são aquelas em que os dados estão representados hierarquicamente, ou seja, existe dependência entre eles. São exemplos de estruturas hierárquicas: árvore, grafos, entre outras.

Não importa com que tipo de dados estaremos trabalhando, a primeira operação a ser efetuada sobre ele é a criação. Uma vez criado, podemos fazer inclusões ou remoções de dados na estrutura. A operação que varre todos os dados armazenados numa

estrutura é o percurso, podendo também ser realizada uma busca por algum valor específico na estrutura.

Nenhuma estrutura de dados única funciona bem para todos os propósitos, assim é importante conhecer os pontos fortes e as limitações de várias delas. Segundo Preiss (2000), três coisas são necessárias para desenvolver um sólido conhecimento de uma estrutura de dados: deve-se saber como a informação é armazenada na memória do computador; deve-se ter familiaridade com os algoritmos que manipulam a informação contida em uma estrutura de dados; devem ser compreendidas as características de desempenho das estruturas de dados de modo que seja possível selecionar dentre elas a que mais atende a uma aplicação em particular.

A seguir abordaremos mais algumas características das estruturas lineares e hierárquicas.

3.1. Estruturas Lineares

Conforme explicado anteriormente, nas estruturas lineares os dados estão representados no mesmo nível de acesso e não existe dependência entre eles.

!	Estruturas lineares são aquelas em que os dados estão representados no mesmo nível de acesso, não existindo dependência hierárquica entre eles.

As estruturas lineares possuem algumas características próprias. Supondo uma estrutura formada pelos elementos E_1, E_2,... E_n podemos deduzir as seguintes informações sobre a estrutura:

- É formada por n elementos;
- E_1 é o seu primeiro elemento;
- E_n é o seu último elemento;

- Sobre a disposição dos elementos: E_1 só tem sucessor, e este é E_2; E_n só tem predecessor, e este é E_{n-1}; os demais elementos têm predecessores e sucessores.

São exemplos de estruturas lineares:

- Filas de bancos, em que as pessoas são organizadas e atendidas por ordem de chegada;
- Listas de alunos em um diário acadêmico, em que os nomes são apresentados em ordem alfabética.

As principais operações genéricas, encontradas nos diversos tipos de estruturas lineares, são:

- Inserção: operação que incorpora ou insere um determinado elemento em uma estrutura linear;
- Retirada: operação que retira um determinado elemento de uma lista linear;
- Percurso: operação que percorre uma estrutura linear, varrendo seus elementos;
- Busca: operação que procura um elemento específico na estrutura linear.

Voltando ao exemplo do diário acadêmico, poderíamos imaginar as operações que equivaleriam respectivamente às seguintes ações: varrer o diário, lendo aluno por aluno (percurso); procurar um determinado aluno pelo seu nome (busca); inserir um novo aluno no diário, a partir do seu nome (inserção); retirar um aluno transferido do diário, a partir do seu nome (retirada).
Existem diversos tipos de estruturas lineares, algumas com características bem peculiares, dependendo da forma como seus elementos são manipulados a partir das operações permitidas na estrutura.

Dentre esses tipos, podemos citar:

- Lista Genérica: é uma estrutura de dados que permite representar um conjunto de dados de forma a preservar a

ordem entre eles. Na lista genérica, as operações de inserção e retirada podem ser realizadas em qualquer lugar da estrutura. Um exemplo de lista genérica é uma lista de convidados para uma festa.

- Pilha: representa uma estrutura que requer acesso aos seus dados de forma especial. A ideia fundamental da pilha é que todo o acesso aos seus elementos se dê sempre por meio do topo. Na pilha, quando um novo elemento é inserido, ele passa a ser o elemento do topo. A remoção de um elemento, por sua vez, exclui sempre o elemento do topo. Para entender o funcionamento dessa estrutura, podemos pensar em uma pilha de pratos.

- Fila: a ideia fundamental dessa estrutura de dados é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início da fila. Um exemplo dessa estrutura é uma fila de banco, em que os clientes são atendidos por ordem de chegada.
- Deque: é uma estrutura linear na qual as operações de inserção e retirada são efetuadas no início e/ou no final da estrutura. Um exemplo de deque é a composição de um trem, em que as inserções e retiradas de vagões são feitas nas extremidades.

- Combinação de Listas: é representada por uma combinação de estruturas lineares com o objetivo de compor estruturas mais complexas. Um exemplo dessa estrutura é o conceito matemático de matriz bidimensional, representada a partir da combinação de várias linhas e colunas.

Explicações mais detalhadas sobre algumas das estruturas lineares citadas serão vistas nas próximas seções (combinações de listas não serão tratadas neste livro).

!	Para cada estrutura apresentada, as implementações completas, nas linguagens C e Java, encontram-se localizadas nos apêndices deste livro.

3.2. Estruturas Hierárquicas

Conforme explicamos na seção anterior, as estruturas lineares apresentam uma característica comum que é a relação sequencial entre os seus elementos. Segundo Moraes (2001), as listas apresentam grande flexibilidade sobre as representações contíguas de estruturas de dados, porém sua forte característica sequencial representa o seu ponto fraco. O arranjo estrutural das listas faz com que a movimentação ao longo delas seja feita um nó por vez.

As estruturas não lineares, por sua vez, definem outros tipos de relações entre seus elementos. Nas estruturas hierárquicas, por exemplo, alguns elementos são hierarquicamente subordinados a outros elementos. Segundo Celes *et al.* (2004), a importância das estruturas lineares é inegável, mas elas não são adequadas para representar dados que sejam dispostos de maneira hierárquica (CELES *et al.*, 2004, p. 185). Nesse contexto, se fazem importantes estruturas com outras características, como as estruturas hierárquicas.

!	Estruturas hierárquicas são aquelas em que alguns elementos estão hierarquicamente subordinados a outros.

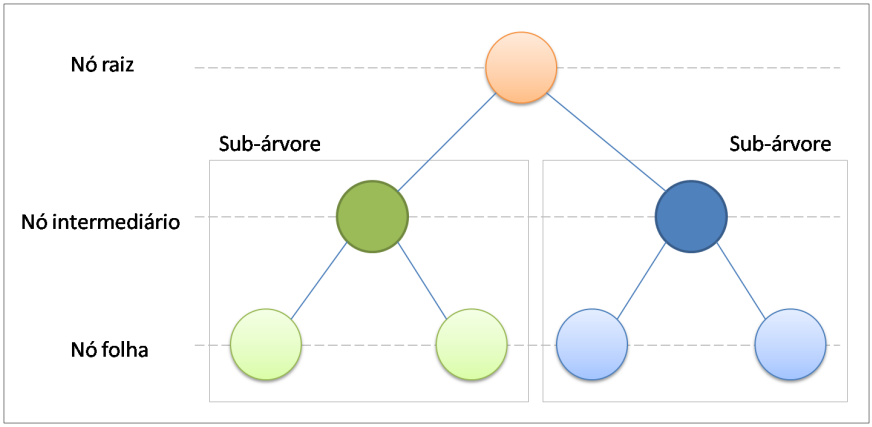
Segundo Preiss (2000), a árvore representa uma das estruturas não lineares mais importantes.
As árvores são estruturas de dados que caracterizam uma relação de hierarquia ou de composição entre os seus elementos, chamados nós.

As árvores possuem algumas características próprias:

- São definidas como sendo um conjunto finito de um ou vários nós;
- Existe um nó específico chamado raiz da árvore que contém zero ou mais subárvores;
- Os demais nós formam conjuntos disjuntos, sendo que cada subconjunto representa uma árvore (subárvore), vazia ou não;
- Nós com filhos são chamados de nós intermediários, e nós sem filhos são chamados de folhas ou nós externos.

Na Figura 3.1 é possível observar graficamente os conceitos apresentados.

Figura 3.1 – Estrutura de dados árvore



São exemplos de árvores:

- A representação de um livro: um livro é dividido em capítulos; cada capítulo é dividido em seções; as seções são divididas em subseções ou tópicos.
- O organograma de uma empresa, representando a hierarquia das responsabilidades e subordinação dos cargos.

- A estrutura de uma universidade: a universidade possui câmpus; os câmpus possuem centros; os centros são compostos por departamentos; os departamentos possuem uma série de disciplinas ligadas a eles; os alunos se matriculam em disciplinas.

Um exemplo de uso de árvores em programação é a organização de pastas (diretórios) em um computador e dos seus respectivos arquivos (documentos, por exemplo). Outros exemplos de uso de árvores é a aplicação dessas estruturas em jogos, como no jogo da velha, e em tomada de decisão em geral.

Moraes (2001) explica que as estruturas de dados organizadas como árvores têm provado ser de grande valor para uma ampla faixa de aplicações, entre outras, representações de expressões algébricas, como um método eficiente para pesquisas em grandes conjuntos de dados, aplicações de inteligência artificial e algoritmos de codificação.

Uma vez que este livro pretende abordar as estruturas lineares, as árvores foram utilizadas apenas para que o leitor entenda a diferença básica entre as representações linear e hierárquica.

3.3. Exercícios Propostos

1) Qual é a principal função de uma estrutura de dados?

2) Uma estrutura de dados serve para qualquer aplicação? Por quê?

3) Como as estruturas podem ser classificadas?

4) O que são estruturas lineares?

5) O que são estruturas não lineares?

6) O que é uma lista genérica?

7) O que é uma pilha?

8) O que é uma fila?

9) O que é um deque?

10) O que é uma combinação de listas?

11) O que é uma árvore?

12) Considerando a estrutura árvores, existe diferença entre nó, nó raiz, nó intermediário e nó folha? Justifique sua resposta (inclua um desenho).

4 Listas Genéricas

Uma lista genérica é uma estrutura linear em que as operações de inserção e retirada podem ser realizadas em qualquer lugar da estrutura.

Supondo uma lista formada pelos elementos E_1, E_2,... E_n, se desejarmos adicionar um novo elemento, ele poderá passar a ser o predecessor ou sucessor de qualquer outro elemento da lista. Ou seja, a inserção desse elemento pode implicar em redefinir a disposição dos demais elementos na lista. Se desejarmos retirar um elemento da lista genérica, de forma semelhante, poderemos excluir qualquer elemento.

As listas podem usar ou não critérios de ordenação. Inicialmente, vamos pensar em uma lista sem critério de ordenação. Um exemplo dessa estrutura é uma lista de convidados para uma festa, em que os nomes dos convidados podem estar em qualquer disposição na lista (não obrigatoriamente em ordem alfabética) e inserções e retiradas também podem ser feitas para qualquer convidado, em qualquer parte da lista. Outro cenário é o uso de uma lista semelhante, mas dessa vez com o uso de algum critério de ordenação. Um exemplo dessa estrutura é uma lista de convidados para uma festa, em que os nomes dos convidados podem estar dispostos em ordem alfabética e, assim, inserções e retiradas de convidados na lista devem respeitar a ordem alfabética, dependendo do seu nome.

Outros exemplos de listas: lista de itens de supermercado, lista de aviões que devem decolar em um controle de tráfego aéreo.

Um exemplo de uso de listas em programação é o uso de uma lista de processos esperando pelo processador em um sistema operacional.

4.1. Operações

Algumas operações são necessárias para estruturar e fornecer informações importantes para execução da estrutura. A primeira operação que deve ser definida é a criação, ou seja, preparação das variáveis para manipular as informações. Uma vez definida e criada (preparada) a estrutura, algumas operações de suporte são indispensáveis, como as de verificação. Verificar se a lista está vazia, por exemplo, é extremamente importante na operação de remoção, pois se não existe nenhum elemento, não poderá haver remoção.

Outra operação básica, com importância equivalente à verificação da lista vazia, é a verificação do seu extremo, ou seja, quando ela está cheia. Essa informação é indispensável no momento da inserção de elementos, pois se a lista está cheia não é possível inserir mais elementos. Na forma de implementação estática, essa preocupação é mais evidente, uma vez que a estrutura é construída com limitação física para armazenamento dos dados estabelecida. Na implementação encadeada (dinâmica) a limitação de armazenamento é a memória livre disponível, sendo muito maior a capacidade de armazenamento.

Obter a quantidade de elementos que estão armazenados na estrutura pode parecer uma operação muito simples ou até mesmo desnecessária, a princípio. No entanto, por se tratar de uma estrutura de dados, a única forma de obter a quantidade de elementos é por meio das operações disponíveis.

A forma como a estrutura manterá organizados seus elementos define diretamente como será a operação de inserção. Uma lista ordenada apenas permitirá adicionar novos elementos seguindo seu critério definido para classificação. Uma lista organizada sem critério de organização poderá admitir, a princípio, elementos em qualquer posição.

Por exemplo, uma lista de amigos, classificada pelo tempo de convivência, jamais permitirá adicionar um novo amigo no início; este certamente irá para o final da lista. Considerando que essa seção trata de listas genéricas, sem critérios predefinidos, serão apresentadas duas formas de inserir elementos: no início e no final da lista.

Com os elementos inseridos na estrutura, surge a necessidade que verificar se essa inserção foi realizada corretamente (inserções no início e/ou final). Acessar os elementos contidos na estrutura, seja para emitir relação de elementos ou buscar por um determinado elemento, é fundamental. Por exemplo, com a emissão da relação dos elementos contidos na estrutura é possível verificar se operações sucessivas de inserção (início e/ou fim) estão sendo realizadas corretamente.

Em alguns casos, apenas imprimir a relação não resolve todos os problemas de manipulação dos elementos e a estrutura precisa detectar se determinado elemento já existe. Em algumas situações, por exemplo, não é permitida a duplicidade de informação, tornando-se indispensável realizar a busca na estrutura para detectar a presença ou não de determinado elemento.

A forma como a lista genérica está proposta neste livro (sem critério de ordenação) não permite a realização de determinadas buscas, como a binária (tipo de busca realizada em uma coleção previamente ordenada). Entretanto, alterações podem ser feitas para resolver essa restrição, ficando a cargo do programador

decidir qual estratégia de busca deseja usar e, se for o caso, adaptar a estrutura e implementar a estratégia escolhida.
Em muitas situações é necessário remover determinado elemento que antes foi inserido, ficando clara a necessidade de discutir algumas operações de remoção. Diversas estratégias de remoção podem ser adotadas. Sendo assim, o programador deverá escolher e implementar as que forem necessárias para atender aos requisitos do sistema. Com o objetivo de abranger o maior número de possibilidades, serão discutidos três tipos de remoção: no início, no final e buscando por determinado elemento.

A remoção no início retira um elemento do início da coleção, enquanto a remoção no final retira o elemento que está posicionado no final da estrutura. A remoção buscando por determinado elemento tem como objetivo procurar um elemento na lista e removê-lo.

O Quadro 4.1 mostra uma sugestão para classificação das operações que serão apresentadas para cada estrutura de dados trabalhada no decorrer deste livro. As operações foram organizadas em 04 (quatro) categorias (básica, inserção, acesso e remoção) que reúnem as operações mais frequentemente discutidas na literatura e utilizadas pelas aplicações.

Quadro 4.1 – Classificação das operações na lista genérica.

Categoria	**Operações**
Básica	Criar, verificar vazia, verificar cheia e tamanho
Inserção	Inserir no início e inserir no final
Acesso	Imprimir e buscar
Remoção	Remover do início, remover do fim e remover buscando determinado elemento

Antes de discutir qualquer operação é apresentado o TAD da lista. Nesse momento, o programador decide o tipo de estrutura que será utilizada para armazenar os dados, ou seja, como será realizado o gerenciamento da memória, conforme discutido na seção 2.4. A seguir serão apresentados aspectos técnicos das duas formas de implementação: estática e dinâmica.

4.2. Lista Genérica Estática

Conforme discutido na seção 2.4, a implementação estática da lista genérica faz uso de um vetor como repositório para os dados. Apenas o vetor não é suficiente para implementar as operações básicas que essa estrutura de dados deve admitir. Por se tratar de um vetor, este possui limitação de espaço, sendo necessário controlar a quantidade de elementos que essa estrutura já armazenou.

Na declaração de vetores, o tamanho deve ser informado antes da compilação e, uma vez definido, não muda ao longo da execução do software. Nas linguagens de programação C e Java, a alocação do vetor pode acontecer em tempo de execução, por meio da alocação dinâmica de memória. Essa estratégia será adotada ao longo deste capítulo.

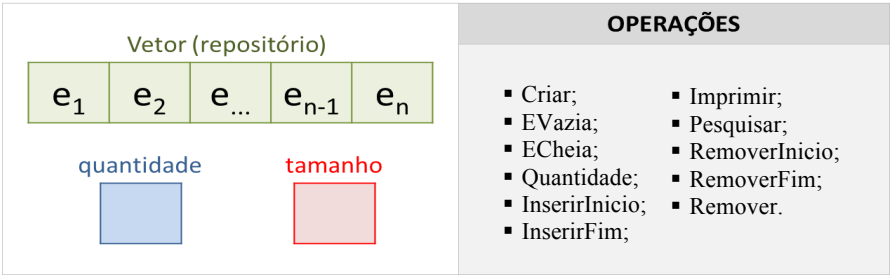
4.2.1. Definição do TAD

Uma vez que a estrutura está definida para realizar a alocação dinâmica de memória, além da referência para esta memória alocada e a quantidade de elementos contidos na estrutura, também será necessário guardar o tamanho da memória alocada.
A partir de então, será verificado se é possível ou não adicionar novos elementos (apenas é possível adicionar elementos se a quantidade de elementos for inferior ao tamanho do repositório).

A Figura 4.1 representa graficamente o TAD Lista Genérica Estática, que é composto por:

- ✓ Vetor como repositório para os dados (que será alocado dinamicamente);
- ✓ Variável inteira para armazenar a quantidade de elementos contidos na estrutura, em um determinado instante;
- ✓ Variável inteira para armazenar a capacidade de armazenamento da estrutura.

Figura 4.1 – TAD Lista Genérica Estática



!	Para deixar a implementação mais genérica, foi estabelecido que o tipo de informação armazenado na estrutura é "Elem". Quando necessário manipular outro tipo de informação, é suficiente alterar "Elem".

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.1) e Java (ver Código Java 4.1) para o problema descrito.

Código C 4.1 – TAD da lista genérica estática

Em C:

```
typedef struct {
   char nome[31];
   int idade;
}Elem;
```

```
typedef struct {
   Elem *elementos;
   int quantidade, tamanho;
}ListaEstatica;
```

A *struct* "ListaEstatica" contém um apontador para "Elem" (elementos). Esse apontador é responsável por armazenar o endereço de memória que será alocado em tempo de execução.

Código Java 4.1 – TAD da lista genérica estática

Em Java:

```
public class ListaEstatica<T>{
   private T[] elementos;
   private int quantidade;
}
```

A classe "ListaEstatica" foi definida parametrizada (T – tipo genérico). Dessa forma, é possível gerenciar qualquer tipo de informação. Essa definição do tipo ocorre quando a classe for instanciada. Java gerencia vetor como um objeto e, por se tratar de um objeto, oferece serviços, dentre eles, a verificação do tamanho do vetor (*length*), não sendo necessário armazenar a quantidade de elementos do vetor (ele conhece seu tamanho).

4.2.2. Operações Básicas

CRIAR

Para criar a lista é necessário indicar a quantidade de memória que deve ser alocada (será o tamanho do vetor). Uma vez alocada a memória com sucesso, a variável que controla a quantidade de elementos recebe valor 0 (zero), já que não existe nenhum elemento armazenado.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.2) e Java (ver Código Java 4.2) para o problema descrito.

Código C 4.2 – Criar estrutura

Em C:

```
int Criar(ListaEstatica *lista, int tam){
  lista->elementos = calloc(tam, sizeof(Elem));
  if (lista->elementos == NULL)
     return FALSE;
  lista->tamanho = tam;
  lista->quantidade = 0;
  return TRUE;
}
```

!	A linguagem C não oferece o tipo lógico (verdadeiro ou falso); para facilitar o entendimento foram definidas duas constantes TRUE (#define TRUE 1) e FALSE (#define FALSE 0).

A função "Criar" pode não conseguir êxito quando o repositório não for possível de ser alocado. Caso isso ocorra, a estrutura não será criada e, consequentemente, não poderá ser utilizada.

Código Java 4.2 – Criar estrutura

Em Java:

```
public ListaEstatica(int tamanho) {
   this.elementos = (T[]) new Object[tamanho];
   this.quantidade = 0;
}
```

A linguagem Java, até a versão utilizada no desenvolvimento deste livro, não permite instanciar um vetor genérico. Uma solução é alocar um vetor de *Object*[1] e fazer a conversão para "T". Uma vez que a classe foi parametrizada para manipular apenas o tipo genérico "T", não haverá problemas de conversão.

[1] http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html

VERIFICAR VAZIA

Uma lista está vazia quando a quantidade de elementos armazenada for igual a 0 (zero). Portanto, para saber se a lista está vazia, é necessário apenas verificar a variável "quantidade", não importando se tem alguma informação contida no vetor.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.3) e Java (ver Código Java 4.3) para o problema descrito.

Código C 4.3 – Verificar estrutura vazia

Em C:

```
int Vazia(ListaEstatica lista){
   return lista.quantidade == 0;
}
```

Não é necessário passar a "ListaEstatica" por referência, uma vez que a função está interessada em apenas ler o valor que está armazenado na variável "quantidade".

Código Java 4.3 – Verificar estrutura vazia

Em Java:

```
public boolean isVazia() {
   return this.quantidade == 0;
}
```

VERIFICAR CHEIA

Uma lista está cheia quando a quantidade de elementos armazenada for igual à quantidade de elementos que a estrutura suporta (tamanho).

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.4) e Java (ver Código Java 4.4) para o problema descrito.

Código C 4.4 – Verificar estrutura cheia

Em C:

```
int Cheia(ListaEstatica lista){
   return lista.quantidade == lista.tamanho;
}
```

Código Java 4.4 – Verificar estrutura cheia

Em Java:

```
public boolean isCheia() {
   return this.quantidade == this.elementos.length;
}
```

VERIFICAR QUANTIDADE DE ELEMENTOS

Essa operação deve apenas informar o valor que está armazenado no atributo que está controlando a quantidade de elementos armazenados na estrutura.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.5) e Java (ver Código Java 4.5) para o problema descrito.

Código C 4.5 – Verificar a quantidade de elementos

Em C:

```
int Quantidade(ListaEstatica lista){
   return lista.quantidade;
}
```

Código Java 4.5 – Verificar a quantidade de elementos

Em Java:

```
public int getQuantidade() {
   return this.quantidade;
}
```

4.2.3. Inserindo Elementos

INSERIR NO INÍCIO

Seguindo boas práticas de programação, o preenchimento do vetor será do início para o final, ou seja, ao inserir um novo elemento no início da lista, implica afirmar que esse elemento será inserido no primeiro índice do vetor.

!	As linguagens de programação C e Java consideram que os índices dos vetores começam com valor 0 (zero). O maior valor possível para o índice é a capacidade de armazenamento menos 1.

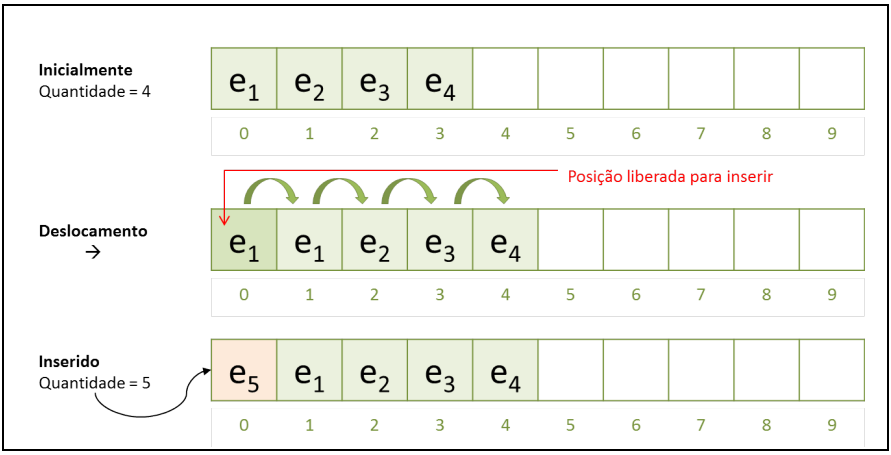
Caso a primeira posição esteja ocupada por um elemento, este elemento será deslocado para a posição sucessiva, e isso deverá ser executado consecutivamente com todos os elementos contidos no vetor.

Na Figura 4.2, está exemplificada a operação inserir no início. O vetor inicialmente possuía quatro elementos (e_1, e_2, e_3 e e_4). Para inserir o elemento "e_5", houve o deslocamento de todos os elementos, que passaram a ocupar uma posição à frente, em direção ao final do vetor.

A primeira posição passou a ficar livre, então nesta foi armazenado o novo valor. Após inserir o novo elemento, a

variável que controla a quantidade de elementos deverá ter seu valor incrementado em uma unidade.

Figura 4.2 – Operação inserir no início



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.6) e Java (ver Código Java 4.6) para o problema descrito.

Código C 4.6 – Inserir no início

Em C:

```
int InserirInicio(ListaEstatica *lista, Elem novo){
  int i;

  if (Cheia(*lista)) return FALSE;
  for (i = lista->quantidade; i > 0; --i)
     lista->elementos[i] = lista->elementos[i - 1];
  lista->elementos[0] = novo;
  lista->quantidade++;
  return TRUE;
}
```

Código Java 4.6 – Inserir no início

Em Java:

```
public void inserirInicio(T novo) {
   if (this.isCheia())
      throw new ListaCheiaException();
   for (int i = this.quantidade; i > 0; --i)
      this.elementos[i] = this.elementos[i - 1];
   this.elementos[0] = novo;
   ++this.quantidade;
}
```

INSERIR NO FINAL

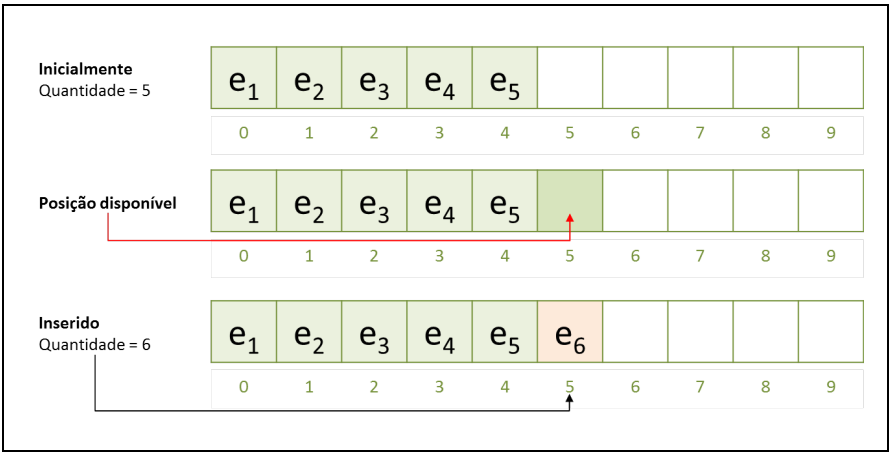
Para inserir elementos no final da lista, ou seja, no final do vetor, é necessário saber qual é a quantidade de elementos armazenados nesse vetor.

Esse valor é controlado pela variável "quantidade", e esta ajudará a determinar onde está o final da lista. Por exemplo, caso a estrutura tenha quatro elementos (quantidade = 4), implica afirmar que os índices 0, 1, 2 e 3, estão ocupados com elementos. A próxima posição disponível para inserir é o índice 4, coincidentemente, o tamanho da estrutura antes de inserir o elemento.

Considerando a lista representada na Figura 4.3, que inicialmente possuía cinco elementos (índices ocupados de 0 até 4), o índice disponível para inserir no final da lista é a quantidade de elementos, ou seja, 5 (cinco).

Semelhante ao que ocorre na inserção no início, aqui também só é possível executar a inserção caso a estrutura não esteja cheia.

Figura 4.3 – Operação Inserir no Fim



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.7) e Java (ver Código Java 4.7) para o problema descrito.

Código C 4.7 – Inserir no Final

Em C:

```
int InserirFim(ListaEstatica *lista, Elem novo){
   if (Cheia(*lista)) return FALSE;
   lista->elementos[lista->quantidade++] = novo;
   return TRUE;
}
```

Código Java 4.7 – Inserir no Final

Em Java:

```
public void inserirFim(T novo) {
   if (this.isCheia())
        throw new ListaCheiaException();
   this.elementos[this.quantidade++] = novo;
}
```

4.2.4. Acessando Elementos

IMPRIMIR

Essa é uma operação de extrema importância, pois permite exibir os elementos que estão contidos na lista. Os elementos são impressos do início até o final da estrutura. A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.8) e Java (ver Código Java 4.8) para o problema descrito.

Código C 4.8 – Imprimir elementos

Em C:

```
void Imprimir(ListaEstatica lista){
   int i;
   for (i = 0; i < lista.quantidade; ++i){
      printf("Nome: %s - ", lista.elementos[i].nome);
      printf("Idade: %d\n", lista.elementos[i].idade);
   }
}
```

Código Java 4.8 – Imprimir elementos

Em Java:

```
public Iterator<T> get() {
   @SuppressWarnings("unchecked")
   T[] temp = (T[]) new Object[this.getQuantidade()];
   for(int i = 0; i < this.quantidade; i++)
       temp[i] = this.elementos[i];
   return Arrays.asList(temp).iterator();
}
```

Na linguagem Java, a impressão de elementos dentro dos métodos é desencorajada. Uma alternativa de solução para esse problema é

fazer com que o método forneça acesso sequencial aos elementos contidos no repositório (*Design Pattern Iterator*[2]).

PESQUISAR ELEMENTO

A pesquisa tem como objetivo verificar se um determinado elemento está contido na lista. Ao encontrar o elemento, essa operação retorna o indicativo de sucesso (por exemplo, a posição do elemento encontrado), caso contrário, retorna indicativo de que não houve êxito na busca.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.9) e Java (ver Código Java 4.9) para o problema descrito.

Código C 4.9 – Pesquisar elemento

Em C:

```
int Pesquisar(ListaEstatica lista, char *nome){
   int i;
   for (i = 0; i < lista.quantidade; ++i)
      if (stricmp(lista.elementos[i].nome, nome) == 0)
         return i;
   return -1;
}
```

Se o elemento que está sendo procurado for encontrado, é retornada sua posição. Caso contrário, é retornado o valor "-1", indicando que a busca não obteve êxito.

!	`stricmp` é uma função definida na biblioteca "string.h", que realiza a comparação entre strings, caractere por caractere, ignorando a diferença entre caracteres maiúsculos e minúsculos.

[2] http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html

Código Java 4.9 – Pesquisar elemento

Em Java:

```
public int get(T elemento) {
   for (int i = 0; i < this.quantidade; ++i)
      if (this.elementos[i].equals(elemento))
         return i;
   throw new ElementoNaoExisteException();
}
```

A implementação em Java indica que não houve sucesso na busca por meio do lançamento da exceção "ElementoNaoExisteException".

4.2.5. Removendo Elementos

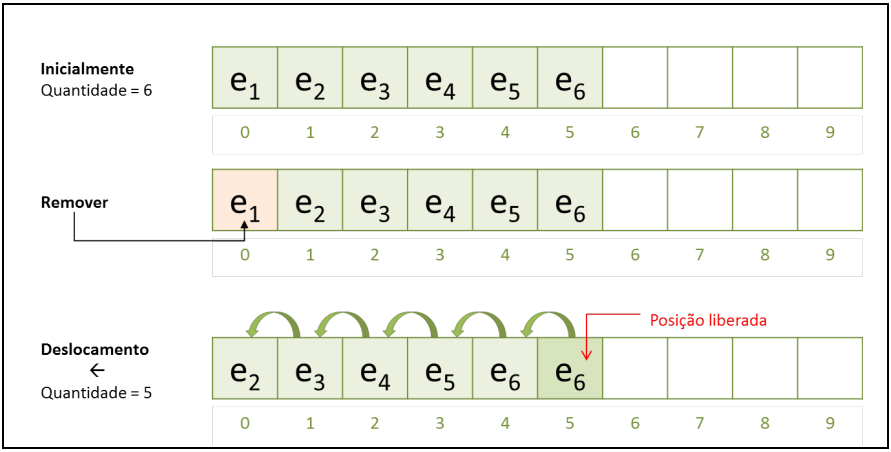
As remoções apresentadas buscam manter a coleção de elementos no início do vetor.

REMOVER DO INÍCIO

Essa operação remove um elemento do início da coleção, ou seja, o elemento que está armazenado na posição 0 (zero) do vetor. Ao realizar essa remoção, todos os outros elementos que estão localizados sucessivamente serão "deslocados" para o início do vetor (do segundo até a o último elemento).

Na Figura 4.4 está exemplificada a operação remover do início. O vetor inicialmente possuía seis elementos (e_1, e_2, e_3, e_4, e_5 e e_6). Ao solicitar a remoção do primeiro elemento, houve o deslocamento de todos os elementos do final para o início do vetor. A última posição (índice 5) passou a ficar livre. Após remoção do elemento, a variável que controla a quantidade de elementos deverá ter seu valor decrementado em uma unidade.

Figura 4.4 – Operação remover do início



Após conclusão da remoção, caso a operação inserir no final seja solicitada, o índice 5 (cinco) será sobrescrito. A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.10) e Java (ver Código Java 4.10) para o problema descrito.

Código C 4.10 – Remover do início

Em C:

```
int RemoverInicio(ListaEstatica *lista, Elem *elem){
   int i;

   if (Vazia(*lista)) return FALSE;
   *elem = lista->elementos[0];
   --lista->quantidade;
   for (i = 0; i < lista->quantidade; ++i)
      lista->elementos[i] = lista->elementos[i + 1];
   return TRUE;
}
```

Código Java 4.10 – Remover do início

Em Java:

```
public T removerInicio() {
   T aux;
   if (this.isVazia())
       throw new ListaVaziaException();
   aux = this.elementos[0];
   --this.quantidade;
   for (int i = 0; i < this.quantidade; ++i)
     this.elementos[i] = this.elementos[i + 1];
   return aux;
}
```

Somente será possível remover elementos se a lista não estiver vazia.

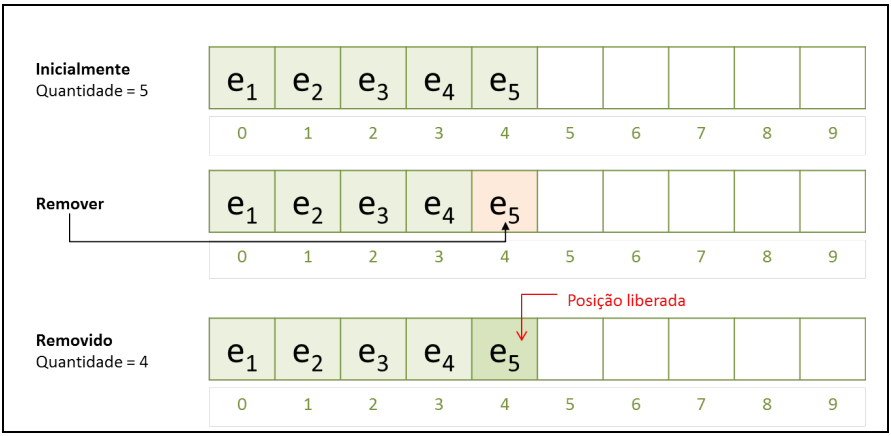
REMOVER DO FINAL

Da mesma forma que ocorre na inserção no final da lista, a quantidade de elementos armazenados no vetor é quem vai indicar a posição em que será realizada a remoção.

Na Figura 4.5, está exemplificada a operação remover do final. O vetor inicialmente possuía cinco elementos (e_1, e_2, e_3, e_4 e e_5).

Ao solicitar remoção, a variável que controla a quantidade de elementos deve ser decrementada, passando a ser quatro, e indicando a posição do elemento que foi removido.

Figura 4.5 – Operação Remover do Fim



Antes de retornar o elemento que está sendo removido, a variável que controla a quantidade de elementos contidos na lista deve ser decrementada em uma unidade, de forma a representar o índice do vetor que contém o último elemento da lista, ou seja, o elemento que está sendo removido.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.11) e Java (ver Código Java 4.11) para o problema descrito.

Código C 4.11 – Remover do final

Em C:

```
int RemoverFim(ListaEstatica *lista, Elem *elem){
   if (Vazia(*lista))
       return FALSE;
   *elem = lista->elementos[--lista->quantidade];
   return TRUE;
}
```

Código Java 4.11 – Remover do final

Em Java:

```
public T removerFim() {
   if (this.isVazia())
       throw new ListaVaziaException();
   return this.elementos[--this.quantidade];
}
```

REMOVER BUSCANDO DETERMINADO ELEMENTO

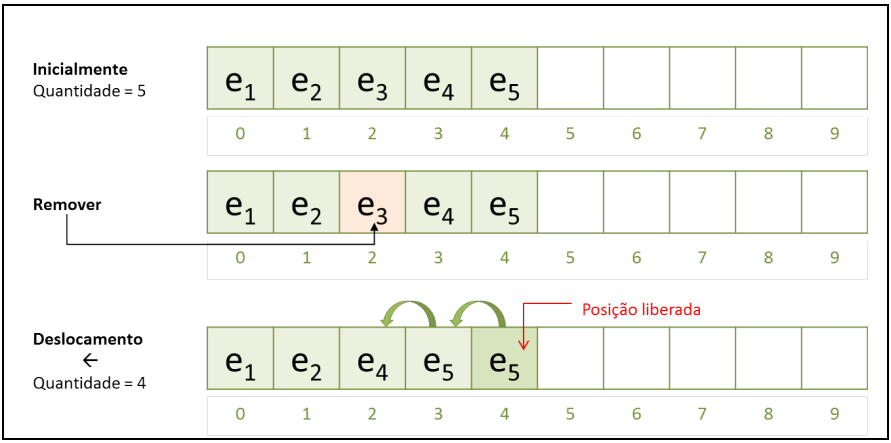
No caso de sucesso na remoção, os elementos que estão localizados consecutivamente na lista devem ser deslocados, semelhante ao que ocorre com a remoção do início.

No caso de o elemento não constar na lista, deverá ser retornada a confirmação do erro na remoção.

Na Figura 4.5 está exemplificada a operação remover buscando determinado elemento. O vetor inicialmente possuía cinco elementos (e_1, e_2, e_3, e_4 e e_5).

Ao solicitar remoção do elemento "e_3", todos os elementos que estavam localizados sucessivamente foram deslocados para o índice imediatamente à esquerda, até a posição em que foi removido o elemento.

Figura 4.6 – Operação Remover Determinado Elemento



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.12) e Java (ver Código Java 4.12) para o problema descrito.

Código C 4.12 – Remover elemento

Em C:

```
int Remover(ListaEstatica *lista, Elem *elem){
   int i, pos = Pesquisar(*lista, elem->nome);
   if (pos < 0)
      return FALSE;
   else{
      *elem = lista->elementos[pos];
      for (i = pos; i < lista->quantidade; ++i)
         lista->elementos[i] = lista->elementos[i + 1];
      lista->quantidade--;

      return TRUE;
   }
}
```

Código Java 4.12 – Remover elemento

Em Java:

```
public void remover(T elemento) {
   int posicao = this.get(elemento);

   if (posicao < 0)
      throw new ElementoNaoExisteException();
   for (int i = posicao; i < this.quantidade; ++i)
     this.elementos[i] = this.elementos[i + 1];
   --this.quantidade;
}
```

4.2.6. API Java

Visando a apoiar o desenvolvimento de aplicações, Java disponibiliza uma API (*Application Programming Interface*) que facilita a resolução de diversos tipos de problema. Isso mesmo, em diversos contextos, o desenvolvedor não precisará perder tempo desenvolvendo códigos genéricos, pois a API Java trará muito código pronto!

Conforme foi dito na seção 4.2.1, Java gerencia vetor como um objeto e, por ser objeto, fornece alguns serviços, como informar a quantidade de elementos que o vetor pode armazenar (ou seja, seu tamanho). Ao longo da apresentação da Lista Genérica Estática foram apresentados os recursos que o vetor de Java oferece, tendo seu comportamento não muito diferente do vetor de C.

Embora não tenha sido um tema explorado nos exemplos deste livro, Java oferece uma classe para manipulação de vetores estáticos, a classe `Arrays`[3] (pacote: `java.util`). Essa faz parte

[3] http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Arrays.html

do *Framework Collections*[4], que é um conjunto de classes e interfaces para manipular coleções.

A classe `Arrays` oferece alguns serviços, considerando a implementação de listas genéricas estáticas na versão ordenada. Dentre esses, dois serviços merecem destaque: `Arrays.sort` e `Arrays.binarySearch`.

Arrays.sort

O serviço "`Arrays.sort`" permite ordenar (na forma crescente) uma coleção de objetos, seguindo o critério de ordenação natural dos elementos contidos no vetor.

Buscando obter melhor desempenho, dependendo do tipo de dado armazenado no vetor, diferentes algoritmos podem ser utilizados para classificação. Cada elemento contido no vetor tem que implementar a interface `Comparable`[5] (pacote: `java.lang`), obrigatoriamente.

No trecho de Código Java 4.13 é apresentado um programa que faz uso da ordenação de vetores com API "`Arrays`". Na linha 4 está sendo instanciado vetor com tamanho quatro, para o tipo `String`[6]. Estão sendo adicionados quatro elementos no vetor (linhas 7 até 10), e esses elementos estão sendo impressos na linha 13 (`Arrays.toString`, converte um vetor em um objeto `String`). A coleção está sendo ordenada na linha 16 (`Arrays.sort`) e impressa novamente na linha 19.

[4] http://docs.oracle.com/javase/1.5.0/docs/guide/collections/index.html
[5] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Comparable.html
[6] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html

Código Java 4.13 – Serviço “Arrays.sort”

```
1  public static void main(String[] args) {
2
3  // vetor de tamanho 4
4  String []nomes = new String[4];
5
6  // adiciona os elementos
7  nomes[0] = "Primeiro";
8  nomes[1] = "Segundo";
9  nomes[2] = "Terceiro";
10 nomes[3] = "Quarto";
11
12 // imprime a lista antes de ordenar
13 System.out.println(Arrays.toString(nomes));
14
15 // ordena a coleção
16 Arrays.sort(nomes);
17
18 // imprime a lista após ordenação
19 System.out.println(Arrays.toString(nomes));
20 }
```

O resultado da execução do código pode ser visualizado na Figura 4.7. Na linha 1 a lista é exibida conforme os elementos foram adicionados. Na linha 2 a lista é exibida após ordenação.

Figura 4.7: Resultado da execução do Código Java 4.13 – Serviço “Arrays.sort”

```
1  [Primeiro, Segundo, Terceiro, Quarto]
2  [Primeiro, Quarto, Segundo, Terceiro]
```

!	A classe `String` implementa a interface `Comparable` nativamente. O mesmo ocorre com as classes de “autoboxing” para os tipos primitivos, por exemplo: `boolean`, `char`, `double`, `float` e `int`.

#	É um erro comum tentar ordenar objetos que não implementam `Comparable`. Quando isso ocorre a exceção `java.lang.ClassCastException` é lançada.

Arrays.binarySearch

O serviço "`Arrays.binarySearch`" realiza busca binária em uma coleção ordenada. No trecho de Código Java 4.14 é apresentada alteração no Código Java 4.13 para acrescentar a busca por determinado elemento. Na linha 21 está sendo exibido o índice do vetor no qual está localizado o elemento "Segundo".

Código Java 4.14 – Serviço "Arrays.binarySearch"

```
1  public static void main(String[] args) {
2
3  String []nomes = new String[4];
4
5  // adiciona os elementos
6  nomes[0] = "Primeiro";
7  nomes[1] = "Segundo";
8  nomes[2] = "Terceiro";
9  nomes[3] = "Quarto";
10
11 // imprime a lista antes de ordenar
12 System.out.println(Arrays.toString(nomes));
13
14 // ordena a coleção
15 Arrays.sort(nomes);
16
17 // imprime a lista após ordenação
18 System.out.println(Arrays.toString(nomes));
19
20 // imprime a posição do elemento "Segundo" na lista
21 System.out.println(Arrays.binarySearch(nomes,
          "Segundo"));
22 }
```

!	Caso o serviço `Arrays.binarySearch` tente localizar um elemento que não está contido na coleção é retornado um valor negativo.

O resultado da execução do código pode ser visualizado na Figura 4.8. Na linha 3 é exibida a posição do elemento "Segundo" no vetor, o índice 2 (dois).

Figura 4.8 – Resultado da execução do Código Java 4.14 – Serviço "Arrays.binarySearch"

```
1 [Primeiro, Segundo, Terceiro, Quarto]
2 [Primeiro, Quarto, Segundo, Terceiro]
3 2
```

A classe `Arrays` oferece mais recursos (`asList`, `deepEquals`, `deepHashCode`, `deepToString`, `equals`, `fill` e `hashCode`) e variações dos métodos apresentados nesta seção.

4.3. Lista Genérica Simplesmente Encadeada

Na seção anterior, foi apresentada a implementação estática (utiliza vetor) da lista genérica. Trata-se de uma estrutura simples, mas que contém sérias limitações, como o fato de o tamanho da coleção não poder mudar ao longo da sua execução. Sucessivas operações de inserção e remoção podem resultar em um tempo de execução elevado, uma vez que o deslocamento dos dados ocorre dentro da estrutura.

A lista genérica simplesmente encadeada representa uma proposta para superar essas limitações apresentadas. Por possuir uma estrutura dinâmica (seção 2.4.2), seus elementos estão dispersos na memória, não existindo a necessidade de alocar um bloco de

memória para “hospedar” todos os elementos. Nesse caso, a restrição quanto ao tamanho será determinada pela quantidade de memória livre disponível.

Na lista genérica simplesmente encadeada, sucessivas operações de inserção e remoção são executadas sem que haja a necessidade do deslocamento dos dados. O que ocorre é a atualização dos “links” entre os elementos contidos na estrutura.

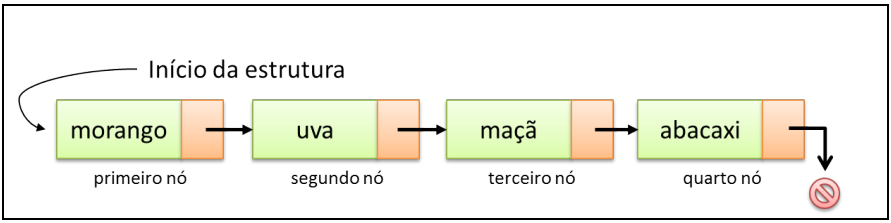
Por se tratar de uma lista simplesmente encadeada, cada nó contém, além do elemento que está sendo armazenado, o link para o próximo elemento do conjunto. Para ter acesso à lista é necessário conhecer o primeiro elemento da coleção, e através deste será possível percorrer, sequencialmente, todos os demais elementos. É importante destacar que caso o programador, por algum motivo, perca o acesso ao primeiro elemento, consequentemente, a lista passará a estar inacessível.

Com relação ao acesso, não é possível manipular diretamente um determinado elemento da coleção, diferente do que ocorre com os vetores. O acesso aos elementos só é possível percorrendo sequencialmente todos os nós (a partir do primeiro) até chegar ao nó que contém o valor que está sendo buscado.

Por exemplo, na Figura 4.9 está representada uma lista de frutas (morango, uva, maçã e abacaxi, nessa sequência). Supondo que se deseja ter acesso ao abacaxi, para isso a lista terá que ser percorrida a partir do início (morango), até chegar ao nó correspondente à fruta procurada (quarto nó).

No percurso, o primeiro nó fornecerá acesso ao segundo, este fornecerá acesso ao terceiro, que por sua vez fornecerá acesso ao quarto e procurado nó.

Figura 4.9 – Lista Encadeada



O último nó da coleção deverá demarcar que não existem mais elementos, ou seja, o campo destinado para referenciar o próximo elemento deverá representar essa característica.

4.3.1. Definição do TAD

A lista simplesmente encadeada é uma estrutura que mantém uma referência para o primeiro nó da coleção, caso exista. Não é necessário que a estrutura saiba a quantidade de elementos que ela contém, uma vez que esse valor poderá ser obtido percorrendo-se a estrutura. Cada nó contém dois campos: elemento armazenado e uma referência para o próximo nó.

Por se tratar de uma estrutura encadeada, cada nó da lista é alocado/desalocado sob demanda, ou seja, de acordo com a necessidade da aplicação. Isso quer dizer que cada nó deverá manter uma referência para um possível nó existente.

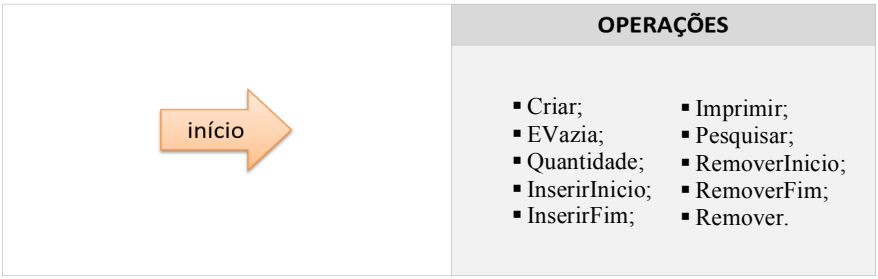
Na linguagem de programação C, essa referência para um provável nó existente é implementada por meio do uso de apontadores. Caso exista o nó, este armazena seu endereço de memória, caso não exista, este mantém o valor "NULL".

Na linguagem de programação Java, semelhante ao que ocorre na linguagem de programação C, também é mantida uma referência para o provável nó (objeto) existente. Caso não exista o próximo elemento, o nó mantém referência "null".

Não é necessário que essa estrutura registre a sua capacidade de armazenamento, tamanho, uma vez que esse valor depende diretamente da memória livre disponível e varia de acordo com a execução.

A Figura 4.10 representa graficamente o TAD da Lista Genérica Simplesmente Encadeada, que é composto por uma referência para o primeiro nó da coleção.

Figura 4.10 – TAD Lista Simplesmente Encadeada



!	Para deixar a implementação o mais genérica possível, foi estabelecido que o tipo de informação armazenado na estrutura é "Elem". Quando necessário manipular outro tipo de informação, é suficiente alterar "Elem".

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.13) e Java (ver Código Java 4.15) para o problema descrito.

Código C 4.13 – TAD da lista genérica simplesmente encadeada

Em C:

```
typedef struct {
   char nome[31];
   int idade;
}Elem;
```

```
typedef struct no{
   Elem elemento;
   struct no *proximo;
}No;

typedef struct {
   No *inicio;
}ListaSimplesmenteEncadeada;
```

A *struct* "No" contém o elemento (Elem) que está sendo manipulado e um apontador para armazenar o endereço do próximo nó (*struct no* *). A *struct* "ListaSimplesmenteEncadeada" mantém um apontador para armazenar o endereço do primeiro nó da coleção.

Código Java 4.15 – TAD da lista genérica simplesmente encadeada

Em Java:

```
public class No <T>{
   private T elemento;
   private No<T> proximo;
}

public class ListaLinearSimplesmenteEncadeada <T> {
   private No<T> inicio;
}
```

A classe que implementa a lista encadeada mantém uma referência para um objeto do tipo "No", e este é o primeiro elemento da lista. A classe "No" registra o elemento que está sendo manipulado e uma referência para um objeto do mesmo tipo, denominado próximo, caso este exista.

4.3.2. Operações Básicas

CRIAR

Diferente do que ocorre na criação das estruturas estáticas, na lista genérica simplesmente encadeada não é necessário indicar o tamanho da estrutura. Inicialmente, por estar vazia, a estrutura não apresenta elementos (nó). Nesse caso, é necessário registrar que o início da lista não possui elemento (está vazia). A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.14) e Java (ver Código Java 4.16) para o problema descrito.

Código C 4.14 – Criar Estrutura Simplesmente Encadeada

Em C:

```
void Criar(ListaSimplesmenteEncadeada *lista){
   lista->inicio = NULL;
}
```

A rotina `criar` está retornando um `void`, o que implica afirmar que para ela não existe um retorno explícito de valores. Realmente não é necessário retornar um valor. É registrado que o início da lista (que é um apontador do tipo "nó") deve armazenar `NULL`.

!	NULL é uma constante da Linguagem de Programação C para indicar o endereço de memória número zero.

Código Java 4.16 – Criar Estrutura Simplesmente Encadeada

Em Java:

```
public ListaSimplesmenteEncadeada (){
   this.inicio = null;
}
```

VERIFICAR VAZIA

Uma lista é considerada vazia quando não existe nó alocado para ela, ou seja, o início da lista é nulo.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.15) e Java (ver Código Java 4.17) para o problema descrito.

Código C 4.15 – Verificar lista encadeada vazia

Em C:

```
int Vazia(ListaSimplesmenteEncadeada lista){
   return (lista.inicio == NULL);
}
```

Código Java 4.17 – Verificar lista encadeada vazia

Em Java:

```
public boolean isVazia() {
   return this.inicio == null;
}
```

O que define a capacidade de armazenamento na lista encadeada é a memória livre disponível. A verificação dessa disponibilidade pode ser realizada no momento em que um novo nó for alocado, ou seja, na inserção de elementos.

Nesse caso, não faz sentido verificar o estado da lista para determinar se está cheia ou não.

VERIFICAR QUANTIDADE DE ELEMENTOS

Para obter a quantidade de elementos que está armazenada na estrutura, por não existir o atributo que guarda esse valor, é necessário percorrer a lista contando os elementos.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.16) e Java (ver Código Java 4.18) para o problema descrito.

Código C 4.16 – Verificar quantidade de elementos na lista encadeada simples

Em C:

```
int Quantidade(ListaSimplesmenteEncadeada lista){
   No *atual;
   int qtde = 0;

   for (atual = lista.inicio; atual != NULL; atual =
atual->proximo)
      qtde++;
   return qtde;
}
```

Código Java 4.18 – Verificar quantidade de elementos na lista encadeada simples

Em Java:

```
public int getQuantidade() {
   int qtde = 0;
   No<T> atual;
   for (atual = this.inicio; atual != null; atual =
atual.getProximo())
      ++qtde;
   return qtde;
}
```

4.3.3. Inserindo Elementos

A lista encadeada é determinada por um conjunto de nós que armazenam os valores que estão sendo manipulados, ou seja, é necessário criar um novo nó para inserir elementos na lista. Uma vez criado o nó, o próximo passo é decidir em qual posição ele será "encaixado".

Semelhante ao exemplo adotado na apresentação da lista estática (seção 4.2.3), uma vez que não apresenta critério de ordenação, novos elementos podem ser acrescentados em qualquer posição.

INSERIR NO INÍCIO

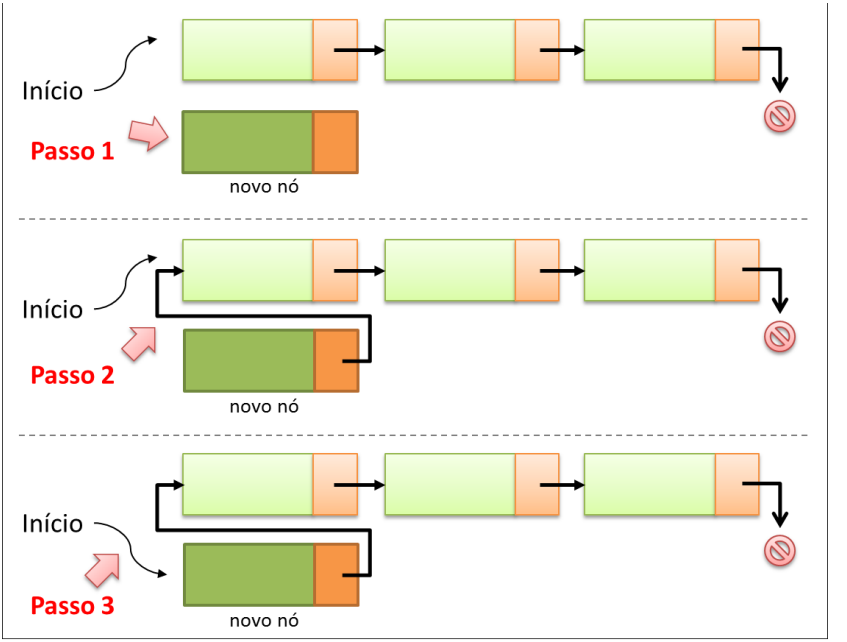
A lista encadeada é definida por um apontador que mantém uma referência para o primeiro elemento da coleção.

Na Figura 4.11 está representada graficamente uma lista que contém 3 (três) elementos. Para inserir um novo (quarto) elemento no início da estrutura, é necessário seguir os três passos:

- Passo 1: Um novo nó é criado e a este é adicionado o elemento a ser inserido na lista;
- Passo 2: O novo nó estabelece uma referência para o atual início da lista;
- Passo 3: O início da lista é atualizado para referenciar o novo nó que foi criado.

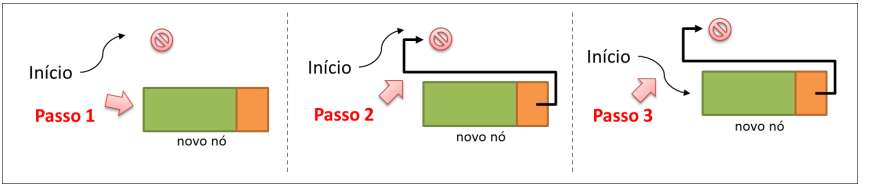
Executados os três passos, o novo elemento estará inserido na lista.

Figura 4.11 – Lista Encadeada Simples - Inserir no Início



Caso a lista esteja vazia, ou seja, a referência para o primeiro elemento não possua valor agregado, ainda assim os três passos explicados devem ser executados da mesma forma. Na Figura 4.12 é possível observar isso com mais clareza.

Figura 4.12 – Lista Encadeada Simples - Inserir no Início da Lista Vazia



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.17) e Java (ver Código Java 4.19) para o problema descrito.

Código C 4.17 – Lista Simplesmente Encadeada - Insere no Início

Em C:

```
int InsereInicio(ListaSimplesmenteEncadeada *lista,
Elem novo_elemento){
   No *novo;

   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;

   novo->elemento = novo_elemento;
   novo->proximo = lista->inicio;
   lista->inicio = novo;

   return TRUE;
}
```

Código Java 4.19 – Lista Simplesmente Encadeada - Insere no Início

Em Java:

```
public void inserirInicio(T elem) {
   No<T> no = new No<T>(elem);

   no.setProximo(this.inicio);

   this.inicio = no;
}
```

É possível perceber que o processo de inserção de elementos na lista encadeada é mais "leve" comparado com a lista estática. Isso se deve ao fato de não ser necessário o deslocamento dos dados, mas apenas a atualização das referências dos nós envolvidos.

Essa forma de implementação faz uma diferença que merece ser considerada em aplicações que precisam realizar inserção de muitos elementos.

INSERIR NO FINAL

Em uma lista encadeada, inserir no final implica em percorrer todos os elementos, a partir do início, e chegar ao último nó.

Isso é necessário porque a estrutura não armazena necessariamente uma referência para o último elemento.

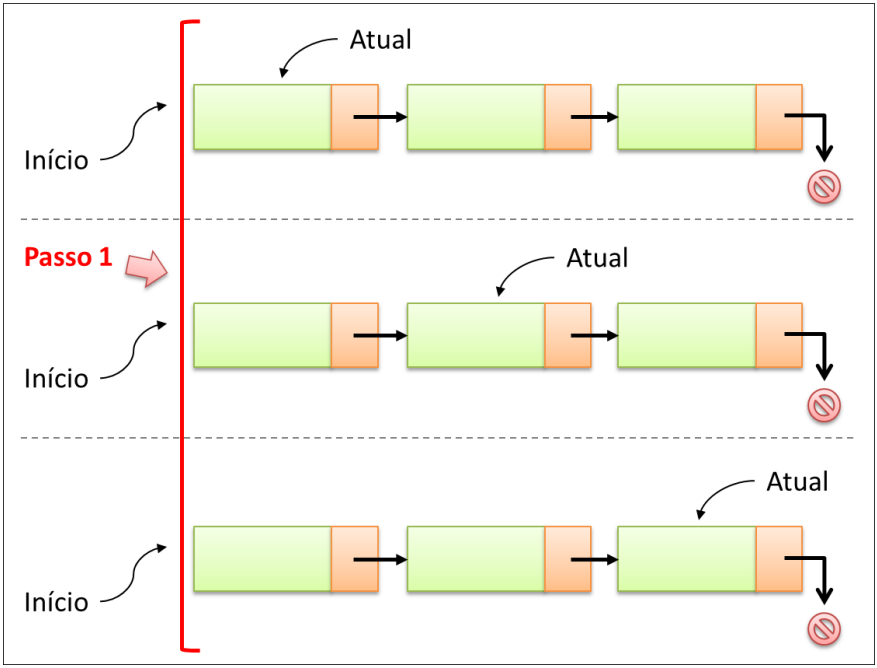
!	A única forma de acesso à lista é por meio da referência para o primeiro nó. Esse valor jamais poderá ser perdido, pois caso isso aconteça, todos os elementos contidos na estrutura também serão perdidos.

Por exemplo, na Figura 4.13 está representada uma lista contendo 03 (três) elementos. Para inserir elementos no final desta lista, o primeiro passo é descobrir o final dela.

Uma vez que a estrutura mantém uma referência para o primeiro elemento e essa referência jamais poderá ser perdida, uma referência auxiliar (denominada "Atual") é responsável por navegar na lista buscando o último elemento.

Esse último elemento é caracterizado por possuir uma referência nula para o próximo elemento (não existe próximo nó).

Figura 4.13 – Lista Simplesmente Encadeada - Inserir no Final, localizando o último nó



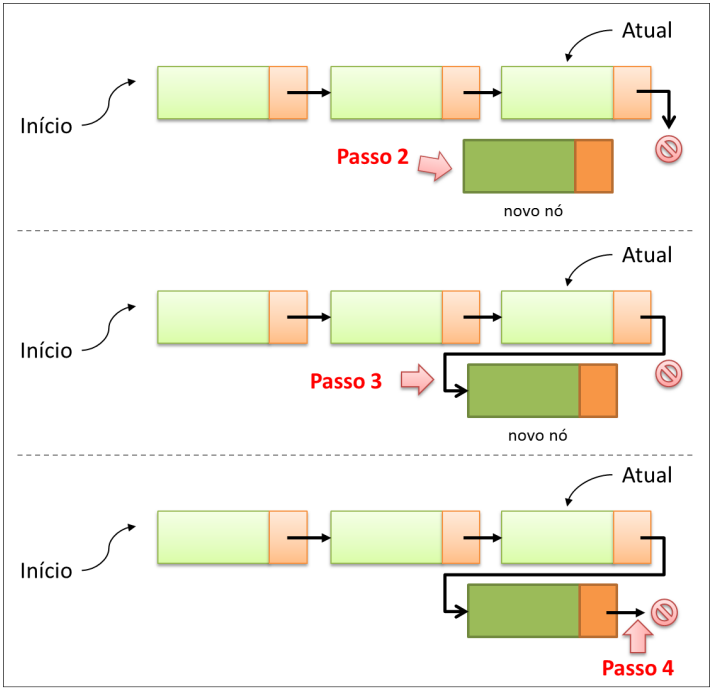
!	No passo 1, o último nó da lista foi localizado. A referência "Atual" mantém acesso a esse elemento.

Uma vez localizado o último elemento, na Figura 4.14 estão representados os próximos 03 (três) passos que devem ser seguidos.

- Passo 2: Um novo nó é criado e a este é adicionado o elemento a ser inserido na lista;
- Passo 3: O próximo elemento da referência "Atual", que era o término da lista, deve ser atualizado para apontar para o novo nó que foi criado;

- Passo 4: A referência para o próximo elemento do novo nó criado deverá ser atualizada para apontar para o final da lista.

Figura 4.14 – Lista Encadeada - Inserir no final



Caso a estrutura esteja vazia (a referência para o primeiro elemento é nula), não será possível executar o primeiro passo (localizar o último nó), uma vez que não existe nó para verificar se o próximo dele é nulo. Uma solução simples é, antes de tentar executar o "passo 1", verificar se a lista está vazia. Caso esteja, deve-se executar a inserção no início da lista (conforme já foi apresentado).

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.18) e Java (ver Código Java 4.20) para o problema descrito.

Código C 4.18 – Lista Simplesmente Encadeada - Inserir no Fim

Em C:

```
int InsereFim(ListaSimplesmenteEncadeada *lista, Elem
novo_elemento){
   No *atual, *novo;

   if (Vazia(*lista))
      return InsereInicio(lista, novo_elemento);

   atual = lista->inicio;
   while (atual->proximo != NULL)
      atual = atual->proximo;

   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;
   atual->proximo = novo;
   novo->proximo = NULL;
   novo->elemento = novo_elemento;
   return TRUE;
}
```

Código Java 4.20 – Lista Simplesmente Encadeada - Inserir no Fim

Em Java:

```
public void inserirFim(T elem) {
   if (this.isVazia()) inserirInicio(elem);
   else{
      No<T> atual, no = new No<T>(elem);
      atual = this.inicio;

      while (atual.getProximo() != null)
         atual = atual.getProximo();
      atual.setProximo(no);
      no.setProximo(null);
   }
}
```

No código Java, a implementação sugerida para a classe `No` ao instanciar novo objeto (construtor) define que o próximo elemento é `null`. Para efeitos didáticos o código apresentado está fazendo essa atualização do próximo elemento explicitamente, ou seja, o comando "`no.setProximo(null);`" pode ser omitido sem qualquer consequência.

4.3.4. Acessando Elementos

IMPRIMIR

Para imprimir os elementos contidos na estrutura, a partir do início, cada nó é percorrido até chegar ao final. Ao acessar o nó, seu respectivo valor é impresso.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.19) e Java (ver Código Java 4.21) para o problema descrito.

Código C 4.19 – Lista Simplesmente Encadeada - Imprimir

Em C:

```
void Imprime(ListaSimplesmenteEncadeada lista){
  No *atual;

  for (atual = lista.inicio; atual != NULL; atual =
atual->proximo){
    printf("Nome: %s - ", atual->elemento.nome);
    printf("Idade: %d\n", atual->elemento.idade);
  }
}
```

Código Java 4.21 – Lista Simplesmente Encadeada - Imprimir

Em Java:

```
public Iterator<T> get() {
   int i = 0;
   No<T> atual;
   @SuppressWarnings("unchecked")
   T[] vetor = (T[]) new Object[this.getQuantidade()];

   for (atual = this.inicio; atual != null; atual =
atual.getProximo())
      vetor[i++] = atual.getElemento();

   return Arrays.asList(vetor).iterator();
}
```

Por Java desencorajar a impressão de elementos por um serviço, semelhante ao adotado na implementação estática, a coleção de elementos é convertida em um vetor e deste é extraído um *iterator* (*Design Pattern Iterator*).

PESQUISAR ELEMENTO

Ao buscar um elemento na lista estática, caso o encontre, faz sentido retornar sua posição, porque é utilizado o vetor como estrutura de armazenamento. Uma vez conhecida a posição, o acesso pode ser feito na forma direta, por meio dos índices do vetor.

No caso da implementação encadeada, não existe o conceito de posição, então não faz sentido retornar essa informação. Por causa disso, na estrutura encadeada a operação pesquisar retornará um indicativo de sucesso ou fracasso na busca por um determinado elemento.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.20) e Java (ver Código Java 4.22) para o problema descrito.

Código C 4.20 – Lista Simplesmente Encadeada - Pesquisando elemento

Em C:

```
int Pesquisa(ListaSimplesmenteEncadeada lista, char
*nome){
   No *atual = lista.inicio;
   while (atual != NULL){
      if (stricmp(atual->elemento.nome, nome) == 0)
        return TRUE;
      atual = atual->proximo;
   }
   return FALSE;
}
```

Caso seja achado o elemento que está sendo procurado, a operação retorna `TRUE` (valor 1), caso contrário, `FALSO` (valor 0).

Código Java 4.22: Lista Simplesmente Encadeada – Pesquisando elemento

Em Java:

```
public boolean get(T elemento) {
   No<T> atual;
   for (atual = this.inicio; atual != null; atual =
atual.getProximo())
      if (atual.getElemento().equals(elemento))
        return TRUE;
   throw new ElementoNaoExisteException();
}
```

No caso da implementação em Java, caso o elemento não seja encontrado, é lançada uma exceção informando que houve o problema.

4.3.5. Removendo Elementos

REMOVER DO INÍCIO

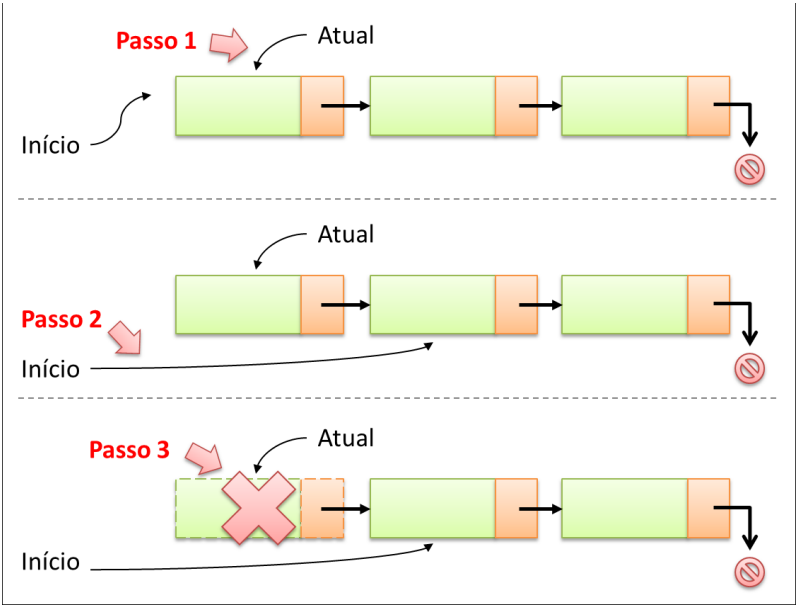
Em uma lista encadeada, remover do início implica em remover o primeiro nó, ou seja, o nó apontado pelo início da lista.

Por exemplo, na Figura 4.15 está representada uma lista contendo 03 (três) elementos.

Para remover um elemento, é necessário seguir os três passos:

- Passo 1: Uma referência "`atual`" aponta para o primeiro nó da estrutura;
- Passo 2: A referência "`início`" deverá apontar para o próximo do "`atual`";
- Passo 3: Uma vez atualizada a referência, é liberada a memória apontada por "`atual`".

Figura 4.15 – Lista Encadeada - Remover do Início



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.21) e Java (ver Código Java 4.23) para o problema descrito.

Código C 4.21 – Lista Simplesmente Encadeada - Remover do Início

Em C:

```
int RemoveInicio(ListaSimplesmenteEncadeada *lista,
Elem *elem){
   No *atual;
   if (Vazia(*lista)) return FALSE;
   atual = lista->inicio;
   lista->inicio = atual->proximo;
   *elem = atual->elemento;
   free(atual);
   return TRUE;
}
```

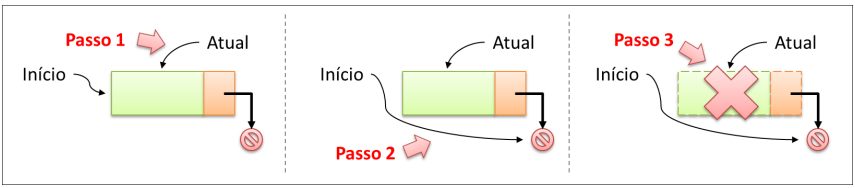
Código Java 4.23 – Lista Simplesmente Encadeada - Remover do Início

Em Java:

```
public T removerInicio() {
   if (this.isVazia())
      throw new ListaVaziaException();
   T elemento = this.inicio.getElemento();
   this.inicio = this.inicio.getProximo();
   return elemento;
}
```

Caso a lista tenha apenas um único elemento, as referências são atualizadas da mesma forma. Na Figura 4.16 é possível observar a remoção com maior clareza. Após execução da remoção, início referencia o final da lista, ou seja, a lista passa a estar vazia.

Figura 4.16 – Lista Simplesmente Encadeada - Remover do Início com um único nó

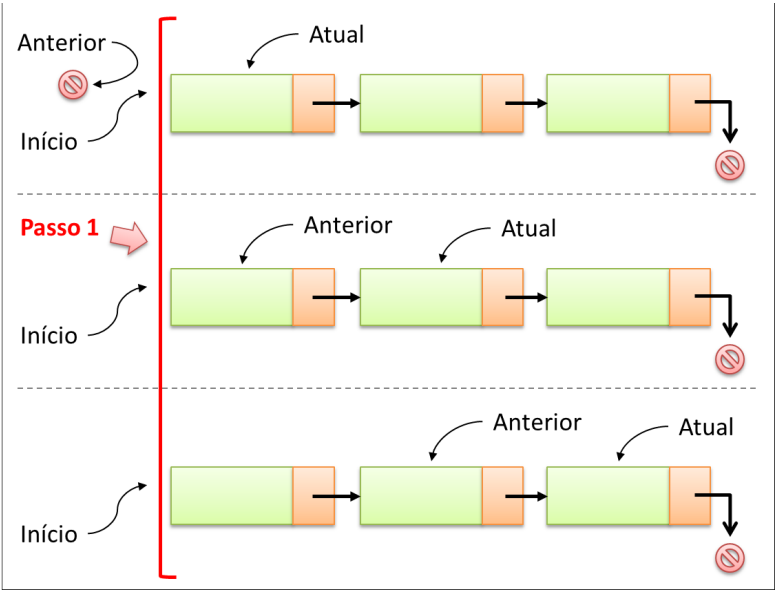


REMOVER DO FINAL

Para remover do final, é necessário percorrer a lista em busca do último elemento. Ao percorrer a lista e atualizar a referência "`atual`", outra referência ("`anterior`") é mantida para armazenar o nó que antecede o nó atual (durante o percurso).

Na Figura 4.17, está representado graficamente o percurso na estrutura em busca do último nó, com a referência "`anterior`" guardando o nó que antecede o último nó da lista.

Figura 4.17 – Lista Simplesmente Encadeada - Remoção do Final, localizando o último nó



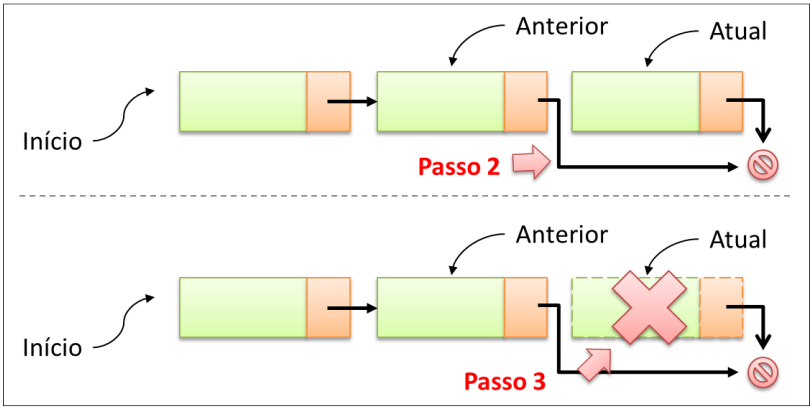
!	No passo 1, o último nó da lista foi localizado. A referência "Atual" mantém acesso a esse elemento. A referência "Anterior" mantém acesso ao penúltimo elemento da lista.

Uma vez localizado o último nó da lista, considerando que "`anterior`" está mantendo uma referência para o nó que antecede "`atual`", para remover o último nó, é necessário seguir mais 02 (dois) passos:

- Passo 2: As referências são atualizadas, sendo que o próximo de "`anterior`" passa a ser o próximo de "`atual`";

- Passo 3: A memória apontada por "`atual`" (ou seja, ocupada pelo último elemento) é liberada.

Figura 4.18 – Lista Simplesmente Encadeada – Remoção do Final



Inicialmente, antes de percorrer a lista, a referência "`anterior`" mantinha valor nulo. Se ao encontrar o último elemento a referência "`anterior`" ainda possuir valor nulo, implica afirmar que a lista contém apenas um único elemento.

Nesse caso, não se tem como atualizar as referências, uma vez que não existe elemento apontado por "`anterior`".

Caso a lista esteja com apenas um único elemento, a remoção no início da lista deverá ser executada, conforme já foi apresentado. Não é preciso reinventar a roda!

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.22) e Java (ver Código Java 4.24) para o problema descrito.

Código C 4.22 – Lista Simplesmente Encadeada - Remoção do Final

Em C:

```
int RemoveFim(ListaSimplesmenteEncadeada *lista, Elem
*elem){
  No *atual, *anterior = NULL;

  if (Vazia(*lista)) return FALSE;

  atual = lista->inicio;
  while (atual->proximo != NULL){
     anterior = atual;
     atual = atual->proximo;
  }

  if (anterior == NULL)
     return RemoveInicio(lista, elem);

  *elem = atual->elemento;
  anterior->proximo = atual->proximo;

  free(atual);

  return TRUE;
}
```

Código Java 4.24 – Lista Simplesmente Encadeada - Remoção do Final

Em Java:

```
public T removerFim() {
   No<T> atual, anterior = null;
   if (this.isVazia())
      throw new ListaVaziaException();
   atual = this.inicio;
   while (atual.getProximo() != null){
      anterior = atual;
      atual = atual.getProximo();
   }
   if (anterior == null) return this.removerInicio();
   anterior.setProximo(atual.getProximo());
   return atual.getElemento();
}
```

REMOVER BUSCANDO DETERMINADO ELEMENTO

O procedimento é semelhante àquele executado para a remoção no final. A diferença é que a remoção buscando determinado elemento procura o nó que contém um determinado valor, em vez de buscar o último elemento da lista.

Encontrado o nó, a referência "`anterior`" é atualizada conforme os demais casos. Caso o elemento esteja posicionado no primeiro nó, a remoção no início é realizada.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.23) e Java (ver Código Java 4.25) para o problema descrito.

Código C 4.23 – Lista Simplesmente Encadeada - Remover buscando determinado elemento

Em C:

```
int Remove(ListaSimplesmenteEncadeada *lista, char
*nome, Elem *elem){

  No *anterior, *atual;

  if (Vazia(*lista))
     return FALSE;

  anterior = NULL;
  atual = lista->inicio;

  while (atual != NULL){
     if (stricmp(atual->elemento.nome, nome) == 0){
       if (anterior == NULL)
            return RemoveInicio(lista, elem);
       anterior->proximo = atual->proximo;
       *elem = atual->elemento;
       free(atual);
       return TRUE;
     }
     anterior = atual;
     atual = atual->proximo;
  }

  return FALSE;
}
```

Código Java 4.25 – Lista Simplesmente Encadeada - Remover buscando determinado elemento

Em Java:

```
public void remover(T elemento) {
   No<T> atual, anterior = null;
   if (this.isVazia())
      throw new ListaVaziaException();
   atual = this.inicio;
   while (atual != null){
      if (atual.getElemento().equals(elemento)){
         if (anterior == null) this.removerInicio();
         else anterior.setProximo(atual.getProximo());
         return;
      }
      anterior = atual;
      atual = atual.getProximo();
   }
   throw new ElementoNaoExisteException();
}
```

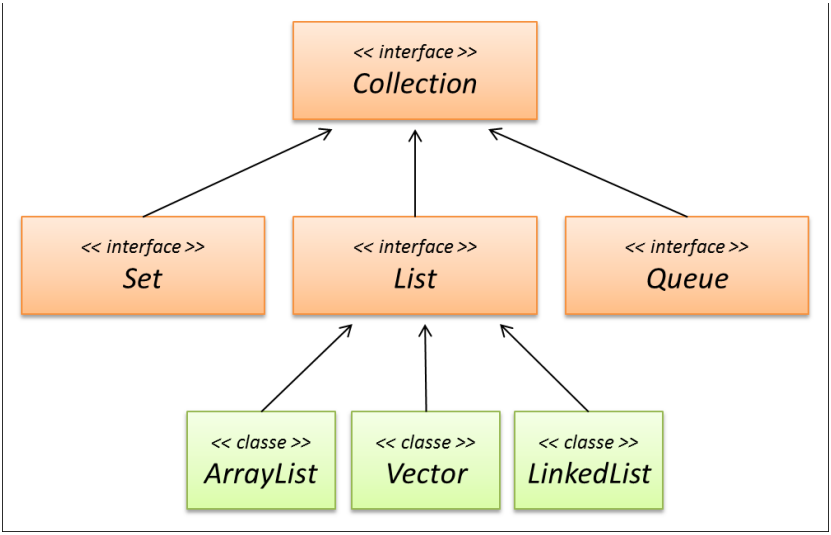
4.3.6. API Java

Manipular coleções genéricas é uma prática muito comum no desenvolvimento de software.

Segundo Deitel e Deitel (2010), Java fornece várias estruturas de dados definidas, chamadas coleções, utilizadas para armazenar grupos de objetos relacionados. Essas classes fornecem métodos eficientes que organizam, armazenam e recuperam seus dados sem que seja necessário conhecer como os dados são armazenados. Isso reduz o tempo de desenvolvimento de aplicativos.

Na Figura 4.19, é possível observar parte da hierarquia de classes/interfaces de Java para manipulação de coleções de objetos. *Collection* é o início de tudo.

Figura 4.19 – Hierarquia Collection (algumas classes e interfaces)



A partir da versão 1.5[7] da linguagem de programação Java, os tipos genéricos foram introduzidos e com isso foi possível determinar exatamente qual é o tipo de objeto que está sendo mantido na coleção. Até a versão 1.4[8], as coleções manipulavam "Object".

Desconsiderando aspectos relacionados ao desempenho das estruturas oferecidas por Java, dentre implementações de uma lista (`interface List`), serão analisados os serviços oferecidos pela classe `LinkedList`[9] que são compatíveis com as operações discutidas nessa seção. No Quadro 4.2 é apresentado o mapeamento das operações discutidas e os serviços que as implementam.

[7] http://docs.oracle.com/javase/1.5.0/docs/api/
[8] http://docs.oracle.com/javase/1.4.2/docs/api/
[9] http://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html

Quadro 4.2 – Lista Simplesmente Encadeada - Mapeamento das Operações em Java (Linkedlist)

Categoria	Operação	Método de Linkedlist
Básica	Criar	`Construtor`
	Verificar Vazia	`isempty()`
	Tamanho	`size()`
Inserção	Inserir Início	`addFirst(<<elem>>)`
	Inserir Final	`addLast(<<elem>>)`
Acesso	Imprimir	`iterator()`
	Pesquisar	`contains(<<elem>>)` `indexOf(<<elem>>)`
Remoção	Remover Início	`removeFirst()`
	Remover Fim	`removeLast()`
	Remover Determinado Elemento	`remove(<<elem>>)`

O quadro acima contempla o uso de todas as operações, referentes às 04 (quatro) categorias, discutidas na apresentação da lista simplesmente encadeada. O exemplo apresentado no Código Java 4.26 faz uso de `LinkedList` para gerenciar uma lista de `String`. Na linha 09 a estrutura é criada (objeto `LinkedList` instanciado). As demais operações básicas estão exemplificadas nas linhas: 12 (verificando se está vazia) e 26 (verificando a quantidade de elementos contidos na estrutura).

Com relação à inserção de elementos, nas linhas 15 e 16 estão sendo inseridos dois elementos no início da estrutura, nas linhas 19 e 20 estão sendo inseridos mais dois elementos. A remoção dos elementos está acontecendo nas linhas 39 (início), 42 (final) e 45/46 (buscando determinado elemento).

Nas linhas 34, 35 e 36 estão sendo realizadas buscas por determinado elemento. Para finalizar o acesso aos dados, entre as linhas 29 e 31 estão sendo impressos os elementos contidos na estrutura.

Código Java 4.26 – Lista Encadeada com LinkedList.

```
1  import java.util.Iterator;
2  import java.util.LinkedList;
3
4  public class UsaLinkedList {
5
6  public static void main(String[] args) {
7
8  // Cria a lista encadeada
9  LinkedList<String> listaEncadeada = new
          LinkedList<String>();
10
11 // Verifica se a lista está vazia
12 System.out.println("Lista Vazia: " +
          listaEncadeada.isEmpty());
13
14 // Insere no início da lista
15 listaEncadeada.addFirst("Primeiro");
16 listaEncadeada.addFirst("Segundo");
17
18 // Insere no final da lista
19 listaEncadeada.addLast("Terceiro");
20 listaEncadeada.addLast("Quarto");
21
22 // Verifica se a lista está vazia
23 System.out.println("Lista Vazia: " +
          listaEncadeada.isEmpty());
24
25 // Verifica a quantidade de elementos adicionados
26 System.out.println("Quantidade: " +
          listaEncadeada.size());
27
28 // Imprime os elementos
29 Iterator<String> it = listaEncadeada.iterator();
30 while (it.hasNext())
```

```
31   System.out.println("Removido: " + it.next());
32
33 // Verifica a existência de determinado elemento
34 System.out.println(listaEncadeada.contains("Primeir
         o"));
35 System.out.println(listaEncadeada.contains("Não
         existe"));
36 System.out.println(listaEncadeada.indexOf("Primeiro
         "));
37
38 // Remove do início da lista
39 System.out.println(listaEncadeada.removeFirst());
40
41 // Remove do final da lista
42 System.out.println(listaEncadeada.removeLast());
43
44 // Remove determinado elemento
45 System.out.println(listaEncadeada.remove("Terceiro"
         ));
46 System.out.println(listaEncadeada.remove("Primeiro"
         ));
47
48 // Verifica se a lista está vazia
49 System.out.println("Lista Vazia: " +
         listaEncadeada.isEmpty());
50 }
51 }
```

O resultado da execução do código pode ser visualizado na Figura 4.20.

Figura 4.20 – Resultado da execução da lista encadeada com LinkedList

```
1 Lista Vazia: true
2 Lista Vazia: false
3 Quantidade: 4
4 Removido: Segundo
5 Removido: Primeiro
6 Removido: Terceiro
```

```
7  Removido: Quarto
8  true
9  false
10 1
11 Segundo
12 Quarto
13 true
14 true
15 Lista Vazia: true
```

!	A classe `LinkedList` ainda oferece outros serviços para manipulação de coleções encadeadas.

4.4. Lista Genérica Duplamente Encadeada

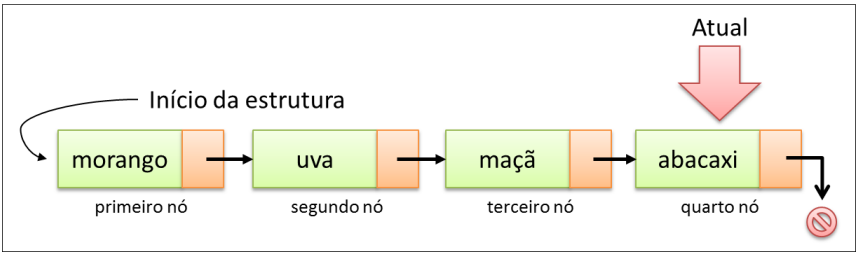
A lista genérica simplesmente encadeada, ou lista simplesmente encadeada, apresentou uma alternativa interessante para implementação de coleções. A principal vantagem é o dinamismo da estrutura, que permite alterar seu tamanho em tempo de execução, de acordo com a necessidade da aplicação. A memória é alocada e liberada sob demanda.

Sucessivas operações de inserção e remoção não modificam a localização da informação na memória (deslocamento dos dados), e apenas os links entre os elementos são atualizados.

Embora as vantagens apresentadas para a lista encadeada possam sugerir uma estrutura muito mais interessante do que a estática, dependendo da aplicação, alguns problemas podem ocorrer: a lista simplesmente encadeada, por exemplo, só pode ser percorrida em um sentido (início para o final). Caso seja necessário percorrer uma lista do final para o início, por meio de uma lista simplesmente encadeada essa operação torna-se praticamente impossível.

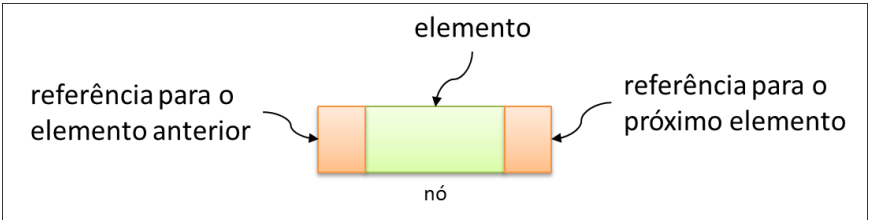
Na Figura 4.21 está exemplificada uma lista que foi percorrida até chegar ao "quarto nó" (atual). Supondo que se deseja ter acesso ao nó que antecede o "atual", isso só será possível percorrendo novamente a lista a partir do início até chegar ao elemento desejado. Dependendo da quantidade de elementos, essa operação poderá ser muito onerosa para o sistema.

Figura 4.21 – Lista Simplesmente Encadeada - Limitação



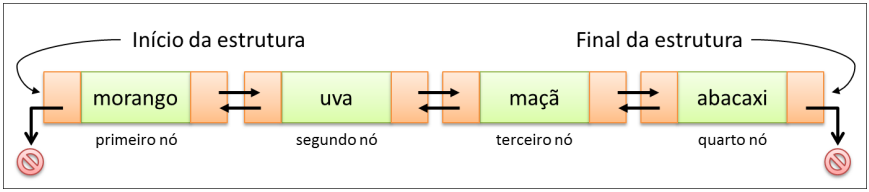
Uma solução é cada nó manter também um link para o nó que o antecede. Dessa forma, o problema anteriormente citado está resolvido. Essa implementação é conhecida como "Duplamente Encadeada". Nesse contexto, a estrutura "nó" contém o elemento que está sendo manipulado, uma referência para o próximo elemento da lista e uma referência para o elemento anterior da lista, conforme pode ser observado na Figura 4.22 .

Figura 4.22 – Nó duplamente encadeado



A principal vantagem do duplo encadeamento é que, partindo de qualquer ponto da lista, a aplicação consegue ter acesso a qualquer outro elemento nela contido.
Na Figura 4.23 está representada graficamente uma "Lista Genérica Duplamente Encadeada", ou seja, uma lista sem critério de ordenação (aceita inserção e remoção de elementos em qualquer posição), em que cada nó possui uma referência para o nó anterior e outra referência para o nó posterior (duplamente encadeada).

Figura 4.23 – Lista Duplamente Encadeada



É possível avançar e retroceder na estrutura a partir de qualquer posição da lista. Em aplicações que realizam muitas buscas, essa característica pode ser determinante para o sucesso da aplicação.

Esse tipo de estrutura pode apresentar melhor resultado caso a lista mantenha uma referência para o primeiro nó e outra referência para o último nó da lista. Operações de inserção e remoção aconteceriam de forma mais rápida em comparação com a implementação simplesmente encadeada.

Por exemplo, analisando a remoção no final da lista, não será mais necessário percorrer toda a lista em busca do elemento que antecede o último nó. Para essa remoção, é suficiente verificar a referência para o nó anterior ao último nó da lista.

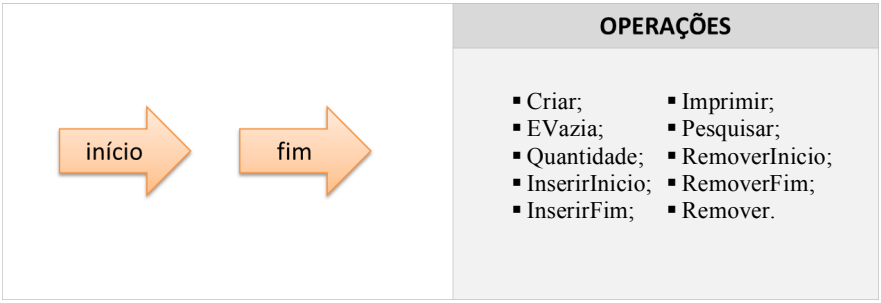
4.4.1. Definição do TAD

Nesse tipo de lista, conforme dito, cada nó deverá manter o elemento que está sendo armazenado e duas referências: uma para o nó anterior e outra para o próximo nó.

Por ser uma estrutura encadeada, não diferente da estratégia adotada na lista simplesmente encadeada, ou seja, a memória é alocada em tempo de execução da aplicação.

Na Figura 4.24 está representado o TAD da Lista Genérica Duplamente Encadeada.

Figura 4.24 – TAD Lista Duplamente Encadeada



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.24) e Java (ver Código Java 4.27) para o problema descrito.

Código C 4.24 – TAD da lista genérica duplamente encadeada

Em C:

```
typedef struct {
   char nome[31];
   int idade;
}Elem;
```

```
typedef struct no{
   Elem elemento;
   struct no *anterior, *proximo;
}No;

typedef struct {
   No *inicio;
   No *fim;
}ListaDuplamenteEncadeada;
```

Além das informações manipuladas em uma lista simplesmente encadeada, a *struct* "Nó" mantém uma referência para o nó anterior (*struct* nó *anterior).

A *struct* `ListaDuplamenteEncadeada` mantém um link para o primeiro elemento da lista e outro para o final da lista. Com essas duas referências é possível realizar as operações de inserção e remoção, tanto no início quanto no final da estrutura, com a mesma eficiência. De fato, não será necessário percorrer toda a lista em busca do último elemento da estrutura, nem ter acesso ao penúltimo elemento, para realizar a operação de remoção no final da lista, conforme acontece na `ListaSimplesmenteEncadeada`.

Código Java 4.27 – TAD da lista genérica duplamente encadeada.

Em Java:

```
public class No <T>{
   private T elemento;
   private No<T> anterior, proximo;
}

public class ListaDuplamenteEncadeada{
   private No<T> inicio, fim;
}
```

A estrutura em Java segue o mesmo princípio da implementação em C.

4.4.2. Operações Básicas

CRIAR

A operação criar possui comportamento muito semelhante ao da `ListaSimplesmenteEncadeada`. A diferença é que na lista duplamente encadeada a estrutura registra ainda que a referência para o final da lista também não possui elemento (está vazia).

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.25) e Java (ver Código Java 4.28) para o problema descrito.

Código C 4.25 – Criar Estrutura Duplamente Encadeada.

Em C:

```
void Cria(ListaDuplamenteEncadeada *lista){
   lista->inicio = lista->fim = NULL;
}
```

Código Java 4.28 – Criar Estrutura Duplamente Encadeada.

Em Java:

```
public ListaDuplamenteEncadeada(){
   this.inicio = this.fim = null;
}
```

As operações para verificar vazia, verificar cheia e obter quantidade possuem implementação igual à apresentada na `ListaSimplesmenteEncadeada` (seção 4.3.2).

4.4.3. Inserindo Elementos

Segue o mesmo princípio da `ListaSimplesmenteEncadeada`, ou seja, para inserir novo elemento, um "nó" é criado e é definida sua posição de "encaixe" na lista. Nesse caso, também não será adotado critério de ordenação.

INSERIR NO INÍCIO

Essa operação busca inserir um novo elemento no início da estrutura.

Na Figura 4.25 está representada uma lista duplamente encadeada com 03 (três) elementos. Para inserir um novo elemento no início da lista é necessário seguir os 05 (cinco) passos:

- Passo 1: Um novo nó é criado e a este é associado o elemento a ser inserido;
- Passo 2: O novo nó estabelece a sua referência para o próximo elemento sendo o atual início da lista;
- Passo 3: A referência para o elemento anterior ao novo nó é definida como nula, ou seja, não existe;
- Passo 4: O elemento apontado pelo atual início da lista atualiza sua referência anterior para indicar o novo nó que está sendo criado;
- Passo 5: O início da lista é atualizado para referenciar o novo nó que foi criado.

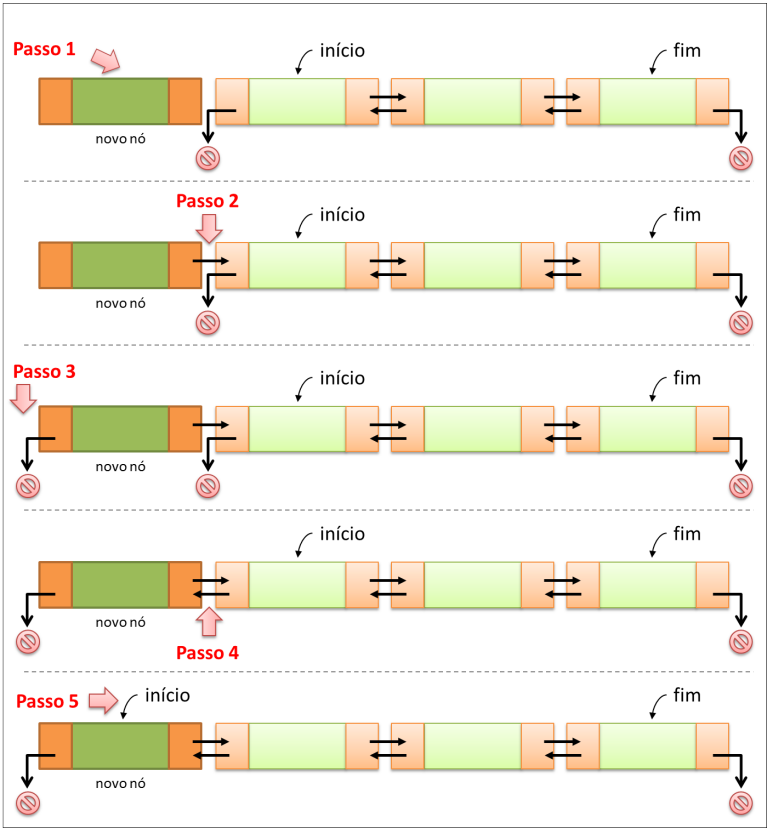
Executados os 05 (cinco) passos, o novo elemento estará inserido na lista.

O "Passo 4" só pode ser executado caso a lista esteja mantendo, no mínimo, um elemento. No caso de estar vazia, essa atualização não é possível de ser executada, pois não existe elemento

apontado pelo início da lista para que este possa atualizar sua referência anterior para o novo nó.

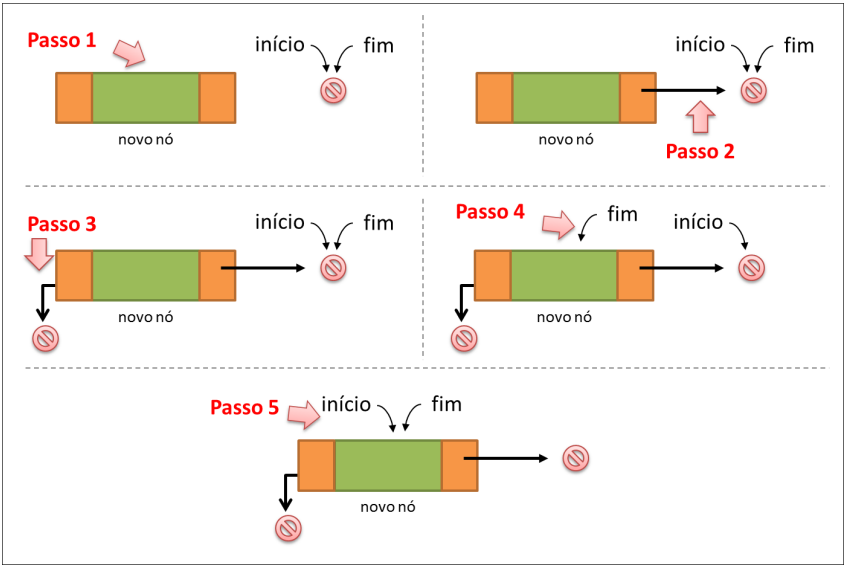
Considerando que a lista está vazia, os passos 1, 2 e 3 devem ser executados normalmente, correspondendo à criação do nó e atualização das suas referências, próximo (este receberá o valor armazenado em início, que é nulo) e anterior. No "passo 4", como esse é o primeiro elemento que está sendo adicionado, a referência para o final da lista deverá apontá-lo. Para finalizar, o "Passo 5" é executado normalmente.

Figura 4.25 – Lista Duplamente Encadeada - Inserir no Início



Na Figura 4.26 é possível observar graficamente os 05 (cinco) passos para inserção de elementos em uma lista duplamente encadeada vazia.

Figura 4.26 – Lista Duplamente Encadeada - Inserir no Início (estrutura vazia)



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.26) e Java (ver Código Java 4.29) para o problema descrito.

Código C 4.26 – Lista Duplamente Encadeada - Insere no Início

Em C:

```
int InsereInicio(ListaDuplamenteEncadeada *lista,
Elem novo_elemento){
   No *novo;

   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;
```

```
   novo->elemento = novo_elemento;
   novo->proximo = lista->inicio;
   novo->anterior = NULL;
   if (Vazia(*lista)) lista->fim = novo;
   else lista->inicio->anterior = novo;
   lista->inicio = novo;
   return TRUE;
}
```

Código Java 4.29 – Lista Duplamente Encadeada - Insere no Início.

Em Java:

```
public void inserirInicio(T elem) {
   No<T> no = new No<T>(elem);

   if (this.isVazia()) this.fim = no;
   else {
      this.inicio.setAnterior(no);
      no.setProximo(this.inicio);
   }
   this.inicio = no;
}
```

A classe "No", ao instanciar novo elemento, atualiza as referências anterior e próximo para receberem valor nulo, ou seja, não existem. Considerando esse comportamento, ao inserir novo "nó" na lista, caso esta não esteja vazia, as referências do novo nó devem ser atualizadas.

INSERIR NO FINAL

Para inserir um novo elemento na lista duplamente encadeada não é necessário percorrer toda a estrutura em busca do último elemento. Uma vez que essa lista mantém referência para o último

nó, é necessário apenas atualizar a referência fim para "encaixar" o novo nó.

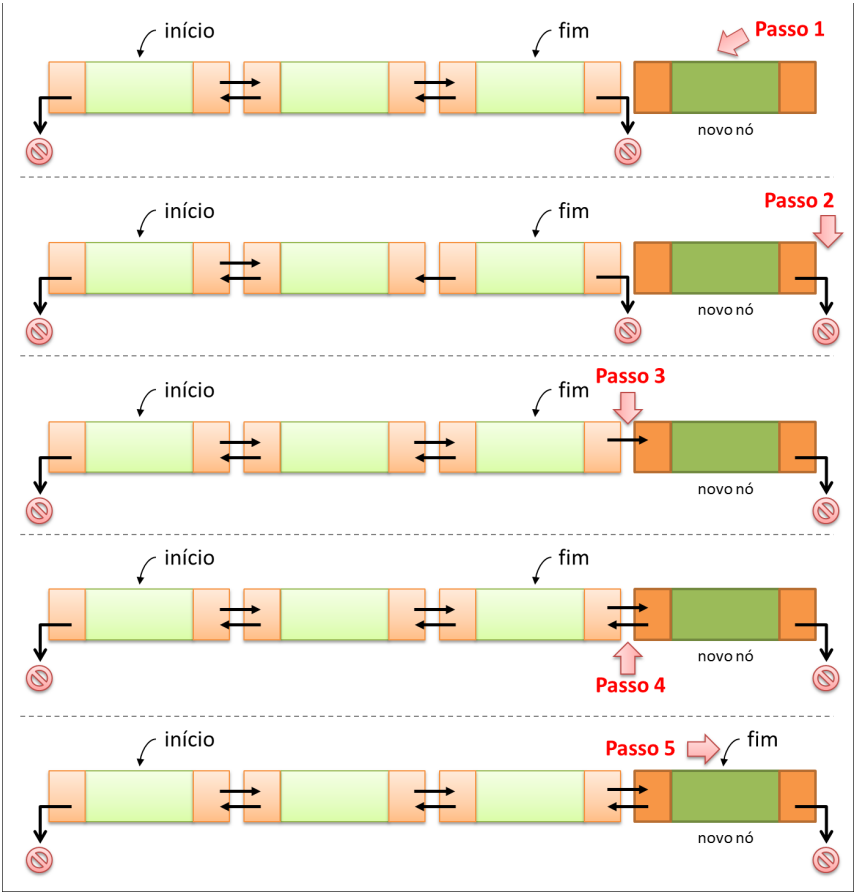
Por exemplo, na Figura 4.27. está representada uma lista duplamente encadeada com 03 (três) elementos. Para inserir um elemento no final da lista, é necessário seguir os 05 (cinco) passos:

- Passo1: Um novo nó é criado e a ele é associado o elemento a ser inserido na lista;
- Passo 2: A referência para o próximo elemento do novo nó é definida como nula, ou seja, não existe;
- Passo 3: A referência para o próximo elemento do final da lista é atualizada para indicar o novo "nó";
- Passo 4: O novo "nó" estabelece referência para o atual final da lista;
- Passo 5: O final da lista é atualizado e referencia o novo "nó".

Semelhante ao que ocorre na inserção no início da lista, o algoritmo sugerido para remoção no final deve considerar que a lista possua no mínimo 01 (um) elemento. No caso de a lista estar vazia, o "Passo 3" não poderá ser executado. Assim, não faz diferença executar a inserção no início ou no final, pois o resultado é uma lista com um único elemento e as duas referências (início e fim) apontando para o mesmo nó.

Uma vez que a sugestão é não "reinventar a roda", já existe a solução pronta no algoritmo "`InsereInicio`". Essa operação poderia verificar se a estrutura está vazia. Caso esteja, executa-se e a inserção no início.

Figura 4.27 – Lista Duplamente Encadeada - Inserir no final



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.27) e Java (ver Código Java 4.30) para o problema descrito.

Código C 4.27 – Lista Duplamente Encadeada - Inserir no Fim.

Em C:

```
int InsereFim(ListaDuplamenteEncadeada *lista, Elem
novo_elemento){
   No *atual, *novo;

   if (Vazia(*lista))
      return InsereInicio(lista, novo_elemento);
   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;

   novo->elemento = novo_elemento;
   novo->proximo = NULL;
   lista->fim->proximo = novo;
   novo->anterior = lista->fim;

   lista->fim = novo;
   return TRUE;
}
```

Código Java 4.30 – Lista Duplamente Encadeada - Inserir no Fim.

Em Java:

```
public void inserirFim(T elem) {
   if (this.isVazia()) this.inserirInicio(elem);
   else{
      No<T> no = new No<T>(elem);
      no.setProximo(null);
      this.fim.setProximo(no);
      no.setAnterior(this.fim);
      this.fim = no;
   }
}
```

No código Java, a implementação sugerida para a classe `No`, ao instanciar novo objeto (construtor), define que o próximo elemento é `null`. Para efeitos didáticos, o código apresentado

está fazendo essa atualização do próximo elemento explicitamente, ou seja, o comando "`no.setProximo(null);`" poderia ser omitido sem qualquer consequência.

4.4.4. Acessando Elementos

Na lista duplamente encadeada, as operações de acesso (imprimir e pesquisar) possuem implementação igual às apresentadas na `ListaSimplesmenteEncadeada` (ver seção 4.3.4).

4.4.5. Removendo Elementos

Semelhante ao que ocorre na inserção de elementos, as operações de remoção são executadas com a mesma eficiência, não importando em que extremidade será feita a inserção. Isso se deve ao fato da estrutura manter as duas referências para as extremidades, conforme explicado na seção 4.4.1.

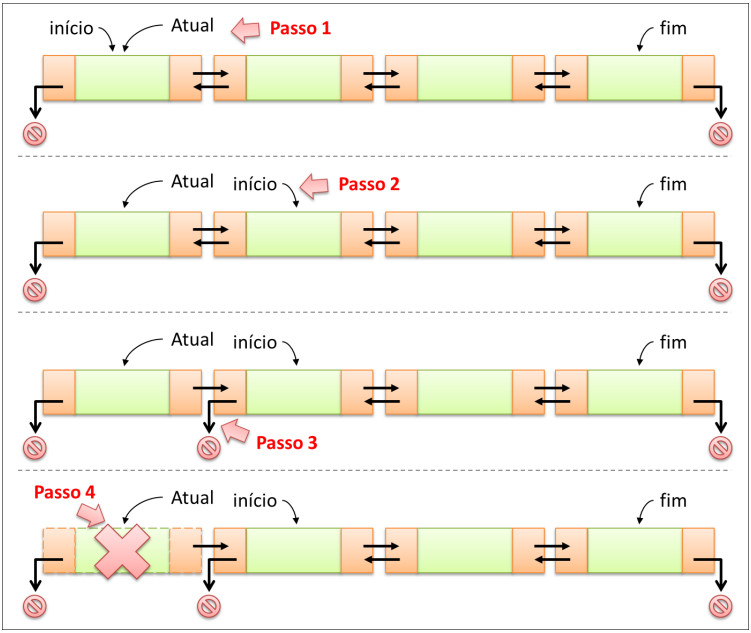
REMOVER DO INÍCIO

Em uma lista duplamente encadeada, para haver remoção do início, a referência que identifica o início da coleção deverá apontar para o próximo elemento da lista.
Por exemplo, na Figura 4.28 é possível observar uma lista duplamente encadeada com 04 (quatro) elementos. Para haver a remoção, é necessário seguir os 04 (quatro) passos indicados na imagem, sendo:

- Passo 1: Uma referência "`atual`" aponta para o primeiro nó da lista;
- Passo 2: A referência "`início`" da lista duplamente encadeada é alterada para indicar o próximo elemento da lista;

- Passo 3: É alterado o *link* "`anterior`" do nó indicado pela referência "`início`" da lista para registrar que não existe elemento, ou seja, o elemento é nulo;
- Passo 4: Uma vez executado o "passo 3", a memória alocada pela referência "atual" é liberada.

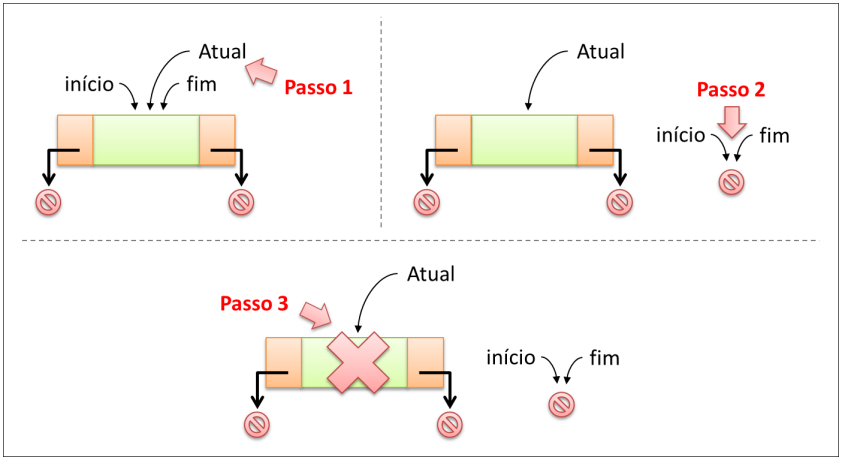
Figura 4.28 – Lista Duplamente Encadeada - Remover do Início



Analisando o algoritmo proposto para este tipo de remoção e considerando o cenário de uma lista contendo apenas um único elemento, o "passo 3" não poderá ser executado, por não existir um próximo elemento ligado ao antigo início.

Caso a coleção tenha apenas um único elemento, a remoção é mais simples, e uma sugestão é definir as referências início e fim para indicar que não existe elemento, ou seja, temos elemento nulo. Na Figura 4.29 é possível observar com maior clareza a sugestão proposta.

Figura 4.29 – Lista Duplamente Encadeada - Remover do Início (lista unitária)



A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.28) e Java (ver Código Java 4.31) para o problema descrito.

Código C 4.28 – Lista Duplamente Encadeada - Remover do início.

Em C:

```
int RemoveInicio(ListaDuplamenteEncadeada *lista,
Elem *elem){
   No *atual;
   if (Vazia(*lista)) return FALSE;
   atual = lista->inicio;
   *elem = atual->elemento;
   if (lista->inicio == lista->fim)
      lista->inicio = lista->fim = NULL;
   else{
      lista->inicio = atual->proximo;
      lista->inicio->anterior = NULL;
   }
   free(atual);
   return TRUE;
}
```

Antes de executar os passos 2 e 3, é verificado se a lista contém apenas um único elemento. Caso verdade, foi adotada a sugestão proposta nesta seção para listas unitárias.

Código Java 4.31 – Lista Duplamente Encadeada - Remover do início.

Em Java:

```
public T removerInicio() {
   if (this.isVazia()) throw new
ListaVaziaException();
   T elemento = this.inicio.getElemento();

   if (this.inicio == this.fim)
      this.inicio = this.fim = null;
   else{
      this.inicio.getProximo().setAnterior(null);
      this.inicio = this.inicio.getProximo();
   }
   return elemento;
}
```

REMOVER DO FINAL

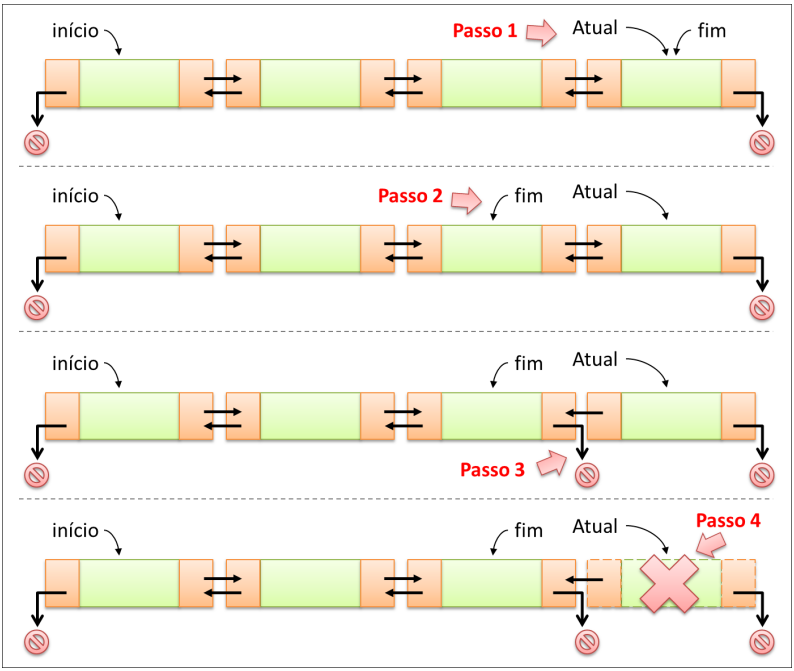
Semelhante ao que ocorre na operação de inserção no final, não é necessário percorrer a lista em busca do último elemento, uma vez que a referência "fim" da lista oferece acesso direto ao último nó.

Para realizar a remoção é necessário apenas que o penúltimo nó da lista atualize sua referência para indicar que não existe próximo elemento, ou seja, este é nulo.

Por exemplo, na Figura 4.30 está representada uma lista duplamente encadeada com 04 (quatro) elementos. Ao solicitar a remoção do final da lista, 04 (quatro) passos deverão ser executados:

- Passo 1: Uma referência "`atual`" aponta para o último nó da lista;
- Passo 2: A referência "`fim`" da lista duplamente encadeada é alterada para indicar o nó antecessor ao fim da lista;
- Passo 3: É alterado o link "`próximo`" do nó indicado pela referência "`fim`" da lista para registrar que não existe elemento, ou seja, este é nulo;
- Passo 4: Uma vez executado o "passo 3", a memória alocada pela referência "atual" é liberada.

Figura 4.30 – Lista Duplamente Encadeada - Remoção do Final



Considerando uma lista duplamente encadeada unitária, ou seja, que possui um único elemento, semelhante ao que ocorre na remoção do início, o "passo 3" não poderá ser executado.

Uma vez que esse tipo de problema já foi resolvido na remoção do início, a sugestão é realizar a operação `RemoverInicio`. Mais uma vez, não precisamos "reinventar a roda".

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.29) e Java (ver Código Java 4.32) para o problema descrito.

Código C 4.29 – Lista Duplamente Encadeada - Remoção do Final.

Em C:

```
int RemoveFim(ListaDuplamenteEncadeada *lista, Elem
*elem){
  No *atual;

  if (Vazia(*lista)) return FALSE;
  atual = lista->fim;

  if (lista->inicio == lista->fim)
     lista->inicio = lista->fim = NULL;
  else{
     atual->anterior->proximo = NULL;
     lista->fim = atual->anterior;
  }

  *elem = atual->elemento;

  free(atual);

  return TRUE;
}
```

Código Java 4.32 – Lista Duplamente Encadeada - Remoção do Final.

Em Java:

```
public T removerFim() {
   if (this.isVazia())
      throw new ListaVaziaException();
   T elemento = this.fim.getElemento();

   if (this.inicio == this.fim)
      this.inicio = this.fim = null;
   else{
      this.fim.getAnterior().setProximo(null);
      this.fim = this.fim.getAnterior();
   }
   return elemento;
}
```

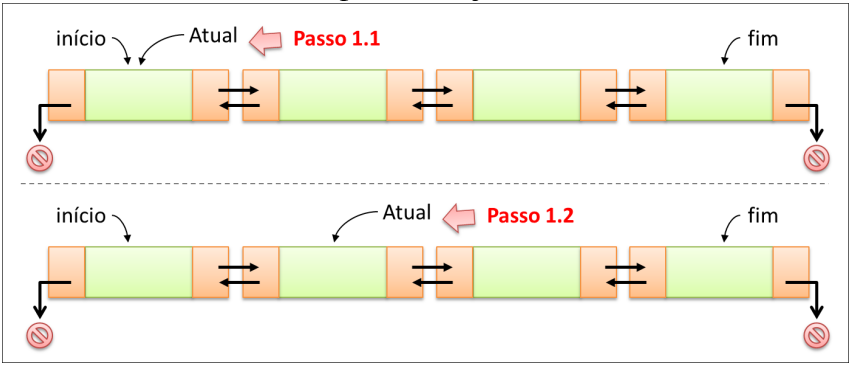
REMOVER BUSCANDO DETERMINADO ELEMENTO

A remoção buscando determinando elemento em uma lista duplamente encadeada tem implementação semelhante à lista simplesmente encadeada.

Ou seja, como foi feito para a lista simplesmente encadeada, o elemento será procurado e, caso encontrado, sugerimos que seja verificado se ele é o primeiro ou último elemento. Por fim, as referências são atualizadas para "encaixar" a remoção do nó.

Por exemplo, considerando uma coleção com 04 (quatro) elementos (Figura 4.31) e supondo que se deseja remover o elemento contido no segundo nó, primeiro é necessário percorrer a lista buscando o determinado elemento (passos 1.1 e 1.2).

Figura 4.31 – Lista Duplamente Encadeada - Buscando elemento para remoção

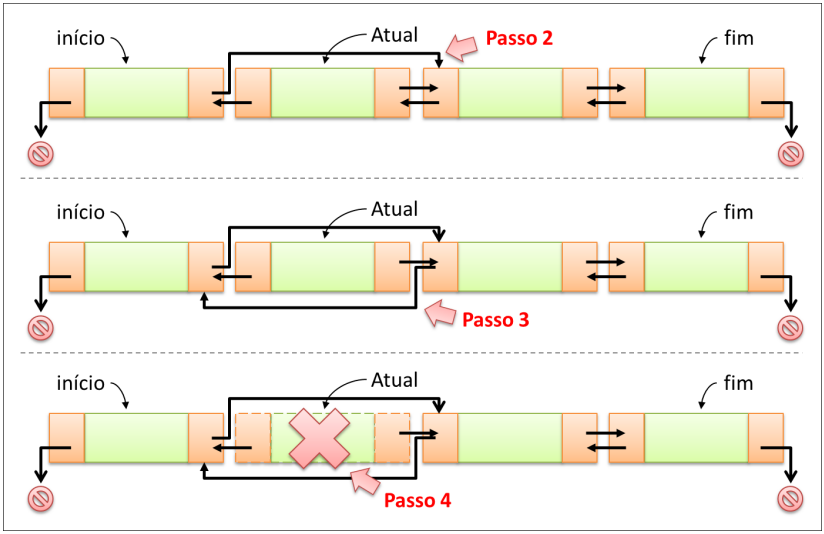


Encontrado o nó, e considerando que este não é o primeiro ou último nó da lista, teremos "`atual`" mantendo uma referência para ele.

Assim sendo, para removê-lo é necessário seguir mais 03 (três) passos, conforme pode ser observado naFigura 4.32.

- Passo 2: A referência "`próximo`" do nó que antecede o nó "`atual`" deve ser atualizada para indicar o próximo nó atualmente apontado por "`atual`";
- Passo 3: A referência "`anterior`" do nó posterior ao "`atual`" deve ser atualizada para indicar o nó anterior atualmente apontado por "`atual`";
- Passo 4: A memória apontada por "`atual`" deve ser liberada.

Figura 4.32 – Lista Duplamente Encadeada - Removendo elemento.



Caso a remoção solicitada seja o primeiro nó da lista ou o último nó da lista (que não é o caso deste exemplo que está sendo analisado), uma vez que esse problema já foi resolvido, essas implementações deverão ser chamadas.

O programador precisa estar atento para esse tipo de situção e, conforme dito no início desse livro, a essência principal das estruturas é não "reiventar a roda".

A seguir, são apresentadas soluções nas linguagens C (ver Código C 4.30) e Java (ver Código Java 4.33) para o problema descrito.

Código C 4.30 – Lista Duplamente Encadeada - Remoção Buscando Elemento

Em C:

```
int Remove(ListaDuplamenteEncadeada *lista, char
*nome, Elem *elem){
  No *atual;

  if (Vazia(*lista)) return FALSE;
  atual = lista->inicio;
  while (atual != NULL){
    if (stricmp(atual->elemento.nome, nome) == 0){
      if (atual == lista->inicio)
           return RemoveInicio(lista, elem);
      else if (atual == lista->fim)
           return RemoveFim(lista, elem);
      atual->anterior->proximo = atual->proximo;
      atual->proximo->anterior = atual->anterior;

      *elem = atual->elemento;
      free(atual);
      return TRUE;
    }
    atual = atual->proximo;
  }
  return FALSE;
}
```

Código Java 4.33 – Lista Duplamente Encadeada - Remoção Buscando Elemento.

Em Java:

```
public void remover(T elem) {
   if (this.isVazia())
      throw new ListaVaziaException();

   No<T> atual = this.inicio;
   while (atual != null){
      if (atual.getElemento().equals(elem)){
         if (atual.getAnterior() == null)
             this.removerInicio();
         else if (atual.getProximo() == null)
             this.removerFim();
         else{

atual.getAnterior().setProximo(atual.getProximo());

atual.getProximo().setAnterior(atual.getAnterior());
         }
         return;
      }
      atual = atual.getProximo();
   }
   throw new ElementoNaoExisteException();
}
```

Até agora foram apresentadas diferentes formas de implementar o TAD "Lista Genérica". Dependendo dos requisitos da aplicação, o programador poderá escolher a implementação que atenda esses requisitos e ofereça o maior número de recursos e maior eficiência.

Os algoritmos apresentados podem ser alterados para agregar mais funcionalidades, por exemplo, a lista poderia adotar critério de ordenação e manter seus elementos ordenados segundo esse

critério, ou seja, quem decide a posição em que o novo elemento será inserido é a estrutura.

A dinâmica do funcionamento das estruturas também poderia ser alterada buscando mais eficiência na execução de suas operações. Por exemplo, todas as implementações apresentadas tratam o percurso da lista partindo do início até o final. Uma vez percorrida a lista, voltar ao início apenas é possível no duplo encadeamento (lista duplamente encadeada) e, dependendo no número de elementos, pode ser uma operação muito onerosa para o sistema.
Uma solução simples para esse cenário é realizar a implementação circular, ou seja, o último elemento da coleção aponta para o primeiro. Dessa forma, além de resolver o problema citado (inclusive para o encadeamento simples entre os elementos), a partir de qualquer ponto da lista, é possível chegar a qualquer outro ponto desta lista.

A lista genérica, conforme o nome sugere, é uma estrutura que permite inserção e remoção em qualquer posição. Em algumas situações é preciso definir políticas de acesso aos elementos de uma coleção e algumas operações não poderiam estar disponíveis para execução. Para exemplificar, suponhamos que um sistema de "*Help Desk*" precise manter uma fila de solicitações que deve ser atendida por ordem de chegada (sistema honesto). Entende-se que o primeiro pedido registrado deve ser o primeiro a ser executado, e não qualquer outro, ou seja, remoções no meio ou no final da estrutura não são permitidas. Nos próximos capítulos serão tratadas essas estruturas.

4.4.6. API Java

A documentação de Java para a classe "`LinkedList`" (citada na implementação Java para as listas simplesmente encadeadas) informa que a estrutura utilizada para ligar os elementos é a

duplamente encadeada. Considerando esse fato, tudo que foi apresentado na seção 4.3.6 também se aplica a essa estrutura.

4.5. Exercícios Propostos

1) Qual é a principal característica de uma lista genérica?

2) As listas genéricas obrigatoriamente devem possuir critério de ordenação para seus elementos? Por quê?

3) Quais são as principais operações realizadas por uma lista genérica?

4) A busca binária possui desempenho melhor ou pior do que a busca sequencial? Por quê?

5) O que é uma lista genérica estática?

6) A lista genérica estática apresenta apenas uma única forma de implementação?

7) A linguagem de programação C permite alocar uma coleção estática (vetor) em tempo de execução? Como?

8) As operações “Cheia” e “Vazia” são dispensáveis em uma lista genérica?

9) Para a estrutura lista, existe diferença entre inserir no início e inserir no final? O que acontece na lista quando essas operações são executadas?

10) Na lista genérica é possível remover um elemento que não esteja localizado no início ou final da estrutura? Por quê?

11) Na lista genérica é possível inserir elementos em qualquer ponto da estrutura? Por quê?

12) Faça as alterações necessárias na lista genérica para adotar critério de ordenação na execução das operações inserir, remover e buscar.

13) Quais são as principais diferenças entre as listas genéricas estática e a encadeada?

14) Quando a lista genérica encadeada está cheia?

15) Existe problema de desempenho na execução das rotinas de inserção e remoção na lista genérica encadeada?

16) Existe vantagem na implementação duplamente encadeada para as listas? Caso exista, explique.

17) Considerando as implementações encadeadas das listas, em qual momento a operação que insere no final vai também alterar a referência que está indicando o primeiro elemento da coleção?

18) É possível percorrer uma lista genérica duplamente encadeada nos dois sentidos da estrutura (do início para o final e do final para o início)? Como?

19) A API de Java traz a implementação de listas? Explique.

20) Desenvolva uma aplicação que faça uso das estruturas apresentadas neste capítulo.

5 Pilha

A pilha é uma estrutura linear em que as operações de inserção e retirada são realizadas em um único ponto da estrutura. Ou seja, o acesso aos elementos deve ser feito a partir do topo.

Supondo uma pilha formada pelos elementos E_1, E_2,... E_n, se desejarmos adicionar um novo elemento, ele será o elemento E_{n+1}. Uma vez que o elemento E_{n+1} representa o elemento do topo, se desejarmos retirar um elemento da pilha, teremos que excluir o elemento E_{n+1}. A pilha é comumente conhecida como estrutura "LIFO" (*last in, first out*).

Um exemplo dessa estrutura é uma pilha de pratos sujos, em que o último prato empilhado (localizado no topo) é o primeiro prato a ser lavado. Ou seja, se vamos colocar (inserir) mais um prato na pilha, colocamos no topo. Se vamos retirar um prato da pilha, retiramos aquele que está no topo.

São exemplos de uso de pilhas em programação: pilha de execução de uma linguagem de programação (em que variáveis locais são armazenadas em pilha), chamadas de subrotinas por um compilador ou sistema operacional, inversão de listas, conversão e avaliação de expressões algébricas (em que operadores para expressões matemáticas são armazenados em pilha).

A literatura que trata as estruturas de dados classifica a pilha como uma estrutura simples e poderosa e, por isso, bastante utilizada pelos desenvolvedores.

Uma das estruturas de dados mais simples é a pilha. Celes *et al.* (2004) explica que possivelmente por essa razão, é a estrutura de dados mais utilizada em programação. Pilha é um conceito muito utilizado na ciência da computação para a solução de problemas. Silva (2007) explica que a pilha é uma estrutura simples e muito poderosa como elemento básico de solução de problemas complexos.

Para Tenembaum *et al.* (1995), um dos conceitos mais úteis na ciência da computação é o de pilha. Segundo estes autores, a pilha desempenha um proeminente papel nas áreas de programação e de linguagens de programação.

5.1. Operações

Por se tratar de uma forma particular de uma lista (pode ser entendida como uma lista com restrições de acesso), a pilha oferece um conjunto de operações restritas. O objetivo é promover o gerenciamento dos dados seguindo a sua filosofia LIFO.

Tomando como base a lista genérica apresentada no capítulo anterior, as pilhas também possuem um conjunto de operações classificadas como: básica, inserção, acesso e remoção. Essas operações gerenciam sua principal região, que é o topo da pilha.

Diferente do que ocorre nas listas genéricas, as pilhas possuem padrão de nomenclatura para suas operações de inserção e remoção, sendo respectivamente *push* (empilhamento ou inserção) e *pop* (desempilhamento ou remoção). Não apenas existe padrão na nomenclatura das operações de inserção e remoção, como seu procedimento também é padrão (todas as implementações seguem o mesmo comportamento) e determina que seja possível realizar apenas inserção e remoção em uma única posição da estrutura, no topo da pilha.

O conjunto de operações e a estrutura utilizada para armazenamento dos dados definem o TAD da pilha. Considerando a estrutura de armazenamento (gerenciamento da memória), semelhante à lista genérica, a pilha também pode usar estratégia estática (vetor) e dinâmica (encadeamento de elementos). A seguir serão apresentados aspectos técnicos dessas duas formas de implementação.

5.2. Pilha Estática

Utilizando o vetor como estrutura de armazenamento e considerando que as operações devem gerenciar apenas uma única extremidade (topo), as opções para realizar as operações de inserção e remoção na pilha devem ser analisadas com muito critério, investigando principalmente a sua eficiência.

As operações de inserção nas extremidades da estrutura foram apresentadas nas listas genéricas, sendo elas: inserção no início e inserção no final. Analisando a inserção no início, no caso do vetor, cada novo elemento adicionado implica em deslocamento de todos os dados contidos na estrutura para a outra extremidade (fim).

Essa estratégia de inserção apresenta desempenho pior a cada novo elemento inserido (mais deslocamentos serão necessários realizar). Por outro lado, as inserções no final da estrutura não necessitam de deslocamento dos dados, já que a inserção é direta e o novo elemento é inserido na última posição disponível no vetor.

É possível concluir que o empilhamento deverá ocorrer no final da estrutura e, consequentemente, as remoções também deverão ocorrer no final. Uma vez escolhido o topo (extremidade), é possível definir seu TAD e seguir com a codificação.

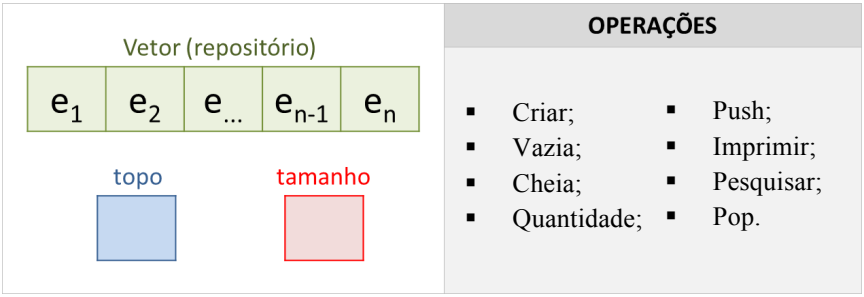
!	Seguindo a estratégia adotada pelo livro para alocação de vetores, a pilha também fará uso de alocação dinâmica de memória.

5.2.1. Definição do TAD

Além do vetor (repositório dos dados), é necessário controlar a quantidade de elementos contidos na estrutura; com isso, será possível realizar a verificação da capacidade de armazenamento da pilha. Uma vez que será adotada a estratégia de adicionar e remover elementos no final da estrutura (topo), fica subentendido que essa quantidade indica exatamente o topo da pilha.

Diante disso, a Figura 5.1 representa graficamente o TAD da Pilha Estática, que é muito semelhante ao TAD da Lista Estática. A diferença é que a pilha estática gerencia a quantidade de elementos através do "`topo`".

Figura 5.1 – Pilha Estática - TAD



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.1) e Java (ver Código Java 5.1) para o problema descrito.

Código C 5.1 – Pilha Estática - TAD

Em C:

```
typedef struct{
      char nome[31];
      int idade;
}Elem;

typedef struct {
      Elem *elementos;
      int tamanho, topo;
}Pilha;
```

Código Java 5.1 – Pilha Estática - TAD

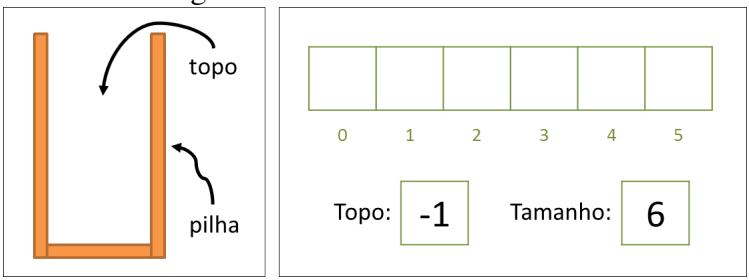
Em Java:

```
public class PilhaEstatica <T>{
      private T[] elementos;
      private int topo;
}
```

5.2.2. Operações Básicas

As quatro operações básicas (Criar, Vazia, Cheia e Tamanho) essencialmente definem e analisam o "topo da pilha" na sua execução. Considerando que "topo" deve indicar o elemento "da vez" (ou seja, sua posição ou índice do vetor), ao ser criada, a estrutura deve registrar que não possui elemento. Para evitar problemas de entendimento, é sugerido não utilizar posição (índice) válida do vetor para representar a pilha vazia. Por exemplo, a pilha vazia poderia receber valor "-1" (um negativo). Na Figura 5.2, é possível observar uma pilha vazia.

Figura 5.2 – Pilha Estática Vazia



Para verificar se a pilha está vazia, é suficiente comparar se o topo é "-1". Caso seja verdade (topo é "-1"), a pilha encontra-se vazia, caso contrário, não está vazia. A operação que verifica se a pilha está cheia deve comparar se o valor do topo acrescido de uma unidade é igual ao tamanho da estrutura. Caso seja verdade, a pilha está cheia, caso contrário, não está cheia. A quantidade de elementos armazenados na estrutura corresponde ao topo da pilha acrescido de uma unidade.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.2) e Java (ver Código Java 5.2) para o problema descrito.

Código C 5.2 – Pilha Estática - Operações Básicas

Em C:

```
// cria a pilha
int Cria(Pilha *pilha, int tamanho){

  if ((pilha->elementos = calloc(tamanho,
sizeof(Elem))) == NULL)
     return FALSE;

  pilha->tamanho = tamanho;
  pilha->topo = -1;

  return TRUE;
}
```

```
// verifica se a pilha está vazia
int Vazia(Pilha pilha){

   return (pilha.topo == -1);
}


// verifica se a pilha está cheia
int Cheia(Pilha pilha){

   return (pilha.topo + 1 == pilha.tamanho);
}


// retorna a quantidade de elementos armazenados na
pilha
int Quantidade(Pilha pilha){

   return pilha.topo + 1;
}
```

Código Java 5.2 – Pilha Estática - Operações Básicas

Em Java:

```
// construção da pilha
public PilhaEstatica(int tamanho) {
   this.elementos = (T[]) new Object[tamanho];
   this.topo = -1;
}

// verificando se a pilha está cheia
public boolean isCheia(){
   return this.topo + 1 == this.elementos.length;
}

// verifica se a pilha está vazia
public boolean isVazia(){
   return this.topo == - 1;
}
```

```
// retorna a quantidade de elementos armazenados na pilha
public int getQuantidade(){
   return this.topo + 1;
}
```

5.2.3. Inserindo Elementos

A única forma de inserir elementos em uma pilha (empilhamento) é através do seu topo e essa operação é denominada `push`.

Uma vez que o topo inicia com valor "`-1`", depois de avaliar que a estrutura não está cheia, e antes de armazenar o elemento, o valor do topo é incrementado em uma unidade.

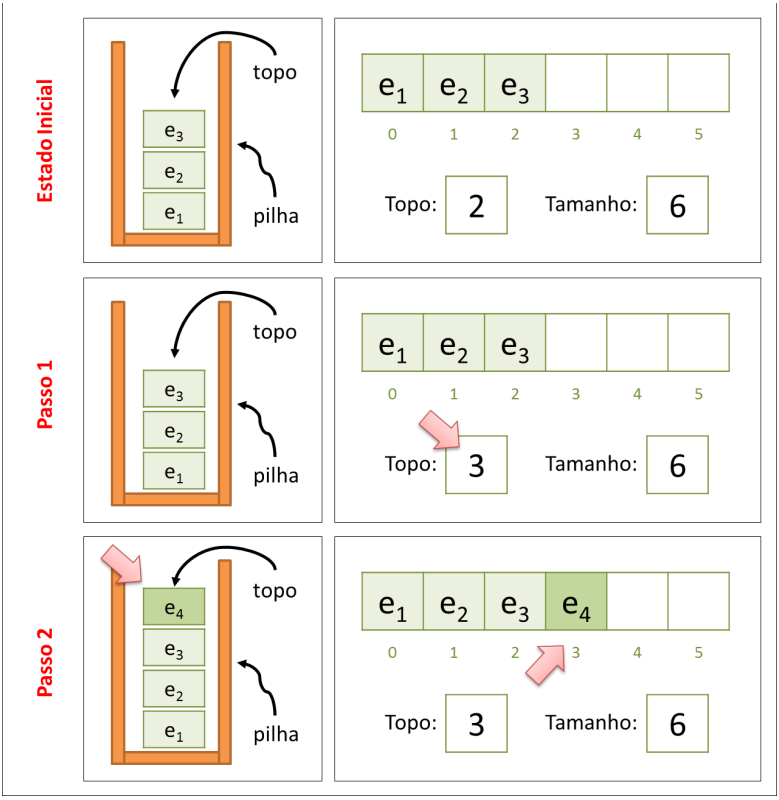
Por exemplo, na Figura 5.3 está representada uma pilha com inicialmente 03 (três) elementos (e_1, e_2 e e_3) e capacidade de armazenamento 06 (seis).

Para adicionar (*push*) um novo elemento (e_4), o topo deverá ser incrementado (passo 1).

Inicialmente, a referência para o topo era 2. Para adicionar o novo elemento, a referência do topo passa a ser 3. Assim sendo, o novo elemento poderá ser inserido (passo2).

!	Apenas será possível executar essa operação se a pilha não estiver **cheia**.

Figura 5.3 – Pilha Estática - Operação Push



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.3) e Java (ver Código Java 5.3) para o problema descrito.

Código C 5.3 – Pilha Estática - Empilhamento (*push*)

Em C:

```
int Push(Pilha *pilha, Elem elem){
   if (Cheia(*pilha)) return FALSE;
   pilha->elementos[++pilha->topo] = elem;
   return TRUE;
}
```

Código Java 5.3 – Pilha Estática - Empilhamento (*push*)

Em Java:

```
public void push(T novo) throws PilhaCheiaException{
   if (this.isCheia())
      throw new PilhaCheiaException();
   this.elementos[++this.topo] = novo;
}
```

5.2.4. Acessando Elementos

As operações de acesso (impressão e pesquisa) têm comportamento semelhante ao da lista estática. A principal diferença é que o percurso da pilha deve considerar o topo da pilha como início da estrutura e índice válido (contém elemento).

Uma vez que não é possível realizar remoção em posição diferente do topo, a busca não precisa retornar a posição em que o elemento está localizado, sendo suficiente sinalizar se o elemento está ou não armazenado na pilha.

O topo da pilha é um elemento de fundamental importância para estrutura; a partir deste é que elementos são inseridos e/ou removidos. Diante de tal importância, se faz necessário definir uma operação que permita verificar qual elemento (caso haja) está posicionado no topo da pilha.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.4) e Java (ver Código Java 5.4) para o problema descrito.

Código C 5.4 – Pilha Estática - Operações de Acesso.

Em C:

```
// Imprime a pilha

void Imprimir(Pilha pilha){
   int i;

   for(i = pilha.topo; i >= 0; --i){
      printf("Nome: %s - ", pilha.elementos[i].nome);
      printf("Idade: %d\n", pilha.elementos[i].idade);

   }
}


// Busca por um determinado elemento na pilha
int Pesquisar(Pilha pilha, char *nome){
   int i;

   for (i = 0; i <= pilha.topo; ++i)
      if (stricmp(pilha.elementos[i].nome, nome) == 0)
         return TRUE;
   return FALSE;
}


// Obter o elemento posicionado no topo da pilha
int Topo(Pilha pilha, Elem *elem){
   if (Vazia(pilha)) return FALSE;
   *elem = pilha.elementos[pilha.topo];
   return TRUE;
}
```

Código Java 5.4 – Pilha Estática - Operações de Acesso

Em Java:

```
// Retorna um "Iterator" para impressão dos elementos
public Iterator<T> get(){
   @SuppressWarnings("unchecked")
   T[] auxiliar = (T[]) new Object[this.topo + 1];
   for(int i = this.topo; i >= 0; i--)
      auxiliar[this.topo - i] = this.elementos[i];
   return Arrays.asList(auxiliar).iterator();
}

// Busca por um determinado elemento na pilha
public boolean get(T elemento) {
   for (int i = 0; i <= this.topo; ++i)
      if (this.elementos[i].equals(elemento))
         return true;
   return false;
}
```

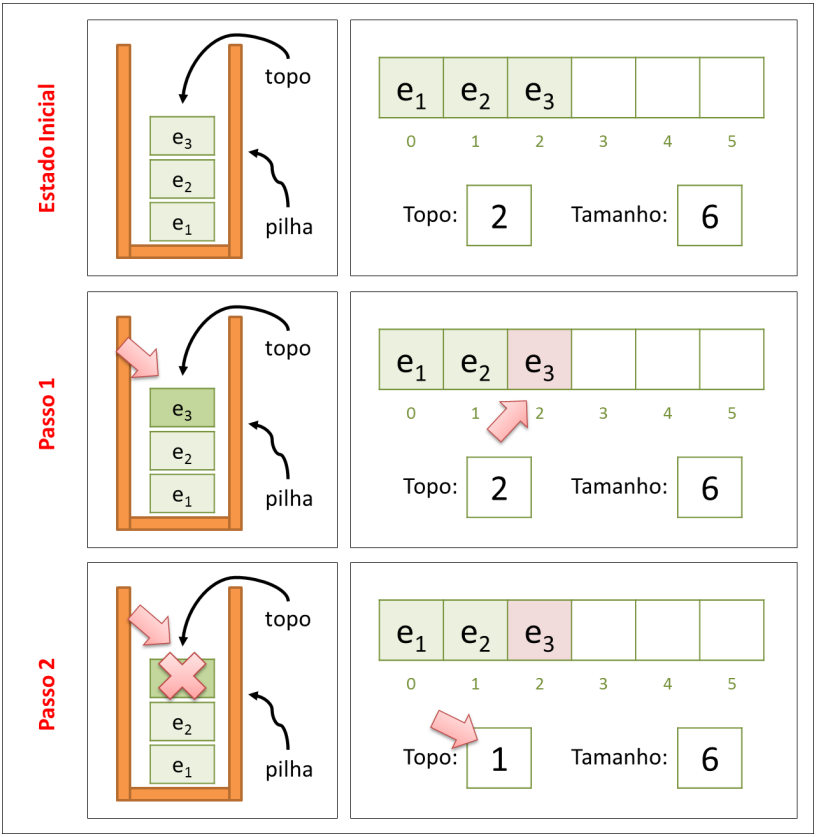
5.2.5. Removendo elementos

Semelhante ao que ocorre na inserção de elementos, as remoções na pilha (desempilhamento) só podem ocorrer no topo, sendo essa operação denominada *pop*. Considerando que o topo indica o índice no vetor que contém o elemento que deverá ser removido, depois de avaliar que a estrutura não está vazia, e após reservar o elemento que está no topo, este deverá ser decrementado em uma unidade.

Na Figura 5.4 está representada graficamente uma pilha contendo inicialmente 03 (três) elementos (e_1,e_2 e e_3). Ao solicitar remoção (desempilhamento), a operação `pop` deverá reservar o elemento que está no topo (passo 1), depois decrementar o valor do topo (passo 2), que era 2 (dois) e passa a ser 1 (um). Apenas será possível executar essa operação se a pilha não estiver vazia.

Caso a operação `push` venha a ser executada, solicitando a inserção do elemento e_4, uma vez que o atual topo é 1 (um), esse novo elemento será inserido no índice 2 (dois), ou seja, haverá uma sobreposição de elementos. Isso não representa um problema, uma vez que o outro elemento (e_3) havia sido removido anteriormente.

Figura 5.4 – Pilha Estática - Operação Pop



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.5) e Java (ver Código Java 5.5) para o problema descrito.

Código C 5.5 – Pilha Estática - Desempilhamento (*pop*)

Em C:

```
int Pop(Pilha *pilha, Elem *elem){
   if (Vazia(*pilha)) return FALSE;
   *elem = pilha->elementos[pilha->topo--];
   return TRUE;
}
```

Código Java 5.5 – Pilha Estática - Desempilhamento (pop)

Em Java:

```
public T pop() throws PilhaVaziaException{
   if (this.isVazia()) throw new
PilhaVaziaException();
   return this.elementos[this.topo--];
}
```

5.3. Pilha Encadeada

Essa forma de implementação é baseada no encadeamento de "nó", em que cada um contém um elemento que está sendo persistido na estrutura. As operações de empilhamento (*push*) e desempilhamento (*pop*) não fazem deslocamento físico dos dados, ou seja, é necessário apenas atualizar as referências dos nós na estrutura para "encaixar" a operação que está sendo executada.

Considerando a pilha simplesmente encadeada (estrutura em que se percebe a conexão do elemento atual com o próximo elemento), para instanciar a mesma lógica adotada na pilha estática (ou seja, lógica em que o topo representa o final da lista), seria necessário percorrer toda a lista para conseguir executar as operações sobre ela. Nesse contexto, quanto maior o número de inserções, pior será o desempenho.

No caso da pilha duplamente encadeada esse problema não existe, uma vez que a estrutura mantém uma referência para o último nó da coleção.

Desconsiderando aspectos de desempenho, para localizar as extremidades da pilha, as operações de inserção e remoção possuem a mesma eficiência, seja qual for a extremidade escolhida, uma vez que apenas serão atualizadas as referências dos nós envolvidos na operação que estará sendo executada.

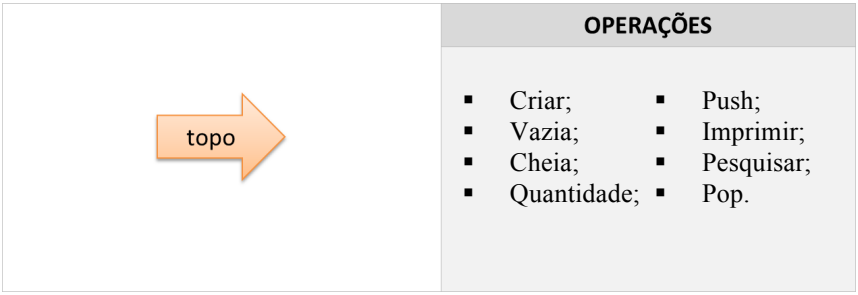
Uma vez que a inserção no início da pilha não apresenta o ônus do percurso em busca do último elemento e promovendo uma solução que atenda as duas formas encadeadas de implementação apresentadas neste livro, é escolhido o início como topo da pilha. Essa definição apresenta comportamento inverso ao da pilha estática.

5.3.1. Definição do TAD

Semelhante ao que ocorre com a lista encadeada (simples e dupla), a pilha deverá manter apenas uma referência para o primeiro nó da estrutura. Essa referência é denominada topo.

Na Figura 5.5 é possível observar o TAD para Pilha Encadeada, sendo considerada a implementação simplesmente encadeada. Para transformar esta representação em uma pilha duplamente encadeada, uma nova referência (fim) deverá ser acrescentada, o que não repercute em grandes vantagens de desempenho para inserção e remoção de elementos.

Figura 5.5 – Pilha Encadeada - TAD



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.6) e Java (ver Código Java 5.6) para o problema descrito.

Código C 5.6 – Pilha Encadeada - TAD

Em C:

```
typedef struct{
   char nome[31];
   int idade;
}Elem;

typedef struct no{
   Elem elemento;
   struct no *proximo;
}No;

typedef struct{
   No *topo;
}Pilha;
```

Código Java 5.6 – Pilha Encadeada - TAD

Em Java:

```
public class No <T>{
   private T elemento;
   private No<T> proximo;
}
```

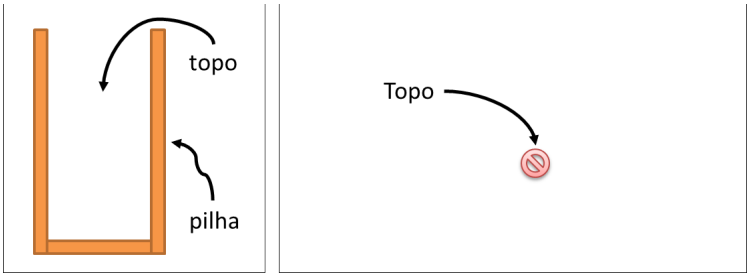
```
public class PilhaEncadeada <T>{
   private No<T> topo;
}
```

5.3.2. Operações Básicas

As operações básicas sobre as pilhas seguem a mesma linha de raciocínio apresentada na seção 5.2.2, essencialmente definindo e verificando o topo para realizar suas operações, exceto a verificação da quantidade de elementos, para o que é necessário percorrer toda a pilha.

Uma alternativa para evitar esse gasto operacional é a pilha manter uma variável para armazenar a quantidade de elementos que nela está contida. Essa alternativa representa um alto risco, uma vez que haverá redundância de informação (a quantidade poderá ser obtida resgatando essa variável e/ou percorrendo a pilha). Redundância de informação pode gerar inconsistência, o que determinaria a não confiabilidade da estrutura e isso não é desejável. Para evitar qualquer problema, dependendo dos requisitos da aplicação, a manutenção dessa variável é totalmente desaconselhável.
A pilha deverá ser criada indicando que não contém elemento, ou seja, o seu topo é `NULL`. A operação para verificar se ela está vazia observará o topo. Caso o topo tenha valor `NULL`, a pilha está vazia. A operação para verificar se a pilha está cheia não se aplica a essa implementação, uma vez que não existe limite previamente estabelecido e, nesse caso, a memória da máquina é quem vai determinar o limite para o número de elementos da estrutura. Enquanto for possível alocar memória, os elementos podem ser adicionados sem problemas. Na Figura 5.6 está representada graficamente a pilha vazia e sua referência para o primeiro elemento (ou seja, o topo), que armazena valor `NULL`.

Figura 5.6 – Pilha Encadeada Vazia



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.7) e Java (ver Código Java 5.7) para o problema descrito.

Código C 5.7 – Pilha Estática - Operações Básicas.

Em C:

```
// cria a pilha
void Cria(Pilha *pilha){
   pilha->topo = NULL;
}

// verifica se a pilha está vazia
int Vazia(Pilha pilha){
   return pilha.topo == NULL;
}

// retorna a quantidade de elementos armazenados na
pilha
int Quantidade(Pilha pilha){
   int qtde = 0;
   No *atual = pilha.topo;
   while(atual != NULL){
      ++qtde;
      atual = atual->proximo;
   }
   return qtde;
}
```

Código Java 5.7 – Pilha Estática - Operações Básicas.

Em Java:

```
// construção da pilha
public PilhaEncadeada(){
   this.topo = null;
}


// verifica se a pilha está vazia
public boolean isVazia(){
   return this.topo == null;
}


// retorna a quantidade de elementos armazenados na
pilha
public int getQuantidade(){
   int qtde = 0;

   No<T> atual = this.topo;

   while (atual != null){
      ++qtde;
      atual = atual.getProximo();
   }

   return qtde;
}
```

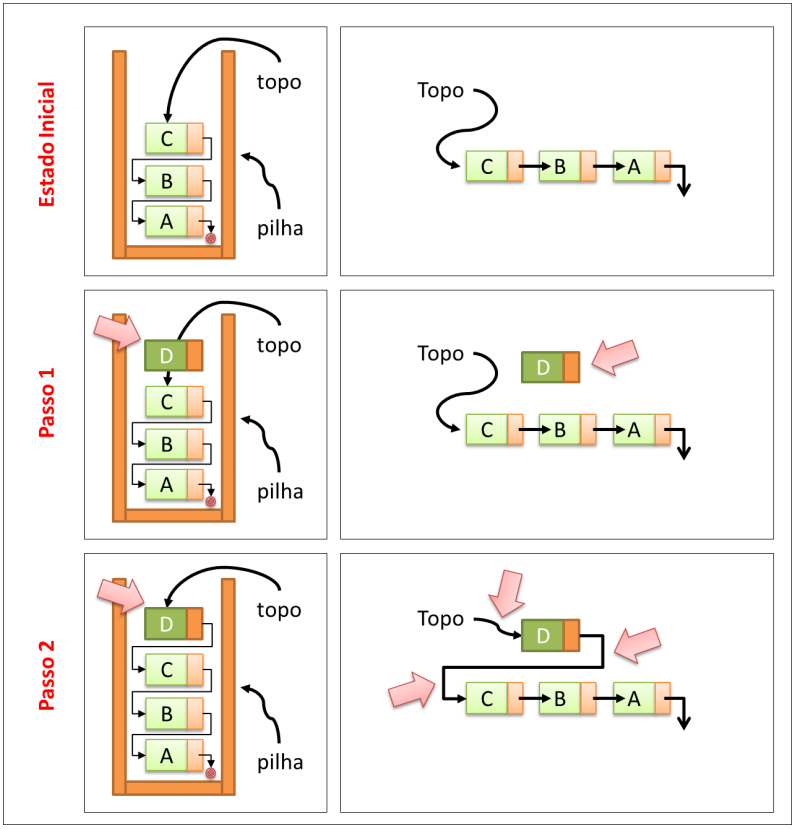
5.3.3. Inserindo Elementos

Não importa a forma de encadeamento (simples ou duplo), a inserção (*push*) será executada no topo da pilha (nesse caso, no início da coleção). Um novo nó deverá ser alocado contendo o elemento que será inserido. Esse nó atualizará sua referência para apontar para o atual início da pilha, e então passará a ser o novo topo.

Na Figura 5.7 está representada graficamente uma pilha encadeada contendo 03 (três) elementos (A, B e C). Para inserir (*push*) o quarto elemento (D), primeiro é necessário alocar memória para esse novo nó (passo 1). Uma vez alocado o novo nó, este deverá ser inserido no topo da pilha, ou seja, no seu início (passo 2).

O resultado final é uma pilha contendo os seguintes elementos: D, C, B e A.

Figura 5.7 – Pilha Encadeada - Operação Push



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.8) e Java (ver Código Java 5.8) para o problema descrito.

Código C 5.8 – Pilha Encadeada - Empilhamento (*push*)

Em C:

```
int Push(Pilha *pilha, Elem novo_elemento){
   No *novo;

   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;
   novo->elemento = novo_elemento;
   novo->proximo = pilha->topo;
   pilha->topo = novo;

   return TRUE;
}
```

Código Java 5.8 – Pilha Encadeada - Empilhamento (*push*)

Em Java:

```
public void push(T elemento){
   No<T> novo = new No<T>(elemento);
   novo.setProximo(this.topo);
   this.topo = novo;
}
```

5.3.4. Acessando Elementos

O acesso aos elementos para imprimir ou pesquisar seguem a mesma implementação da lista encadeada, ou seja, a pilha é percorrida do início (topo) até o final da estrutura para busca ou impressão dos elementos. O mesmo ocorre para verificação do elemento posicionado no topo da pilha.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.9) e Java (ver Código Java 5.9) para o problema descrito.

Código C 5.9 – Pilha Encadeada - Operações de Acesso

Em C:

```
// Imprime a pilha
void Imprime(Pilha pilha){
   No *atual = pilha.topo;
   while (atual != NULL){
      printf("Nome: %s - ", atual->elemento.nome);
      printf("Idade: %d\n", atual->elemento.idade);
      atual = atual->proximo;
   }
}

// Busca por um determinado elemento na pilha
int Pesquisar(Pilha pilha, char *nome){
   No *atual = pilha.topo;
   while (atual != NULL){
      if (stricmp(atual->elemento.nome, nome))
         return TRUE;
      atual = atual->proximo;
   }
   return FALSE;
}

// Obter o elemento posicionado no topo da pilha
int Topo(Pilha pilha, Elem *elem){
   if (Vazia(pilha)) return FALSE;
   *elem = pilha.topo->elemento;
   return TRUE;
}
```

Código Java 5.9: Pilha Encadeada - Operações de Acesso

Em Java:

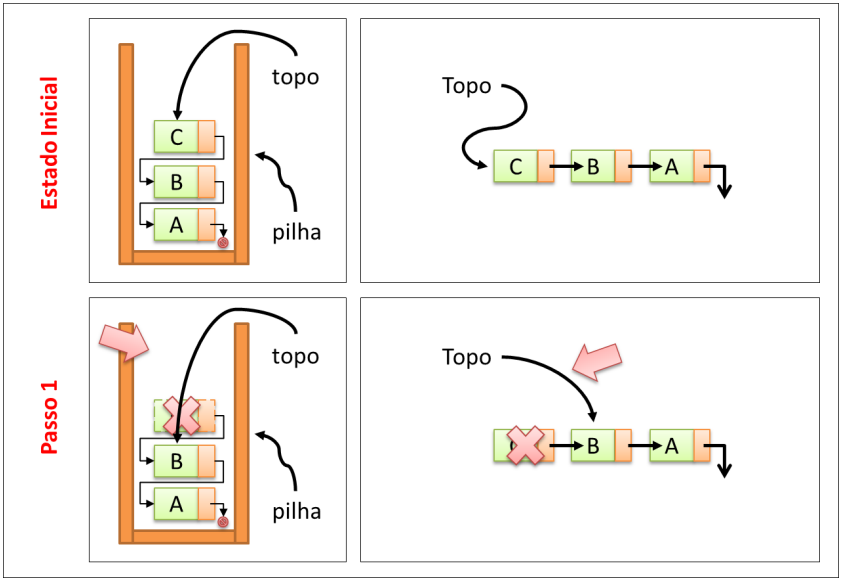
```
// Imprime a pilha
public Iterator<T> get() {
   int i = 0;

   @SuppressWarnings("unchecked")
   T[] vetor = (T[]) new Object[this.getQuantidade()];

   No<T> atual = this.topo;
   while(atual != null) {
      vetor[i++] = atual.getElemento();
      atual = atual.getProximo();
   }
   return Arrays.asList(vetor).iterator();
}


// Busca por um determinado elemento na pilha
public int get(T elem) {
   int i = 0;
   No<T> atual = this.topo;
   while (atual != null){
      if (atual.getElemento().equals(elem)) return i;
      atual = atual.getProximo();
      i++;
   }
   throw new ElementoNaoExisteException();
}


// Obter o elemento posicionado no topo da pilha
public T getTopo() throws PilhaVaziaException{
   if (this.isVazia())
      throw new PilhaVaziaException();
   return this.topo.getElemento();
}
```

5.3.5. Removendo Elementos

Caso a pilha contenha elemento (não esteja vazia), será removido o que estiver posicionado no topo (início do encadeamento de "nó"). Para que haja essa remoção, o topo fará referência ao elemento que ele aponta como próximo (ou seja, o próximo elemento do topo).

Na Figura 5.8 está representada uma pilha contendo 03 (três) elementos (C, B e A) inicialmente. Ao executar a operação "`pop`", o elemento que está no topo (C) será removido. Para que isso ocorra (Passo 1), o topo que estava apontando para o nó contendo o elemento "C" agora irá referenciar o próximo elemento a partir dele, ou seja, o nó contendo o elemento "D".

Figura 5.8 – Pilha Encadeada - Operação Pop



A seguir, são apresentadas soluções nas linguagens C (ver Código C 5.10) e Java (ver Código Java 5.10) para o problema descrito.

Código C 5.10 – Pilha Encadeada - Desempilhamento (*pop*)

Em C:

```
int Pop(Pilha *pilha, Elem *elem){
   No *atual;

   if (Vazia(*pilha)) return FALSE;
   atual = pilha->topo;
   pilha->topo = atual->proximo;
   *elem = atual->elemento;
   free(atual);
   return TRUE;
}
```

Código Java 5.10 – Pilha Encadeada - Desempilhamento (*pop*)

Em Java:

```
public T pop() throws PilhaVaziaException{
   T elem;
   if (this.isVazia())
      throw new PilhaVaziaException();
   elem = this.topo.getElemento();
   this.topo = this.topo.getProximo();
   return elem;
}
```

5.4. API Java

O *framework Collections*[10] traz uma implementação para a estrutura de dados "Pilha": é a classe Stack[11] (`java.util.Stack<E>`). Essa classe herda da classe *Vector*[12]

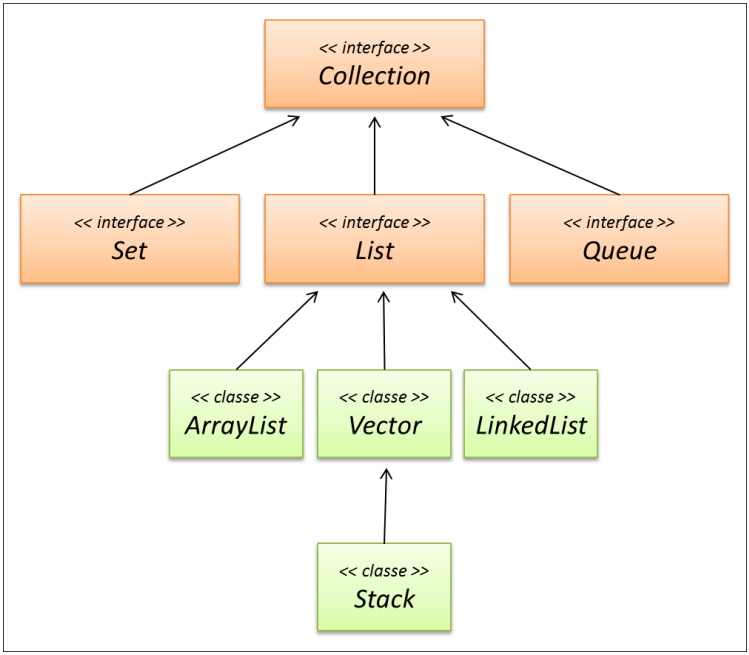
[10] http://docs.oracle.com/javase/6/docs/technotes/guides/collections/index.html
[11] http://docs.oracle.com/javase/6/docs/api/java/util/Stack.html
[12] http://docs.oracle.com/javase/6/docs/api/java/util/Vector.html

(`java.util.Vector<E>`), que por sua vez é uma implementação da interface *List*.

!	Por *Stack* ser uma herança da classe *Vector*, além dos serviços específicos que ela implementa, também oferece como opção todos os serviços herdados da classe *Vector*.

Na Figura 5.9, é possível observar a hierarquia de interfaces e classes do *Framework Collections* até a classe *Stack*.

Figura 5.9 – Hierarquia Collection (algumas classes e interfaces).



A classe *Stack* oferece um conjunto de serviços que permite realizar as operações inerentes ao funcionamento de uma Pilha. No Quadro 5.1 é apresentado o mapeamento das operações discutidas e respectivo serviço que o implementa.

Quadro 5.1 – Pilha - Mapeamento das Operações em Java (Stack)

Categoria	Operação	Método de Linkedlist
Básica	Criar	`Construtor`
	Verificar Vazia	`empty()`
	Tamanho	`size()`
Inserção		`push(<<elem>>)`
Acesso	Imprimir	`iterator()`
	Pesquisar	`contains(<<elem>>)`
	Ver Topo	`peek()`
Remoção		`pop()`

É importante destacar que os serviços disponibilizados pela classe *Vector* também estão disponíveis na classe *Stack*.

Por exemplo, o serviço *size* é herdado da classe Vector.

Algumas outras diferenças podem ser relacionadas:

- Para verificar se a pilha está vazia, além do serviço `empty`(*Stack*), pode ser utilizado o serviço `isEmpty`(*Vector*);
- Além do serviço `push`(*Stack*), é possível inserir elementos por meio dos serviços `add`(*Vector*) e `addElement`(*Vector*);
- O desempilhamento pode ser feito por meio do serviço `pop`(*Stack*) e `remove`(*Vector*).

!	O serviço `remove`(*index*), que recebe a posição em que deseja remover o elemento, ou `remove`(*object*) que indica o objeto que deseja remover, estão disponíveis. A execução de qualquer um desses serviços viola o comportamento LIFO.

No trecho de Código Java 5.11 é apresentado programa que faz uso da pilha "Stack". Na linha 6 (seis) está sendo instanciada a pilha para o tipo `String`[13]. Na linha 9 (nove) está sendo verificada se a pilha está vazia ou não.

Conforme citado anteriormente, inserção (empilhamento) de elementos pode ocorrer por meio de 3 (três) serviços: `push`, `add` e `addElement`.

Nas linhas de código entre 12 (doze) e 17 (dezessete) estão sendo realizadas inserções de elementos na Pilha; todos os serviços seguem a regra que determina o empilhamento em uma das extremidades (topo).

!	Por ser uma extensão da classe `Vector`, a classe `Stack` também permite inserção indicando a posição dentro da estrutura. Atenção, isso fere o comportamento LIFO da pilha. Muito cuidado!!

Na linha 20 (vinte) está sendo executado o serviço `size` implementado em Vector. O elemento que está posicionado no topo da pilha está sendo verificado na linha 23, por meio do serviço `peek`.

Nas linhas 26 (vinte e seis) e 27 (vinte e sete) está havendo desempilhamento (`pop`) enquanto ainda existem elementos armazenados na pilha (`empty`).

[13]http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html

Código Java 5.11 – Pilha com Stack

```
1  import java.util.Stack;
2
3  public class UsaPilhaJava {
4   public static void main(String[] args) {
5     // criando a pilha
6     Stack<String> pilha = new Stack<String>();
7
8     // verificando se a pilha está vazia
9     System.out.println("Vazia: " + pilha.empty());
10
11    // inserindo elementos (empilhando)
12    pilha.push("Primeiro");
13    pilha.push("Segundo");
14    pilha.add("Terceiro");
15    pilha.add("Quarto");
16    pilha.addElement("Quinto");
17    pilha.addElement("Sexto");
18
19    // verificando a quantidade de elementos
   contidos na pilha
20    System.out.println("Quantidade:"+ pilha.size());
21
22    // verificando o elemento que está no topo da
   pilha
23    System.out.println("Topo: " + pilha.peek());
24
25    // removendo elementos (desempilhando)
26    while (!pilha.empty())
27       System.out.println(pilha.pop());
28 }
29 }
```

O resultado da execução do código pode ser conferido na Figura 5.10: na linha 1 é exibido o resultado da verificação que indica se a pilha está vazia; a quantidade de elementos contidos na estrutura está sendo exibida na linha 2; o topo da pilha (`peek`) está sendo exibido na linha 3; nas linhas 4 até 9, os elementos foram removidos (desempilhamento – `pop`) e exibidos.

Figura 5.10 – Resultado da execução do Código Java 5.11 – Pilha com Stack

```
1 Vazia: true
2 Quantidade: 6
3 Topo: Sexto
4 Sexto
5 Quinto
6 Quarto
7 Terceiro
8 Segundo
9 Primeiro
```

5.5. Exercícios Propostos

1) O que é uma estrutura LIFO?

2) Como é a dinâmica de funcionamento de uma pilha?

3) É possível inserir um elemento em qualquer posição da pilha?

4) Faz sentido manter uma pilha com critério de ordenação para seus elementos? Por quê?

5) Faz sentido disponibilizar um serviço na pilha que permite remover apenas os elementos que estão localizados nas extremidades da pilha? Por quê?

6) Quando não é possível remover (pop) elementos na pilha?

7) Tem diferença a escolha da extremidade para implementar o topo da pilha nas implementações estática e dinâmica?

8) Desenvolva uma aplicação que faça uso das estruturas apresentadas neste capítulo.

6 Fila

A fila é uma estrutura linear na qual as operações de inserção são efetuadas apenas no final e as de retirada só podem ser realizadas no início da estrutura. Celes *et al.* (2004) explica que a estrutura de fila é uma analogia natural com o conceito de fila que usamos no dia a dia: quem primeiro entra numa fila é o primeiro a ser atendido ou a sair da fila.

Segundo Horowitz e Sahni (1987), pilhas e filas são casos especiais de uma lista.

Supondo uma estrutura formada pelos elementos E_1, E_2,... E_n, se desejarmos adicionar um novo elemento, ele será o elemento E_{n+1}. Se desejarmos retirar um elemento da lista, teremos que excluir o elemento E_1. A fila é comumente conhecida como estrutura "FIFO" (*first in, first out*).

Um exemplo dessa estrutura é uma fila de banco, em que os clientes que chegam ficam no final da fila (inserção) e os que são atendidos (e portanto deixam o banco) são os do início da fila (retirada por ordem de chegada).

São exemplos de uso de filas em programação: gerenciamento de processos em um sistema operacional (gerenciamento do uso da máquina pelos Jobs do usuário, por exemplo), implementação de uma fila de impressão (as requisições de impressão são colocadas em fila).

Tanto pilhas quanto filas têm ampla aplicação no mundo da computação. São estruturas simples e de grande poder para solucionar problemas computacionais (SILVA, 2007, p. 140).

6.1. Operações

A fila é outra forma particular de lista, ou seja, também apresenta características de restrição de acesso aos elementos. O objetivo dessa política de acesso definida é promover o seu padrão de comportamento FIFO. Na definição do seu comportamento também possui um conjunto de operações, classificadas em básica, inserção, acesso e remoção. Essas operações gerenciam o início e o final da fila, para inserir e remover elementos.

Diferente do que ocorre com a pilha, não existe um padrão de nomenclatura para suas operações, mas existe um padrão de comportamento que determina ser possível apenas inserir no final da fila e remover do início da fila. Dessa forma, é garantido que o último elemento a ser inserido será o último a ser removido, como também, o primeiro elemento a ser inserido será o primeiro a ser removido.

Uma vez especificado o comportamento particular (FIFO) da fila, o TAD é definido refletindo esse aspecto e escolhendo a estrutura de gerenciamento da memória, podendo ser: estático (vetor) ou dinâmico (encadeamento de elementos). A seguir, serão analisados alguns aspectos técnicos dessas duas formas de implementação.

6.2. Fila Estática

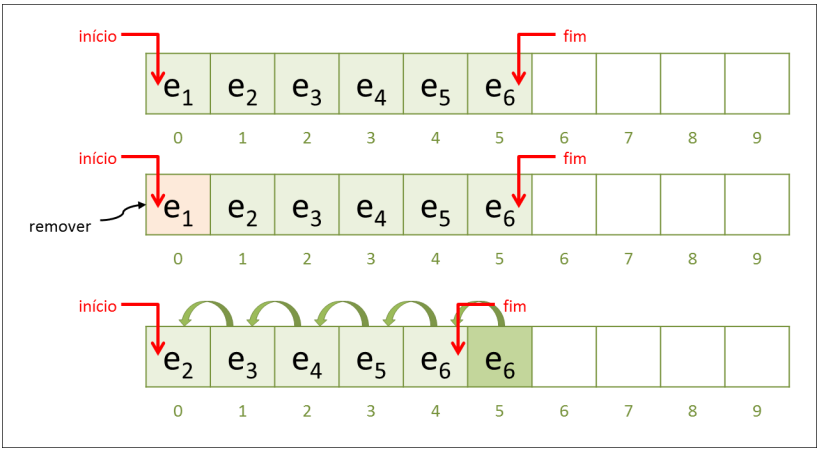
Conforme citado anteriormente, esse tipo de implementação utiliza como repositório de dados (gerenciamento da memória) o vetor, que é uma coleção estática. Embora algumas rotinas para manipulação da lista estática já tenham sido apresentadas

anteriormente, por ser a fila considerada uma forma particular de lista, essas rotinas merecem uma análise criteriosa.

A fila só permite inserir elementos no final – essa operação, por sua vez, já foi apresentada na lista estática (seção 4.2.3). Considerando a remoção, esta só poderá ocorrer no início da lista – essa operação também já foi apresentada na lista estática (seção 4.2.5).

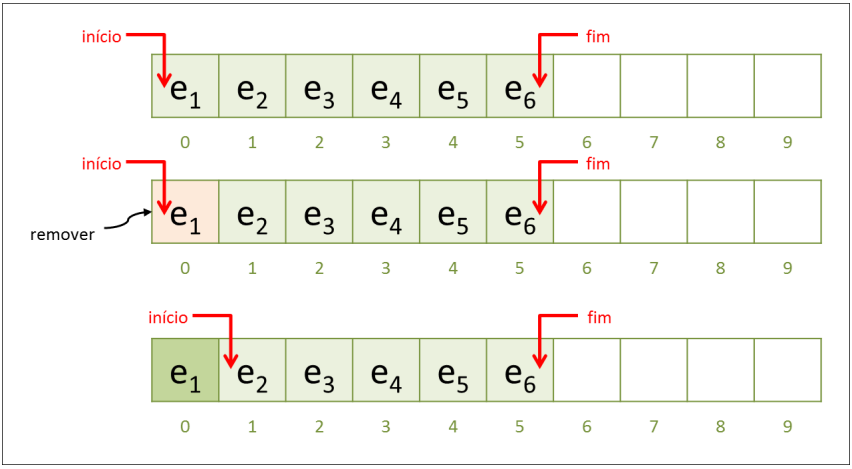
Analisando especificamente o processo de remoção dos elementos (início da fila), toda execução implica em deslocamento físico dos dados, o que pode ser um processo bastante oneroso, principalmente considerando o elevado número de inserções (quanto maior, pior). Por exemplo, na Figura 6.1 está representada uma fila com 6 (seis) elementos. Ao ser solicitada a remoção (só pode ser no início, pois trata-se de uma fila), todos os demais elementos que estavam localizados imediatamente à direita foram deslocados para o início. Quanto maior o número de elementos, pior é o desempenho dessa operação.

Figura 6.1 – Fila Estática - Removendo deslocando elementos



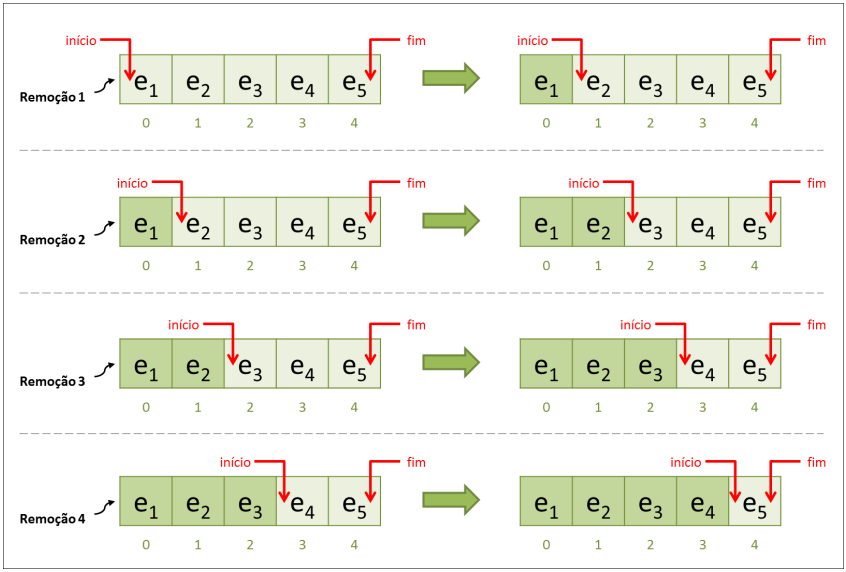
Esse problema não ocorre com a inserção, uma vez que esta é executada de forma direta, no final do vetor. O problema está na remoção dos elementos, que em termos de eficiência, pode representar um sério problema para o sistema.
Uma solução para o problema proposto é remover o elemento desejado e, para evitar o deslocamento físico dos dados, promover o deslocamento do início da fila. Dessa forma, a remoção ocorre com ônus muito inferior ao da solução anteriormente apresentada. Na Figura 6.2, a mesma fila de 6 (seis) elementos está representada e a remoção está ocorrendo com deslocamento do início da fila para posição imediatamente à direita.

Figura 6.2 – Fila Estática - Removendo deslocando início



O problema dessa solução é que sucessivas remoções vão promover o deslocamento da referência para o início da fila até que esta encontre a referência para o final da fila. Na Figura 6.3 está representado um caso extremo: inicialmente havia uma fila com 5 (cinco) elementos; foram promovidas 4 (quatro) remoções sucessivas, o que implicou no encontro da referência início com a referência fim da fila.

Figura 6.3 – Fila Estática - Removendo deslocando início - Problema das remoções sucessivas

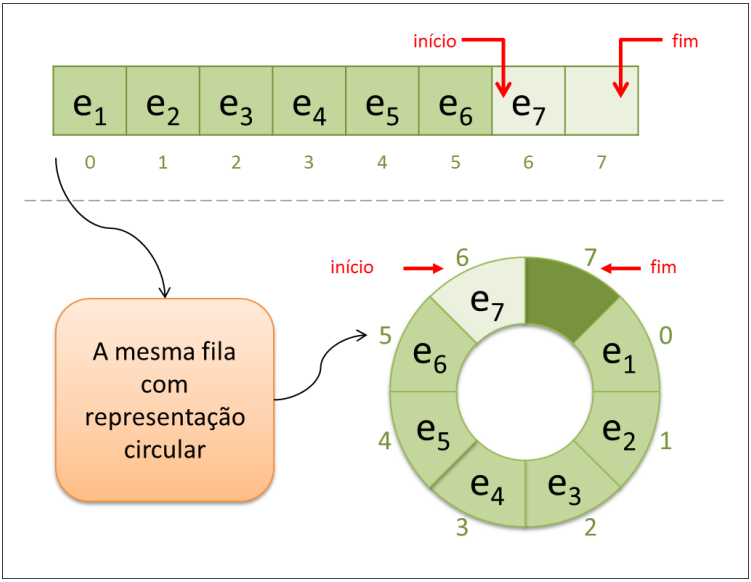


Supondo que seja necessário adicionar novos elementos nessa fila, a operação que faz a inserção vai perceber que a referência para o final está indicando que a estrutura está cheia, ou seja, não podem ser inseridos elementos. No entanto, percebe-se que a fila contém 4 (quatro) posições disponíveis (índices de 0 até 3).

Uma solução para esse problema é fazer o gerenciamento circular da estrutura, ou seja, ao percorrer o vetor, automaticamente volta-se ao início da fila. Na Figura 6.4 está representada uma fila de tamanho 8 (oito), contendo 7 (sete) elementos, com 6 (seis) posições vazias (índices de 0 até 5).

Ao solicitar a inserção de um elemento, esta será realizada no final na estrutura (índice 7) e a referência para o final da fila ao ser incrementada deve voltar ao início da fila. Não tem problema, de fato, o índice 0 (zero) está disponível (vazio).

Figura 6.4 – Fila Estática Circular



Essa solução apresentada resolve os dois problemas discutidos nesta seção, ou seja, não é necessário realizar o deslocamento físico dos dados (que é uma operação muito onerosa para o sistema) e o gerenciamento dos índices do vetor é realizado com eficiência (o vetor poderá trabalhar com a maior capacidade possível de armazenamento, sem desperdício de memória).

Por ser uma solução interessante e eficiente, este livro adotará, como forma de implementação para a fila estática, o gerenciamento circular do vetor.

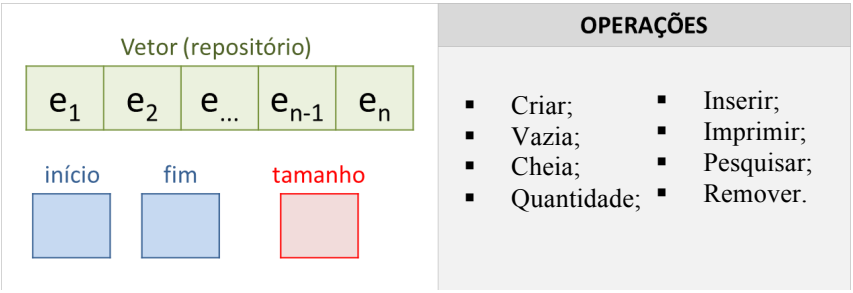
Uma vez definida a estratégia para gerenciamento da memória, é possível definir o TAD e seguir com a codificação.

6.2.1. Definição do TAD

Definido que será utilizado o vetor como repositório para os dados, semelhante às outras implementações estáticas deste livro, será adotada a alocação dinâmica de memória para esse vetor, o que implica concluir que a estrutura também deverá armazenar o seu tamanho.

Por se tratar de uma fila, é necessário manter uma referência para o início da coleção e outra referência para o final.

Figura 6.5 – Fila Estática - TAD



A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.1) e Java (ver Código Java 6.1) para o problema descrito.

Código C 6.1 – Fila Estática - TAD

Em C:

```
typedef struct{
   char nome[31];
   int idade;
}Elem;

typedef struct {
   Elem *elementos;
   int tamanho, inicio, fim;
}Fila;
```

Código Java 6.1 – Fila Estática - TAD

Em Java:

```
public class FilaEstatica<T>{
   private T[] elementos;
   private int inicio, fim, tamanho;
}
```

6.2.2. Operações Básicas

As operações básicas da fila devem analisar e gerenciar o vetor circular e as variáveis de controle (referência para o início, referência para fim e o tamanho da estrutura).

CRIAR

Por se tratar do gerenciamento circular do vetor, alguns detalhes merecem atenção: é necessário, por exemplo, definir o posicionamento das referências início e fim da estrutura no momento da sua criação. Uma sugestão para solução desse problema é adotar que início e fim começarão juntos na posição 0 (zero) do vetor.

As inserções sempre são executadas no final da estrutura (FIFO). Ao inserir um novo elemento, a referência fim deverá armazenar esse novo elemento e posteriormente deslocar-se para o próximo índice do vetor disponível. Dessa forma, fica determinado que a referência "fim" sempre estará indicando o índice disponível para acréscimo de elementos. A referência início estará indicando onde começa a lista, ou seja, onde serão executadas as remoções.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.2) e Java (ver Código Java 6.2) para o problema descrito.

Código C 6.2 – Fila Estática - Criar

Em C:

```
int Cria(Fila *fila, int tamanho){
   fila->elementos = calloc(tamanho, sizeof(Elem));
   if (fila->elementos == NULL) return FALSE;
   fila->inicio = fila->fim = 0;
   fila->tamanho = tamanho;
   return TRUE;
}
```

Código Java 6.2 – Fila Estática - Criar

Em Java:

```
public FilaEstatica(int tamanho){
   this.elementos = (T[]) new Object[tamanho];

   this.inicio = this.fim = 0;
   this.tamanho = tamanho;
}
```

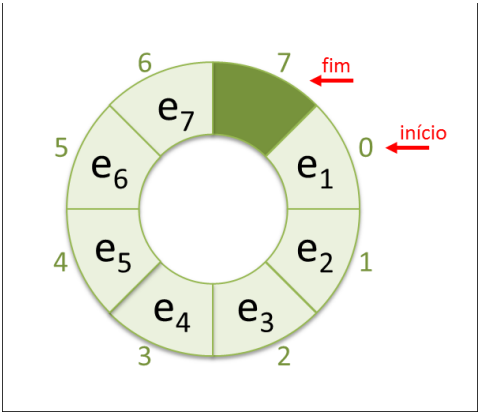
VERIFICAR VAZIA E CHEIA

Ao ser criada, a fila estática define que as referências início e fim começam no índice 0 (zero) do vetor. Sucessivas inserções e remoções deslocarão essas referências ao longo do vetor. Devido à implementação circular, ao incrementar uma referência (início ou fim) e esta chegar ao último índice do vetor, ela deverá voltar ao seu início (ou seja, índice 0).

A referência "`início`" indica o primeiro elemento da coleção e a referência "`fim`" indica a próxima posição disponível para inserir novos elementos, caso a estrutura não esteja cheia.

Com base nessas afirmações, considerando uma fila com capacidade para armazenar 8 (oito) elementos e supondo que foram inseridos 7 (sete) elementos, será encontrada a situação ilustrada pela Figura 6.6.

Figura 6.6 – Fila Estática Cheia



Ao se tentar adicionar um novo elemento, a referência "`fim`" será incrementada e encontrará a referência "`início`". Uma vez que "`fim`" indica a próxima posição disponível para acrescimento (ou seja, próxima posição vazia) e esta posição também é o início da estrutura, parecerá que a fila está vazia, o que não é verdade. Ao contrário, a fila está cheia e não pode receber novos elementos.

Entendendo esse cenário, o índice que antecede o início da fila não pode receber novos valores, sendo chamado de "**índice nulo**". Esse índice nulo não necessariamente é o 0 (zero), podendo ser qualquer índice, desde que anteceda o início. Para verificar se a fila está cheia, é necessário analisar essa condição.

Uma fila é considerada vazia quando as referências "`início`" e "`fim`" são iguais.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.3) e Java (ver Código Java 6.3) para o problema descrito.

Código C 6.3 – Fila Estática - Operações Cheia e Vazia

Em C:

```
// Verificar se está vazia
int Vazia(Fila fila){
   return (fila.inicio == fila.fim);
}

// Verificar se está cheia
int Cheia(Fila fila){
   return (((fila.fim + 1) % fila.tamanho) ==
fila.inicio);
}
```

Código Java 6.3 – Fila Estática - Operações Cheia e Vazia

Em Java:

```
// Verificar se está vazia
public boolean isVazia(){
   return this.inicio == this.fim;
}

// Verificar se está vazia
public boolean isCheia(){
   return ((this.fim + 1) % this.tamanho) ==
this.inicio;
}
```

VERIFICAR QUANTIDADE DE ELEMENTOS

A lista deverá ser percorrida a partir do início até encontrar o "índice nulo", ou seja, o índice que antecede a referência "`fim`" da lista.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.4) e Java (ver Código Java 6.4) para o problema descrito.

Código C 6.4 – Fila Estática - Operação para verificar a quantidade

Em C:

```
int Quantidade(Fila fila){
   int aux = fila.inicio, qtde = 0;
   while (aux != fila.fim){
      ++qtde;
      aux = (++aux)%fila.tamanho;
   }
   return qtde;
}
```

Código Java 6.4 – Fila Estática - Operação para verificar a quantidade

Em Java:

```
public int getQuantidade() {
   int aux = this.inicio, qtde = 0;
   while (aux != this.fim){
      ++qtde;
      this.fim = (++this.fim) % this.tamanho;
   }
   return qtde;
}
```

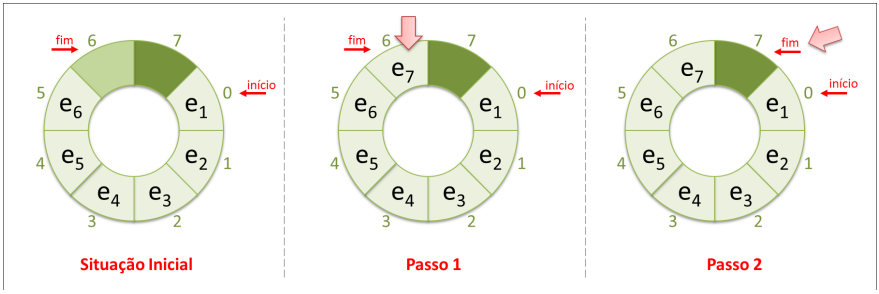
6.2.3. Inserindo Elementos

Novos elementos são inseridos no final da estrutura, ou seja, no índice indicado pela referência "`fim`". Apenas é possível inserir elementos caso a lista não esteja cheia, ou seja, o "índice nulo" não pode receber elementos.

Considerando que a fila não esteja cheia, a referência "`fim`" deverá receber o novo elemento. Posteriormente, seu valor será incrementado considerando a implementação circular, ou seja, caso esteja no último índice do vetor, este deverá ir para o início.

Por exemplo, na Figura 6.7 está representada uma fila de tamanho 8 (oito) contendo 6 (seis) elementos (e_1, e_2, e_3, e_4, e_5 e e_6), indicada pela "Situação inicial". Para inserir o elemento "e_7", "passo 1", este é armazenado na posição indicada pela referência "`fim`", que é o índice 6 (seis). Após inserir o elemento, a referência "`fim`" é incrementada, conforme pode ser visto no "Passo 2".

Figura 6.7 – Fila Estática Circular - Inserindo elemento



!	Com a inserção executada, a referência "`fim`" passa a indicar o "**índice nulo**", ou seja, não será possível acrescentar novos elementos, pois a fila está cheia.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.5) e Java (ver Código Java 6.5) para o problema descrito.

Código C 6.5 – Fila Estática - Operação para verificar a quantidade

Em C:

```
int Insere(Fila *fila, Elem elem){
   if (Cheia(*fila)) return FALSE;
   fila->elementos[fila->fim] = elem;
   fila->fim = (++fila->fim % fila->tamanho);
   return TRUE;
}
```

Código Java 6.5 – Fila Estática - Operação para verificar a quantidade

Em Java:

```
public void inserir(T elem) throws FilaCheiaException
{
   if (this.isCheia()) throw new FilaCheiaException();
   this.elementos[this.fim] = elem;
   this.fim = (++this.fim) % this.tamanho;
}
```

6.2.4. Acessando Elementos

O acesso aos elementos armazenados na estrutura para impressão e/ou busca, semelhante ao que ocorre na lista estática, também começa do início e termina no final. O único detalhe neste caso é o gerenciamento circular do vetor, o que exige bastante atenção do programador.

Na manipulação da fila, em determinados momentos, é necessário verificar os elementos que estão armazenados nas suas extremidades, ou seja, início e fim. Para isso, são oferecidas operações para verificação desses valores.

É importante destacar que a referência "`início`" indica o primeiro elemento da coleção, enquanto que a referência "`fim`" indica o próximo índice após o último elemento da estrutura. Isso implica dizer que para verificar o último elemento, a referência "`fim`" deverá ser decrementada antes de recuperar o valor que está armazenado.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.6) e Java (ver Código Java 6.6) para os problemas descritos.

Código C 6.6 – Fila Estática - Operação para verificar a quantidade

Em C:

```
// Buscar um elemento na Fila
int Buscar(Fila fila, char *nome){
   int i = fila.inicio;
   while (i != fila.fim){
      if (stricmp(fila.elementos[i].nome, nome) == 0)
         return i;
      i = (++i)%fila.tamanho;
   }
   return -1;
}

// Imprimir a Fila
void Imprimir(Fila fila){
   int i = fila.inicio;
   while (i != fila.fim){
      printf("Nome: %s - ", fila.elementos[i].nome);
      printf("Idade: %d\n", fila.elementos[i].idade);
      i = (++i)%fila.tamanho;
   }
}
```

```
// Ver o início da Fila
int VerInicio(Fila fila, Elem *elem){
   if (Vazia(fila)) return FALSE;
   *elem = fila.elementos[fila.inicio];
   return TRUE;
}

// Ver o final da Fila
int VerFim(Fila fila, Elem *elem){
   int aux;
   if (Vazia(fila)) return FALSE;
   if ((aux = fila.fim - 1) < 0) aux += fila.tamanho;
   *elem = fila.elementos[aux];
   return TRUE;
}
```

#	É um erro comum de programação decrementar o valor da referência "fim" sem considerar que esse valor pode ser negativo. Supondo que a referência "fim" está indicando o índice 0 (zero), ao decrementá-la, ela deverá ir para o final do vetor e não assumir valor negativo.

Código Java 6.6 – Fila Estática - Operação para verificar a quantidade

Em Java:

```
// Buscar um elemento na Fila
public int get(T elem) {
   int aux = this.inicio;
   while (aux != this.fim){
      if (this.elementos[aux].equals(elem)) return
aux;
      aux = ++aux % this.tamanho;
   }
   return -1;
}
```

```
// Obter coleção para posterior impressão da Fila
public Iterator<T> get() {
   @SuppressWarnings("unchecked")
   T[] temp = (T[]) new Object[this.getQuantidade()];
   int aux = this.inicio, i = 0;
   while (aux != this.fim){
      temp[i++] = this.elementos[aux];
      aux = ++aux % this.tamanho;
   }
   return Arrays.asList(temp).iterator();
}

// Ver o início da Fila
public T getInicio() throws FilaVaziaException{
   if (this.isVazia()) throw new FilaVaziaException();
   return this.elementos[this.inicio];
}

// Ver o final da Fila
public T getFim() throws FilaVaziaException{
   int aux;
   if (this.isVazia()) throw new FilaVaziaException();
   if ((aux = this.fim - 1) < 0) aux += this.tamanho;
   return this.elementos[aux];
}
```

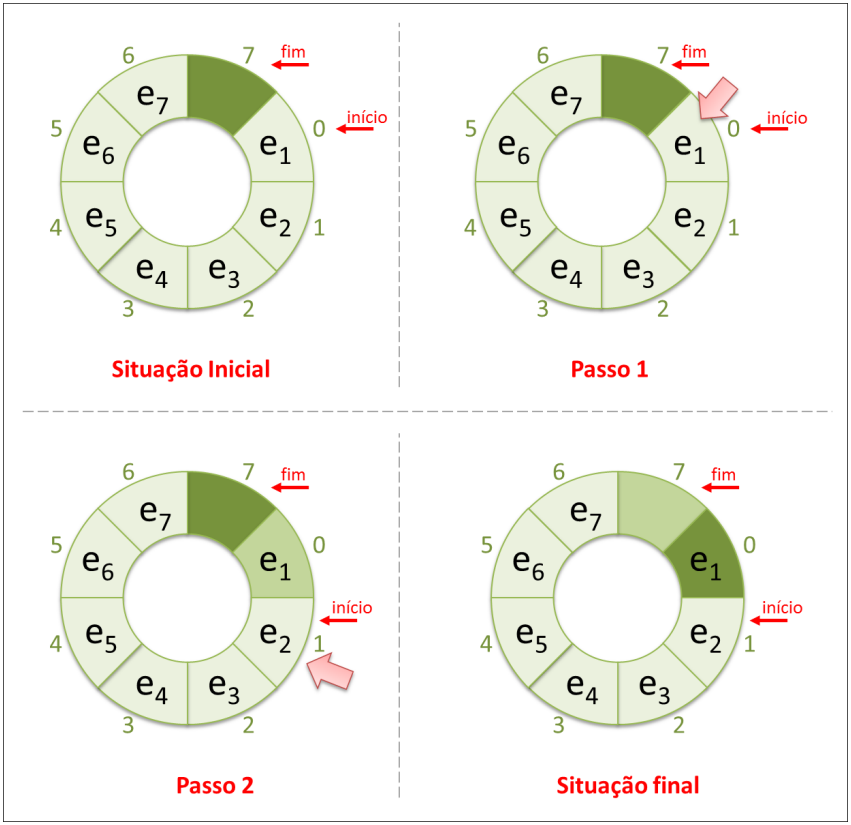
6.2.5. Removendo Elementos

A fila só permite remoção no início da estrutura (FIFO). Para haver a remoção antes, deve ser verificado se existe o elemento que se deseja remover. Considerando que a fila não está vazia, o elemento que está posicionado na referência "`início`" deve ser retornado e posteriormente o valor da referência deve ser incrementado, considerando a implementação circular do vetor.

Na Figura 6.8 está representada uma fila de tamanho 8 (oito), contendo 7 (sete) elementos (e_1, e_2, e_3, e_4, e_5, e_6 e e_7), na "Situação inicial" (fila cheia). Para haver a remoção, o elemento indicado

pela referência "`início`" deverá ser retornado (Passo 1). Posteriormente, essa referência deverá ser incrementada (Passo 2).

Figura 6.8 – Fila Estática - Remoção



Com a remoção de um elemento, o "**índice nulo**" que na situação inicial era o índice 7 (sete), agora passou a ser o índice 0 (zero), uma vez que este é o novo índice que antecede o início da fila.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 6.7) e Java (ver Código Java 6.7) para o problema descrito.

Código C 6.7 – Fila Estática - Operação Remover

Em C:

```
int Remove(Fila *fila, Elem *elem){
   if (Vazia(*fila)) return FALSE;
   *elem = fila->elementos[fila->inicio];
   fila->inicio = ++fila->inicio % fila->tamanho;
   return TRUE;
}
```

Código Java 6.7 – Fila Estática - Operação Remover

Em Java:

```
public void inserir(T elem) throws FilaCheiaException
{
   if (this.isCheia()) throw new FilaCheiaException();
   this.elementos[this.fim] = elem;
   this.fim = (++this.fim) % this.tamanho;
}
```

6.3. Fila Encadeada

A forma de implementação encadeada possui como base o encadeamento de "nó", cada elemento que está sendo manipulado está contido em um nó. Por manter o encadeamento de "nó", não precisa fazer o deslocamento físico dos dados nas operações de inserção e/ou remoção, que é uma grande vantagem, comparando com a implementação estática.

Considerando aspectos de desempenho, a fila encadeada não possui problemas com inserções e remoções sucessivas, ou no aumento do número de elementos.

Conforme citado anteriormente, a fila é uma lista com restrições de acesso, seguem o padrão FIFO, inserções realizadas no final e remoções no início da estrutura. Implica afirmar que essas operações apenas atualizam as referências na extremidade da estrutura.

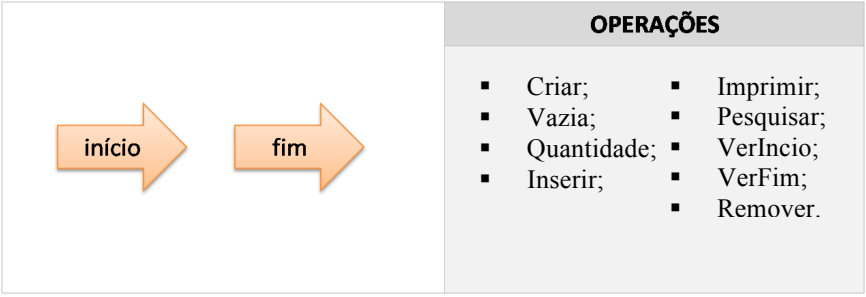
O encadeamento pode ser simples ou duplo, depende da necessidade da aplicação. Em seções anteriores, foram apresentadas, respectivamente, a lista simplesmente encadeada e a lista duplamente encadeada.

6.3.1. Definição do TAD

Independente da forma de encadeamento (simples ou duplo), a fila mantém uma referência para o início e outra para o final da estrutura. Dessa forma é possível garantir que as operações de inserção e remoção possuem o menor custo para a estrutura.

Na Figura 6.9 é possível observar o TAD para a fila encadeada.

Figura 6.9: Fila encadeada - Definição do TAD



A seguir, são apresentadas soluções nas linguagens C e Java para o problema descrito.

Código C 6.8 – Fila Estática – Operação Remover

Em C:

```
typedef struct{
   char nome[31];
   int idade;
}Elem;

typedef struct no{
   Elem elemento;
   struct no *proximo;
}No;

typedef struct{
   No *inicio, *fim;
}Fila;
```

Código Java 6.8 – Fila Estática – Operação Remover

Em Java:

```
public class No <T>{
   private T elemento;
   private No<T> proximo;
}

public class FilaEncadeada <T>{
   private No<T> inicio, fim;
}
```

6.3.2. Operações Básicas

A implementação das quatro operações básicas (Criar, Vazia, Cheia e Tamanho) pode ser entendida como uma simplificação da implementação adotada na lista duplamente encadeada, uma vez que esta estrutura não necessariamente precisa manter duas referências (anterior e próximo) em cada nó.

A seguir, são apresentadas soluções nas linguagens C e Java para os problemas descritos.

Código C 6.9 – Fila Encadeada - Operações Básicas

Em C:

```
// cria a fila
void Cria(Fila *fila){
   fila->inicio = fila->fim = NULL;
}

// verifica se a fila estah vazia
int Vazia(Fila fila){
   return (fila.inicio == NULL);
}

// verifica se a fila estah cheia
int Cheia(Fila fila){
   return FALSE;
}

// retorna a quantidade de elementos armazenados na
fila
int Quantidade(Fila fila){
   int qtde = 0;

   No *aux = fila.inicio;

   while (aux != NULL){
      ++qtde;
      aux = aux->proximo;
   }
   return qtde;
}
```

Código Java 6.9 – Fila Encadeada - Operações Básicas

Em Java:

```
// construção da fila
public FilaEncadeada(){
   this.inicio = this.fim = null;
}

// verifica se a fila estah vazia
public boolean isVazia() {
   return this.inicio == null;
}

// retorna a quantidade de elementos da fila
public int getQuantidade() {
   No<T> atual = this.inicio;
   int qtde = 0;

   while (atual != null){
      ++qtde;
      atual = atual.getProximo();
   }

   return qtde;
}
```

6.3.3. Inserindo Elementos

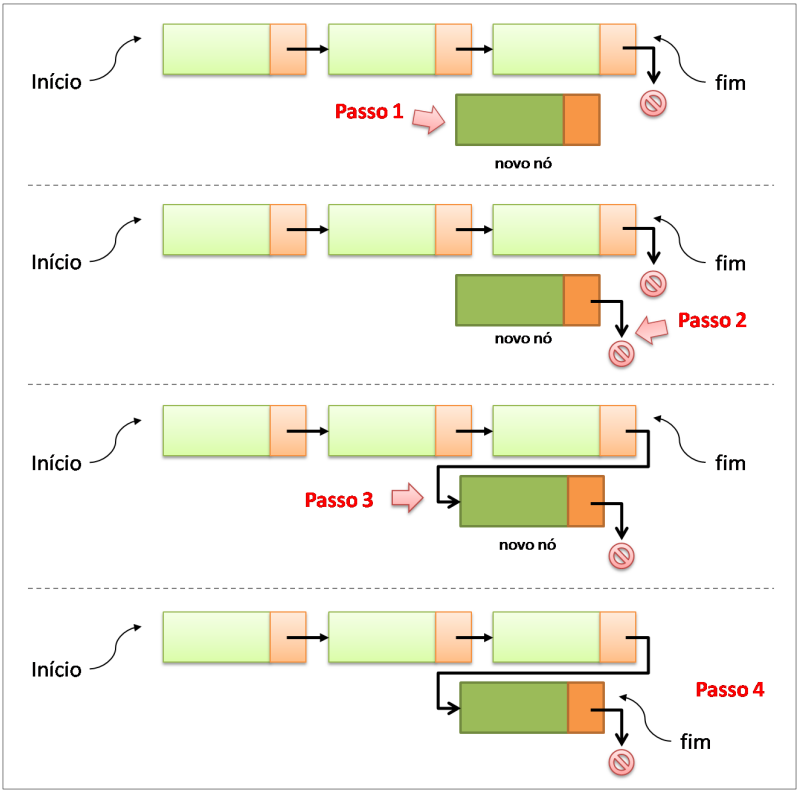
Por seguir o padrão FIFO, a inserção de elementos é realizada no final da estrutura. Diferente do que ocorre na lista simplesmente encadeada, não é necessário percorrer toda a estrutura para encontrar o último nó, uma vez que a fila mantém uma referência para este último elemento.

Na Figura 6.10 está representada uma lista contendo três elementos. A referência "inicio" está indicando o começo da fila, enquanto que a referência "fim" está indicando onde a fila

termina. Para inserir novo elemento, é necessário seguir os 04 (quatro) passos:

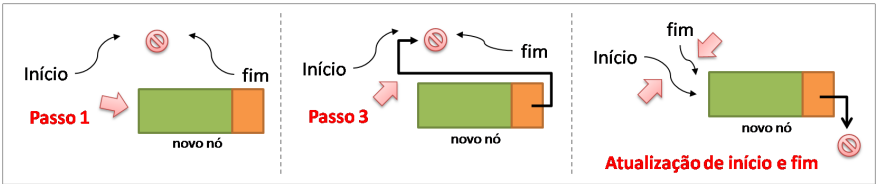
- Passo 1: Um novo nó é criado e adicionado o elemento a ser inserido na fila;
- Passo 2: O novo nó terá como próximo elemento o final da fila (`null`);
- Passo 3: O nó referenciado pelo fim terá como próximo elemento o novo nó criado;
- Passo 4: A referência fim é atualizada para indicar o novo nó que foi inserido.

Figura 6.10 – Fila Encadeada – Inserindo Elemento



Caso a fila esteja vazia, na operação para inserir um novo elemento é suficiente executar os passos 1 e 3, após execução as referências "`início`" e "`fim`" deverão apontar para esse novo elemento criado, conforme pode ser observado na Figura 6.11.

Figura 6.11 – Fila Encadeada - Inserindo na fila vazia



A seguir, são apresentadas soluções nas linguagens C e Java para o problema descrito.

Código C 6.10 – Fila Encadeada – Inserindo Elemento

Em C:

```
int Insere(Fila *fila, Elem elem){
   No *novo;

   if (Cheia(*fila)) return FALSE;

   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;

   novo->elemento = elem;
   novo->proximo = NULL;

   if (Vazia(*fila)) fila->inicio = novo;

   else fila->fim->proximo = novo;
   fila->fim = novo;

   return TRUE;
}
```

Código Java 6.10 – Fila Encadeada – Inserindo Elemento

Em Java:

```
public void inserir(T elem) {
   No<T> novo = new No<T>(elem);
   if (this.isVazia()) this.inicio = novo;
   else this.fim.setProximo(novo);
   this.fim = novo;
}
```

6.3.4. Acessando Elementos

O acesso (buscar e imprimir) aos elementos segue o mesmo padrão adotado na implementação das listas estática e encadeada. Os elementos são percorridos a partir do início até o final da estrutura.

Semelhante ao apresentado na fila estática, também é necessário acrescentar duas operações para visualizar os elementos que estão posicionados no início no final da estrutura. A seguir, são apresentadas soluções nas linguagens C e Java para os problemas descritos.

Código C 6.11 – Fila Estática – Operação para verificar a quantidade

Em C:

```
// Buscar um elemento na Fila
int Buscar(Fila fila, char *nome){
   No *aux = fila.inicio;
   while (aux != NULL){
      if (stricmp(aux->elemento.nome, nome) == 0)
         return TRUE;
      aux = aux->proximo;
   }
   return FALSE;
}
```

```
// Imprimir a Fila
void Imprime(Fila fila){
   No *aux = fila.inicio;

   while (aux != NULL){
      printf("Nome: %s - ", aux->elemento.nome);
      printf("Idade: %d\n", aux->elemento.idade);
      aux = aux->proximo;
   }
}


// Ver o início da Fila
int VerInicio(Fila fila, Elem *elem){
   if (Vazia(fila)) return FALSE;

   *elem = fila.inicio->elemento;
   return TRUE;
}


// Ver o final da Fila
int VerFim(Fila fila, Elem *elem){
   if (Vazia(fila)) return FALSE;

   *elem = fila.fim->elemento;
   return TRUE;
}
```

Código Java 6.11 – Fila Estática – Operação para verificar a quantidade

Em Java:

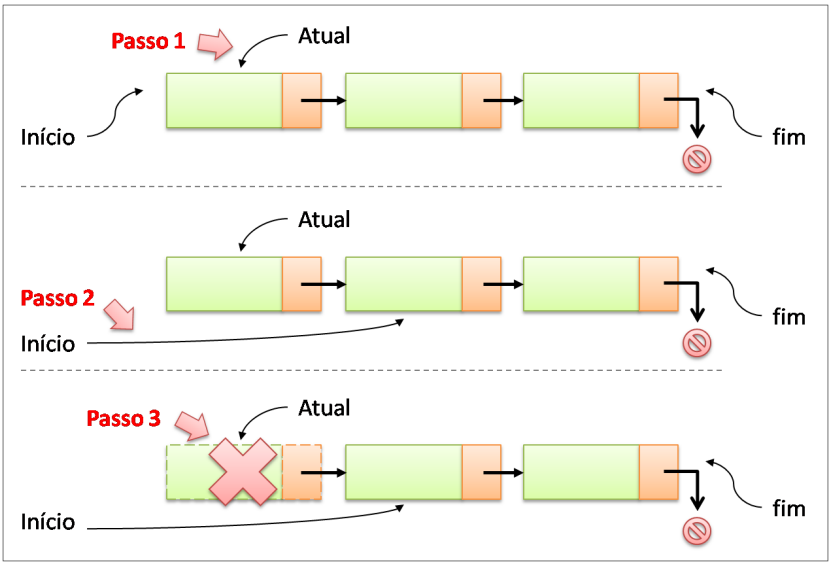
```
// Buscar um elemento na Fila
public int get(T elem) {
   int i = 0;
   No<T> atual = this.inicio;
   while (atual != null){
      if (atual.getElemento().equals(elem)) return i;
      atual = atual.getProximo();
      i++;
   }
   throw new ElementoNaoExisteException();
}

// Obter coleção para posterior impressão da Fila
public Iterator<T> get() {
   int i = 0;
   @SuppressWarnings("unchecked")
   T[] vetor = (T[]) new Object[this.getQuantidade()];
   No<T> atual = this.inicio;
   while(atual != null) {
      vetor[i++] = atual.getElemento();
      atual = atual.getProximo();
   }
      return Arrays.asList(vetor).iterator();
}

// Ver o início da Fila
public T getInicio() throws FilaVaziaException{
   if (this.isVazia()) throw new FilaVaziaException();
   return this.inicio.getElemento();
}

// Ver o final da Fila
public T getFim() throws FilaVaziaException{
   if (this.isVazia()) throw new FilaVaziaException();
   return this.fim.getElemento();
}
```

6.3.5. Removendo Elementos

A remoção de elementos ocorre apenas no início da estrutura, ou seja, será removido o primeiro nó da fila. Na Figura 6.12 está representada uma fila encadeada contendo 03 (três) elementos, para realizar a remoção é necessário executar os três passos, a seguir:

- Passo 1: Uma referência "`atual`" aponta para o primeiro nó da estrutura;
- Passo2: A referência "`início`" deverá apontar para o próximo do "`atual`";
- Passo 3: Uma vez atualizada a referência é liberada a memória apontada por "`atual`".

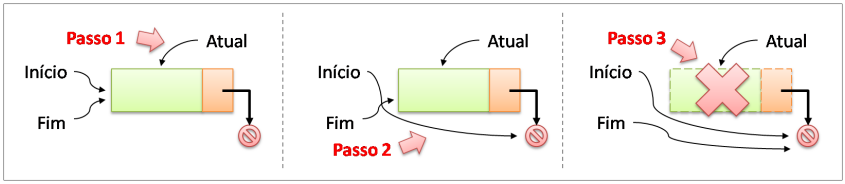
Figura 6.12 – Fila Encadeada - Removendo Elemento



Caso a fila seja unitária (contenha apenas um único elemento), é necessário ficar atento para indicar que a referência "`fim`"

também deve ser alterada para indicar que a fila passa a estar vazia (as duas referências são nulas). Na Figura 6.13 é possível visualizar graficamente a solução para esse problema.

Figura 6.13 – Fila Encadeada - Removendo Elemento na Fila Unitária



A fila pode ser codificada de diversas outras formas, por exemplo: duplamente encadeada ou circular. Fica a cargo de o programador decidir o que é necessário, de acordo com os requisitos da aplicação.
Uma vez conhecida a estrutura de dados fila e formas de implementação, na próxima seção está sendo analisado o que a Linguagem Java trás implementado para esse problema.

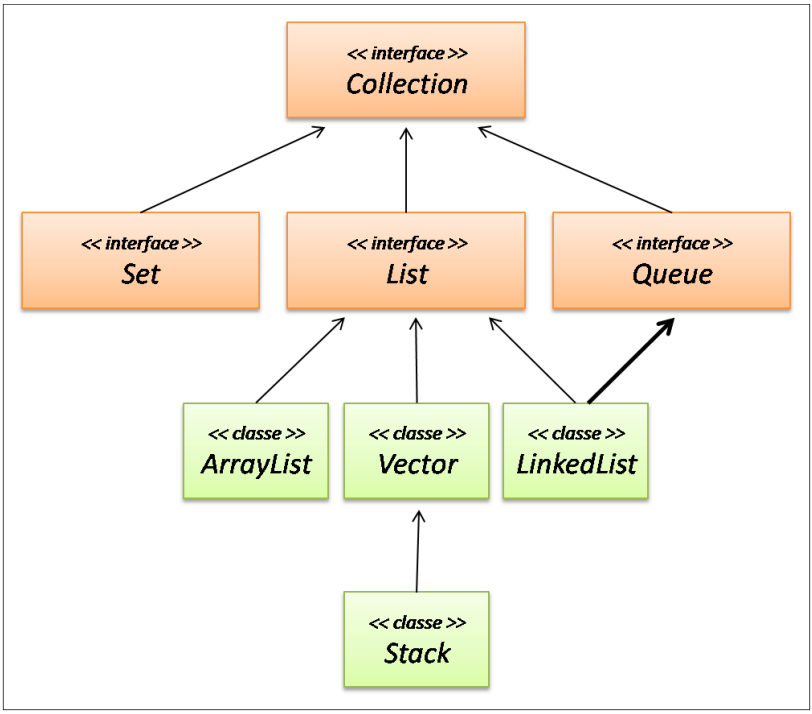
6.4. API Java

Semelhante ao que ocorre com a estrutura de dados pilha (*Stack*), o *framework Collections* traz uma definição para a fila, é a interface "*Queue*"[14] (`java.util.Queue<E>`) e a classe "*LinkedList*" implementa essa especificação.

Conforme pode ser observado na Figura 6.14.

[14] http://docs.oracle.com/javase/6/docs/api/java/util/Queue.html

Figura 6.14 – Hierarquia Collection (interface Queue)



A interface "*Queue*" define o comportamento FIFO e oferece um conjunto de serviços inerentes ao funcionamento de uma fila. É importante destacar que é uma interface, ou seja, não é uma implementação e sim uma definição de comportamento.

Conforme dito anteriormente, a classe "*LinkedList*" além dos serviços definidos na interface "*List*", também implementa os serviços definidos na interface "*Queue*".

No Quadro 6.1, é apresentado mapeamento das operações discutidas nessa seção e respectivo serviço que o implementa.

Quadro 6.1 – Fila - Mapeamento das Operações em Java (Queue)

Categoria	Operação	Método de Linkedlist
Básica	Criar	`Construtor`
	Verificar Vazia	`isEmpty()`
	Tamanho	`size()`
Inserção		`add(<<elem>>)` `offer(<<elem>>)`
Acesso	Imprimir	`iterator()`
	Pesquisar	`contains(<<elem>>)`
	Ver Inicio	`peek()`
Remoção		`poll()` `remove()`

É importante destacar que se uma referência "`Queue`" está gerenciando um "`LinkedList`", está garantida a segurança que apenas estarão disponíveis operações que seguem o padrão FIFO de comportamento.

No trecho de Código Java 6.12 é apresentado um programa que faz uso da fila "Queue", através da classe "LinkedList". Na linha 8 a fila está sendo instanciada.

Na linha 11 está sendo verificado se a fila está vazia (`isEmpty`), o que é verdade, a fila acabou de ser criada, consequentemente não possui elementos.

Os serviços para inserir (`add` e `offer`) estão sendo executados entre as linhas 15 e 20. A lista agora contém os elementos: Primeiro, Segundo, Terceiro, Quarto, Quinto e Sexto (nesta ordem). Ao verificar a quantidade de elementos contidos na fila (linha 23) está sendo retornado o valor 6 (seis).

O serviço "peek", linha 26, permite verificar o elemento que está posicionado no início (cabeça) da fila, que é o "Primeiro". Para remoção estão disponíveis os serviços "poll" e "remove", executados respectivamente nas linhas 29 e 30, removendo os elementos "Primeiro" e "Segundo". A fila passou a ter 4 (quatro) elementos.

Na linha 33 está sendo verificado se existe o elemento "Quarto" na fila, através do serviço "contains", o que retorna verdadeiro. O percurso da fila está sendo realizado através do "Iterator"[15], linhas 37 até 39.

Código Java 6.12 – Pilha com Queue e LinkedList

```
1  import java.util.Iterator;
2  import java.util.LinkedList;
3  import java.util.Queue;
4
5  public class UsaFilaJava {
6   public static void main(String[] args) {
7     // criar a lista
8     Queue<String> fila = new LinkedList<String>();
9
10    // verificar se a fila está vazia
11    if (fila.isEmpty()) System.out.println("Fila
    Vazia");
12    else System.out.println("Fila não vazia");
13
14    // adicionar elementos
15    fila.offer("Primeiro");
16    fila.offer("Segundo");
17    fila.offer("Terceiro");
18    fila.add("Quarto");
19    fila.add("Quinto");
20    fila.add("Sexto");
```

[15] http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html

```
21
22    // verificar quantidade
23    System.out.println("Quantidade: " +
    fila.size());
24
25    // verificar elemento que está no início da fila
26    System.out.println("Início: " + fila.peek());
27
28    // Remover elemento
29    System.out.println("Removido: " + fila.poll());
30    System.out.println("Removido: " +
    fila.remove());
31
32    // Procurar um determinado elemento
33    System.out.println("'Quarto': " +
    fila.contains("Quarto"));
34
35    // Percorrer a fila para imprimir
36    System.out.print("Elementos na fila: ");
37    Iterator<String> it = fila.iterator();
38    while (it.hasNext())
39       System.out.printf("%s ", it.next());
40  }
41 }
```

6.5. Exercícios Propostos

1) O que é uma estrutura FIFO?

2) Como é a dinâmica de funcionamento de uma fila?

3) Quais são as vantagens da implementação circular para a fila?

4) O que é o "**índice nulo**"? É fixo no vetor?

5) É possível inserir um elemento em qualquer posição da fila? Por quê?

6) Faz sentido manter uma fila com critério de ordenação para seus elementos? Por quê?

7) Tem diferença a escolha da extremidade para implementar o início e o fim das filas nas implementações estática e dinâmica? Por quê?

8) Desenvolva uma aplicação que faça uso das estruturas apresentadas neste capítulo.

7 Deque

O deque é uma estrutura linear em que as operações de inserção e retirada são efetuadas no início e/ou no final da estrutura.

Supondo uma estrutura formada pelos elementos E_1, E_2,... E_n, se desejarmos adicionar um novo elemento, o deque passará a ter n+1 elementos. Desse modo, o elemento inserido poderá passar a ser o elemento E_1 – caso seja inserido no início do deque, e assim todos os demais elementos serão reorganizados (passando a ocupar as próximas posições na estrutura) – ou será o elemento E_{n+1} – caso seja inserido no final do deque. Considerando o deque de n+1 elementos, se desejarmos retirar um elemento, teremos que excluir ou o elemento E_1 ou o elemento E_{n+1}.

Um exemplo dessa estrutura é a composição de um trem, onde as inserções e retiradas de vagões são feitas nas extremidades.

Um exemplo de uso de deque em programação é o jogo de dominó. No jogo de dominó, a representação das peças lançadas no tabuleiro pode ser gerenciada por meio de um deque, sendo possível agregar novas peças apenas nas suas duas extremidades.

7.1. Operações

O deque é outra forma particular de lista que possui restrição quanto ao acesso aos elementos. O objetivo dessa estrutura é fornecer a infraestrutura de software que permite realizar operações de inserção e remoção em qualquer das extremidades. Nesse sentido, no deque não é permitido acessar os elementos que

não estejam localizados nas extremidades da estrutura, denominadas "esquerda" (início) e "direita" (fim). Essa estrutura também é conhecida como uma "fila de cabeça dupla".

Na definição do seu comportamento, também possui um conjunto de operações, classificadas em básica, inserção, acesso e remoção. Essas operações gerenciam a esquerda e a direita do deque, para inserir e remover elementos.

Semelhante ao que ocorre com a fila, não existe um padrão para nomenclatura das operações, apenas existe a restrição de comportamento que define inserções e remoções em quaisquer das extremidades.

Uma vez entendido o comportamento de um deque, o TAD é definido refletindo esse aspecto e escolhendo a estrutura de gerenciamento da memória, podendo ser: estático (vetor) ou dinâmico (encadeamento de elementos). A seguir, serão analisados alguns aspectos técnicos dessas duas formas de implementação.

7.2. Deque Estático

Essa forma de gerenciamento da memória utiliza vetor como repositório dos dados. Conforme discutido na seção 6.2 (Fila Estática), o vetor apresenta sérios problemas de desempenho para executar sucessivas operações de inserção e remoção, e quanto maior o número de elementos contidos na estrutura, mais grave o problema fica. A solução encontrada para a fila foi utilizar a implementação circular do vetor, o que também se aplica ao caso do deque, ou seja, este também fará uso de um vetor circular para armazenamento dos dados.

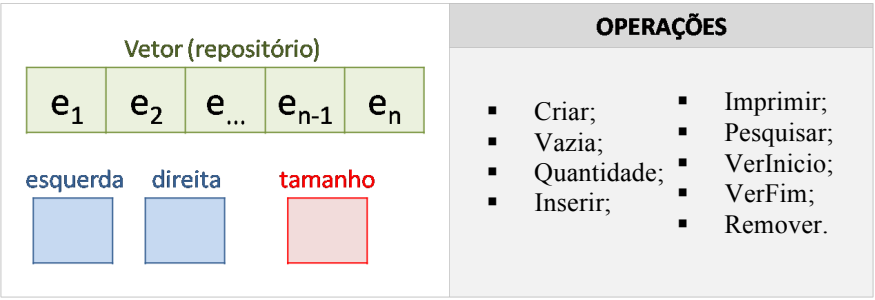
Uma vez conhecida a estratégia para gerenciamento da memória é possível definir o TAD e seguir com a codificação.

7.2.1. Definição do TAD

Definido que será utilizado o vetor como repositório para os dados, semelhante às outras implementações estáticas deste livro, será adotada a alocação dinâmica de memória, o que implica dizer que a estrutura deverá armazenar o seu tamanho. Por se tratar de um deque, é necessário manter uma referência para o início da coleção (esquerda) e outra referência para o final (direita).

Na Figura 7.1 está representado o TAD do deque, contendo o vetor, as referências para as extremidades e uma variável para manter o tamanho (capacidade de armazenamento) da estrutura.

Figura 7.1 – Deque Estático - TAD



A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.1) e Java (ver Código Java 7.1) para o problema descrito.

Código C 7.1 – Deque Estático - TAD

Em C:

```
typedef struct{
   char nome[31];
   int idade;
}Elem;
```

```
typedef struct {
   Elem *elementos;
   int tamanho, esquerda, direita;
}Deque;
```

Código Java 7.1 – Deque Estático - TAD

Em Java:

```
public class DequeEstatico <T>{
   private T[] elementos;
   private int esquerda, direita;
}
```

7.2.2. Operações Básicas

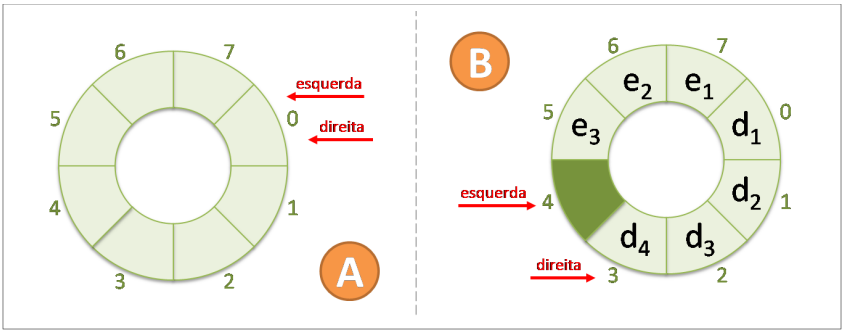
As operações básicas do deque devem analisar e gerenciar o vetor circular e as variáveis de controle (referência para extremidades esquerda, referência para a extremidade direita e o tamanho da estrutura).

Seguindo o mesmo padrão de solução adotado para a fila (vetor circular), ao criar um deque, as referências esquerda e direita devem indicar a posição 0 (zero). A referência "`direita`" indica o elemento que está posicionado na extremidade mais à direita, enquanto a referência "`esquerda`" indica qual é a próxima posição disponível para inserir novos elementos à esquerda. Lembramos que a inserção pode ocorrer nas duas extremidades.

Sempre que as referências "`direita`" e "`esquerda`" indicarem a mesma posição do vetor, poderemos dizer que o deque está vazio. É possível verificar se o deque está cheio por meio da análise das referências "`direita`" e "`esquerda`".

Por exemplo, consideramos um deque de tamanho 8 (oito), com a inserção de 3 (três) elementos à esquerda (e_1, e_2 e e_3) e 4 (quatro) elementos à direita (d_1, d_2, d_3 e d_4), representados na Figura 7.2. No lado "A", o deque está vazio; no Lado "B", o deque reflete as inserções executadas.

Figura 7.2 – Deque Estático – Operações Básicas



Se ao incrementarmos a referência "`direita`" chegarmos ao índice indicado pela referência "`esquerda`" ou se decrementarmos a referência "`esquerda`" e encontrarmos o índice indicado pela referência "`direita`", o deque estará cheio, que é o cenário exemplificado na figura.

Nesse caso, o "**índice nulo**" (não permite armazenar elemento) é o 4 (quatro).

A seguir, são apresentadas soluções nas linguagens C (ver Código Java 7.2) e Java (ver Código Java 7.2) para os problemas descritos.

Código C 7.2 – Deque Estático - Operações Básicas

Em C:

```
// Criar a estrutura
int Criar(Deque *deque, int tamanho){
   deque->elementos = calloc(tamanho, sizeof(Elem));
   if (deque->elementos == NULL) return FALSE;
   deque->esquerda = deque->direita = 0;
   deque->tamanho = tamanho;
   return TRUE;
}
```

```
// Verificar se está vazio
int Vazia(Deque deque){
   return (deque.esquerda == deque.direita);
}

// Verificar se está cheio
int Cheia(Deque deque){
   return (((deque.direita + 1) % deque.tamanho) ==
deque.esquerda);
}

// Verificar a quantidade de elementos
int Quantidade(Deque deque){
   int qtde = 0, aux = deque.esquerda;
   while (aux != deque.direita){
      ++qtde;
      aux = ++aux%deque.tamanho;
   }
   return qtde;
}
```

Código Java 7.2 – Deque Estático - Operações Básicas

Em Java:

```
// Criar a estrutura
public DequeEstatico(int tamanho) {
   this.elementos = (T[]) new Object[tamanho];
   this.esquerda = this.direita = 0;
}

// Verificar se está vazia
public boolean isVazia(){
   return this.esquerda == this.direita;
}

// Verificar se está cheia
public boolean isCheia(){
    return (((this.direita + 1) %
this.elementos.length) == this.esquerda);
}
```

```
// Verificar a quantidade de elementos
public int getQuantidade(){
   int qtde = 0, aux = this.esquerda;
   while (aux != this.direita){
      ++qtde;
      aux = ++aux % this.elementos.length;
   }
   return qtde;
}
```

7.2.3. Inserindo Elementos

Os elementos podem ser inseridos tanto na extremidade esquerda quanto na extremidade direita, devendo-se ter cuidado para não adicionar elementos no "**índice nulo**".

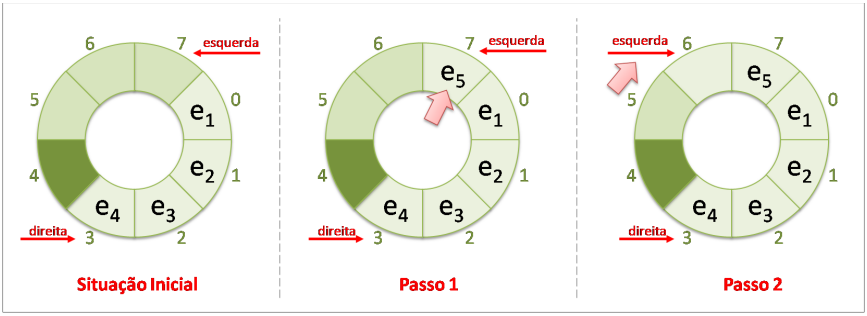
Caso a inserção seja realizada na extremidade esquerda, o elemento é adicionado no índice indicado pela referência "`esquerda`" e o valor da referência é decrementando observando os aspectos da implementação circular do vetor.

Por exemplo, na Figura 7.3 está representado um deque de capacidade de armazenamento 8 (oito), contendo 4 (quatro) elementos (e_1, e_2, e_3 e e_4), na "Situação inicial".

Para inserir um elemento (e_5) à esquerda, primeiro esse novo elemento é armazenado na posição indicada pela referência "`esquerda`" (Passo 1).

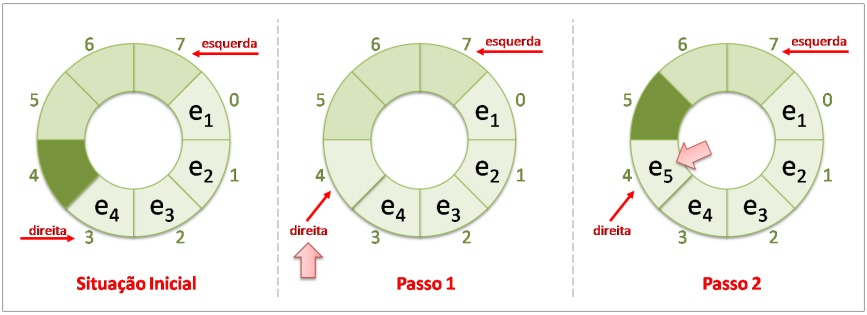
Uma vez armazenado no vetor, o valor da referência "`esquerda`" deve ser decrementado em uma unidade. O "**índice nulo**" nesse caso é o 4 (quatro).

Figura 7.3 – Deque Estático - Inserir à Esquerda



No caso da inserção ser realizada na extremidade direita, primeiro o valor da referência deve ser incrementado (também observando os aspectos da implementação circular do vetor) para depois ser armazenado o elemento que está sendo inserido. Por exemplo, a Figura 7.4 está representando um deque de capacidade de armazenamento 8 (oito), contendo 4 (quatro) elementos (e_1, e_2, e_3 e e_4), na "Situação inicial". Para inserir um elemento (e_5) à direita, o valor da referência deve ser incrementado em uma unidade (Passo 1). Após incremento, o novo valor é armazenado no índice indicado pela referência "`direita`" (Passo 2). O "**índice nulo**" inicialmente era o 4 (quatro) e após inserção passou a ser o índice 5 (cinco).

Figura 7.4 – Deque Estático - Inserir à Direita



!	Na implementação circular, ao incrementar uma referência, caso o valor obtido seja maior do que o último índice permitido pelo vetor (o que não pode ser feito), esta referência volta ao início do vetor, ou seja, ao índice 0 (zero).

!	Na implementação circular, ao decrementar uma referência, caso o valor obtido seja negativo (o que não pode ser feito), esta referência vai para o final do vetor, ou seja, assume o valor do maior índice permitido (tamanho do vetor - 1).

A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.3) e Java (ver Código Java 7.3) para o problema descrito.

Código C 7.3 – Deque Estático - Operações Básicas

Em C:

```
// Inserir à direita
int Inserir_Direita(Deque *deque, Elem elem){
  if (Cheia(*deque)) return FALSE;

  deque->direita = ++deque->direita % deque->tamanho;
  deque->elementos[deque->direita] = elem;

  return TRUE;
}

// Inserir à esquerda
int Inserir_Esquerda(Deque *deque, Elem elem){
  if (Cheia(*deque)) return FALSE;
  deque->elementos[deque->esquerda] = elem;
  if((deque->esquerda = --deque->esquerda % deque->
tamanho) < 0)
     deque->esquerda += deque->tamanho;
  return TRUE;
}
```

Código Java 7.3 – Deque Estático - Operações Básicas.

Em Java:

```
// Inserir à direita
public void inserirDireita(T novo) throws
DequeCheioException{
   if (this.isCheia())
      throw new DequeCheioException();
   this.direita = ++this.direita %
this.elementos.length;
   this.elementos[this.direita] = novo;
}

// Inserir à esquerda
public void inserirEsquerda(T novo) throws
DequeCheioException{
   if (this.isCheia())
      throw new DequeCheioException();
   this.elementos[this.esquerda] = novo;
   if ((this.esquerda = --this.esquerda %
this.elementos.length) < 0)
      this.esquerda += this.elementos.length;
}
```

7.2.4. Acessando Elementos

O acesso aos elementos armazenados na estrutura para impressão e/ou busca pode ser realizado partindo-se de qualquer extremidade para a outra, ou seja, da esquerda para direita ou da direita para esquerda. O programador decide como proceder e precisa estar atento para a implementação circular do vetor.

Semelhante ao que ocorre na fila, no deque é importante verificar quais são os elementos que estão posicionados nas extremidades da estrutura. É interessante destacar que a referência "`direita`" indica o elemento que está posicionado mais à direita da estrutura, enquanto que a referência "`esquerda`" indica a próxima referência mais à esquerda disponível para inserir elementos (desde que este não seja a "**referência nula**").

A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.4) e Java (ver Código Java 7.4) para os problemas descritos.

Código C 7.4 – Deque Estático - Acessando Elementos

Em C:

```
// Imprimir pela direita
void Imprimir_Direita(Deque deque){
   int i;
   for (i = deque.direita; i != deque.esquerda;){
      printf("Nome: %s - ", deque.elementos[i].nome);
      printf("Idade: %d\n", deque.elementos[i].idade);
      if((i = --i % deque.tamanho) < 0)
         i += deque.tamanho;
   }
}

// Imprimir pela esquerda
void Imprimir_Esquerda(Deque deque){
   int i = deque.esquerda + 1 % deque.tamanho;
   while (i != (deque.direita + 1) % deque.tamanho){
      printf("Nome: %s - ", deque.elementos[i].nome);
      printf("Idade: %d\n", deque.elementos[i].idade);
      i = ++i%deque.tamanho;
   }
}

// Ver elemento que está mais à direita
int Ver_Direita(Deque deque, Elem *elem){
   if (Vazia(deque)) return FALSE;
   *elem = deque.elementos[deque.direita];
   return TRUE;
}

// Ver elemento que está mais à esquerda
int Ver_Esquerda(Deque deque, Elem *elem){
   if (Vazia(deque)) return FALSE;
   *elem = deque.elementos[(deque.esquerda + 1) %
deque.tamanho];
   return TRUE;
}
```

Código Java 7.4 – Deque Estático - Operações Básicas

Em Java:

```
// Imprimir pela direita
public Iterator<T> getDireita(){
   @SuppressWarnings("unchecked")
   T[] aux = (T[]) new Object[this.getQuantidade()];
   int i = this.direita, j = 0;
   while (i != this.esquerda){
      aux[j++] = this.elementos[i];
      if ((i = --i % this.elementos.length) < 0)
         i += this.elementos.length;
   }

   return Arrays.asList(aux).iterator();
}

// Imprimir pela esquerda
public Iterator<T> getEsquerda(){
   @SuppressWarnings("unchecked")
   T[] aux = (T[]) new Object[this.getQuantidade()];
   int i = this.esquerda + 1 % this.elementos.length,
j = 0;

   while (i != (this.direita + 1) %
this.elementos.length){
      aux[j++] = this.elementos[i];
      i = ++i % this.elementos.length;
   }
   return Arrays.asList(aux).iterator();
}


// Ver elemento mais à direita
public T getElementoDireita() throws
DequeVazioException{
   if (this.isVazia())
      throw new DequeVazioException();
   return this.elementos[this.direita];
}
```

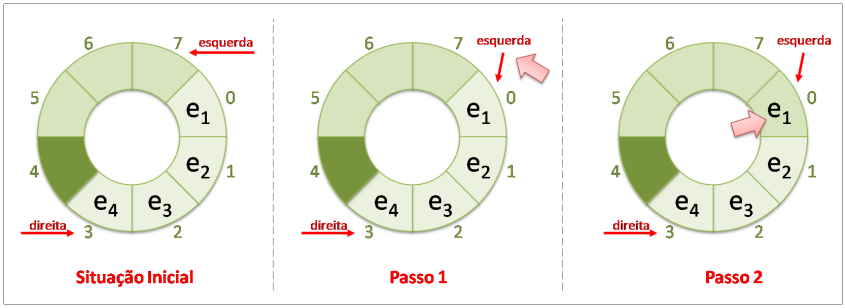
```
// Ver elemento mais à esquerda
public T getElementoEsquerda() throws
DequeVazioException{
   if (this.isVazia())
      throw new DequeVazioException();
   return this.elementos[(this.esquerda + 1) %
this.elementos.length];
}
```

7.2.5. Removendo Elementos

A remoção pode ser realizada em qualquer uma das extremidades. Para isso, é necessário apenas verificar a referência correspondente à extremidade que deseja realizar a remoção e executar o procedimento.

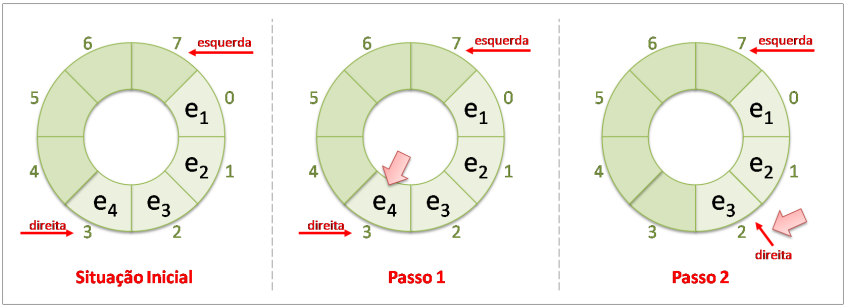
No caso da remoção na extremidade esquerda, na Figura 7.5 está representado um deque de tamanho 8 (oito) com 4 (quatro) elementos (e_1, e_2, e_3 e e_4), na "Situação inicial". Uma vez que a referência "`esquerda`" indica o índice mais à esquerda disponível para inserir elementos, é necessário incrementar esse valor (Passo 1) e, posteriormente, retirar o elemento (e_1) da estrutura (Passo 2).

Figura 7.5 – Deque Estático - Remover à Esquerda



Considerando a remoção na extremidade direita, na Figura 7.6 está representado um deque de tamanho 8 (oito) com 4 (quatro) elementos (e_1, e_2, e_3 e e_4), na "Situação inicial". A referência "`direita`" indica o elemento que está posicionado mais à direita na estrutura, ou seja, o elemento que será removido (Passo 1). Após identificar o elemento que será removido, é necessário realizar o decremento da referência "`direita`" (Passo 2).

Figura 7.6 – Deque Estático - Remover à Direita



A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.5) e Java (ver Código Java 7.5) para o problema descrito.

Código C 7.5 – Deque Estático - Operações Básicas

Em C:

```
// Remover direita
int Remover_Direita(Deque *deque, Elem *elem){
   if (Vazia(*deque)) return FALSE;
   *elem = deque->elementos[deque->direita];
   if((deque->direita = --deque->direita % deque-
>tamanho) < 0)
      deque->direita += deque->tamanho;
   return TRUE;
}
```

```
// Remover esquerda
int Remove_Esquerda(Deque *deque, Elem *elem){
   if (Vazia(*deque)) return FALSE;
   deque->esquerda = ++deque->esquerda % deque->
tamanho;
   *elem = deque->elementos[deque->esquerda];
   return TRUE;
}
```

Código Java 7.5 – Deque Estático - Operações Básicas

Em Java:

```
// Remover Direita
public T removerDireita() throws DequeVazioException{
   if (this.isVazia())
      throw new DequeVazioException();
   T elem = this.elementos[this.direita];
   if ((this.direita = --this.direita %
this.elementos.length) < 0)
      this.direita += this.elementos.length;
   return elem;
}

// Remover esquerda
public T removerEsquerda() throws
DequeVazioException{
   if (this.isVazia())
      throw new DequeVazioException();
   this.esquerda = ++this.esquerda %
this.elementos.length;
   return this.elementos[this.esquerda];
}
```

7.3. Deque Encadeado

O deque encadeado tem resultados semelhantes ao da fila encadeada (seção 6.3), ou seja, o fato de usar encadeamento de "nó" não necessita fazer o deslocamento físico dos dados nas operações de inserção e/ou remoção.

Uma vez que mantem referências para suas extremidades (esquerda e direita), também não apresenta problemas de desempenho ao realizar essas operações.

Diferente das outras estruturas apresentadas (pilha e fila), o percurso do deque ocorre nas duas direções (esquerda para direita e direta para esquerda).

Diante desse fato, este livro traz a implementação duplamente encadeada para o deque, uma vez que esta permite (com maior flexibilidade) o percurso na estrutura em qualquer direção. O encadeamento duplo foi apresentado na seção que trata da lista genérica duplamente encadeada.

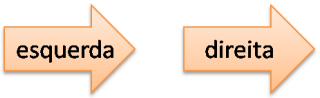
Uma vez conhecida a estratégia para gerenciamento da memória, é possível definir o TAD e seguir com a codificação.

7.3.1. Definição do TAD

Independente do encadeamento (simples ou duplo), esse tipo de estrutura mantém uma referência para a extremidade esquerda e outra para a extremidade direita. Com essas referências é garantido que as operações de inserção e remoção vão possuir melhor desempenho.

Na Figura 7.7 é possível observar o TAD para o deque encadeado.

Figura 7.7 – Deque Encadeado – TAD



OPERAÇÕES

- Criar;
- Vazio;
- Cheio;
- Quantidade;
- InserirEsquerda;
- InserirDireita;
- RemoverDireita;
- RemoverEsquerda;
- VerDireita;
- VerEsquerda;
- ImprimirDireita;
- ImprimirEsquerda.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.6) e Java (ver Código Java 7.6) para o problema descrito.

Código C 7.6 – Deque Encadeado - TAD

Em C:

```
typedef struct{
   char nome[31];
   int idade;
}Elem;

typedef struct no{
   Elem elemento;
   struct no *anterior;
   struct no *proximo;
}No;

typedef struct {
   No *esquerda, *direita;
}TDeque;
```

Código Java 7.6 – Deque Encadeado - TAD

Em Java:

```
public class No <T>{
   private T elemento;
   private No<T> anterior, proximo;
}

public class DequeEncadeado <T>{
   public No<T> esquerda, direita;
}
```

7.3.2. Operações Básicas

As operações básicas seguem o mesmo comportamento definido da lista genérica duplamente encadeada. A referência "`esquerda`" da estrutura do deque possui comportamento compatível com a referência "`início`" da lista duplamente encadeada. O mesmo ocorre com a referência "`direita`" do deque, que possui comportamento compatível com a referência "`fim`" da lista duplamente encadeada. A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.7) e Java (ver Código Java 7.7) para os problemas descritos.

Código C 7.7 – Deque Encadeado - Operações Básicas

Em C:

```
// Criar o deque encadeado
void Cria(TDeque *deque){
   deque->esquerda = deque->direita = NULL;
}

// Verificar se o deque encadeado está vazio
int Vazia(TDeque deque){
   if (deque.esquerda == NULL) return TRUE;
   else return FALSE;
}
```

```
// Obter a quantidade de elementos contidos no deque encadeado
int Quantidade(TDeque deque){
   No *aux;
   int qtde = 0;

   for (aux = deque.esquerda; aux != NULL; aux = aux->proximo) ++qtde;
   return qtde;
}
```

Código Java 7.7 – Deque Encadeado - Operações Básicas

Em Java:

```
// Criar o deque encadeado
public DequeEncadeado(){
   this.esquerda = this.direita = null;
}

// Verificar se o deque encadeado está vazio
public boolean isVazio(){
   return this.esquerda == null;
}

// Obter a quantidade de elementos contidos no deque encadeado
public int getQuantidade(){
   No<T> aux;
   int qtde = 0;
   aux = this.esquerda;
   while (aux != null){
      ++qtde;
      aux = aux.getProximo();
   }
   return qtde;
}
```

7.3.3. Inserindo Elementos

O deque é uma estrutura que permite inserir elementos em quaisquer das extremidades (esquerda e/ou direita).

Uma vez que existe equivalência entre a referência "`esquerda`" do deque e a referência "`início`" da lista duplamente encadeada, as operações de inserção também seguem o mesmo padrão. Ou seja, a inserção à esquerda (deque) é compatível com a inserção no início (lista), como também, a inserção à direita (deque) é compatível com a inserção no fim (lista). As operações de inserção na lista duplamente encadeada foram apresentadas na seção 4.4.3.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.8) e Java (ver Código Java 7.8) para o problema descrito.

Código C 7.8 – Deque Encadeado - Inserir

Em C:

```
int Inserir_Direita(TDeque *deque, Elem elem){
   No *novo;

   if (Cheia(*deque)) return FALSE;

   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;

   novo->elemento = elem;
   novo->proximo = NULL;
   novo->anterior = deque->direita;

   if (Vazia(*deque)) deque->esquerda = novo;
   else deque->direita->proximo = novo;
   deque->direita = novo;
   return TRUE;
}
```

```
int Inserir_Esquerda(TDeque *deque, Elem elem){
  No *novo;

  if (Cheia(*deque)) return FALSE;
  if ((novo = malloc(sizeof(No))) == NULL)
     return FALSE;
  novo->elemento = elem;
  novo->proximo = deque->esquerda;
  novo->anterior = NULL;
  if (Vazia(*deque)) deque->direita = novo;
  else deque->esquerda->anterior = novo;
  deque->esquerda = novo;
  return TRUE;
}
```

Código Java 7.8 – Deque Encadeado - Inserir

Em Java:

```
public void inserirDireita(T elem){
  No<T> novo = new No<T>(elem);
  novo.setAnterior(this.direita);
  if (this.isVazio()) this.esquerda = novo;
  else this.direita.setProximo(novo);
  this.direita = novo;
}

public void inserirEsquerda(T elem){
  No<T> novo = new No<T>(elem);
  novo.setProximo(this.esquerda);
  if (this.isVazio()) this.direita = novo;
  else this.esquerda.setAnterior(novo);
  this.esquerda = novo;
}
```

7.3.4. Acessando Elementos

Por possuir dupla cabeça, o acesso aos elementos contidos no deque pode ser realizado nas duas direções (esquerda para direita e direita para esquerda). Operações para impressão ou busca de elementos podem considerar esse aspecto na sua implementação.

Da mesma forma que ocorre na implementação estática dessa estrutura, também é interessante verificar quais são os elementos contidos nas duas extremidades.

A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.9) e Java (ver Código Java 7.9) para o problema descrito.

Código C 7.9 – Deque Encadeado - Acessando Elementos

Em C:

```
// Imprimir pela direita
void Imprimir_Direita(TDeque deque){
   No *aux;

   for (aux = deque.direita; aux != NULL; aux = aux-
>anterior)
      printf("%s (%d)\n", aux->elemento.nome, aux-
>elemento.idade);
}

// Imprimir pela esquerda
void Imprimir_Esquerda(TDeque deque){
   No *aux;

   for (aux = deque.esquerda; aux != NULL; aux = aux-
>proximo)
      printf("%s (%d)\n", aux->elemento.nome, aux-
>elemento.idade);
}
```

```
// Ver elemento que está na extremidade Direita
int Ver_Direita(TDeque deque, Elem *elem){
  if (Vazia(deque)) return FALSE;

  *elem = deque.direita->elemento;
  return TRUE;
}


// Ver elemento que está na extremidade Esquerda
int Ver_Esquerda(TDeque deque, Elem *elem){
  if (Vazia(deque)) return FALSE;

  *elem = deque.esquerda->elemento;
  return TRUE;
}

// Buscar elemento partindo da direita
int Buscar_Direita(TDeque deque, char *nome){
  No *aux;

  for (aux = deque.direita; aux != NULL; aux = aux-
>anterior)
    if (stricmp(aux->elemento.nome, nome) == 0)
return TRUE;

  return FALSE;
}

// Buscar elemento partindo da esquerda
int Buscar_Esquerda(TDeque deque, char *nome){
  No *aux;

  for (aux = deque.esquerda; aux != NULL; aux = aux-
>proximo)
    if (stricmp(aux->elemento.nome, nome) == 0)
return TRUE;

  return FALSE;
}
```

Código Java 7.9 – Deque Encadeado - Acessando Elementos

Em Java:

```
// Imprimir pela direita
public Iterator<T> getDireta(){
   No<T> atual = this.direita;
   @SuppressWarnings("unchecked")
   T[] aux = (T[]) new Object[this.getQuantidade()];
   int i = 0;
   while (atual != null){
      aux[i++] = atual.getElemento();
      atual = atual.getAnterior();
   }
   return Arrays.asList(aux).iterator();
}


// Imprimir pela esquerda
public Iterator<T> getEsquerda(){
   No<T> atual = this.esquerda;
   @SuppressWarnings("unchecked")
   T[] aux = (T[]) new Object[this.getQuantidade()];
   int i = 0;
   while (atual != null){
      aux[i++] = atual.getElemento();
      atual = atual.getProximo();
   }
   return Arrays.asList(aux).iterator();
}


// Obter elemento que está na extremidade Direita
public T getElementoDireita() throws
DequeVazioException{
   if (this.isVazio())
      throw new DequeVazioException();
   return this.direita.getElemento();
}
```

```
// Obter elemento que está na extremidade Esquerda
public T getElementoEsquerda() throws
DequeVazioException{
   if (this.isVazio())
      throw new DequeVazioException();
   return this.esquerda.getElemento();
}

// Buscar elemento partindo da extremidade Direita
public boolean getDireta(T elem){
   No<T> atual = this.direita;
   while (atual != null){
      if (atual.getElemento().equals(elem))
        return true;
      atual = atual.getAnterior();
   }
   return false;
}

// Buscar elemento partindo da extremidade Esquerda
public boolean getEsquerda(T elem){
   No<T> atual = this.esquerda;
   while (atual != null){
      if (atual.getElemento().equals(elem))
        return true;
      atual = atual.getProximo();
   }
   return false;
}
```

7.3.5. Removendo Elementos

Semelhante ao comportamento da inserção, o deque é uma estrutura que permite remover elementos em quaisquer das extremidades (esquerda e/ou direita).

Uma vez que existe equivalência entre a referência "`direta`" do deque e a referência "`fim`" da lista duplamente encadeada, as operações de remoção também seguem o mesmo padrão.

Ou seja, a remoção à esquerda (deque) é compatível com a remoção no início (lista), como também, a remoção à direita (deque) é compatível com a remoção no fim (lista). As operações de remoção na lista duplamente encadeada foram apresentadas na seção 4.4.5. A seguir, são apresentadas soluções nas linguagens C (ver Código C 7.10) e Java (ver Código Java 7.10) para os problemas descritos.

Código C 7.10 – Deque Encadeado - Inserir

Em C:

```
int Remover_Esquerda(TDeque *deque, Elem *elem){
   No *atual;
   if (Vazia(*deque)) return FALSE;
   atual = deque->esquerda;
   if (deque->direita == deque->esquerda)
      deque->direita = deque->esquerda = NULL;
   else{
      deque->esquerda->proximo->anterior = NULL;
      deque->esquerda = deque->esquerda->proximo;
   }
   *elem = atual->elemento;
   free(atual);
   return TRUE;
}

int Remover_Direita(TDeque *deque, Elem *elem){
   No *atual;
   if (Vazia(*deque)) return FALSE;
   atual = deque->direita;
   *elem = atual->elemento;
   if (deque->direita == deque->esquerda)
      deque->direita = deque->esquerda = NULL;
   else{
      deque->direita->anterior->proximo = NULL;
      deque->direita = deque->direita->anterior;
   }
   free(atual);
   return TRUE;
}
```

Código Java 7.10 – Deque Encadeado - Inserir

Em Java:

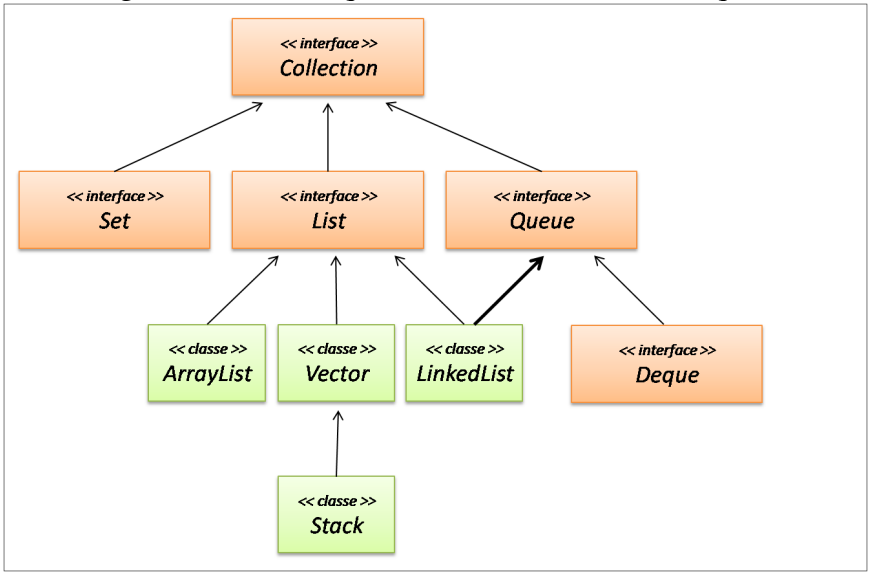
```
public T removerEsquerda() throws
DequeVazioException{
   No<T> atual = this.esquerda;

   if (this.isVazio())
      throw new DequeVazioException();

   if (this.direita == this.esquerda)
      this.direita = this.esquerda = null;
   else{
      this.esquerda.getProximo().setAnterior(null);
      this.esquerda = this.esquerda.getProximo();
   }

   return atual.getElemento();
}


public T removerDireita() throws DequeVazioException{
   No<T> atual = this.direita;
   if (this.isVazio())
      throw new DequeVazioException();

   if (this.direita == this.esquerda)
      this.direita = this.esquerda = null;
   else{
      this.direita.getAnterior().setProximo(null);
      this.direita = this.direita.getAnterior();
   }
   return atual.getElemento();
}
```

7.4. API Java

Semelhante ao que ocorre com as estruturas de dados pilha (*Stack*) e fila (*Queue*), o *framework Collections* também traz uma definição para o deque, a interface "*Deque*"[16] (`Java.util.Deque<E>`), e a classe "LinkedList" implementa essa especificação, conforme pode ser observado na Figura 7.8.

A interface "*Deque*" define o comportamento da fila de cabeça dupla e oferece um conjunto de serviços inerentes ao funcionamento de um deque. É importante destacar que é uma interface, ou seja, não é uma implementação e sim uma definição de comportamento. Conforme dito anteriormente, a classe "*LinkedList*", além dos serviços definidos na interface "*List*", também implementa os serviços definidos na interface "*Queue*".

Figura 7.8 – Hierarquia *Collection* (interface Deque)



[16] http://docs.oracle.com/javase/6/docs/api/java/util/Deque.html

As operações definidas na interface "Deque" possuem duas formas de implementação: uma gerencia a execução, podendo lançar exceção caso ocorra algum problema, enquanto que a outra gerencia a execução por meio de valores retornados pelos seus serviços, para indicar o status da execução (sucesso ou não). No Quadro 7.1 é apresentado o mapeamento das operações discutidas nesta seção e respectivo serviço que o implementa.

Quadro 7.1 – Deque - Mapeamento das Operações em Java (Deque)

Categoria	Operação	Deque (com exceção)	Deque (sem exceção)
Básica	Criar	`Construtor`	`Construtor`
	Verificar Vazia	`isEmpty()`	`isEmpty()`
	Tamanho	`size()`	`size()`
Inserção	Esquerda	`addFirst()`	`offerFirst()`
	Direita	`addLast()`	`OfferLast()`
Acesso	Imprimir	`iterator()`	
	Pesquisar	`contains(<<elem>>)`	
	Ver Inicio	`getFirst()`	`peekFirst()`
	Ver Fim	`getLast()`	`peekLast()`
Remoção	Esquerda	`removeFirst()`	`pollFirst()`
	Direita	`removeLast()`	`pollLast()`

No trecho de Código Java 7.11 é apresentado um programa que faz uso do deque "Deque", por meio da classe "LinkedList".

Na linha 6, o deque está sendo criado (instanciado). Após criação, está sendo verificado se ele está vazio (linha 8).

Conforme citado, a interface "`Deque`" oferece duas formas de inserção, considerando o gerenciamento dos erros (lançamento de exceção) provenientes da execução dos serviços. Entre as linhas 12 e 25, elementos estão sendo armazenados fazendo uso desses serviços.

Ao executar as inserções, o deque estará contendo (da esquerda para direita) os seguintes elementos: Quarto, Terceiro, Segundo, Primeiro, Quinto, Sexto, Sétimo e Oitavo, contabilizando 8 (oito) elementos.

Na linha 28 verifica-se a quantidade de elementos persistidos na estrutura.

Da mesma forma que ocorre com a inserção, para remover elementos, a interface "`Deque`" permite realizar a operação com lançamento de exceção ou indicação de sucesso (retorno do método).

Entre as linhas 31 e 40 estão sendo realizadas remoções, obtendo elementos na seguinte ordem: Quarto, Terceiro, Oitavo e Sétimo.

Para visualização dos elementos contidos nas extremidades do deque, também existem serviços com lançamento de exceção e serviços com verificação de erros (retorno do método); entre as linhas 43 e 52 estão sendo verificados esses valores, obtendo como resultado: Segundo, Segundo, Sexto e Sexto.

Após execução dessas operações, ainda persistem no deque os seguintes elementos (da esquerda para direita): Segundo, Primeiro, Quinto e Sexto.

Na linha 54 esses elementos estão sendo impressos.

Código Java 7.11 – Deque com LinkedList

```
1 import java.util.Deque;
2 import java.util.LinkedList;
3 public class UsaDequeJava {
4  public static void main(String[] args) {
5    // Criar o deque
6    Deque<String> deque = new LinkedList<String>();

7    // Verificar se o deque está vazio
8    if (deque.isEmpty()) System.out.println("Deque
   vazio");
9    else System.out.println("Deque não vazio");
10
11   // Inserir elementos à esquerda (podendo lançar
   exceção)
12   deque.addFirst("Primeiro");
13   deque.addFirst("Segundo");
14
15   // Inserir elementos à esquerda (sem lançar
   exceção)
16   deque.offerFirst("Terceiro");
17   deque.offerFirst("Quarto");
18
19   // Inserir elementos à direita (podendo lançar
   exceção)
20   deque.addLast("Quinto");
21   deque.addLast("Sexto");
22
23   // Inserir elementos à direita (sem lançar
   exceção)
24   deque.offerLast("Sétimo");
25   deque.offerLast("Oitavo");
26
27   // Verificar a quantidade de elementos
28   System.out.println("Quantidade: " +
   deque.size());
29
30   // remover elemento esquerda (podendo lançar
   exceção)
```

```
31   System.out.println("Removido esquerda: "+
   deque.removeFirst());
32
33   // remover elemento esquerda (sem lançar
   exceção)
34   System.out.println("Removido esquerda: " +
   deque.pollFirst());
35
36   // Remover elemento à direita (podendo lançar
   exceção)
37   System.out.println("Removido direita: "+
   deque.removeLast());
38
39   // Remover elemento à direita (sem lançar
   exceção)
40   System.out.println("Removido direita: " +
   deque.pollLast());
41
42   // Ver elemento mais à esquerda (podendo lançar
   exceção)
43   System.out.println("Esquerda: " +
   deque.getFirst());
44
45   // Ver elemento mais à esquerda (sem lançar
   exceção)
46   System.out.println("Esquerda: " +
   deque.peekFirst());
47
48   // Ver elemento mais à direita (podendo lançar
   exceção)
49   System.out.println("Direita: " +
   deque.getLast());
50
51   // Ver elemento mais à direita (sem lançar
   exceção)
52   System.out.println("Direita: " +
   deque.peekLast());
53   // Imprimir
54   System.out.println(deque);
55 }
56 }
```

7.5. Exercícios Propostos:

1) O que é um deque?

2) Como é a dinâmica de funcionamento de um deque?

3) Quais são as vantagens da implementação circular para o deque?

4) O que é o "**índice nulo**"? É fixo no vetor?

5) É possível inserir um elemento em qualquer posição do deque? Por quê?

6) Não tem diferença a escolha das extremidades esquerda e direita nas implementações estática e dinâmica? Por quê?

7) Existe semelhança entre o deque encadeado e a lista genérica duplamente encadeada? Explique.

8) Desenvolva uma aplicação que faça uso das estruturas apresentadas neste capítulo.

Referências

CELES, Waldemar *et al.* **Introdução a estruturas de dados:** com técnicas de programação em C. Rio de Janeiro: Elsevier, 2004.

CORMEN, Thomas H. *et al.* **Algoritmos:** teoria e prática. Rio de Janeiro: Elsevier, 2002.

DAMAS, Luis. **Linguagem C**. Rio de Janeiro: LTC, 2007.

DEITEL, Paul; DEITEL, Harvey. **Java – Como Programar**. Rio de Janeiro: Prentice Hall Brasil, 2010.

HOROWITZ, Ellis; SAHNI, Sartaj. **Fundamentos de estruturas de dados**. Rio de Janeiro: Campus, 1987.

MORAES, Celso Roberto. **Estruturas de dados e algoritmos – uma abordagem didática**. São Paulo: Berkeley Brasil, 2001.

PREISS, Bruno R**. Estruturas de dados e algoritmos:** Padrões de projetos orientados a objeto com Java. Rio de Janeiro: Elsevier, 2000.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estruturas de dados, com aplicações em Java.** São Paulo: Pearson Prentice Hall, 2003.

SILVA, Osmar Quirino da. **Estrutura de Dados e Algoritmos Usando C – Fundamentos e Aplicações.** Rio de Janeiro: Editora Ciência Moderna Ltda., 2007.

SOMMERVILLE, Ian. **Engenharia de Software.** São Paulo: Pearson Addison-Wesley, 2011.

TENEMBAUM, Aaron M. *et al.* **Estrutura de dados usando C.** São Paulo: Pearson Makron Books, 1995.

VELOSO, Paulo *et al.* **Estruturas de dados.** Rio de Janeiro: Campus, 1993.

VILLAS, Marcos Vianna *et al.* **Estruturas de dados:** conceitos e técnicas de implementação. Rio de Janeiro: Campus, 1993.

Linguagem C: lista_estatica.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

typedef struct {
   char nome[31];
   int idade;
}Elem;

typedef struct {
   Elem *elementos;
   int quantidade, tamanho;
}ListaEstatica;

int Criar(ListaEstatica *lista, int tam);
int EVazia(ListaEstatica lista);
int ECheia(ListaEstatica lista);
int Quantidade(ListaEstatica lista);

int InserirInicio(ListaEstatica *lista, Elem
novo_elemento);
int InserirFim(ListaEstatica *lista, Elem
novo_elemento);

void Imprimir(ListaEstatica lista);
int Pesquisar(ListaEstatica lista, char *nome);

int RemoverInicio(ListaEstatica *lista, Elem *elem);
int RemoverFim(ListaEstatica *lista, Elem *elem);
int Remover(ListaEstatica *lista, Elem *elem);
```

Linguagem C: lista_estatica.c

```
#include "lista_estatica.h"

int Criar(ListaEstatica *lista, int tam){
  if ((lista->elementos = calloc(tam, sizeof(Elem)))
== NULL)
    return FALSE;
  lista->tamanho = tam;
  lista->quantidade = 0;
  return TRUE;
}

int EVazia(ListaEstatica lista){
  return lista.quantidade == 0;
}

int ECheia(ListaEstatica lista){
  return lista.quantidade == lista.tamanho;
}
int Quantidade(ListaEstatica lista){
  return lista.quantidade;
}

int InserirInicio(ListaEstatica *lista, Elem
novo_elemento){
  int i;

  if (ECheia(*lista)) return FALSE;
  for (i = lista->quantidade; i > 0; --i)
    lista->elementos[i] = lista->elementos[i - 1];
  lista->elementos[0] = novo_elemento;
  lista->quantidade++;
  return TRUE;
}

int InserirFim(ListaEstatica *lista, Elem
novo_elemento){
  if (ECheia(*lista)) return FALSE;
  lista->elementos[lista->quantidade++] =
novo_elemento;
  return TRUE;
}
```

```
void Imprimir(ListaEstatica lista){
   int i;
   for (i = 0; i < lista.quantidade; ++i){
     printf("Nome: %s - ", lista.elementos[i].nome);
     printf("Idade: %d\n", lista.elementos[i].idade);
   }
}



int Pesquisar(ListaEstatica lista, char *nome){
   int i;
   for (i = 0; i < lista.quantidade; ++i)
     if (stricmp(lista.elementos[i].nome, nome) == 0)
        return i;
   return -1;
}



int RemoverInicio(ListaEstatica *lista, Elem *elem){
   int i;

   if (EVazia(*lista)) return FALSE;
   *elem = lista->elementos[0];
   --lista->quantidade;
   for (i = 0; i < lista->quantidade; ++i)
     lista->elementos[i] = lista->elementos[i + 1];
   return TRUE;
}



int RemoverFim(ListaEstatica *lista, Elem *elem){
   if (EVazia(*lista)) return FALSE;
   *elem = lista->elementos[--lista->quantidade];
   return TRUE;
}
```

```
int Remover(ListaEstatica *lista, Elem *elem){
  int i, pos = Pesquisar(*lista, elem->nome);

  if (pos < 0) return FALSE;
  else{
    *elem = lista->elementos[pos];
    for (i = pos; i < lista->quantidade; ++i)
      lista->elementos[i] = lista->elementos[i + 1];
    lista->quantidade--;
    return TRUE;
  }
}
```

Linguagem Java: ListaEstatica.java

```
import java.util.Arrays;
import java.util.Iterator;

public class ListaEstatica<T>{
  private T[] elementos;
  private int quantidade;

  @SuppressWarnings("unchecked")
  public ListaEstatica(int tamanho) {
    this.elementos = (T[]) new Object[tamanho];
    this.quantidade = 0;
  }

  public boolean isCheia() {
    return this.quantidade == this.elementos.length;
  }

  public boolean isVazia() {
    return this.quantidade == 0;
  }

  public int getQuantidade() {
    return this.quantidade;
  }
```

```
  public void inserirInicio(T novo) {
    if (this.isCheia())
      throw new ListaCheiaException();

    for (int i = this.quantidade; i > 0; --i)
      this.elementos[i] = this.elementos[i - 1];

    this.elementos[0] = novo;
    ++this.quantidade;
  }



  public void inserirFim(T novo) {
    if (this.isCheia())
                throw new ListaCheiaException());
    this.elementos[this.quantidade++] = novo;
  }




  public Iterator<T> get() {
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new
Object[this.getQuantidade()];
    for(int i = 0; i < this.quantidade; i++)
      temp[i] = this.elementos[i];
    return Arrays.asList(temp).iterator();
  }

  public int get(T elemento) {
    for (int i = 0; i < this.quantidade; ++i)
      if (this.elementos[i].equals(elemento)) return
i;
    throw new ElementoNaoExisteException();
  }
```

```
  public T removerInicio() {
     T aux;
     if (this.isVazia())
        throw new ListaVaziaException();
     aux = this.elementos[0];
     --this.quantidade;
     for (int i = 0; i < this.quantidade; ++i)
       this.elementos[i] = this.elementos[i + 1];

     return aux;
  }


  public T removerFim() {
     if (this.isVazia())
        throw new ListaVaziaException();
     return this.elementos[--this.quantidade];
  }


  public void remover(T elemento) {
     int posicao = this.get(elemento);
     if (posicao < 0)
        throw new ElementoNaoExisteException();
     for (int i = posicao; i < this.quantidade; ++i)
       this.elementos[i] = this.elementos[i + 1];
     --this.quantidade;
  }
}
```

Apêndice B - Lista Genérica Simplesmente Encadeada

Linguagem C: lista_simplesmente_encadeada.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

typedef struct {
   char nome[31];
   int idade;
}Elem;

typedef struct no{
   Elem elemento;
   struct no *proximo;
}No;

typedef struct {
   No *inicio;
}ListaSimplesmenteEncadeada;

void Criar(ListaSimplesmenteEncadeada *lista);
int EVazia(ListaSimplesmenteEncadeada lista);
int ECheia(ListaSimplesmenteEncadeada lista);
int Quantidade(ListaSimplesmenteEncadeada lista);

int InsereInicio(ListaSimplesmenteEncadeada *lista,
Elem novo_elemento);
int InsereFim(ListaSimplesmenteEncadeada *lista, Elem
novo_elemento);

void Imprime(ListaSimplesmenteEncadeada lista);
int Pesquisa(ListaSimplesmenteEncadeada lista, char
*nome);

int RemoveFim(ListaSimplesmenteEncadeada *lista, Elem
*elem);
int RemoveInicio(ListaSimplesmenteEncadeada *lista,
Elem *elem);
```

```
int Remove(ListaSimplesmenteEncadeada *lista, char
*nome, Elem *elem);
```

Linguagem C: lista_simplesmente_encadeada.c

```
#include "lista_simplesmente_encadeada.h"

void Criar(ListaSimplesmenteEncadeada *lista){
  lista->inicio = NULL;
}

int EVazia(ListaSimplesmenteEncadeada lista){
  return (lista.inicio == NULL);
}

int ECheia(ListaSimplesmenteEncadeada lista){
  return FALSE;
}
int Quantidade(ListaSimplesmenteEncadeada lista){
  No *atual;
  int qtde = 0;

  for (atual = lista.inicio; atual != NULL; atual =
atual->proximo)
    qtde++;
  return qtde;
}

int InsereInicio(ListaSimplesmenteEncadeada *lista,
Elem novo_elemento){
  No *novo;

  if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;
  novo->elemento = novo_elemento;
  novo->proximo = lista->inicio;
  lista->inicio = novo;
  return TRUE;
}
```

```
int InsereFim(ListaSimplesmenteEncadeada *lista, Elem
novo_elemento){
   No *atual, *novo;

   if (EVazia(*lista))
      return InsereInicio(lista, novo_elemento);

   atual = lista->inicio;
   while (atual->proximo != NULL)
       atual = atual->proximo;
   if ((novo = malloc(sizeof(No))) == NULL)
       return FALSE;
   atual->proximo = novo;
   novo->proximo = NULL;
   novo->elemento = novo_elemento;
   return TRUE;
}


void Imprime(ListaSimplesmenteEncadeada lista){
   No *atual;

   for (atual = lista.inicio; atual != NULL; atual =
atual->proximo){
     printf("Nome: %s - ", atual->elemento.nome);
     printf("Idade: %d\n", atual->elemento.idade);
   }
}


int Pesquisa(ListaSimplesmenteEncadeada lista, char
*nome){
   No *atual = lista.inicio;
   while (atual != NULL){
     if (stricmp(atual->elemento.nome, nome) == 0)
        return TRUE;
     atual = atual->proximo;
   }
   return FALSE;
}
```

```
int RemoveInicio(ListaSimplesmenteEncadeada *lista,
Elem *elem){
  No *atual;

  if (EVazia(*lista)) return FALSE;
  atual = lista->inicio;
  lista->inicio = atual->proximo;
  *elem = atual->elemento;

  free(atual);

  return TRUE;
}
```

```
int RemoveFim(ListaSimplesmenteEncadeada *lista, Elem
*elem){
  No *atual, *anterior = NULL;

  if (EVazia(*lista)) return FALSE;

  atual = lista->inicio;
  while (atual->proximo != NULL){
    anterior = atual;
    atual = atual->proximo;
  }

  if (anterior == NULL)
    return RemoveInicio(lista, elem);
  *elem = atual->elemento;
  anterior->proximo = atual->proximo;

  free(atual);

  return TRUE;
}
```

```
int Remove(ListaSimplesmenteEncadeada *lista, char
*nome, Elem *elem){
   No *anterior, *atual;

   if (EVazia(*lista)) return FALSE;
   anterior = NULL;
   atual = lista->inicio;
   while (atual != NULL){
      if (stricmp(atual->elemento.nome, nome) == 0){
         if (anterior == NULL)
              return RemoveInicio(lista, elem);
         anterior->proximo = atual->proximo;
         *elem = atual->elemento;
         free(atual);
         return TRUE;
      }
      anterior = atual;
      atual = atual->proximo;
   }
   return FALSE;
}
```

Linguagem Java: No.java

```
public class No <T>{
   private T elemento;
   private No<T> proximo;

   public No(T elemento) {
      this.elemento = elemento;
      this.proximo = null;
   }

   public T getElemento() {
      return elemento;
   }

   public void setElemento(T elemento) {
      this.elemento = elemento;
   }
```

```
   public No<T> getProximo() {
      return proximo;
   }

   public void setProximo(No<T> proximo) {
      this.proximo = proximo;
   }

   public String toString(){
      return String.format("%s", this.elemento);
   }
}
```

Linguagem Java: ListaSimplesmenteEncadeada.java

```
import java.util.Arrays;
import java.util.Iterator;


public class ListaSimplesmenteEncadeada <T>{
   private No<T> inicio;

   public ListaSimplesmenteEncadeada(){
      this.inicio = null;
   }


   public boolean isCheia() {
      throw new NaoSeAplicaException("Lista
encadeada");
   }


   public boolean isVazia() {
      return this.inicio == null;
   }


   public int getQuantidade() {
      int qtde = 0;
      No<T> atual = this.inicio;
      while (atual != null){
         atual = atual.getProximo();
         ++qtde;
      }
      return qtde;
   }


   public void inserirInicio(T elem) {
      No<T> no = new No<T>(elem);
      no.setProximo(this.inicio);
      this.inicio = no;
   }
```

```
  public void inserirFim(T elem) {
    if (this.isVazia()) inserirInicio(elem);
    else{
      No<T> atual, no = new No<T>(elem);
      atual = this.inicio;
      while (atual.getProximo() != null)
           atual = atual.getProximo();
      atual.setProximo(no);
      no.setProximo(null);
    }
  }

  public Iterator<T> get() {
    int i = 0;

    @SuppressWarnings("unchecked")
    T[] vetor = (T[]) new
Object[this.getQuantidade()];

    No<T> atual = this.inicio;
    while (atual != null){
      vetor[i++] = atual.getElemento();
      atual = atual.getProximo();
    }

    return Arrays.asList(vetor).iterator();
  }

  public int get(T elemento) {
    int i = 0;
    No<T> atual = this.inicio;
    while (atual != null){
      if (atual.getElemento().equals(elemento))
         return i;
      atual = atual.getProximo();
      ++i;
    }

    throw new ElementoNaoExisteException();
  }
```

```
  public T removerInicio() {
    if (this.isVazia())
       throw new ListaVaziaException();
    T elemento = this.inicio.getElemento();
    this.inicio = this.inicio.getProximo();
    return elemento;
  }

  public T removerFim() {
    No<T> atual, anterior = null;
    if (this.isVazia())
       throw new ListaVaziaException();
    atual = this.inicio;
    while (atual.getProximo() != null){
      anterior = atual;
      atual = atual.getProximo();
    }
    if (anterior == null)
      return this.removerInicio();
    anterior.setProximo(atual.getProximo());
    return atual.getElemento();
  }

  public void remover(T elemento) {
    No<T> atual, anterior = null;
    if (this.isVazia())
       throw new ListaVaziaException();
    atual = this.inicio;
    while (atual != null){
      if (atual.getElemento().equals(elemento)){
           if (anterior == null)
            this.removerInicio();
           else
            anterior.setProximo(atual.getProximo());
           return;
      }
      anterior = atual;
      atual = atual.getProximo();
    }
    throw new ElementoNaoExisteException();
  }
}
```

Apêndice C - Lista Genérica Duplamente Encadeada

Linguagem C: lista_duplamente_encadeada.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

typedef struct {
  char nome[31];
  int idade;
}Elem;

typedef struct no{
  Elem elemento;
  struct no *anterior;
  struct no *proximo;
}No;

typedef struct {
  No *inicio;
  No *fim;
}ListaDuplamenteEncadeada;

void Cria(ListaDuplamenteEncadeada *lista);
int EVazia(ListaDuplamenteEncadeada lista);
int ECheia(ListaDuplamenteEncadeada lista);
int Quantidade(ListaDuplamenteEncadeada lista);

int InsereInicio(ListaDuplamenteEncadeada *lista,
Elem novo_elemento);
int InsereFim(ListaDuplamenteEncadeada *lista, Elem
novo_elemento);

int Pesquisa(ListaDuplamenteEncadeada lista, char
*nome);

int RemoveFim(ListaDuplamenteEncadeada *lista, Elem
*elem);
int RemoveInicio(ListaDuplamenteEncadeada *lista,
Elem *elem);
```

```
int Remove(ListaDuplamenteEncadeada *lista, char
*nome, Elem *elem);
```

Linguagem C: lista_duplamente_encadeada.c

```
#include "lista_duplamente_encadeada.h"


void Cria(ListaDuplamenteEncadeada *lista){
   lista->inicio = lista->fim = NULL;
}



int EVazia(ListaDuplamenteEncadeada lista){
   if (lista.inicio == NULL) return TRUE;
   else return FALSE;
}



int ECheia(ListaDuplamenteEncadeada lista){
   return FALSE;
}



int Quantidade(ListaDuplamenteEncadeada lista){
   No *atual = lista.inicio;
   int qtde = 0;

   for (atual = lista.inicio; atual != NULL; atual =
atual->proximo)
      qtde++;
   return qtde;
}
```

```
int InsereInicio(ListaDuplamenteEncadeada *lista,
Elem novo_elemento){
  No *novo;

  if ((novo = malloc(sizeof(No))) == NULL)
     return FALSE;
  novo->elemento = novo_elemento;
  novo->proximo = lista->inicio;
  novo->anterior = NULL;
  if (EVazia(*lista)) lista->fim = novo;
  else lista->inicio->anterior = novo;
  lista->inicio = novo;

  return TRUE;
}
```

```
int InsereFim(ListaDuplamenteEncadeada *lista, Elem
novo_elemento){
  No *atual, *novo;

  if (EVazia(*lista))
      return InsereInicio(lista, novo_elemento);
  if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;

  novo->elemento = novo_elemento;
  novo->proximo = NULL;
  lista->fim->proximo = novo;
  novo->anterior = lista->fim;

  lista->fim = novo;

  return TRUE;
}
```

```
void Imprime(ListaDuplamenteEncadeada lista){
  No *atual;

  for (atual = lista.inicio; atual != NULL; atual =
atual->proximo)
     printf("%s - %d\n", atual->elemento.nome, atual-
>elemento.idade);
}


int Pesquisa(ListaDuplamenteEncadeada lista, char
*nome){
  No *atual = lista.inicio;
  while (atual != NULL){
    if (stricmp(atual->elemento.nome, nome) == 0)
      return TRUE;
    atual = atual->proximo;
  }
  return FALSE;
}


int RemoveInicio(ListaDuplamenteEncadeada *lista,
Elem *elem){
  No *atual;

  if (EVazia(*lista)) return FALSE;
  atual = lista->inicio;
  *elem = atual->elemento;

  if (lista->inicio == lista->fim)
    lista->inicio = lista->fim = NULL;
  else{
    lista->inicio = atual->proximo;
    lista->inicio->anterior = NULL;
  }

  free(atual);
  return TRUE;
}
```

```
int RemoveFim(ListaDuplamenteEncadeada *lista, Elem
*elem){
  No *atual;

  if (EVazia(*lista)) return FALSE;

  atual = lista->fim;
  if (lista->inicio == lista->fim)
    lista->inicio = lista->fim = NULL;
  else{
    atual->anterior->proximo = NULL;
    lista->fim = atual->anterior;
  }

  *elem = atual->elemento;
  free(atual);
  return TRUE;
}

int Remove(ListaDuplamenteEncadeada *lista, char
*nome, Elem *elem){
  No *atual;

  if (EVazia(*lista)) return FALSE;
  atual = lista->inicio;
  while (atual != NULL){
    if (stricmp(atual->elemento.nome, nome) == 0){
      if (atual == lista->inicio)
           return RemoveInicio(lista, elem);
      else if (atual == lista->fim)
           return RemoveFim(lista, elem);
      atual->anterior->proximo = atual->proximo;
      atual->proximo->anterior = atual->anterior;

      *elem = atual->elemento;
      free(atual);
      return TRUE;
    }
    atual = atual->proximo;
  }
  return FALSE;
}
```

Linguagem Java: No.java (nó duplamente encadeado)

```
public class No <T>{
   private T elemento;
   private No<T> anterior, proximo;

   public No(T elemento) {
      this.elemento = elemento;
      this.anterior = this.proximo = null;
   }

   public T getElemento() {
      return elemento;
   }

   public void setElemento(T elemento) {
      this.elemento = elemento;
   }

   public No<T> getAnterior() {
      return anterior;
   }

   public No<T> getProximo() {
      return proximo;
   }

   public void setAnterior(No<T> proximo) {
      this.anterior = proximo;
   }

   public void setProximo(No<T> proximo) {
      this.proximo = proximo;
   }

   public String toString(){
      return String.format("%s", this.elemento);
   }
}
```

Linguagem Java: ListaDuplamenteEncadeada.java

```
import java.util.Arrays;
import java.util.Iterator;


public class ListaDuplamenteEncadeada <T> {
   private No<T> inicio, fim;

   public ListaDuplamenteEncadeada(){
      this.inicio = this.fim = null;
   }

   public boolean isCheia() {
      throw new NaoSeAplicaException("Lista
encadeada");
   }
   public boolean isVazia() {
      return this.inicio == null;
   }

   public int getQuantidade() {
      int qtde = 0;
      No<T> atual = this.inicio;
      while(atual != null){
         ++qtde;
          atual = atual.getProximo();
      }
      return qtde;
   }

   public void inserirInicio(T elem) {
      No<T> no = new No<T>(elem);

      if (this.isVazia()) this.fim = no;
      else {
         this.inicio.setAnterior(no);
         no.setProximo(this.inicio);
      }
      this.inicio = no;
   }
```

```
  public void inserirFim(T elem) {
    if (this.isVazia()) this.inserirInicio(elem);
    else{
      No<T> no = new No<T>(elem);
      no.setProximo(null);
      this.fim.setProximo(no);
      no.setAnterior(this.fim);
      this.fim = no;
    }
  }


  public Iterator<T> get() {
    int i = 0;

    @SuppressWarnings("unchecked")
    T[] vetor = (T[]) new
Object[this.getQuantidade()];

    No<T> atual = this.inicio;
    while(atual != null) {
      vetor[i++] = atual.getElemento();
      atual = atual.getProximo();
    }

    return Arrays.asList(vetor).iterator();
  }


  public int get(T elem) {
    int i = 0;
    No<T> atual = this.inicio;
    while (atual != null){
      if (atual.getElemento().equals(elem)) return
i;
      atual = atual.getProximo();
      i++;
    }
    throw new ElementoNaoExisteException();
  }
```

```
public T removerInicio() {
   if (this.isVazia())
     throw new ListaVaziaException();

   T elemento = this.inicio.getElemento();

   if (this.inicio == this.fim)
     this.inicio = this.fim = null;
   else{
     this.inicio.getProximo().setAnterior(null);
     this.inicio = this.inicio.getProximo();
   }

   return elemento;
}



public T removerFim() {
   if (this.isVazia())
      throw new ListaVaziaException();

   T elemento = this.fim.getElemento();

   if (this.inicio == this.fim)
     this.inicio = this.fim = null;
   else{
     this.fim.getAnterior().setProximo(null);
     this.fim = this.fim.getAnterior();
   }

   return elemento;
}
```

```
  public void remover(T elem) {
    if (this.isVazia())
      throw new ListaVaziaException();
    No<T> atual = this.inicio;
    while (atual != null){
      if (atual.getElemento().equals(elem)){
           if (atual.getAnterior() == null)
                this.removerInicio();
           else if (atual.getProximo() == null)
                this.removerFim();
           else{
atual.getAnterior().setProximo(atual.getProximo());
atual.getProximo().setAnterior(atual.getAnterior());
           }
           return;
      }
      atual = atual.getProximo();
    }
    throw new ElementoNaoExisteException();
  }
}
```

Apêndice D – Pilha Estática

Linguagem C: pilha_estatica.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

typedef struct{
  char nome[31];
  int idade;
}Elem;

typedef struct {
  Elem *elementos;
  int tamanho, topo;
}Pilha;

int Cria(Pilha *pilha, int tamanho);
int Vazia(Pilha pilha);
int Cheia(Pilha pilha);
int Quantidade(Pilha pilha);
int Pop(Pilha *pilha, Elem *elem);
int Push(Pilha *pilha, Elem elem);
int Topo(Pilha pilha, Elem *elem);
void Imprimir(Pilha pilha);
int Pesquisar(Pilha pilha, char *nome);
```

Linguagem C: pilha_estatica.c

```
#include "pilha_estatica.h"

int Cria(Pilha *pilha, int tamanho){
  if ((pilha->elementos = calloc(tamanho,
sizeof(Elem))) == NULL)
     return FALSE;
  pilha->tamanho = tamanho;
  pilha->topo = -1;
  return TRUE;
}
```

```
int Vazia(Pilha pilha){
   return (pilha.topo == -1);
}

int Cheia(Pilha pilha){
   return (pilha.topo + 1 == pilha.tamanho);
}

int Quantidade(Pilha pilha){
   return pilha.topo + 1;
}

int Pop(Pilha *pilha, Elem *elem){
   if (Vazia(*pilha)) return FALSE;
   *elem = pilha->elementos[pilha->topo--];
   return TRUE;
}

int Push(Pilha *pilha, Elem elem){
   if (Cheia(*pilha)) return FALSE;
   pilha->elementos[++pilha->topo] = elem;
   return TRUE;
}

int Topo(Pilha pilha, Elem *elem){
   if (Vazia(pilha)) return FALSE;
   *elem = pilha.elementos[pilha.topo];
   return TRUE;
}

void Imprimir(Pilha pilha){
   int i;
   for(i = pilha.topo; i >= 0; --i){
     printf("Nome: %s - ", pilha.elementos[i].nome);
     printf("Idade: %d\n", pilha.elementos[i].idade);

   }
}
```

```
int Pesquisar(Pilha pilha, char *nome){
  int i;
  for (i = 0; i <= pilha.topo; ++i)
    if (stricmp(pilha.elementos[i].nome, nome) == 0)
      return i;
  return -1;
}
```

Linguagem Java: pilha_estatica.java

```
import java.util.Arrays;
import java.util.Iterator;

public class PilhaEstatica <T>{
  private T[] elementos;
  private int topo;

  @SuppressWarnings("unchecked")
  public PilhaEstatica(int tamanho) {
    this.elementos = (T[]) new Object[tamanho];
    this.topo = -1;
  }

  public boolean isCheia(){
    return this.topo + 1 == this.elementos.length;
  }

  public boolean isVazia(){
    return this.topo == - 1;
  }

  public int getQuantidade(){
    return this.topo + 1;
  }

  public void push(T novo) throws
PilhaCheiaException{
    if (this.isCheia())
       throw new PilhaCheiaException();
    this.elementos[++this.topo] = novo;
  }
```

```
  public T pop() throws PilhaVaziaException{
    if (this.isVazia())
      throw new PilhaVaziaException();
    return this.elementos[this.topo--];
  }

  public Iterator<T> get(){
    @SuppressWarnings("unchecked")
    T[] auxiliar = (T[]) new Object[this.topo + 1];
    for(int i = this.topo; i >= 0; i--)
      auxiliar[this.topo - i] = this.elementos[i];
    return Arrays.asList(auxiliar).iterator();
  }

  public boolean get(T elemento) {
    for (int i = 0; i <= this.topo; ++i)
      if (this.elementos[i].equals(elemento))
         return true;
    return false;
  }
}
```

Apêndice E – Pilha Encadeada

Linguagem C: pilha_encadeada.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

typedef struct{
  char nome[31];
  int idade;
}Elem;

typedef struct no{
  Elem elemento;
  struct no *proximo;
}No;

typedef struct{
  No *topo;
}Pilha;

void Cria(Pilha *pilha);
int Vazia(Pilha pilha);
int Cheia(Pilha pilha);
int Quantidade(Pilha pilha);

int Pop(Pilha *pilha, Elem *elem);
int Push(Pilha *pilha, Elem novo_elemento);
int Topo(Pilha pilha, Elem *elem);

void Imprime(Pilha pilha);
int Pesquisar(Pilha pilha, char *nome);
```

Linguagem C: pilha_encadeada.c

```
#include "pilha_encadeada.h"


void Cria(Pilha *pilha){
   pilha->topo = NULL;
}


int Vazia(Pilha pilha){
   return pilha.topo == NULL;
}


int Cheia(Pilha pilha){
   return FALSE;
}


int Quantidade(Pilha pilha){
   int qtde = 0;
   No *atual = pilha.topo;
   while(atual != NULL){
      ++qtde;
      atual = atual->proximo;
   }
   return qtde;
}


int Pop(Pilha *pilha, Elem *elem){
   No *atual;

   if (Vazia(*pilha)) return FALSE;
   atual = pilha->topo;
   pilha->topo = atual->proximo;
   *elem = atual->elemento;
   free(atual);
   return TRUE;
}
```

```
int Push(Pilha *pilha, Elem novo_elemento){
  No *novo;

  if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;
  novo->elemento = novo_elemento;
  novo->proximo = pilha->topo;
  pilha->topo = novo;
  return TRUE;
}

int Topo(Pilha pilha, Elem *elem){
  if (Vazia(pilha)) return FALSE;
  *elem = pilha.topo->elemento;
  return TRUE;
}

void Imprime(Pilha pilha){
  No *atual = pilha.topo;
  while (atual != NULL){
     printf("Nome: %s - ", atual->elemento.nome);
     printf("Idade: %d\n", atual->elemento.idade);
     atual = atual->proximo;
  }
}

int Pesquisar(Pilha pilha, char *nome){
  No *atual = pilha.topo;
  while (atual != NULL){
     if (stricmp(atual->elemento.nome, nome))
       return TRUE;
     atual = atual->proximo;
  }
  return FALSE;
}
```

Linguagem Java: PilhaEncadeada.java

```
import java.util.Arrays;
import java.util.Iterator;


public class PilhaEncadeada <T>{
   private No<T> topo;

   public PilhaEncadeada(){
      this.topo = null;
   }

   public boolean isVazia(){
      return this.topo == null;
   }

   public int getQuantidade(){
      int qtde = 0;
      No<T> atual = this.topo;
      while (atual != null){
         ++qtde;
         atual = atual.getProximo();
      }
      return qtde;
   }

   public void push(T elemento){
      No<T> novo = new No<T>(elemento);
      novo.setProximo(this.topo);
      this.topo = novo;
   }

   public T pop() throws PilhaVaziaException{
      T elem;
      if (this.isVazia())
         throw new PilhaVaziaException();
      elem = this.topo.getElemento();
      this.topo = this.topo.getProximo();
      return elem;
   }
```

```
  public Iterator<T> get() {
    int i = 0;

    @SuppressWarnings("unchecked")
    T[] vetor = (T[]) new
Object[this.getQuantidade()];

    No<T> atual = this.topo;
    while(atual != null) {
      vetor[i++] = atual.getElemento();
      atual = atual.getProximo();
    }

    return Arrays.asList(vetor).iterator();
  }


  public int get(T elem) {
    int i = 0;
    No<T> atual = this.topo;
    while (atual != null){
      if (atual.getElemento().equals(elem))
         return i;
      atual = atual.getProximo();
      i++;
    }
    throw new ElementoNaoExisteException();
  }


  public T getTopo() throws PilhaVaziaException{
    if (this.isVazia())
       throw new PilhaVaziaException();
    return this.topo.getElemento();
  }
}
```

Apêndice F – Fila Estática

Linguagem C: fila_estatica.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

typedef struct{
   char nome[31];
   int idade;
}Elem;

typedef struct {
   Elem *elementos;
   int tamanho, inicio, fim;
}Fila;

int Cria(Fila *fila, int tamanho);
int Vazia(Fila fila);
int Cheia(Fila fila);
int Quantidade(Fila fila);
int Insere(Fila *fila, Elem elem);
int Remove(Fila *fila, Elem *elem);
int VerInicio(Fila fila, Elem *elem);
int VerFim(Fila fila, Elem *elem);
int Buscar(Fila fila, char *nome);
void Imprime(Fila fila);
```

Linguagem C: fila_estatica.c

```
#include "fila_estatica.h"

int Cria(Fila *fila, int tamanho){
   fila->elementos = calloc(tamanho, sizeof(Elem));
   if (fila->elementos == NULL) return FALSE;
   fila->inicio = fila->fim = 0;
   fila->tamanho = tamanho;
   return TRUE;
}
```

```
int Vazia(Fila fila){
  return (fila.inicio == fila.fim);
}

int Cheia(Fila fila){
  return (((fila.fim + 1) % fila.tamanho) ==
fila.inicio);
}


int Quantidade(Fila fila){
  int aux = fila.inicio, qtde = 0;
  while (aux != fila.fim){
    ++qtde;
    aux = (++aux)%fila.tamanho;
  }
  return qtde;
}


int Insere(Fila *fila, Elem elem){
  if (Cheia(*fila)) return FALSE;
  fila->elementos[fila->fim] = elem;
  fila->fim = (++fila->fim % fila->tamanho);
  return TRUE;
}


int Remove(Fila *fila, Elem *elem){
  if (Vazia(*fila)) return FALSE;
  *elem = fila->elementos[fila->inicio];
  fila->inicio = ++fila->inicio % fila->tamanho;
  return TRUE;
}


int VerInicio(Fila fila, Elem *elem){
  if (Vazia(fila)) return FALSE;
  *elem = fila.elementos[fila.inicio];
  return TRUE;
}
```

```
int VerFim(Fila fila, Elem *elem){
   int aux;
   if (Vazia(fila)) return FALSE;
   if ((aux = fila.fim - 1) < 0) aux += fila.tamanho;
   *elem = fila.elementos[aux];
   return TRUE;
}

int Buscar(Fila fila, char *nome){
   int i = fila.inicio;
   while (i != fila.fim){
      if (stricmp(fila.elementos[i].nome, nome) == 0)
         return i;
      i = (++i)%fila.tamanho;
   }
   return -1;
}

void Imprimir(Fila fila){
   int i = fila.inicio;
   while (i != fila.fim){
      printf("Nome: %s - ", fila.elementos[i].nome);
      printf("Idade: %d\n", fila.elementos[i].idade);
      i = (++i)%fila.tamanho;
   }
}
```

Linguagem Java: fila_estatica.java

```
import java.util.Arrays;
import java.util.Iterator;

public class FilaEstatica<T>{
   private T[] elementos;
   private int inicio, fim;

   @SuppressWarnings("unchecked")
   public FilaEstatica(int tamanho){
      this.elementos = (T[]) new Object[tamanho];
      this.inicio = this.fim = 0;
   }
```

```
  public boolean isVazia(){
    return this.inicio == this.fim;
  }


  public boolean isCheia(){
    return ((this.fim + 1) % this.elementos.length)
== this.inicio;
  }


  public int getQuantidade() {
    int aux = this.inicio, qtde = 0;
    while (aux != this.fim){
      ++qtde;
      this.fim = (++this.fim) %
this.elementos.length;
    }
    return qtde;
  }


  public void inserir(T elem) throws
FilaCheiaException {
    if (this.isCheia())
      throw new FilaCheiaException();
    this.elementos[this.fim] = elem;
    this.fim = (++this.fim) % this.elementos.length;
  }


  public Iterator<T> get() {
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new
Object[this.getQuantidade()];
    int aux = this.inicio, i = 0;
    while (aux != this.fim){
      temp[i++] = this.elementos[aux];
      aux = ++aux % this.elementos.length;
    }
    return Arrays.asList(temp).iterator();
  }
```

```
  public int get(T elem) {
    int aux = this.inicio;
    while (aux != this.fim){
      if (this.elementos[aux].equals(elem))
         return aux;
      aux = ++aux % this.elementos.length;
    }
    return -1;
  }


  public T getInicio() throws FilaVaziaException{
    if (this.isVazia())
      throw new FilaVaziaException();
    return this.elementos[this.inicio];
  }

  public T getFim() throws FilaVaziaException{
    int aux;
    if (this.isVazia())
      throw new FilaVaziaException();
    if ((aux = this.fim - 1) < 0)
      aux += this.elementos.length;
    return this.elementos[aux];
  }

  public T remover() throws FilaVaziaException {
    T elem;
    if (this.isVazia())
      throw new FilaVaziaException();
    elem = this.elementos[this.inicio];
    this.inicio = ++this.inicio %
this.elementos.length;
    return elem;
  }
}
```

Apêndice G – Fila Encadeada

Linguagem C: fila_encadeada.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

typedef struct{
  char nome[31];
  int idade;
}Elem;

typedef struct no{
  Elem elemento;
  struct no *proximo;
}No;

typedef struct{
  No *inicio, *fim;
}Fila;

void Cria(Fila *fila);
int Vazia(Fila fila);
int Cheia(Fila fila);
int Quantidade(Fila fila);

int Insere(Fila *fila, Elem elem);
int Remove(Fila *fila, Elem *elem);

int VerInicio(Fila fila, Elem *elem);
int VerFim(Fila fila, Elem *elem);

void Imprime(Fila fila);
```

Linguagem C: fila_encadeada.c

```
#include "fila_encadeada.h"


void Cria(Fila *fila){
   fila->inicio = fila->fim = NULL;
}


int Vazia(Fila fila){
   return (fila.inicio == NULL);
}


int Cheia(Fila fila){
   return FALSE;
}


int Quantidade(Fila fila){
   int qtde = 0;
   No *aux = fila.inicio;
   while (aux != NULL){
      ++qtde;
      aux = aux->proximo;
   }
   return qtde;
}


int Insere(Fila *fila, Elem elem){
   No *novo;
   if (Cheia(*fila)) return FALSE;
   if ((novo = malloc(sizeof(No))) == NULL)
      return FALSE;
   novo->elemento = elem;
   novo->proximo = NULL;
   if (Vazia(*fila)) fila->inicio = novo;
   else fila->fim->proximo = novo;
   fila->fim = novo;
   return TRUE;
}
```

```
int Remove(Fila *fila, Elem *elem){
  No *aux;
  if (Vazia(*fila)) return FALSE;
  *elem = fila->inicio->elemento;
  fila->inicio = fila->inicio->proximo;
  if (fila->inicio == NULL) fila->fim = NULL;
  return TRUE;
}

int VerInicio(Fila fila, Elem *elem){
  if (Vazia(fila)) return FALSE;
  *elem = fila.inicio->elemento;
  return TRUE;
}

int VerFim(Fila fila, Elem *elem){
  if (Vazia(fila)) return FALSE;
  *elem = fila.fim->elemento;
  return TRUE;
}

void Imprime(Fila fila){
  No *aux = fila.inicio;
  while (aux != NULL){
    printf("Nome: %s - ", aux->elemento.nome);
    printf("Idade: %d\n", aux->elemento.idade);
    aux = aux->proximo;
  }
}

int Buscar(Fila fila, char *nome){
  No *aux = fila.inicio;
  while (aux != NULL){
    if (stricmp(aux->elemento.nome, nome) == 0)
      return TRUE;
    aux = aux->proximo;
  }
  return FALSE;
}
```

Linguagem Java: FilaEncadeada.java

```
import java.util.Arrays;
import java.util.Iterator;


public class FilaEncadeada <T>{
   private No<T> inicio, fim;


   public FilaEncadeada(){
      this.inicio = this.fim = null;
   }


   public boolean isVazia() {
      return this.inicio == null;
   }


   public int getQuantidade() {
      No<T> atual = this.inicio;
      int qtde = 0;
      while (atual != null){
         ++qtde;
         atual = atual.getProximo();
      }

      return qtde;
   }


   public void inserir(T elem) {
      No<T> novo = new No<T>(elem);

      if (this.isVazia()) this.inicio = novo;
      else this.fim.setProximo(novo);
      this.fim = novo;
   }
```

```
  public Iterator<T> get() {
    int i = 0;

    @SuppressWarnings("unchecked")
    T[] vetor = (T[]) new
Object[this.getQuantidade()];

    No<T> atual = this.inicio;
    while(atual != null) {
      vetor[i++] = atual.getElemento();
      atual = atual.getProximo();
    }

    return Arrays.asList(vetor).iterator();
  }


  public int get(T elem) {
    int i = 0;
    No<T> atual = this.inicio;
    while (atual != null){
      if (atual.getElemento().equals(elem))
        return i;
      atual = atual.getProximo();
      i++;
    }
    throw new ElementoNaoExisteException();
  }


  public T remover() throws FilaVaziaException {
    if (this.isVazia())
       throw new FilaVaziaException();
    T elem = this.inicio.getElemento();

    if (this.inicio == this.fim)
      this.inicio = this.fim = null;
    this.inicio = this.inicio.getProximo();
    return elem;
  }
```

```
  public T getInicio() throws FilaVaziaException{
    if (this.isVazia())
       throw new FilaVaziaException();
    return this.inicio.getElemento();
  }

  public T getFim() throws FilaVaziaException{
    if (this.isVazia())
       throw new FilaVaziaException();
    return this.fim.getElemento();
  }
}
```

Apêndice H – Deque Estático

Linguagem C: deque_estatico.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

typedef struct{
  char nome[31];
  int idade;
}Elem;

typedef struct {
  Elem *elementos;
  int tamanho, esquerda, direita;
}Deque;

int Criar(Deque *deque, int tamanho);
int Vazia(Deque deque);
int Cheia(Deque deque);
int Quantidade(Deque deque);

int Inserir_Direita(Deque *deque, Elem elem);
int Inserir_Esquerda(Deque *deque, Elem elem);
int Remover_Direita(Deque *deque, Elem *elem);
int Remove_Esquerda(Deque *deque, Elem *elem);

int Ver_Direita(Deque deque, Elem *elem);
int Ver_Esquerda(Deque deque, Elem *elem);

void Imprimir_Direita(Deque deque);
void Imprimir_Esquerda(Deque deque);
```

Linguagem C: deque_estatico.c

```
#include "deque_estatico.h"


int Criar(Deque *deque, int tamanho){
   deque->elementos = calloc(tamanho, sizeof(Elem));
   if (deque->elementos == NULL) return FALSE;
   deque->esquerda = deque->direita = 0;
   deque->tamanho = tamanho;
   return TRUE;
}


int Vazia(Deque deque){
   return (deque.esquerda == deque.direita);
}


int Cheia(Deque deque){
   return (((deque.direita + 1) % deque.tamanho) ==
deque.esquerda);
}


int Quantidade(Deque deque){
   int qtde = 0, aux = deque.esquerda;
   while (aux != deque.direita){
      ++qtde;
      aux = ++aux%deque.tamanho;
   }
   return qtde;
}


int Inserir_Direita(Deque *deque, Elem elem){
   if (Cheia(*deque)) return FALSE;
   deque->direita = ++deque->direita % deque->tamanho;
   deque->elementos[deque->direita] = elem;
   return TRUE;
}
```

```
int Inserir_Esquerda(Deque *deque, Elem elem){
  if (Cheia(*deque)) return FALSE;
  deque->elementos[deque->esquerda] = elem;
  if((deque->esquerda = --deque->esquerda % deque-
>tamanho) < 0)
     deque->esquerda += deque->tamanho;
  return TRUE;
}


int Remover_Direita(Deque *deque, Elem *elem){
  if (Vazia(*deque)) return FALSE;
  *elem = deque->elementos[deque->direita];
  if((deque->direita = --deque->direita % deque-
>tamanho) < 0)
     deque->direita += deque->tamanho;
  return TRUE;
}


int Remove_Esquerda(Deque *deque, Elem *elem){
  if (Vazia(*deque)) return FALSE;
  deque->esquerda = ++deque->esquerda % deque-
>tamanho;
  *elem = deque->elementos[deque->esquerda];
  return TRUE;
}


int Ver_Direita(Deque deque, Elem *elem){
  if (Vazia(deque)) return FALSE;
  *elem = deque.elementos[deque.direita];
  return TRUE;
}


int Ver_Esquerda(Deque deque, Elem *elem){
  if (Vazia(deque)) return FALSE;
  *elem = deque.elementos[(deque.esquerda + 1) %
deque.tamanho];
  return TRUE;
}
```

```
void Imprimir_Direita(Deque deque){
   int i;

   for (i = deque.direita; i != deque.esquerda;){
      printf("Nome: %s - ", deque.elementos[i].nome);
      printf("Idade: %d\n", deque.elementos[i].idade);
      if((i = --i % deque.tamanho) < 0)
         i += deque.tamanho;
   }
}

void Imprimir_Esquerda(Deque deque){
   int i = deque.esquerda + 1 % deque.tamanho;
   while (i != (deque.direita + 1) % deque.tamanho){
      printf("Nome: %s - ", deque.elementos[i].nome);
      printf("Idade: %d\n", deque.elementos[i].idade);
      i = ++i%deque.tamanho;
   }
}
```

Linguagem Java: DequeEstatico.java

```
import java.util.Arrays;
import java.util.Iterator;

public class DequeEstatico <T>{
   private T[] elementos;
   private int esquerda, direita;

   @SuppressWarnings("unchecked")
   public DequeEstatico(int tamanho) {
      this.elementos = (T[]) new Object[tamanho];
      this.esquerda = this.direita = 0;
   }

   public boolean isVazia(){
      return this.esquerda == this.direita;
   }
```

```
  public boolean isCheia(){
     return (((this.direita + 1) %
this.elementos.length) == this.esquerda);
  }

  public int getQuantidade(){
     int qtde = 0, aux = this.esquerda;
     while (aux != this.direita){
        ++qtde;
        aux = ++aux % this.elementos.length;
     }
     return qtde;
  }

  public void inserirDireita(T novo) throws
DequeCheioException{
     if (this.isCheia())
         throw new DequeCheioException();
     this.direita = ++this.direita %
this.elementos.length;
     this.elementos[this.direita] = novo;
  }

  public void inserirEsquerda(T novo) throws
DequeCheioException{
     if (this.isCheia())
         throw new DequeCheioException();
     this.elementos[this.esquerda] = novo;
     if ((this.esquerda=--this.esquerda %
this.elementos.length)< 0)
        this.esquerda += this.elementos.length;
  }

  public T removerDireita() throws
DequeVazioException{
     if (this.isVazia())
         throw new DequeVazioException();
     T elem = this.elementos[this.direita];
     if ((this.direita=--this.direita %
this.elementos.length) < 0)
        this.direita += this.elementos.length;
     return elem;
  }
```

```
  public T removerEsquerda() throws
DequeVazioException{
     if (this.isVazia())
        throw new DequeVazioException();
     this.esquerda = ++this.esquerda %
this.elementos.length;
     return this.elementos[this.esquerda];
  }

  public T getElementoDireita() throws
DequeVazioException{
     if (this.isVazia())
       throw new DequeVazioException();
     return this.elementos[this.direita];
  }

  public T getElementoEsquerda() throws
DequeVazioException{
     if (this.isVazia())
        throw new DequeVazioException();
     return
this.elementos[(this.esquerda+1)%this.elementos.lengt
h];
  }

  public Iterator<T> getDireita(){
     @SuppressWarnings("unchecked")
     T[] aux = (T[]) new
Object[this.getQuantidade()];
     int i = this.direita, j = 0;

     while (i != this.esquerda){
       aux[j++] = this.elementos[i];
       if ((i = --i % this.elementos.length) < 0)
            i += this.elementos.length;
     }

     return Arrays.asList(aux).iterator();
  }
```

```
  public Iterator<T> getEsquerda(){
    @SuppressWarnings("unchecked")
    T[] aux = (T[]) new
Object[this.getQuantidade()];
    int i = this.esquerda + 1 %
this.elementos.length, j = 0;

    while (i != (this.direita + 1) %
this.elementos.length){
      aux[j++] = this.elementos[i];
      i = ++i % this.elementos.length;
    }
    return Arrays.asList(aux).iterator();
  }
}
```

Apêndice I – Deque Encadeado

Linguagem C: deque_encadeado.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

typedef struct{
   char nome[31];
   int idade;
}Elem;

typedef struct no{
   Elem elemento;
   struct no *anterior;
   struct no *proximo;
}No;

typedef struct {
   No *esquerda, *direita;
}TDeque;

void Cria(TDeque *deque);
int Vazia(TDeque deque);
int Cheia(TDeque deque);
int Quantidade(TDeque deque);

int Inserir_Direita(TDeque *deque, Elem elem);
int Inserir_Esquerda(TDeque *deque, Elem elem);

int Remover_Direita(TDeque *deque, Elem *elem);
int Remover_Esquerda(TDeque *deque, Elem *elem);

int Ver_Direita(TDeque deque, Elem *elem);
int Ver_Esquerda(TDeque deque, Elem *elem);

void Imprimir_Direita(TDeque deque);
void Imprimir_Esquerda(TDeque deque);
```

```
int Buscar_Esquerda(TDeque deque, char *nome);
int Buscar_Direita(TDeque deque, char *nome);
```

Linguagem C: deque_encadeado.c

```
#include "deque_encadeado.h"

void Cria(TDeque *deque){
  deque->esquerda = deque->direita = NULL;
}

int Vazia(TDeque deque){
  if (deque.esquerda == NULL) return TRUE;
  else return FALSE;
}

int Cheia(TDeque deque){
  return FALSE;
}

int Quantidade(TDeque deque){
  No *aux;
  int qtde = 0;
  for (aux = deque.esquerda; aux != NULL; aux = aux-
>proximo) ++qtde;
  return qtde;
}

int Inserir_Direita(TDeque *deque, Elem elem){
  No *novo;

  if (Cheia(*deque)) return FALSE;
  if ((novo = malloc(sizeof(No))) == NULL)
     return FALSE;
  novo->elemento = elem;
  novo->proximo = NULL;
  novo->anterior = deque->direita;
  if (Vazia(*deque)) deque->esquerda = novo;
  else deque->direita->proximo = novo;
  deque->direita = novo;
  return TRUE;
}
```

```
int Inserir_Esquerda(TDeque *deque, Elem elem){
  No *novo;

  if (Cheia(*deque)) return FALSE;

  if ((novo = malloc(sizeof(No))) == NULL)
     return FALSE;

  novo->elemento = elem;
  novo->proximo = deque->esquerda;
  novo->anterior = NULL;

  if (Vazia(*deque)) deque->direita = novo;

  else deque->esquerda->anterior = novo;
  deque->esquerda = novo;

  return TRUE;
}


int Remover_Esquerda(TDeque *deque, Elem *elem){
  No *atual;

  if (Vazia(*deque)) return FALSE;

  atual = deque->esquerda;
  if (deque->direita == deque->esquerda)
    deque->direita = deque->esquerda = NULL;
  else{
    deque->esquerda->proximo->anterior = NULL;
    deque->esquerda = deque->esquerda->proximo;
  }

  *elem = atual->elemento;

  free(atual);

  return TRUE;
}
```

```
int Remover_Direita(TDeque *deque, Elem *elem){
  No *atual;

  if (Vazia(*deque)) return FALSE;
  atual = deque->direita;
  *elem = atual->elemento;

  if (deque->direita == deque->esquerda)
    deque->direita = deque->esquerda = NULL;
  else{
    deque->direita->anterior->proximo = NULL;
    deque->direita = deque->direita->anterior;

  }

  free(atual);
  return TRUE;
}


int Ver_Esquerda(TDeque deque, Elem *elem){
  if (Vazia(deque)) return FALSE;
  *elem = deque.esquerda->elemento;
  return TRUE;
}


int Ver_Direita(TDeque deque, Elem *elem){
  if (Vazia(deque)) return FALSE;
  *elem = deque.direita->elemento;
  return TRUE;
}


void Imprimir_Esquerda(TDeque deque){
  No *aux;
  for (aux = deque.esquerda; aux != NULL; aux = aux-
>proximo)
    printf("%s (%d)\n", aux->elemento.nome, aux-
>elemento.idade);
}
```

```
void Imprimir_Direita(TDeque deque){
   No *aux;
   for (aux = deque.direita; aux != NULL; aux = aux-
>anterior)
      printf("%s (%d)\n", aux->elemento.nome, aux-
>elemento.idade);
}

int Buscar_Direita(TDeque deque, char *nome){
   No *aux;
   for (aux = deque.direita; aux != NULL; aux = aux-
>anterior)
      if (stricmp(aux->elemento.nome, nome) == 0)
return TRUE;
   return FALSE;
}

int Buscar_Esquerda(TDeque deque, char *nome){
   No *aux;
   for (aux = deque.esquerda; aux != NULL; aux = aux-
>proximo)
      if (stricmp(aux->elemento.nome, nome) == 0)
return TRUE;
   return FALSE;
}
```

Linguagem Java: DequeEncadeado.java

```
import java.util.Arrays;
import java.util.Iterator;


public class DequeEncadeado <T>{
   public No<T> esquerda, direita;

   public DequeEncadeado(){
      this.esquerda = this.direita = null;
   }

   public boolean isVazio(){
      return this.esquerda == null;
   }
```

```
    public int getQuantidade(){
    No<T> aux;
    int qtde = 0;
    aux = this.esquerda;
    while (aux != null){
      ++qtde;
      aux = aux.getProximo();
    }
    return qtde;
  }

  public void inserirDireita(T elem){
    No<T> novo = new No<T>(elem);
    novo.setAnterior(this.direita);
    if (this.isVazio()) this.esquerda = novo;
    else this.direita.setProximo(novo);
    this.direita = novo;
  }

  public void inserirEsquerda(T elem){
    No<T> novo = new No<T>(elem);
    novo.setProximo(this.esquerda);
    if (this.isVazio()) this.direita = novo;
    else this.esquerda.setAnterior(novo);
    this.esquerda = novo;
  }

  public T removerEsquerda() throws
DequeVazioException{
    No<T> atual = this.esquerda;
    if (this.isVazio())
      throw new DequeVazioException();
    if (this.direita == this.esquerda)
      this.direita = this.esquerda = null;
    else{
      this.esquerda.getProximo().setAnterior(null);
      this.esquerda = this.esquerda.getProximo();
    }
    return atual.getElemento();
  }
```

```
  public T removerDireita() throws
DequeVazioException{
     No<T> atual = this.direita;
     if (this.isVazio())
       throw new DequeVazioException();
     if (this.direita == this.esquerda)
       this.direita = this.esquerda = null;
     else{
       this.direita.getAnterior().setProximo(null);
       this.direita = this.direita.getAnterior();
     }
     return atual.getElemento();
  }


  public T getElementoDireita() throws
DequeVazioException{
     if (this.isVazio())
       throw new DequeVazioException();
     return this.direita.getElemento();
  }


  public T getElementoEsquerda() throws
DequeVazioException{
     if (this.isVazio())
       throw new DequeVazioException();
     return this.esquerda.getElemento();
  }


  public Iterator<T> getDireta(){
     No<T> atual = this.direita;
     @SuppressWarnings("unchecked")
     T[] aux = (T[]) new
Object[this.getQuantidade()];
     int i = 0;
     while (atual != null){
       aux[i++] = atual.getElemento();
       atual = atual.getAnterior();
     }return Arrays.asList(aux).iterator();
  }
```

```
  public Iterator<T> getEsquerda(){
    No<T> atual = this.esquerda;
    @SuppressWarnings("unchecked")
    T[] aux = (T[]) new
Object[this.getQuantidade()];
    int i = 0;
    while (atual != null){
      aux[i++] = atual.getElemento();
      atual = atual.getProximo();
    }return Arrays.asList(aux).iterator();
  }



  public boolean getDireta(T elem){
    No<T> atual = this.direita;
    while (atual != null){
      if (atual.getElemento().equals(elem))
         return true;
      atual = atual.getAnterior();
    }return false;
  }



  public boolean getEsquerda(T elem){
    No<T> atual = this.esquerda;
    while (atual != null){
      if (atual.getElemento().equals(elem))
         return true;
      atual = atual.getProximo();
    }
    return false;
  }
}
```

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA

ISBN 978-85-63406-61-3