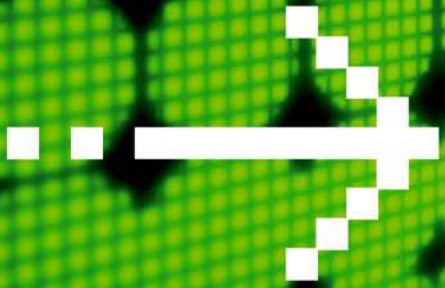
18

série livros didáticos informática ufrgs



estruturas de dados

nina edelweiss
renata galante

bookman®

estruturas de dados

as autoras

Nina Edelweiss é engenheira eletricista e doutora em Ciência da Computação pela UFRGS. Atualmente é professora colaboradora do Instituto de Informática da mesma universidade, onde atua como professora e orientadora do Programa de Pós-Graduação em Ciência da Computação. Durante muitos anos lecionou em cursos de engenharia e no bacharelado em Ciência da Computação na UFRGS, na PUCRS e na UFSC. É co-autora de três livros, tendo publicado diversos artigos em periódicos e em anais de congressos internacionais e nacionais. Desenvolve pesquisas na área de banco de dados, com ênfase em bancos de dados temporais e com versões.

Renata Galante tem doutorado em Ciência da Computação pela UFRGS e é professora adjunta no Instituto de Informática. Está envolvida com atividades de extensão, ensino de graduação e pós-graduação e orientação de alunos de mestrado e doutorado. Tem trabalhado em diversos projetos de pesquisa financiados por agências de fomento como CNPq, Finep e Fapergs. É autora de diversos artigos científicos em periódicos e anais de conferência. É co-editora da Revista de Iniciação Científica da SBC. Desenvolve pesquisa nas áreas de banco de dados, XML, teoria de banco de dados, semântica de linguagens de consulta, sistemas de informação e sistemas na web.

E21e Edelweiss, Nina.
Estruturas de dados [recurso eletrônico] / Nina Edelweiss, Renata Galante. – Dados eletrônicos. – Porto Alegre : Bookman, 2009.

Editado também como livro impresso em 2009.
ISBN 978-85-7780-450-4

Estruturas de dados. 2. Tipos de dados. 3. Árvores. 4. Árvores binárias. I. Galante, Renata. II. Título.

CDU 004.422.63

Catalogação na publicação: Renata de Souza Borges – CRB-10/Prov-021/08

nina edelweiss
renata galante

estruturas de dados

bookman®

2009

Leitura final: *Monica Stefani*

Supervisão editorial: *Arysinha Jacques Affonso*

Capa e projeto gráfico: *Tatiana Sperhacke – TAT studio*

Imagem de capa: © *iStockphoto.com/Alexey Khlobystov*

Editoração eletrônica: *Techbooks*

SÃO PAULO
Av. Angélica, 1.091 - Higienópolis
01227-100 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

apresentação

A Série *Livros Didáticos* do Instituto de Informática da Universidade Federal do Rio Grande do Sul tem como objetivo a publicação de material didático para disciplinas ministradas em cursos de graduação em Computação e Informática, ou seja, para os cursos de Bacharelado em Ciência da Computação, de Bacharelado em Sistemas de Informação, de Engenharia da Computação e de Licenciatura em Informática. A série é desenvolvida tendo em vista as Diretrizes Curriculares do MEC e é resultante da experiência dos professores do Instituto de Informática e dos colaboradores externos no ensino e na pesquisa.

Os primeiros títulos, *Fundamentos da Matemática Intervalar* e *Programando em Pascal XSC* (esgotados), foram publicados em 1997 no âmbito do Projeto Aritmética Intervalar Paralela (ArInPar), financiados pelo ProTeM - CC CNPq/ Fase II. Essas primeiras experiências serviram de base para os volumes subseqüentes, os quais se caracterizam como livros-texto para disciplinas dos cursos de Computação e Informática.

Em seus títulos mais recentes, a Série Livros Didáticos tem contado com a colaboração de professores externos que, em parceria com professores do Instituto, estão desenvolvendo livros de alta qualidade e valor didático. Hoje a série está aberta a qualquer autor de reconhecida capacidade.

O sucesso da experiência com esses livros, aliado à responsabilidade que cabe ao Instituto na formação de professores e pesquisadores em Computação e Informática, conduziu à ampliação da abrangência e à institucionalização da série.

Em 2008, um importante passo foi dado para a consolidação e ampliação de todo o trabalho: a publicação dos livros pela Artmed Editora S.A., por meio

do selo Bookman. Hoje são 15 os títulos publicados – uma lista completa, incluindo os próximos lançamentos, encontra-se nas orelhas desta obra – ampliando a oferta aos leitores da série. Sempre com a preocupação em manter nível compatível com a elevada qualidade do ensino e da pesquisa desenvolvidos no âmbito do Instituto de Informática da UFRGS e no Brasil.

Prof. Paulo Blauth Menezes
Comissão Editorial da Série Livros Didáticos
Instituto de Informática da UFRGS

prefácio

Este livro é o resultado da experiência acumulada pelas autoras ao longo de vários anos ministrando a disciplina "Estruturas de Dados" nos cursos de Bacharelado em Ciência da Computação e de Engenharia da Computação da Universidade Federal do Rio Grande do Sul (UFRGS).

O livro tem por objetivo servir de base para o ensino das estruturas de dados "listas" e "árvores" em nível de graduação na área de Computação e Informática. O conteúdo do livro é adequado para uma disciplina de um semestre, sendo assumido o conhecimento prévio de técnicas de desenvolvimento de algoritmos. Não são feitas análises de complexidade dos algoritmos nem de sua otimização: a clareza dos algoritmos é priorizada, para facilitar sua compreensão.

O conteúdo está de acordo com a proposta de Currículo de Referência da Sociedade Brasileira de Computação (SBC) para cursos de graduação na área de Computação e Informática. A disciplina "Estruturas de Dados" é um dos assuntos tratados nos Fundamentos de Computação, e consta do currículo dos cursos de Bacharelado em Ciência da Computação e de Engenharia da Computação. Nesta proposta, o conteúdo de "Estruturas de Dados" compreende:

> *Listas lineares e suas generalizações: listas ordenadas, listas encadeadas, pilhas e filas. Aplicações de listas. Árvores e suas generalizações: árvores binárias, árvores de busca, árvores balanceadas (AVL), árvores B e B+. Aplicações de árvores.*

Somente o estudo de árvores B e B+ não foi incluído neste livro, por tratar de armazenamento em memória secundária, assunto que não está sendo

tratado aqui. Todos os algoritmos apresentados consideram o armazenamento dos dados em memória principal. Com base no objetivo de servir de base para uma disciplina de um semestre, não haveria tempo hábil para incluir mais este assunto (memória secundária) com todos os seus desdobramentos. Fica a sugestão de que este assunto seja tratado em uma disciplina posterior, nomeada no Currículo de Referência de "Pesquisa e Ordenação".

Uma apostila editada em 2000 por Santos e Edelweiss, seguindo exatamente a seqüência das aulas ministradas na UFRGS, serviu de base para este livro. Nela, os textos simplesmente descreviam as lâminas projetadas nas aulas. Os conteúdos aqui apresentados são mais abrangentes. O mesmo conjunto de operações básicas é detalhado para cada uma das estruturas analisadas, passível de ser ampliado e adaptado para diferentes aplicações.

As autoras agradecem aos professores, monitores e alunos das aulas da disciplina de Estruturas de Dados, ministrada no Departamento de Informática Aplicada do Instituto de Informática da UFRGS, que utilizaram o material, apontaram problemas e propuseram melhorias.

Muito do que aqui é apresentado se deve às aulas de Clesio Saraiva dos Santos, excelente mestre e querido amigo, nosso professor em momentos diferentes de nossas trajetórias acadêmicas. A ele devemos ter despertado em nós o interesse por esta área de pesquisa, o prazer pelo desafio de desenvolver um algoritmo elegante e correto, a alegria gratificante de transmitir novos conhecimentos aos nossos alunos.

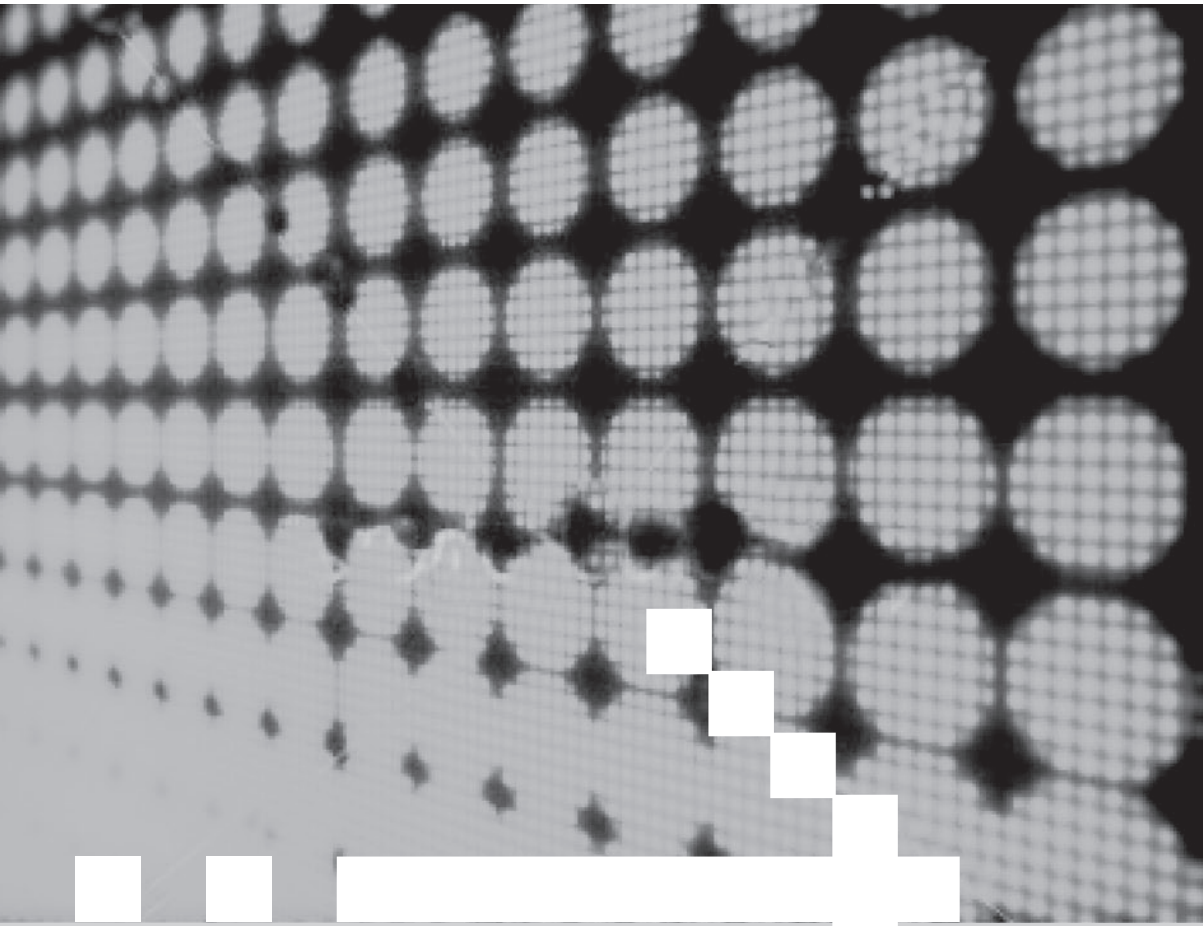
Nina Edelweiss
Renata Galante

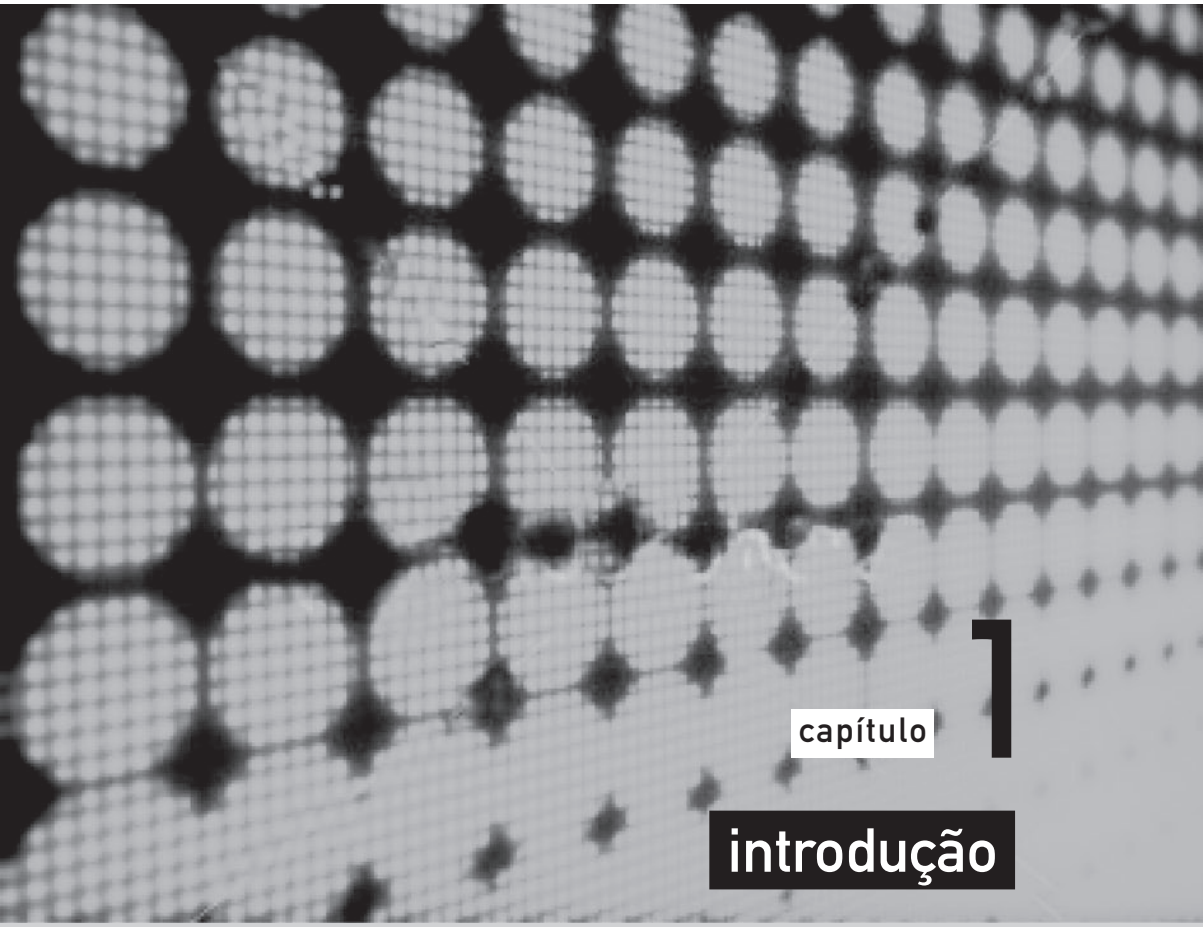
sumário

4 pilhas e filas 125

lista de algoritmos

lista de figuras





capítulo 1

introdução

A representação dos valores manipulados por uma aplicação pode ser feita por diferentes estruturas de dados. As principais linguagens de programação oferecem um elenco de tipos primitivos, como inteiro (para valores inteiros), real (para valores fracionários), lógico (para valores booleanos) e caractere (para representar caracteres). Oferecem, ainda, alguns tipos estruturados para representar conjuntos de valores, como arranjos (também denominados de vetores e matrizes), registros, conjuntos e seqüências.

Entretanto, os tipos estruturados presentes nas linguagens de programação têm grandes limitações quando se quer representar estruturas de dados mais complexas, que envolvem diferentes tipos e associações entre eles. Para tornar os programas mais eficientes, surgiu a necessidade de serem definidas estruturas de dados mais apropriadas, com a definição de operações eficientes para cada uma delas.

Neste livro são analisadas duas estruturas de dados que, embora muito comuns em programas de aplicação, não são fornecidas diretamente pelas linguagens de programação: `listas lineares` e `árvores`. Elas são compostas por conjuntos de elementos onde são armazenadas as informações, denominados `nodos`. O que as diferencia entre si é o relacionamento lógico entre estes nodos: nas `listas lineares` o relacionamento é de ordem; nas `árvores`, existe um relacionamento de subordinação.

Para cada uma destas estruturas serão analisadas formas de organizar os dados, como armazenar fisicamente a estrutura, e algoritmos que implementam as principais operações para sua manipulação.

O objetivo deste livro é servir como apoio a disciplinas de `Estruturas de Dados` em cursos de graduação em Ciência da Computação. Não se pretende, de modo algum, esgotar a análise destas estruturas de dados. Para um estudo mais aprofundado dos assuntos apresentados o leitor deve consultar as leituras recomendadas. É importante ressaltar que é considerada somente a representação de informações em memória principal. Técnicas de armazenamento em memória secundária são tratadas em outra(s) disciplina(s), como `Organização de Arquivos`.

1.1 → pré-requisitos

Para acompanhar este livro são necessários os seguintes conhecimentos prévios:

- programação básica, utilizando alguma linguagem de programação procedimental (Pascal, C, Fortran, …);
- além da utilização de variáveis simples, o conhecimento de estruturas presentes nas linguagens de programação, como arranjos e registros;
- utilização de ponteiros e de alocação dinâmica de variáveis.

1.2 → estrutura do texto

Os demais capítulos deste livro estão estruturados como segue.

No `capítulo 2` são brevemente apresentados alguns conceitos básicos, necessários para o desenvolvimento do restante do texto. Inicialmente é visto o conceito de Tipos Abstratos de Dados, utilizado em linguagens de programação e como uma das bases para as estruturas estudadas a seguir. Todos os algoritmos apresentados utilizam este conceito, o que facilita a sua implementação. Em seguida são analisadas duas formas para armazenar fisicamente as estruturas vistas a seguir. Em cada uma delas serão analisadas estas duas formas de armazenamento físico.

A partir do capítulo seguinte são detalhadas duas estruturas de dados que têm larga utilização em problemas de aplicação. São analisadas as diferentes formas de armazenamento físico e as operações básicas que podem ser executadas sobre elas, com apresentação dos algoritmos correspondentes.

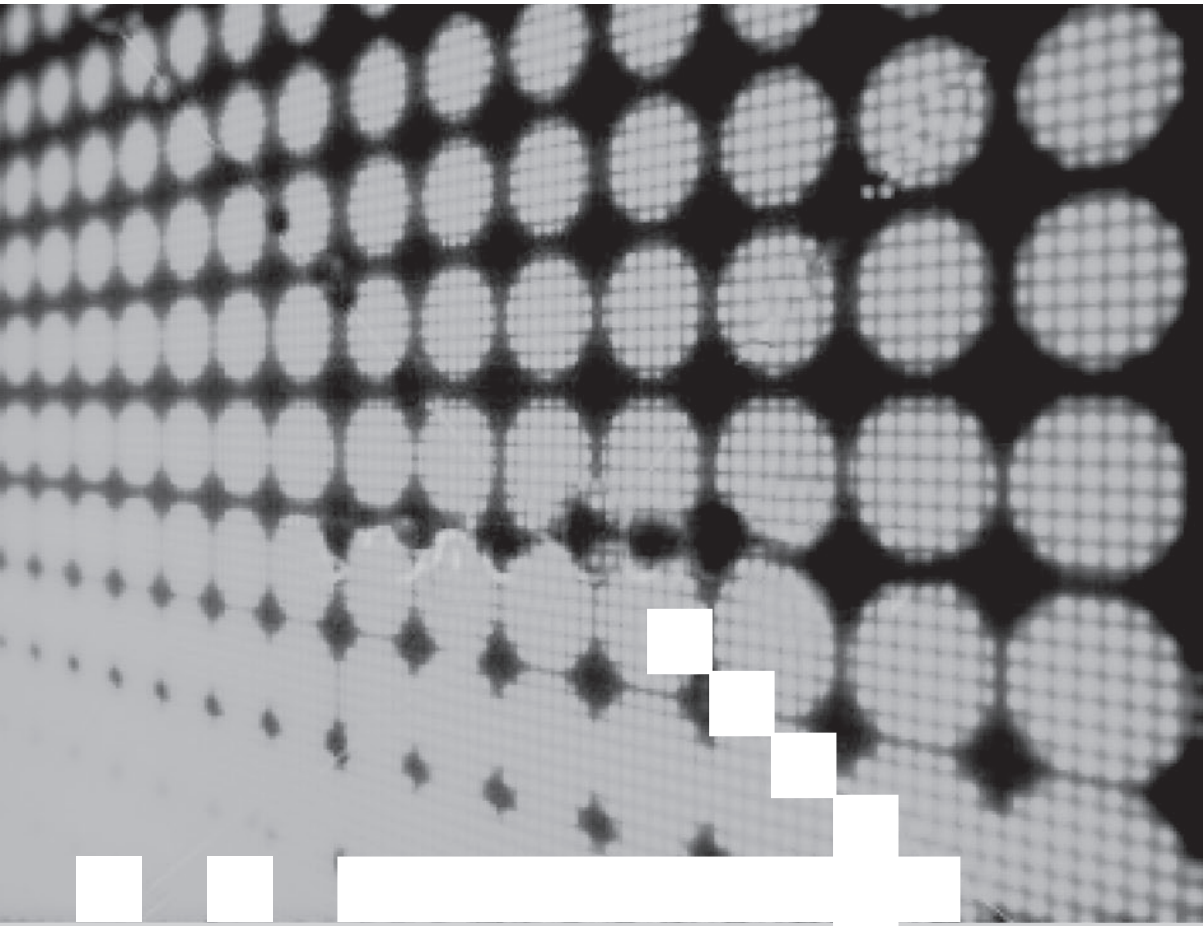
O `capítulo 3` detalha a estrutura particular denominada Lista Linear. Filas e pilhas, tipos particulares de listas com limitações na disciplina de acesso, são analisadas no `capítulo 4`.

O `capítulo 5` introduz a estrutura denominada árvore. Um tipo particular denominado árvore binária é detalhado no `capítulo 6`. Árvores Binárias de Pesquisa e árvores AVL, tipos especiais de árvores binárias, são também analisadas neste capítulo.

No final de cada capítulo é apresentada uma `lista de exercícios sugeridos`, agrupados por assunto, finalizando com algumas aplicações.

Todos os algoritmos ao longo do texto são escritos na pseudolinguagem apresentada no `Anexo`.

Finalmente, as leituras recomendadas reúnem alguns livros disponíveis no Brasil, que podem complementar os conceitos aqui apresentados. Selecionamos somente livros em português, com exceção de três (Aho 1983, Knuth 1968 e Knuth 1973) que influenciaram a maior parte das pesquisas em estruturas de dados realizadas ao longo dos últimos anos.



capítulo 2

conceitos básicos

Neste capítulo são apresentados alguns conceitos necessários ao entendimento das diferentes estruturas a serem vistas neste livro. Em primeiro lugar é feita a distinção entre tipos de dados e estruturas de dados, sendo em seguida introduzidos os Tipos Abstratos de Dados, utilizados nos algoritmos apresentados ao longo de todo o livro. Por fim, são analisadas diferentes alternativas para a representação física das estruturas a serem estudadas sem, no entanto, entrar em detalhes de implementação.

2.1 → tipos de dados e estruturas de dados

Uma aplicação em Ciência da Computação é basicamente um programa de computador que manipula dados. A representação dos dados manipulados por uma aplicação pode ser feita por diferentes estruturas de dados. Um fator que determina o papel dessas estruturas no processo de programação de aplicações é a identificação de quão bem as estruturas de dados coincidem com o domínio do problema a ser tratado. Portanto, é essencial que as linguagens de programação tenham suporte a uma variedade de tipos e estruturas, para que representem adequadamente os dados manipulados pelas aplicações. Embora os termos "tipos de dados" e "estruturas de dados" sejam semelhantes, eles têm significados diferentes.

Um `tipo de dado` consiste da definição do conjunto de valores (denominado domínio) que uma variável pode assumir ao longo da execução de um programa e do conjunto de operações que podem ser aplicadas sobre ele. Por exemplo, o tipo de dado `inteiro` pode assumir valores inteiros (negativos, zero, positivos), como `<..., -2, -1, 0, 1, 2,...>;` as operações possíveis sobre este tipo de dado são: soma, subtração, multiplicação, divisão, entre outras. As principais linguagens de programação oferecem uma grande variedade de tipos de dados, classificados em básicos e estruturados.

Os `tipos de dados básicos`, também denominados `tipos primitivos`, não possuem uma estrutura sobre seus valores, ou seja, não é possível decompor o tipo primitivo em partes menores. Os tipos básicos são, portanto, indivisíveis. Por exemplo, uma variável do tipo `lógico` somente pode assumir os valores `verdadeiro` e `falso`. As principais linguagens de programação oferecem um elenco de tipos primitivos, como `inteiro` (para valores inteiros), `real` (para valores fracionários), `lógico` (para valores booleanos) e `caractere` (para representar caracteres).

Os `tipos de dados estruturados` permitem agregar mais do que um valor em uma variável, existindo uma relação estrutural entre seus elementos. As linguagens de programação oferecem mecanismos para estruturar dados complexos, quando esses dados são compostos por diversos campos. Os principais tipos de dados estruturados fornecidos pelas linguagens de programação são: `arranjos` (também denominados vetores e matrizes, utilizados para agregar componentes do mesmo tipo, com um tamanho máximo predefinido), `registros` (para agregar componentes de tipos diferentes),

`seqüências` (para coleções ordenadas de componentes do mesmo tipo) e `conjuntos` (para definir, para um tipo básico, uma faixa de valores que seus componentes podem assumir), entre outros.

Existem ainda os `tipos de dados definidos pelo usuário`. São também tipos de dados estruturados, construídos hierarquicamente através de componentes, os quais são de fato tipos de dados primitivos e/ou estruturados. Um tipo definido pelo usuário é constituído por um conjunto de componentes, que podem ser de tipos diferentes, agrupados sob um único nome. Normalmente, os elementos desse tipo estruturado têm alguma relação semântica. Essa construção hierárquica de tipos é a maneira fornecida pelas linguagens de programação para estruturar os dados e manipulá-los de forma organizada.

Dentro desse contexto surgem as estruturas de dados que, por sua vez, especificam conceitualmente os dados, de forma a refletir um relacionamento lógico entre os dados e o domínio de problema considerado. Além disso, as estruturas de dados incluem operações para manipulação dos seus dados, que também desempenham o papel de caracterização do domínio de problema considerado. Cabe lembrar que esse nível conceitual de abstração não é fornecido diretamente pelas linguagens de programação, as quais fornecem os tipos de dados e os operadores que permitem a construção de uma estrutura de dados flexível para o problema que está sendo definido. A forma mais próxima de implementação de uma estrutura de dados é através dos tipos abstratos de dados, apresentados na próxima seção.

2.2 ···→ tipos abstratos de dados

Os `tipos abstratos de dados` (`TADs`) são estruturas de dados capazes de representar os tipos de dados que não foram previstos no núcleo das linguagens de programação e que, normalmente, são necessários para aplicações específicas. Essas estruturas são divididas em duas partes: os dados e as operações. A especificação de um TAD corresponde à escolha de uma boa maneira de armazenar os dados e à definição de um conjunto adequado de operações para atuar sobre eles.

A característica essencial de um TAD é a separação entre conceito e implementação, ou seja, existe uma distinção entre a definição do tipo e a sua

representação, e a implementação das operações. Um TAD é, portanto, uma forma de definir um novo tipo de dado juntamente com as operações que manipulam esse novo tipo de dado. As aplicações que utilizam esse TAD são denominadas clientes do tipo de dado.

Como exemplo, um tipo de dados interessante e do qual necessitamos freqüentemente é o tipo `Data`. Seguidamente precisamos fazer cálculos envolvendo datas e nem sempre isto é fácil. Por exemplo, *quantos dias existem entre 24 de abril e 2 de fevereiro?* Apesar de ser possível efetuar este cálculo, ele não é trivial, já que os meses têm número de dias diferentes. Assim, parece uma boa idéia implementar um TAD para `Data`.

Formalmente, um TAD pode ser definido como um par `(v,o)`, onde `v` representa um conjunto de valores e `o` representa o conjunto de operações aplicáveis sobre esses valores. Por exemplo, um TAD para representar uma data pode ser dado pelo par `(v,o)`, onde:

- `v` – é uma tripla formada por dia-mês-ano; e
- `o` – são as operações aplicáveis sobre o tipo `data`, como verificar se uma data é válida, calcular o dia da semana de uma determinada data, calcular a data do Carnaval de um determinado ano, entre outras.

Uma vez que o TAD foi caracterizado, o próximo passo é escolher sua estrutura de representação, ou seja, a estrutura de dados propriamente dita que irá suportar as operações definidas. A representação será uma coleção de campos primitivos ou mesmo uma estrutura complexa formada por vários campos primitivos. Considerando o TAD `Data`, podemos criar uma estrutura com três campos do tipo inteiro para representar, respectivamente, o dia, o mês e o ano, como mostrado a seguir:

```
Data = registro
          Dia: inteiro
          Mês: inteiro
          Ano: inteiro
       fim registro
```

Além da estrutura de dados, a representação do TAD envolve a especificação de um conjunto de funções e procedimentos que podem ser executados

sobre esse novo tipo de dado. Por exemplo, podemos definir as seguintes operações aplicáveis sobre o TAD `Data`:

- **`Procedimento InicializaData`**
  Entradas: `Dia, Mês, Ano (inteiro)`
  Saída: `D(Data)`
  Função que recebe três parâmetros inteiros, informando `Dia`, `Mês` e `Ano`, e retorna `D`, inicializado na data resultante;

- **`Função AcrescentaDias`**
  Entradas: `D (Data), Dias (inteiro)`
  Retorno: `(Data)`
  Soma um determinado número de `Dias` a uma data recebida como parâmetro e retorna o resultado. Caso a operação não seja possível, retorna uma data cujo dia seja `-1`;

- **`Função EscreveExtenso`**
  Entrada: `D (Data)`
  Retorno: `(lógico)`
  Recebe uma data e a escreve por extenso. Por exemplo: 10/03/2007 deve ser escrito como 10 de março de 2007. Retorna `verdadeiro` se a operação foi realizada com sucesso e `falso`, caso contrário.

A representação do TAD é, portanto, composta pela definição da estrutura de dados e pela especificação das operações aplicáveis sobre ele. Em linguagens de programação essa representação é denominada de `interface do TAD`. A visibilidade da estrutura interna do tipo fica limitada às operações que são explicitamente especificadas na interface como exportáveis, agregando a idéia de ocultamento das informações. O cliente do TAD recebe a especificação da estrutura de dados e do conjunto de operações, mas a implementação permanece invisível e inacessível. O cliente do TAD tem acesso somente à forma abstrata do tipo, com base simplesmente nas funcionalidades oferecidas pelo tipo. A forma como ele foi implementado torna-se um detalhe de implementação, que não deve interferir no uso do TAD em outros contextos. Essa separação entre a definição do TAD e sua implementação permite que alterações de implementação não influam nas aplicações que utilizam o TAD.

Após a especificação do TAD e a associação de suas operações é possível implementar as operações definidas para o TAD. Cabe lembrar que a característica essencial é separar o conceito da implementação. Os TADs são geralmente implementados em linguagens de programação através do conceito de `bibliotecas`. A implementação das operações propriamente dita é feita em um arquivo separado, sendo que qualquer alteração na implementação dessas operações implica somente na compilação do módulo envolvido.

Por fim, basta somente fazer a ligação da interface e da implementação com as aplicações do usuário para gerar um programa executável. Cabe novamente salientar que a forma de implementação das operações não deve alterar o TAD, ou seja, a implementação das funções e dos procedimentos pode ser alterada e recompilada, mantendo as aplicações em funcionamento sem alteração.

Em resumo, um TAD especifica um conjunto de operações e a semântica dessas operações (o que elas fazem). Essa especificação é denominada `tipo abstrato`. A implementação propriamente dita das operações é denominada `tipo concreto`. As aplicações que utilizam esse TAD são denominadas `clientes do tipo de dado`. Deste modo, duas importantes vantagens são identificadas na utilização do conceito de TADs: (1) a possibilidade de sua utilização em diversas aplicações diferentes; e (2) a possibilidade de alterar o tipo sem alterar as aplicações que o utilizam.

O projeto de um TAD envolve a escolha de operações adequadas para uma estrutura de dados, definindo seu comportamento. Algumas dicas interessantes são:

- definir um número pequeno de operações, com soluções simples, que combinadas possam realizar funções mais complexas;
- o conjunto de operações deve ser adequado o suficiente para realizar as computações necessárias às aplicações que utilizarão o TAD. A obtenção das informações básicas não deve ser simples e direta para as aplicações;
- cada operação deve ter um propósito definido, com um comportamento constante e coerente, sem muitos casos especiais e exceções.

Os TADs podem ainda ser classificados como genéricos ou específicos. Os TADs `genéricos` são estruturas de dados nas quais é possível acrescentar qualquer item de dados. Por exemplo, um TAD `listas` pode ser utilizado para representar uma lista de freqüência de alunos, uma lista telefônica, etc. Os TADs `específicos` são definidos para um dado domínio de aplicação, como uma agenda de telefones. Não se deve, entretanto, misturar características genéricas com características específicas de domínio. No contexto deste livro vamos trabalhar com TADs genéricos para listas, pilhas, filas e árvores.

2.3 → alternativas de representação física

Um aspecto que deve ser considerado na análise das estruturas de dados a serem estudadas é a forma utilizada para representar fisicamente os relacionamentos lógicos entre os dados. Por representação física de uma estrutura de dados consideramos somente a forma utilizada para implementar os relacionamentos entre os diferentes nodos, e não a representação física das informações internas de cada nodo. Por exemplo, na implementação física de uma lista linear, a ser vista no capítulo 3, considera-se somente como será implementada a relação de ordem entre os seus nodos.

Existem duas alternativas básicas de representação física de dados, analisadas a seguir: contigüidade física e encadeamento. Em qualquer uma delas, é importante notar que a posição de um componente na estrutura lógica determina sua posição na estrutura física escolhida.

2.3.1 contigüidade física

Nesta alternativa, os dados são armazenados em posições contíguas na memória. A ordem entre os nodos da estrutura armazenada é definida implicitamente pela posição ocupada pelos nodos na memória. Assim, cada posição contígua na memória armazena o conjunto de informações correspondente a um nodo, que pode ser simples ou complexo, apresentando um ou vários campos.

Para que um dado esteja efetivamente representado, é necessário que, a partir de sua representação física, se possa reconstituir tanto sua estrutura

lógica quanto seus componentes. A representação por contigüidade física é intuitiva e natural para estruturas que apresentam um relacionamento lógico de ordem linear entre os nodos. Existem, entretanto, estruturas que apresentam outras formas de relacionamentos entre os nodos, como relacionamentos de subordinação, para as quais esta forma de representação física não é intuitiva.

As **vantagens** da representação das estruturas de dados por contigüidade física são:

`proteção de memória` – a alocação é feita antes do início da execução do programa, garantindo a proteção da memória;
`transferência de dados` – como todos os dados estão alocados em bloco, a transferência de dados entre memória principal e secundária fica facilitada;
`estruturas simples` – é apropriada para armazenar estrutura simples, principalmente aquelas que utilizam a ordem física em sua representação;
`representação` – algumas estruturas de dados possuem uma representação lógica semelhante à contigüidade física, simplificando desta maneira a representação dos dados;
`acesso` – qualquer nodo pode ser diretamente acessado a qualquer momento, através de um índice associado à sua posição.

As **desvantagens** da representação das estruturas de dados por contigüidade física são:

`compartilhamento de memória` – este tipo de alocação não permite o compartilhamento de memória;
`previsão de espaço físico` – é necessário definir, antes da execução da aplicação, o número máximo de nodos a serem alocados. Isto pode constituir um grave problema quando não for possível estimar, *a priori*, o número de nodos que a estrutura vai apresentar;
`estruturas complexas` – não é apropriado para estruturas complexas, devido à natureza seqüencial;
`inserção e exclusão de componentes` – estas duas operações geralmente implicam no deslocamento de um número considerável de informações,

de modo a preservar as relações lógicas representadas, ou seja, existe a necessidade de reestruturação dos componentes da estrutura de dados, durante sua manipulação.

2.3.2 encadeamento

O espaço necessário para a representação dos dados pode ser alocado à medida que se torne necessário, através de alocação dinâmica. Sempre que um espaço de memória para armazenar um dado seja necessário, este é solicitado pela aplicação, e alocado pelo gerenciador de memória em algum espaço livre da memória, sendo seu endereço devolvido em uma variável especial. As principais linguagens de programação permitem esta forma de alocação através de variáveis do tipo ponteiro (também denominadas de apontadores).

Uma estrutura armazenada através de encadeamento apresenta seus nodos alocados em posições aleatórias na memória, e não lado a lado, como na representação por contigüidade física. A disposição física dos componentes de uma estrutura independe de sua posição na estrutura lógica. Deste modo, para percorrer a estrutura é necessário que os nodos, além do(s) campo(s) de informação, apresentem algum campo adicional, onde é colocado o endereço físico do próximo nodo.

As **vantagens** da representação das estruturas de dados por encadeamento são:

`compartilhamento de memória` – uma vez que os nodos de uma estrutura são indicados através de seus endereços, os mesmos nodos poderiam fazer parte de mais de uma estrutura;

`maleabilidade` – a alocação e a liberação de memória feita de forma dinâmica favorece a maleabilidade dos programas;

`facilidade para inserção e remoção de componente` – a inserção e a remoção de nodos é facilmente realizada, sendo somente necessário ajustar os campos de elo dos nodos envolvidos na operação, sem a necessidade de deslocamento de informações.

As **desvantagens** desta forma de representação física são:

`transferência de dados` – é dificultada neste tipo de representação, uma vez que os dados estão espalhados na memória;

`gerência de memória mais onerosa` – toda a manipulação da estrutura é feita através de alocação e/ou liberação de memória, o que deve ser realizado pelo gerenciador de memória;

`procedimentos menos intuitivos` – a utilização de alocação dinâmica implica na construção de procedimentos que envolvam alocação e liberação de memória, e no encadeamento dos nodos. Isto faz com que os procedimentos escritos para as operações sobre os dados sejam mais complexos;

`acesso` – o processamento dos dados encadeados deve ser feito de forma serial, isto é, um nodo deve ser sempre acessado a partir de outro acessado anteriormente. Não é possível, como nos casos dos arranjos, acessar qualquer nodo a qualquer momento, a menos que todos os endereços sejam armazenados individualmente.

2.3.3 representação física mista

Em alguns casos é interessante utilizar uma representação física mista – parte da estrutura armazenada através de contigüidade física, e parte por encadeamento. Esta forma é muito utilizada para armazenar estruturas complexas, com operações específicas. Neste texto não será considerada esta forma de armazenamento físico.

2.4 → exercícios

exercício 1 Especifique um TAD que seja capaz de armazenar uma data composta por dia, mês e ano e implemente o seguinte conjunto de operações para manipular esse tipo `Data`:

- uma função que recebe como parâmetro o dia, o mês, o ano e uma estrutura do tipo `Data`, onde o resultado deve ser armazenado, e retorna `verdadeiro` se a data estiver válida; caso contrário, retorna `falso`. A validação da data pode ser feita da seguinte forma:
  - meses com 30 dias: 04, 06, 09 e 11;
  - meses com 31 dias: 01, 03, 05, 07, 08, 10,12;

  - cálculo de ano bissexto: cada ano divisível por 4 é um ano bissexto (E); cada ano divisível por 100 não é bissexto (OU); todo ano divisível por 400 sempre é um ano bissexto;
- uma função que recebe como parâmetro uma data do tipo `string` (no formato DD/MM/AAAA) e uma estrutura do tipo `Data`, onde o resultado deve ser armazenado, e retorna `verdadeiro` se a data estiver válida; caso contrário, retorna `falso`;
- uma função que recebe como parâmetro uma estrutura do tipo `Data` e um número inteiro, e retorna a soma do número de dias da data recebida;
- uma função que recebe como parâmetro uma estrutura do tipo `Data` e um número inteiro, e retorna a subtração do número de dias da data recebida;
- uma função que recebe como parâmetro uma data e escreve essa data por extenso. Por exemplo, 25/03/2007 deve ser escrito como *25 de março de 2007*. A função deve retornar `verdadeiro` se a operação for realizada com sucesso e `falso`, caso contrário;
- uma função que receba como parâmetro o ano e retorne a data da Páscoa. O domingo de Páscoa é o primeiro domingo depois da primeira lua cheia do outono. Utilize o seguinte algoritmo, criado em 1800 pelo matemático Carl Gauss:
  - suponha que `y` seja o ano (como 1800 ou 2001);
  - divida `y` por 19 e chame o resto de `a.` Ignore o quociente;
  - divida `y` por 100 para obter o quociente `b` e o resto `c;`
  - divida `b` por 4 para obter o quociente `d` e o resto `e;`
  - divida `8 * b + 13` por `25` para obter o quociente `g`. Ignore o resto;
  - divida `19 * a + b - d - g + 15` por `30` para obter o quociente `h`. Ignore o resto;
  - divida `c` por 4 para obter o quociente `j` e o resto `k;`
  - divida `a + 11 * h` por `319` para obter o quociente `m`. Ignore o resto;
  - divida `2 * e + 2 * j - k - h + m + 32` por `7` para obter o resto `r`. Ignore o quociente;
  - divida `h - m + r + 90` por `25` para obter o quociente `n`. Ignore o resto;
  - divida `h - m + r + n + 19` por `32` para obter o resto `p`. Ignore o quociente.

exercício 2 Considere uma aplicação para armazenar os seguintes dados de uma pessoa em uma agenda de endereços: nome, endereço e telefone. Especifique um TAD para armazenar os dados das pessoas e as operações necessárias para inserir, consultar e excluir os dados das pessoas.

exercício 3 Considere uma aplicação para armazenar os seguintes dados de carros para uma garagem: placa, marca/modelo e cor. Especifique um TAD para armazenar os dados dos carros e as operações necessárias para inserir, consultar e excluir os dados das pessoas.

exercício 4 Considere uma empresa que precisa armazenar os seguintes dados de um cliente:

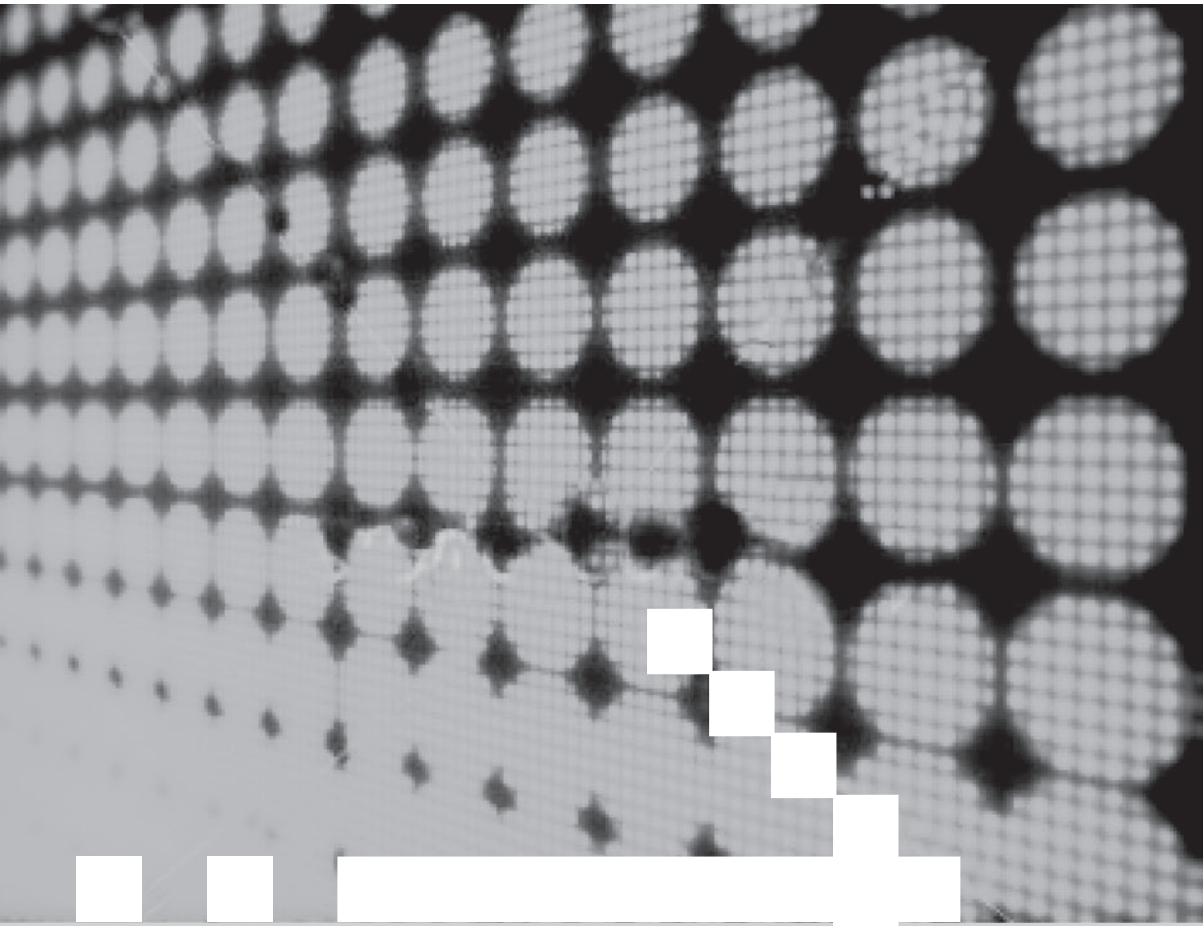
- nome completo;
- ano de nascimento;
- renda mensal do cliente.

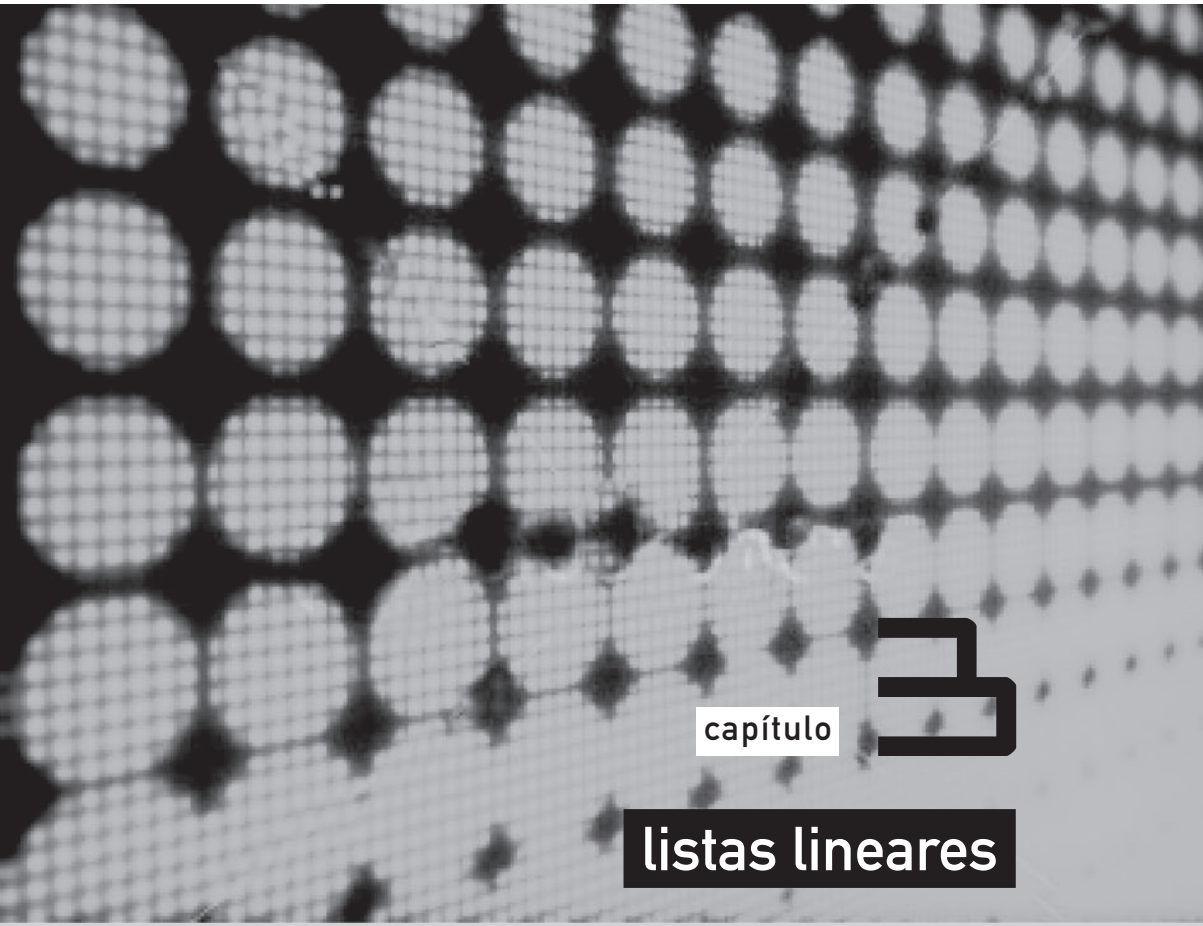
Especifique um TAD para armazenar os dados de um cliente e as operações necessárias para inserir, consultar e excluir os dados dos clientes. Especifique também operações para exibir o número de clientes com renda mensal acima da média, e exibir o número de clientes que nasceram entre 1980 e 2000.

exercício 5 Considere um conjunto de informações relativas a alunos, constituído de nome, número de matrícula e data de nascimento. Especifique um TAD para armazenar os dados dos alunos e as operações necessárias para inserir, consultar e excluir esses dados. Implemente uma aplicação que utilize o tipo `Aluno` e ainda execute as seguintes operações:

- imprima os nomes e números de matrícula dos alunos que nasceram após uma determinada data (passada como parâmetro);
- imprima as informações relativas a um determinado aluno, cujo número de matrícula é passado como parâmetro.

exercício 6 Os dados relativos aos clientes de uma empresa estão armazenados em um arquivo. Para cada cliente são registrados um código, o nome, o endereço, o telefone, a data em que fez sua primeira compra na empresa, a data da última compra e o valor da última compra. Especifique o TAD `Clientes` para armazenar os dados dos clientes e as operações necessárias para inserir, consultar e excluir esses dados. Implemente uma aplicação que utilize o tipo `Clientes`.





capítulo 3

listas lineares

Lista Linear é um conjunto de elementos de mesmo tipo, denominados nodos, entre os quais existe uma relação de ordem linear (ou total). O relacionamento entre os nodos de uma lista linear é definido somente por sua posição em relação aos outros nodos. Os nodos de uma lista podem conter, cada um deles, um dado primitivo ou um dado composto. Quando a lista apresenta todos os nodos com a mesma estrutura interna, a lista é denominada homogênea. Listas compostas de nodos com diferentes estruturas internas, denominadas não heterogêneas, não são consideradas neste texto.

Toda lista linear apresenta um nodo que encabeça a lista, que é o primeiro nodo da lista. A partir deste segue uma seqüência de nodos, conforme a ordem definida entre eles, até o último nodo. Todo nodo de uma lista linear apresenta um nodo diretamente antes dele (com exceção do primeiro), e outro imediatamente depois dele (exceto o último). O número de nodos de uma lista é denominado de `comprimento da lista`.

A Figura 3.1 apresenta uma lista linear composta por cinco nodos, sendo cada um identificado por uma letra (a, b, … e) que simboliza o conteúdo de seu(s) campo(s) de informação. O primeiro nodo é aquele que apresenta a informação a, seguido daquele de informação b; o último nodo apresenta a informação e.

Inúmeras aplicações apresentam dados que, por sua natureza, são organizados em listas lineares. Alguns exemplos de informações geralmente organizadas em listas lineares são:

- as notas de cada aluno de uma turma, sendo a ordem das notas definida pela posição que o aluno ocupa na turma, ou de acordo com o número de sua matrícula;
- o cadastro de funcionários de uma empresa, organizado de acordo com seu número de matrícula, ou simplesmente pela ordem em que foram admitidos na empresa;
- os 7 dias da semana constituem uma lista, onde a segunda-feira precede a terça-feira e vem depois do domingo;

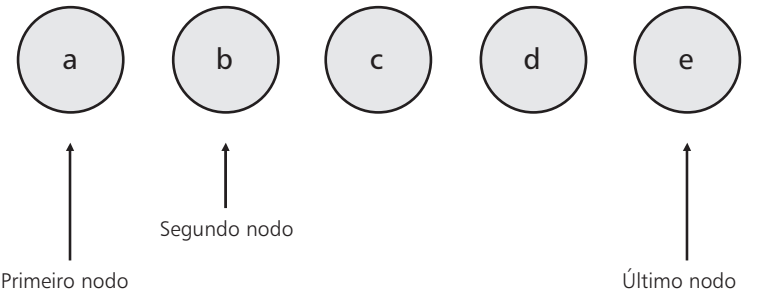


Figura 3.1 Lista linear.

- valores obtidos através de medições, quando a ordem em que estas foram feitas é relevante para a análise dos dados. Por exemplo, registro de temperaturas de uma localidade.

A estrutura interna de cada componente de uma lista linear pode apresentar qualquer nível de complexidade. Pode ser um tipo simples, um arranjo, um registro, ou outro TAD qualquer. A estrutura interna dos nodos é abstraída na manipulação da lista, interessando somente as relações de ordem entre os diferentes nodos.

Formalmente, uma lista linear pode ser definida da seguinte maneira: uma coleção de $n \geq 0$ nodos $x_1, x_2, \ldots, x_n$, todos do mesmo tipo, cujas propriedades estruturais relevantes envolvem apenas as posições relativas lineares entre os nodos:

```
n = 0:      lista vazia, apresenta zero nodos
n > 0:      x1 é o primeiro nodo
            xn é o último nodo
1 < k < n:  xk é precedido por xk-1 e sucedido por xk+1
```

operações sobre listas lineares

Listas lineares, como quaisquer TADs, requerem a definição das operações que podem ser realizadas sobre elas. Várias operações de manipulação de uma lista podem ser necessárias em uma aplicação. Para cada uma delas devem ser definidos procedimentos específicos, de modo que este tipo de dado possa ser facilmente manipulado. O programa de aplicação deverá se preocupar somente com a utilização da lista, estando as operações básicas já disponíveis. Atenção redobrada deve ser dada à eficiência dos procedimentos que implementam as operações mais utilizadas, por estarem presentes em quase todas as aplicações que usam listas lineares. As operações básicas sobre listas lineares são as seguintes:

`criação de uma lista` – a primeira operação a ser executada, através da qual são alocadas as variáveis necessárias para a definição da lista e inicializadas suas variáveis de controle. As demais operações ficam habilitadas somente depois da execução desta operação;

`inserção de um nodo` – é a maneira de formar a lista, inserindo os nodos um a um. A inserção de um nodo pode ser feita no início da lista, no final da lista, ou em alguma posição dentro da lista;

`exclusão de um nodo` – é a forma de excluir um nodo da lista. A exclusão também pode ser feita no início da lista, no final da lista, ou em alguma posição no meio da lista;

`acesso a um nodo` – para consulta ou alteração de seus valores internos. O nodo pode ser identificado por sua posição na lista ou através de alguma informação que ele contém;

`destruição de uma lista` – operação executada quando uma lista existente não é mais necessária. A lista não poderá mais ser utilizada depois da execução desta operação. Pode ocorrer que a lista, na realidade, continue existindo fisicamente, mas sua utilização não é mais permitida.

É importante lembrar que estas são somente as operações básicas sobre listas – outras operações podem ser necessárias em aplicações específicas, devendo ser implementadas de acordo com a necessidade.

A representação física utilizada para armazenar uma lista linear influi diretamente nos algoritmos definidos para as operações a serem definidas sobre ela. A escolha entre uma dessas formas de representação está relacionada a dois fatores principais: a freqüência com que as operações são executadas sobre a lista e o volume de dados envolvido nas aplicações. Neste capítulo são discutidos os algoritmos que lidam com listas lineares e suas estruturas de armazenamento. Inicialmente são consideradas as listas representadas através de contigüidade física e, em seguida, as mesmas operações considerando listas representadas através de encadeamento.

Uma forma alternativa e eficiente de representar listas lineares, que facilita o acesso a seus nodos, é usar variáveis especiais, aqui denominadas de `descritores`. Um descritor apresenta informações sobre alguns nodos especiais e sobre a lista propriamente dita. Esta possibilidade também será considerada nos algoritmos vistos a seguir.

3.1 → listas lineares implementadas através de contigüidade física

Listas lineares implementadas através de contigüidade física utilizam a seqüencialidade da memória do computador para representar a ordem dos nodos na lista. Endereços fisicamente adjacentes na memória representam nodos logicamente adjacentes na lista. Deste modo, o relacionamento lógico representado pela posição de um nodo na lista não precisa ser explicitamente representado.

A forma mais usual de implementar uma lista linear através de contigüidade física é através da utilização de um arranjo de uma dimensão (vetor). Cada elemento do arranjo representa um nodo da lista. Qualquer nodo da lista pode ser acessado diretamente através do índice do arranjo, que representa sua posição na lista. A Figura 3.2 ilustra uma lista linear de 6 nodos, representada pelo arranjo `LL` de 6 elementos. O primeiro nodo desta lista (`L1`) está armazenado em `LL[1]`, o segundo em `LL[2]` e o último em `LL[6]`.

A possibilidade de acessar diretamente um nodo é uma característica importante deste tipo de implementação, pois não é necessário percorrer toda a lista, a partir de seu início, para que um determinado nodo seja acessado.

Todos os elementos de um arranjo apresentam o mesmo tipo de dado, que representa o conjunto de informações do nodo. Como exemplo, consideremos uma lista linear que agrupa e ordena os produtos vendidos em uma

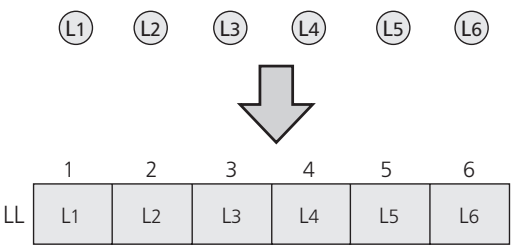


Figura 3.2 Lista linear implementada através de contigüidade física.

loja. Cada nodo desta lista traz informações relativas a um produto, como o nome do produto, seu código e seu preço de venda. Considerando que a loja venda no máximo N produtos diferentes, os seguintes tipos de dados devem ser utilizados:

```
TipoNodo = registro
              Nome: string
                    Código: inteiro
                    Valor: real
              fim registro
TipoLista = arranjo [1.. N] de TipoNodo
```

A natureza dinâmica das listas faz com que seu comprimento varie durante uma aplicação, pela inserção de novos nodos e pela remoção de nodos existentes. A sucessiva inclusão e remoção de nodos em uma lista pode levar a situações em que suas extremidades (primeiro e último nodo) não coincidam com as extremidades do arranjo. Isto implica que se tenha, além dos indicadores dos índices de início e final do arranjo (aqui denominados de IA e FA), indicadores das posições onde estão alocados o primeiro e o último nodo da lista (aqui denominados respectivamente IL e FL). Por exemplo, na Figura 3.3 o arranjo LL de 10 elementos implementa uma lista linear de 5 nodos, iniciando esta lista no índice IL do arranjo (no exemplo mostrado, índice 3) e terminando no índice FL (no exemplo, índice 7).

É possível armazenar mais de uma lista em um mesmo arranjo. Por exemplo, na Figura 3.4 é representado o arranjo LL implementando duas listas – a primeira (de 2 nodos) inicia no índice IL1 e termina no índice FL1, enquanto que a segunda (de 3 nodos) inicia no índice IL2 e termina no índice FL2. Ao fazer as inclusões de novos nodos em listas implementadas desta forma, deve-se tomar cuidado para que não seja utilizado o espaço que está sendo ocupado pela outra lista.

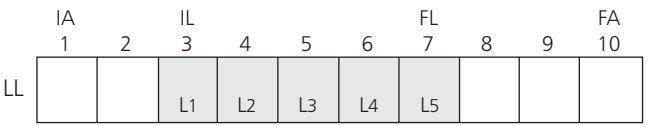


Figura 3.3 Lista linear utilizando parte de um arranjo.

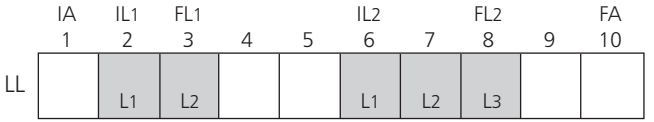


Figura 3.4 Várias listas lineares implementadas sobre um mesmo arranjo.

Caso uma aplicação utilize um só arranjo para implementar mais de uma lista, é aconselhada a definição *a priori* da área que será disponibilizada para cada um deles. Isto é feito definindo os índices `IA` e `FA` para cada lista, de modo que uma não possa avançar sobre o espaço reservado para a outra.

A seguir serão vistos algoritmos correspondentes às principais operações que podem ser executadas sobre listas lineares implementadas sobre arranjos. É importante lembrar que os algoritmos apresentados não esgotam o assunto, já que outras operações podem ser necessárias em aplicações específicas.

3.1.1 criação de uma lista linear vazia

Criar uma lista linear significa alocar a área física onde a lista será armazenada e inicializar suas variáveis de controle (primeiro nodo, último nodo, etc.). A criação de uma lista linear não implica na necessidade de ela apresentar imediatamente nodos com informações. Uma lista linear geralmente é criada `vazia`, ou seja, sem nodo. Mais tarde, à medida que a aplicação necessitar, os nodos serão inseridos na lista.

A criação de uma lista linear implementada através de contigüidade física requer a alocação física do arranjo e a inicialização das variáveis que irão informar os índices do primeiro e do último nodo da lista.

A posição das extremidades de uma lista sobre o arranjo pode ser informada de diversas formas. Para a indicação do primeiro nodo da lista pode ser utilizado diretamente o índice de sua posição no arranjo, ou então a distância que ocupa a partir do primeiro elemento do arranjo (que não necessariamente inicia com índice um). O último nodo da lista também pode ser indicado através de seu índice no arranjo, ou então esta posição pode ser calculada através do número de nodos máximo que a lista poderá apresentar. A forma de indicar a posição dos limites da lista evidentemente irá influir nos algoritmos desenvolvidos para manipular esta lista.

Assim, a criação de uma lista linear implementada sobre um arranjo requer que inicialmente seja alocado o arranjo LL onde a lista será implementada, assim como as variáveis que contém os índices de seus limites, ou seja, os índices da primeira posição no arranjo que esta lista poderá ocupar (denominado de IA) e o da última (denominado FA).

Em seguida, devem ser inicializadas as variáveis que indicam o início e o final da lista (denominadas de IL e FL). No processo de inicialização de uma lista, os valores utilizados para inicializar os indicadores de início e final da lista devem trazer a informação de que a lista está vazia. Isto pode ser feito de várias maneiras, por exemplo, indicando para início da lista (IL) um índice menor do que o da primeira posição que a lista pode ocupar no arranjo. A inicialização do final da lista (FL) geralmente só é necessária para servir de base para atualizar este indicador, quando o primeiro nodo for inserido na lista. O importante é que os procedimentos que implementam as demais operações sobre a lista conheçam a estratégia utilizada para inicializá-la, podendo desta maneira reconhecer quando a lista está vazia.

O algoritmo apresentado a seguir recebe como parâmetro o índice do início da área que pode ser utilizada para a lista sobre o arranjo (IA). Em todos os algoritmos a seguir, a indicação de que a lista está vazia é feita inicializando as variáveis de IL e FL em um índice imediatamente anterior ao início da área disponível. Considera-se aqui que a alocação do arranjo e a definição da área disponível para a lista já foram realizadas pela aplicação.

```
Algoritmo 3.1 – InicializarLLArr
    Entradas: IA (inteiro)
    Saídas: IL, FL (inteiro)
início
    IL ← FL ← IA – 1
fim
```

3.1.2 inserção de um novo nodo

Para realizar a operação de inserção de um novo nodo em uma lista linear devem ser informados (1) a posição do novo nodo na lista (sua ordem na lista), e (2) o valor do campo de informação deste novo nodo. Duas situações são identificadas nas quais é impossível realizar a inserção:

- quando a posição solicitada não for coerente com a lista atualmente existente – por exemplo, a lista apresenta 10 nodos e se quer inserir o 13º;

- quando o espaço que a lista pode ocupar no arranjo estiver totalmente preenchido.

O algoritmo que implementa a operação de inserção deve verificar estas condições e, caso não seja possível realizar a operação, comunicar o fato à aplicação. Uma vez verificada a possibilidade de realizar a inserção, a implementação da operação vai depender da posição onde o novo nodo deverá ser inserido – se no início, no meio ou no final da lista. A seguir são analisadas separadamente estas três situações de inserção.

Inserção como primeiro nodo da lista. Quando a lista inicia na primeira posição disponível no arranjo, todos os nodos da lista devem ser deslocados para a posição seguinte, liberando a primeira posição para o novo nodo. O indicador de final da lista (FL) deve ser atualizado para a posição seguinte no arranjo. A Figura 3.5 exemplifica esta operação, considerando um arranjo de 10 posições, e uma lista com 6 nodos, que inicia na primeira posição do arranjo. Após o deslocamento de todos os nodos da lista, o novo nodo é inserido na primeira posição, e a lista passa a ter 7 nodos.

No caso de a lista não iniciar na primeira posição do arranjo (`IL > IA`), não há necessidade de realizar o deslocamento de todos os nodos, uma vez que o novo nodo pode ser inserido diretamente na posição logo abaixo do início da

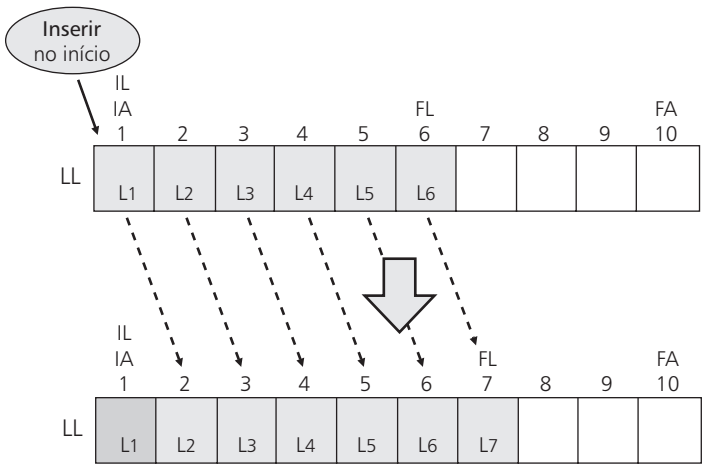


Figura 3.5 Inserção no início de uma lista, com deslocamento.

lista, desde que esta esteja livre. O indicador de início da lista deve ser atualizado para a posição agora ocupada pelo novo nodo (Figura 3.6).

A seguir é apresentado um algoritmo que insere um nodo no início de uma lista linear implementada sobre o arranjo `LL`, sendo a área disponível para a lista limitada pelos índices `IA` e `FA`. A lista, antes da inserção do novo nodo, ocupa as posições entre os índices `IL` e `FL`. A função recebe, ainda, as informações que devem ser inseridas no novo nodo (`InfoNodo`). Após a inserção, o procedimento devolve o arranjo e os índices de limites da lista atualizados. Caso haja espaço para inserir o novo nodo, o algoritmo inicialmente verifica se existe espaço livre no início, caso em que basta mudar o índice de início da lista (`IL`) para uma posição antes daquela que ocupava. No caso da lista iniciar na primeira posição disponível do arranjo, é feito o deslocamento de toda a lista para abrir espaço para o novo nodo. O parâmetro `Sucesso` retorna verdadeiro caso a inserção tenha sucesso, e falso no caso contrário.

```
Algoritmo 3.2 - InserirIniLLArr
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             InfoNodo (TipoNodo)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variável auxiliar: Ind (inteiro)
```

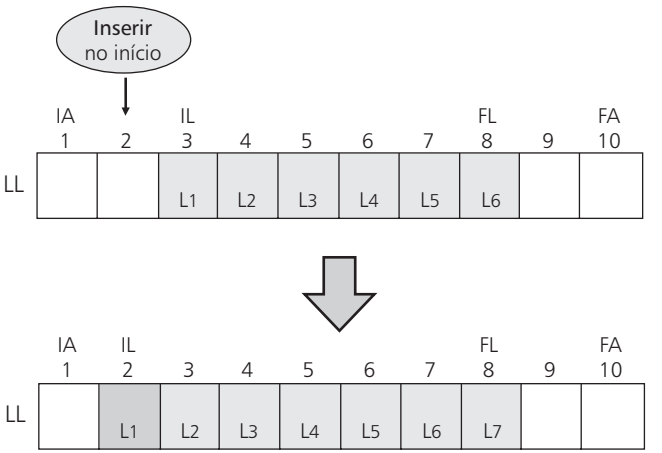


Figura 3.6 Inserção no início de uma lista, sem deslocamento.

```
início
  se (IA = IL) e (FA = FL)
  então Sucesso ← falso
  senão início
        se IL = 0
        então IL ← FL ← IA
        senão se IL > IA
              então IL ← IL-1
              senão início  {DESLOCAR NODOS PARA CIMA}
                      para Ind de FL incr -1 até IL faça
                       LL[Ind+1] ← LL[Ind]
                    FL ← FL+1
                    fim
        LL[IL] ← InfoNodo
        Sucesso ← verdadeiro
        fim
fim
```

Inserção como último nodo da lista. Para inserir um novo nodo no final da lista somente é necessário avançar o indicador de final da lista, não sendo alteradas as posições do restante da lista. Pode ocorrer o caso de a lista estar ocupando as últimas posições do arranjo, mas ainda existirem posições livres no início – neste caso todos os nodos da lista deverão ser deslocados uma posição para baixo, de modo a liberar a última posição, na qual será inserido o novo nodo (Figura 3.7).

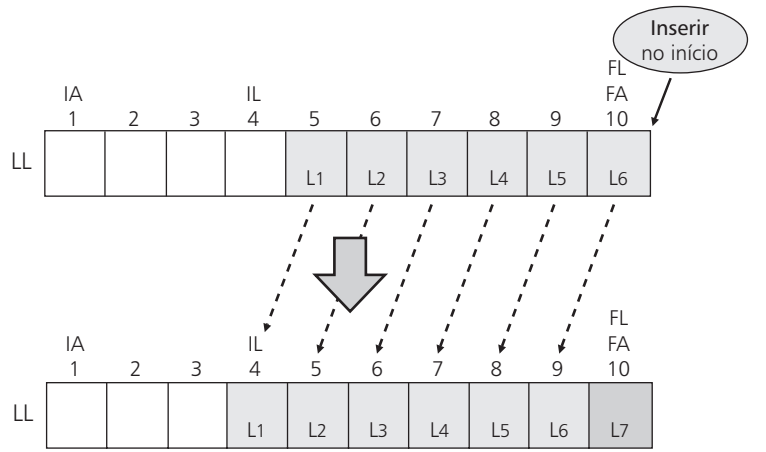


Figura 3.7 Inserção no final de uma lista, com deslocamento.

O algoritmo apresentado a seguir insere um nodo no final de uma lista linear implementada sobre o arranjo LL, de limites IA e FA. O algoritmo recebe o nome do arranjo e seus limites, os limites da lista e as informações a serem inseridas no novo nodo (InfoNodo). De forma análoga ao algoritmo anterior, caso haja espaço para inserir o novo nodo, verifica se existe espaço livre no final para a inserção, senão é efetuado o deslocamento de toda a lista uma posição para baixo, liberando a última posição do arranjo. O parâmetro Sucesso retorna verdadeiro caso a inserção tenha sucesso, e falso caso contrário. O algoritmo devolve o arranjo e os índices de limites da lista atualizados.

```
Algoritmo 3.3 - InserirFimLLArr
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             InfoNodo (TipoNodo)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variável auxiliar: Ind (inteiro)
início
  se (IA = IL) e (FA = FL)
  então Sucesso ← falso
  senão início
        se IL = 0    {LISTA VAZIA}
        então IL ← FL ← IA
        senão se FL < FA    {TEM ESPAÇO ATRÁS}
              então FL ← FL+1
              senão início    {DESLOCAR NODOS PARA BAIXO}
                    para Ind de IL incr 1 até FL faça
                        LL[Ind-1] ← LL[Ind]
                    IL ← IL-1
                    fim
        LL[FL] ← InfoNodo
        Sucesso ← verdadeiro
        fim
fim
```

Inserção no meio da lista. Esta inserção requer a movimentação de todos os nodos acima ou abaixo da posição onde vai ser feita a inserção do novo nodo. A Figura 3.8 apresenta a inserção de um novo nodo na terceira posição de uma lista linear. Para que esta inserção possa ser realizada, os nodos

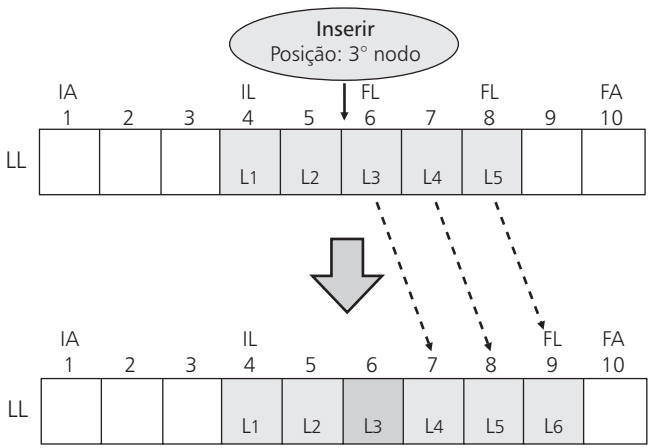


Figura 3.8 Inserção no meio de uma lista linear.

que ocupam as posições a partir da terceira devem ser deslocados para a posição seguinte no arranjo. O indicador de final da lista deve ser acrescido de uma unidade.

A seguir é apresentado um algoritmo que insere um novo nodo em uma lista linear, implementada por contigüidade física. É utilizado um arranjo `LL`, podendo a lista ocupar as posições de índices `IA` até `FA`. A função recebe como entradas a posição `K` (ordem na lista) de inserção do novo nodo e as informações que devem ser inseridas em seu campo de informação (`InfoNodo`), além dos índices do início e do final da lista (`IL` e `FL`).

Antes de realizar a inserção do novo nodo na lista, o algoritmo analisa se a inserção é possível, verificando:

- se existe espaço no arranjo para realizar a inserção – a inserção não poderá ser realizada caso o arranjo esteja todo ocupado;
- se a posição de inserção é válida – a posição recebida deverá ser um valor maior do que zero; se a lista estiver vazia (condição representada no procedimento por `IL = 0`), a inserção poderá ser realizada somente na primeira posição; se a lista não for vazia, a posição não poderá ser maior do que o número de nodos apresentado pela lista antes da inserção, incrementado de uma unidade.

Se alguma destas condições falhar, o parâmetro lógico Sucesso retorna falso, indicando que a inserção não pode ser realizada. Se o nodo é inserido, Sucesso retorna verdadeiro.

O algoritmo aqui mostrado inclui os outros dois apresentados nesta seção: se K=1, a inserção é realizada no início da lista; e se K=FL-IL+2, o novo nodo é inserido no final da lista.

```
Algoritmo 3.4 - InserirLLArrPosK
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             K (inteiro)
             InfoNodo (TipoNodo)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variável auxiliar: Ind (inteiro)
início
  se (IA=IL e FA=FL) ou (K > FL-IL+2) ou (K≤0) ou (IL=0 e K≠1)
  então Sucesso ← falso
  senão início
        se IL = 0
        então IL ← FL ← IA
        senão se FL ≠ FA
              então início     {DESLOCAR NODOS PARA CIMA}
                    para Ind de FL incr -1 até IL+K-1 faça
                       LL[Ind+1] ← LL[Ind]
                    FL ← FL+1
                    fim
              senão início     {DESLOCAR NODOS PARA BAIXO}
                    para Ind de IL incr 1 até IL+K-2 faça
                       LL[Ind-1] ← LL[Ind]
                    IL ← IL - 1
                    fim
        LL[IL+K-1] ← InfoNodo
        Sucesso ← verdadeiro
        fim
fim
```

Uma forma de otimizar a inserção é procurar diminuir o número de deslocamentos de nodos. Isto pode ser feito verificando onde fica a posição em que o nodo deve ser inserido. Caso esta ficar mais próxima do início da lista, o deslocamento deve ser feito para baixo, desde que haja espaço livre deste

lado da lista; se a posição ficar mais próxima do final da lista, o deslocamento deve ser feito para cima, também condicionado à existência de espaço deste lado da lista. O algoritmo de inserção otimizado é apresentado a seguir. A inserção do primeiro nodo é realizada já no meio do arranjo, permitindo deslocamentos para os dois lados do arranjo.

```
Algoritmo 3.5 - InserirLLArrPosKOt
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             K (inteiro)
             InfoNodo (TipoNodo)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variável auxiliar: Ind (inteiro)
início
  se (IA=IL e FA=FL) ou (K > FL-IL+2) ou (K≤0) ou (IL=0 e K≠1)
  então Sucesso ← falso
  senão início
        se IL = 0
        então IL ← FL ← (FA-IA+1) div 2   {INSERE NO MEIO}
        senão se IL = IA ou ((FL ≠ FA) e (K > (FL-IL+2/2))
              então início     {DESLOCAMENTO PARA CIMA}
                    para Ind de FL incr -1 até IL+K-1 faça
                         LL[Ind+1] ← LL[Ind]
                    FL ← FL+1
                    fim
              senão início     {DESLOCAMENTO PARA BAIXO}
                    para Ind de IL incr 1 até IL+K-2 faça
                         LL[Ind-1] ← LL[Ind]
                    IL ← IL - 1
                    fim
        LL[IL+K-1] ← InfoNodo
        Sucesso ← verdadeiro
        fim
fim
```

3.1.3 remoção de um nodo

A próxima operação a ser analisada remove um determinado nodo de uma lista linear. Este nodo pode ser identificado através de sua ordem na lista, ou através de alguma informação contida no próprio nodo.

Será considerado aqui o caso em que a identificação do nodo a ser removido é feita através de sua ordem na lista – será removido o "k-ésimo" nodo da lista, ou seja, nodo de ordem K a partir do início da lista. Dada a alocação contígua dos nodos, o nodo buscado é facilmente localizado assim que for identificada a posição onde inicia a lista, bastando para isso somar sua ordem ao índice da primeira posição ocupada. É importante lembrar que a posição do nodo buscado não é contada a partir do início do arranjo, mas sim a partir do primeiro nodo da lista, que pode estar localizado em qualquer posição do arranjo.

A remoção do "k-ésimo" nodo somente terá sucesso se a ordem do nodo solicitado (índice K) for válida, ou seja, estiver contida na lista. Caso a operação seja realizada, a remoção do nodo implicará no rearranjo dos demais nodos, de modo a não deixar espaço livre entre os nodos da lista – os nodos que o seguem devem ser adiantados uma posição e o índice de final da lista (FL) deve ser atualizado. Caso o nodo removido seja o único da lista, ela resulta vazia, devendo então os sinalizadores de início e final da lista serem ajustados para fornecer esta informação. Por exemplo, conforme feito na criação da lista (Algoritmo 3.1), IL e FL são colocados em um índice uma unidade menor do que o do limite inferior da área disponível para a lista no arranjo. A Figura 3.9 mostra a remoção do terceiro nodo da lista linear.

A seguir é apresentado um algoritmo que remove o "k-ésimo" nodo de uma lista linear implementada através de contigüidade física. Inicialmente é feita a verificação da validade da ordem do nodo a ser removido em relação à lista existente; em seguida os nodos posteriores àquele que deve ser removido são deslocados uma posição para baixo, sobre-escrevendo o nodo a ser removido; finalmente, o indicador de final de lista é atualizado (passa a indicar a posição anterior àquela que apontava). Caso a lista tenha resultado vazia, é feita a atualização dos indicadores de início e de final da lista. Sucesso retorna verdadeiro caso tenha sido possível efetuar a remoção do nodo solicitado.

```
Algoritmo 3.6 - RemoverKLLArr
   Entradas: LL (TipoLista)
             IL, FL (inteiro)
             K (inteiro)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variável auxiliar: Ind (inteiro)
```

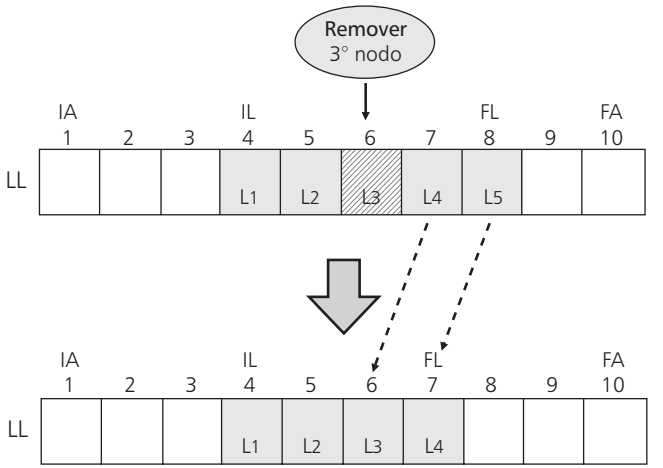


Figura 3.9 Remoção do "k-ésimo" nodo de uma lista linear.

```
início
  se (K ≤ 0) ou (K > FL-IL+1)
  então Sucesso ← falso
  senão início
        para Ind de IL+K-1 incr 1 até FL-1 faça
             LL[Ind] ← LL[Ind+1]
        FL ← FL-1
        se FL = IL–1
        então IL ← FL ← 0
        Sucesso ← verdadeiro
        fim
fim
```

Este algoritmo também pode ser otimizado: se o nodo a ser removido estiver localizado na primeira metade da lista, o deslocamento deve ser feito para baixo; caso contrário, o deslocamento deve ser feito para cima.

3.1.4 acesso a um nodo

Acessar um nodo de uma lista linear significa identificar um determinado nodo desta lista, para fins de consulta, alteração ou remoção. A identificação do nodo pode ser feita pela sua posição na lista ou através de alguma informação contida no nodo. Estas duas alternativas são analisadas a seguir.

acesso a nodo identificado por sua ordem na lista

Vamos considerar inicialmente o caso em que a identificação do nodo seja através de sua ordem na lista. Esta operação somente poderá ser executada se a ordem do nodo a ser acessado corresponder a um nodo da lista.

A seguir é apresentado um algoritmo para acessar o "k-ésimo" nodo (nodo de ordem K) da lista. Ele recebe, através de parâmetros, a ordem do nodo a ser acessado (K), o nome do arranjo onde a lista está armazenada (LL) e os índices IL e FL de controle do início e do final da lista. Se o acesso foi efetuado com sucesso, Sucesso retorna verdadeiro; caso contrário, retorna falso. O campo de informação do nodo acessado é retornado através do parâmetro InfoNodo.

Inicialmente o algoritmo verifica se o nodo solicitado pode ser acessado e, em caso afirmativo, faz o acesso diretamente ao nodo. Para que o nodo possa ser acessado, é necessário que: (1) o nodo faça parte da lista, isto é, que não seja solicitado um nodo de ordem superior ao comprimento da lista, definido pelos índices IL e FL; e (2) que a lista não seja vazia.

```
Algoritmo 3.7 - AcessarKLLArr
   Entradas: LL (TipoLista)
             IL, FL (inteiro)
             K (inteiro)
   Saídas: InfoNodo (TipoNodo)
           Sucesso (lógico)
início
  se (K ≤ 0) ou (K > FL-IL+1) ou (IL = 0)
  então Sucesso ← falso
  senão início
        InfoNodo ← LL[IL+K-1]
        Sucesso ← verdadeiro
        fim
fim
```

acesso a nodo identificado através de seu conteúdo

Quando a identificação do nodo a ser acessado é feita buscando um determinado valor contido em algum campo de informação, a lista deve ser percorrida, a partir de seu primeiro nodo, até que seja encontrado o valor buscado. Para este percurso é utilizada a posição física contígua dos nodos na memória, ou seja, são percorridos em ordem os índices do arranjo que implementa

a lista. Caso a informação buscada possa estar repetida em mais de um nodo, geralmente o acesso é feito ao primeiro nodo que a possui. Utilizaremos aqui um tipo de nodo que apresenta dois campos de informação, o campo `Valor`, que contém o valor buscado e o campo `Info`, que contém o restante das informações no nodo:

```
TipoNodo =  registro
               Valor: inteiro
               Info : TipoInfo
            fim registro
```

O algoritmo a seguir devolve a `Posição` (o índice no arranjo) do nodo para o qual o campo `Valor` é igual ao valor buscado (`ValBuscado`), que deve ser passado como parâmetro. Caso esta informação não seja encontrada no arranjo, é devolvido o valor `zero`. Este algoritmo somente poderá ser utilizado caso o espaço disponível para esta lista no arranjo não apresente um elemento de índice zero. O algoritmo deve receber, ainda, o nome do arranjo que implementa a lista e os índices de início e de final da lista.

Algoritmo 3.8 - PosValLLArr

```
   Entradas: LL (TipoLista)
             IL, FL (inteiro)
             ValBuscado (TipoValor)
   Saída: Posição(inteiro)
   Variáveis auxiliares:
            I (inteiro)
            Achou (lógico)
início
  Achou ← falso
  Posição ← 0
  I ← IL
  enquanto (I ≤ FL) e (não Achou)
  faça se LL[I].Valor = ValBuscado
       então início
             Posição ← I
             Achou ← verdadeiro
             fim
       senão I ← I+1
fim
```

Esta forma de busca é facilitada caso se tenha uma `lista ordenada`. Uma lista ordenada é uma lista linear onde são estabelecidas regras de precedência

entre os elementos da lista, ou seja, é uma coleção ordenada de nodos do mesmo tipo. Podem ser encontrados na literatura diversos algoritmos que fazem buscas em listas ordenadas. Como exemplo, apresentamos a seguir uma função que executa uma `busca binária` em uma lista ordenada. O método consiste na comparação do valor procurado com o nodo localizado no endereço médio do arranjo alocado para a lista. Se o valor procurado for menor que o valor contido naquele nodo, o processo é repetido para a primeira metade do arranjo; caso contrário, para a segunda metade. Se for igual, a busca encerra-se com sucesso. Enquanto o valor procurado não for encontrado, o processo de divisão ao meio é repetido até que não seja mais possível dividir. Nesse caso, constata-se que o valor procurado não está presente no arranjo.

O algoritmo a seguir devolve a `Posição` (o índice no arranjo) do nodo para o qual o campo `Valor` é igual ao valor buscado (`ValBuscado`), que deve ser passado como parâmetro. Note que agora se supõe que a lista esteja ordenada em ordem crescente do campo `Valor`. Caso esta informação não seja encontrada no arranjo, é retornada a `Posição` zero. O algoritmo deve receber, ainda, o nome do arranjo que implementa a lista, e os índices de início e final da lista.

```
Algoritmo 3.9 - PosValLLArrOrd
   Entradas: LL (TipoLista)
             IL, FL (inteiro)
             ValBuscado (TipoNodo)
   Saídas: Posição(inteiro)
   Variáveis auxiliares:
            Meio, Inf, Sup (inteiro)
            Achou (lógico)
início
  Achou ← falso
  Posição ← 0
  Inf ← IL
  Sup ← FL
  enquanto (Inf ≤ Sup) e (não Achou)
  faça início
       Meio ← (Inf+Sup) div 2    {DIVISÃO INTEIRA}
       se LL[Meio].Valor = ValBuscado
       então início
             Posição ← Meio
             Achou ← verdadeiro
             fim
       senão se LL[Meio].Valor < ValBuscado
             então Inf ← Meio+1    {BUSCA SEGUE SÓ NA METADE SUPERIOR}
```

```
            senão Sup ← Meio-1    {BUSCA SEGUE SÓ NA METADE INFERIOR}
      fim
fim
```

3.2 → listas lineares implementadas por contigüidade física com descritor

Para facilitar o acesso e a manipulação dos nodos de uma lista podem ser utilizados `descritores`. Descritores são variáveis que, como diz o nome, "descrevem" a lista – isto é, armazenam informações adicionais relativas à lista.

Os valores que o descritor deve armazenar são definidos pelo tipo de aplicação, de modo a facilitar as operações mais usuais. As informações contidas no descritor podem, por exemplo, mostrar:

- `localização da lista` – índices que delimitam o espaço do arranjo disponível para a lista;
- `acesso à lista` – índices de determinados nodos da lista, como o índice do primeiro nodo (permite percorrer a lista a partir do início), o índice do último nodo (para inclusão de novos nodos no final da lista), o índice do nodo que está no meio da lista (para pesquisa no caso de lista com valores ordenados), entre outros;
- `estrutura da lista` – informações relevantes a respeito da lista, como quantos nodos a lista apresenta no momento (comprimento da lista);
- `conteúdo da lista` – informações sobre o conteúdo dos campos de alguns nodos da lista. Por exemplo, se a aplicação utiliza muitas vezes o maior valor contido em algum campo da lista, este maior valor pode ser armazenado diretamente no descritor evitando, assim, que a lista seja freqüentemente percorrida em busca deste valor.

Como exemplo, a Figura 3.10 apresenta um descritor `DescrLL` para a lista implementada sobre o arranjo `LL`. A variável que representa o descritor apresenta cinco campos, todos inteiros, armazenando:

1 índice do primeiro nodo da lista;
2 índice do último nodo da lista;
3 comprimento da lista;
4 índice do nodo que contém o menor valor da lista;
5 índice do nodo que contém o maior valor da lista.

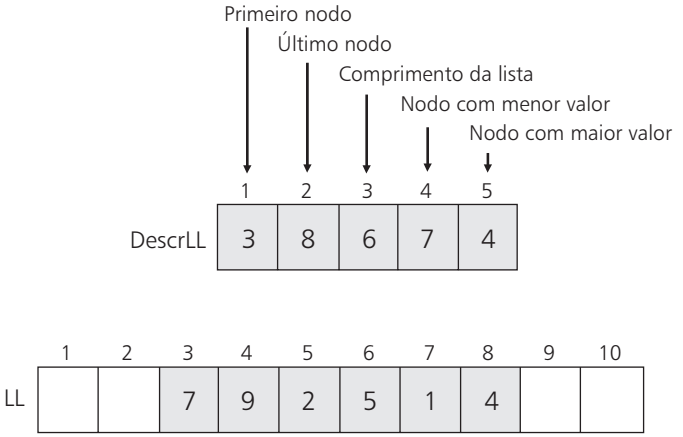


Figura 3.10 Descritor para uma lista linear implementada por contigüidade física.

O acesso a uma lista com descritor sempre deve ser feito através deste descritor. Por isso, é importante que o descritor tenha todas as informações necessárias para acessar e manipular a lista. Deste modo, detalhes da implementação da lista ficam afastados do usuário. Como exemplo, suponhamos que uma lista linear está implementada sobre um arranjo `LL` de `N` elementos, do tipo `TipoNodo` apresentado na Seção 3.1.4. Caso esta lista seja representada pelo descritor `DescrLL` da Figura 3.10 (um arranjo de cinco elementos), as informações a seguir seriam obtidas da seguinte forma:

`LL[DescrLL [1]].Info` – informação contida no primeiro nodo da lista;
`LL[DescrLL [2]].Valor` – `Valor` contido no último nodo da lista;
`LL[DescrLL [5]].Valor` – maior `Valor` contido na lista;
`DescrLL [3]` – comprimento da lista. A informação está contida diretamente no descritor, não sendo necessário percorrer a lista para obter seu comprimento.

Suponhamos agora que esta lista tem outro descritor, `DL`, no formato de um registro, onde o campo `IL` guarda o índice do início da lista, e o campo `FL`, o índice do final. Um terceiro campo, `MaiorValor`, traz o maior `Valor` contido na lista:

```
TipoDescritor = registro
                      IL: inteiro
                      FL: inteiro
```

```
        MaiorValor: inteiro
    fim registro
```

Algumas informações que podem ser obtidas desta lista são:

`LL[DL.IL].Info` – informação contida no primeiro nodo da lista;
`LL[DL.FL].Valor` – `Valor` contido no último nodo da lista;
`DL.MaiorValor` – maior valor contido no campo `Valor` de todos os nodos da lista. Neste caso, não é necessário acessar o arranjo, pois a informação já está contida no descritor.

Os algoritmos das operações apresentadas na seção 3.1 para as listas lineares implementadas sobre arranjos precisam ser adaptados caso seja utilizado um descritor para a lista. A principal alteração nos algoritmos é na forma do acesso à lista, que sempre deverá ser feito através das informações contidas no descritor. Nos algoritmos apresentados a seguir será utilizado o seguinte descritor:

```
TipoDescr = registro
              IA : inteiro
              IL : inteiro
              N : inteiro
              FL : inteiro
              FA : inteiro
            fim registro
```

onde `IA` é o índice de início da área disponível para a lista, `IL` é o índice do primeiro nodo da lista, `N` é o comprimento da lista, `FL` é o índice do último nodo da lista e `FA` é o índice do final da área disponível para a lista.

Lembramos que este descritor não é o único que pode ser definido para este tipo de lista – caso seja definida outra estrutura para o descritor, os algoritmos a seguir devem ser adaptados a ele.

3.2.1 criação de uma lista linear vazia com descritor

A criação de uma lista linear vazia com descritor consiste em alocar e inicializar seu descritor. A indicação de lista vazia, quando a lista é implementada por contigüidade física com descritor, vai depender dos campos presentes neste descritor. Diferentes estratégias podem ser utilizadas para dar esta indicação – os algoritmos que manipulam a lista devem conhecer a estratégia utilizada, que deve ser padronizada em uma aplicação. Se, por exemplo, o

descritor apresentar um campo com o número de nodos da lista, a indicação de lista vazia é feita diretamente através deste campo. Quando este campo não estiver presente, as indicações de início e de final da lista no momento considerado devem trazer a indicação da lista vazia – por exemplo, a indicação de início da lista indicando um índice menor do que o primeiro índice do arranjo. A Figura 3.11 ilustra duas estratégias para representar a lista vazia, considerando que a lista pode ocupar os elementos de índice 11 até 20 do arranjo LL. A estratégia utilizada para indicar que a lista está vazia influencia diretamente os algoritmos construídos para esta lista.

O algoritmo apresentado a seguir inicializa o descritor de uma lista linear. Note que este algoritmo independe do nome do arranjo utilizado para implementar a lista. Os valores dos índices de início e de final da área a ser ocupada pela lista devem ser fornecidos como parâmetros.

```
Algoritmo 3.10 – InicLLArrDescr
  Entradas: DL (TipoDescr)
            IniArea, FimArea (inteiro)
  Saída: DL (TipoDescr)
início
  DL.IL ← 0
  DL.FL ← 0
  DL.N ← 0
  DL.IA ← IniArea
  DL.FA ← FimArea
fim
```

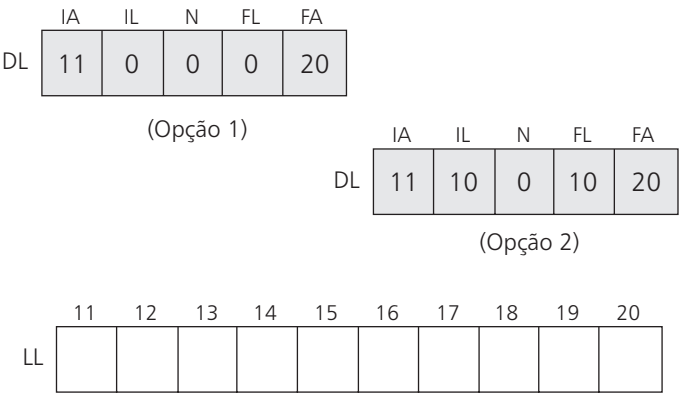


Figura 3.11 Estratégias para representar uma lista vazia.

3.2.2 inserção de um novo nodo

A seguir, são analisadas separadamente as três possíveis posições para inserção de um novo nodo: no início, no final e no meio da lista. Conforme visto na Seção 3.1.2, poderá ser necessário deslocar parte ou toda a lista existente para abrir espaço para o novo nodo.

Inserção no início da lista. Caso a lista não inicie na primeira posição disponível no arranjo (indicada pelo índice `IA`), a inserção de um novo nodo no início da lista é feita na posição anterior à primeira ocupada pela lista, sendo isto indicado no campo correspondente do descritor (`IL`). A Figura 3.12 ilustra esta situação. Caso no arranjo não haja espaço disponível antes do primeiro nodo, todos os outros nodos da lista devem ser deslocados para a frente, de modo a abrir espaço na frente (Figura 3.13). Isto só poderá ser feito se houver espaço no final.

Inserção no final da lista. De forma semelhante à inserção no início, caso haja espaço disponível no final da lista, a inserção é feita logo após a última posição ocupada, indicando no descritor que o final da lista avança uma posição (Figura 3.14). Caso não haja espaço no final da lista, mas

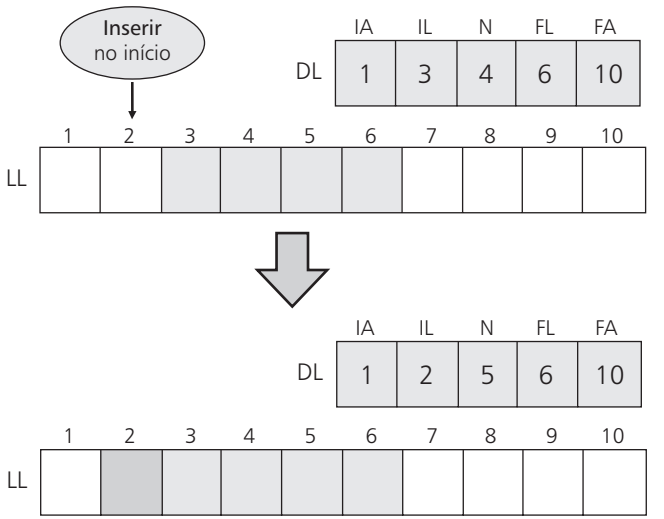


Figura 3.12 Inserção no início de lista, com descritor.

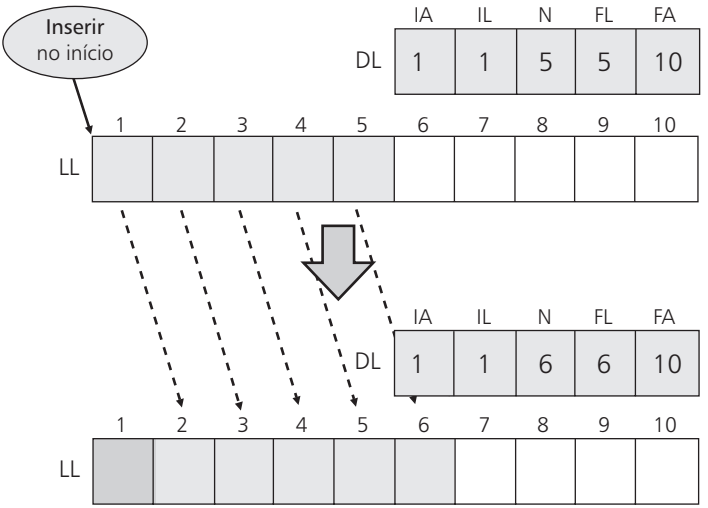


Figura 3.13 Inserção no início de lista, com descritor.

exista alguma posição disponível no início do arranjo, será necessário deslocar todos os outros nodos para baixo de modo a abrir espaço no final (Figura 3.15).

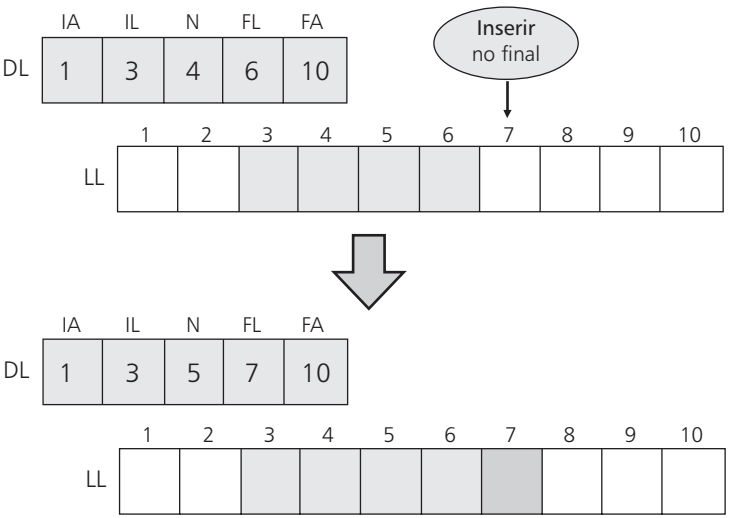


Figura 3.14 Inserção no final de lista, com descritor.

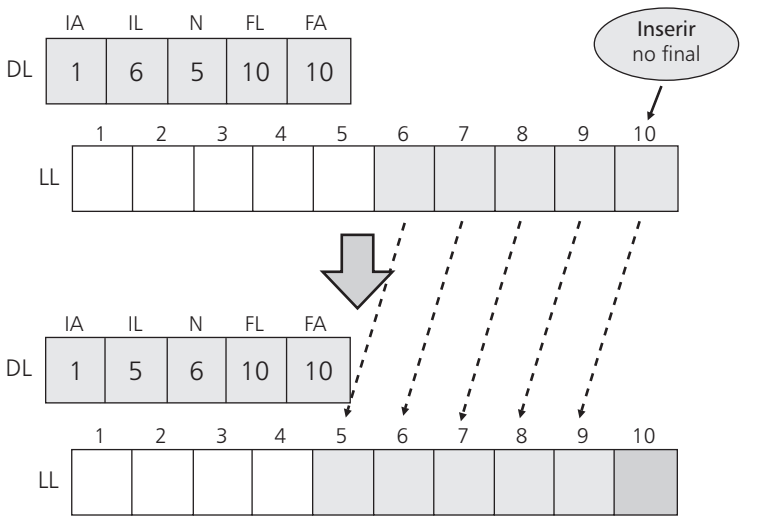


Figura 3.15 Inserção no final de lista com descritor, com deslocamento.

Inserção no meio da lista. Na inserção no meio de uma lista sempre haverá necessidade de deslocamento de nodos, para cima ou para baixo. O procedimento de inserção pode ser otimizado fazendo este deslocamento do lado mais próximo da posição onde o novo nodo será inserido (início ou final).

A seguir é apresentado um algoritmo que realiza esta operação de inserção. Ele recebe o nome do descritor (`DL`), as informações a serem inseridas no novo nodo (`Info`), a posição (ordem na lista) em que o novo nodo deverá ser inserido (`Pos`) e o nome do arranjo (`LL`). No final do processamento, o algoritmo devolve o arranjo `LL` e o descritor `DL` atualizados, além de informar se foi ou não possível realizar a inserção através do parâmetro lógico `Sucesso`. A inserção não será realizada caso não haja mais espaço disponível no arranjo para este novo nodo, ou quando a posição de inserção for inválida. É usada a estratégia de inserir o primeiro nodo da lista no meio do espaço disponível, procurando deste modo diminuir a necessidade de deslocamentos nas inserções posteriores.

```
Algoritmo 3.11 - InserirLLArrDesc
   Entradas: LL (TipoLista)
             DL (TipoDescr)
             Info (TipoNodo)
```

```
            Pos (inteiro)
   Saídas: LL (TipoLista)
           DL (TipoDescr)
           Sucesso (lógico)
   Variáveis auxiliares:
           K (inteiro)
           IndPos (inteiro)
início
  se (DL.N = DL.FA-DL.IA+1) ou (Pos > DL.N+1) ou (Pos < 1)
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        DL.N ← DL.N + 1
        se Pos = 1   {INSERIR NO INÍCIO}
        então início
              se DL.N = 1
              então início
                    DL.IL ← (DL.FA-DL.IA+1) div 2
                    DL.FL ← DL.IL
                    fim
              senão se DL.IL > DL.IA
                    então DL.IL ← DL.IL-1
                    senão início
                          para K de DL.FL até DL.IL incr-1 faça
                               LL[K+1] ← LL[K]
                          DL.FL ← DL.FL+1
                          fim
              LL[DL.IL] ← Info
              fim
        senão se Pos = DL.N   {INSERIR COMO ÚLTIMO}
              então início
                    se DL.FL < DL.FA
                    então DL.FL ← DL.FL+1
                    senão início
                          para K de DL.IL até DL.FL faça
                               LL[K-1] ← LL[K]
                          DL.IL ← DL.IL-1
                          fim
                    LL[DL.FL] ← Info
                    fim
              senão início   {INSERIR NO MEIO}
                    se DL.FL < DL.FA
                    então início
                          IndPos ← DL.IL+Pos-1
```

```
                        para K de DL.FL até IndPos incr-1 faça
                              LL[K+1] ← LL[K]
                        DL.FL ← DL.FL+1
                        fim
                  senão início
                        IndPos ← DL.IL+Pos-2
                        para K de DL.IL até IndPos faça
                             LL[K-1] ← LL[K]
                        DL.IL ← DL.IL–1
                        fim
                  LL[IndPos] ← Info
                  fim
      fim
fim
```

3.2.3 remoção de um nodo

A seguir é analisada a operação de remoção de um nodo de uma lista, sendo este identificado através de sua ordem na lista (remoção do "k-ésimo" nodo da lista). O nodo somente poderá ser removido se o valor fornecido para sua posição corresponder efetivamente a um dos nodos da lista, ou seja, se não for menor do que um ou maior do que o número de nodos da lista (N).

No algoritmo que implementa esta operação, apresentado a seguir, a remoção é realizada deslocando uma posição para baixo todos os nodos que seguem o nodo envolvido, de modo a não deixar espaço livre. Além disso, no descritor são atualizadas as indicações de final da lista (FL) e de tamanho da lista (N). Caso a operação de remoção resulte em uma lista vazia, os campos do descritor utilizados para fornecer esta informação também são atualizados. Além da ordem do nodo a ser removido (K), o algoritmo deve receber o nome do arranjo (LL) e o nome de seu descritor (DL). Ao término de sua execução, Sucesso informa se a remoção foi ou não executada com sucesso.

```
Algoritmo 3.12 - RemoverKLLArrDesc
   Entradas: LL (TipoLista)
             DL (TipoDescr)
             K (inteiro)
   Saídas: LL (TipoLista)
           DL (TipoDescr)
           Sucesso (lógico)
   Variável auxiliar: Ind (inteiro)
início
```

```
  se (K ≤ 0) ou (K > DL.N)
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        para Ind de DL.IL+K-1 até DL.FL-1 faça
             LL[Ind] ← LL[Ind+1]
        DL.FL ← DL.FL-1
        DL.N ← DL.N-1
        se DL.FL = DL.IL–1
        então DL.IL ← DL.FL ← 0
        fim
fim
```

De forma semelhante à remoção de um nodo de uma lista sem descritor, esta operação pode ser otimizada fazendo o deslocamento para o lado que apresente o menor número de nodos, atualizando adequadamente o descritor.

3.2.4 acesso a um nodo

Novamente podem ser consideradas duas situações: o acesso a um determinado nodo a partir de sua posição dentro da lista, ou a partir de uma informação contida no nodo.

acesso ao "k-ésimo" nodo de um lista com descritor

Para acessar um nodo definido por sua posição na lista é necessário fornecer o nome do arranjo que implementa a lista (`LL`) e o nome de seu descritor (`DL`), além da posição do nodo na lista (`K`). O algoritmo apresentado a seguir devolve as informações contidas na posição correspondente do arranjo através do parâmetro `InfoNodo`. No final, o parâmetro `Sucesso` terá valor falso caso a posição fornecida não esteja compreendida na lista, não tendo sido possível acessar o nodo requerido.

```
Algoritmo 3.13 - AcessarKLLArrDesc
   Entradas: LL (TipoLista)
             DL (TipoDescr)
             K (inteiro)
   Saídas: InfoNodo (TipoNodo)
           Sucesso (lógico)
início
```

```
  se (K ≤ 0) ou (K > DL.FL-DL.IL+1) ou (DL.N = 0)
  então Sucesso ← falso
  senão início
        InfoNodo ← LL[DL.IL+K-1]
        Sucesso ← verdadeiro
        fim
fim
```

acesso a nodo identificado através de campo de informação

Utilizando novamente o `TipoNodo` anteriormente definido, consideremos a operação de acesso a um nodo a partir da busca de um determinado valor contido em seu campo de informação. Supondo uma lista não ordenada, o algoritmo a seguir devolve a `Posição` (o índice no arranjo) do nodo cujo campo `Valor` seja igual ao valor buscado, passado como parâmetro (`ValBuscado`). Caso esta informação não seja encontrada no arranjo, é retornada `Posição` zero, assumindo que o espaço disponível para esta lista no arranjo não apresente algum elemento com este índice. O algoritmo deve receber, ainda, o nome do arranjo (`LL`) sobre o qual a lista está implementada, e o nome do descritor da lista (`DL`).

Algoritmo 3.14 - PosValLLArrDesc

```
   Entradas: LL (TipoLista)
             DL (TipoDescr)
             ValBuscado (TipoValor)
   Saída: Posição (inteiro)
   Variáveis auxiliares:
            I (inteiro)
            Achou (lógico)
início
  Achou ← falso
  Posição ← 0
  I ← DL.IL
  enquanto (I ≤ DL.FL) e (não Achou)
  faça se LL[I].Valor = Val
       então início
             Posição ← I
             Achou ← verdadeiro
             fim
       senão I ← I+1
fim
```

3.3 → listas lineares com ocupação circular do arranjo

Quando uma lista linear é implementada sobre um arranjo, a sucessiva inclusão e remoção de nodos pode levar a uma situação em que o final da lista esteja na última posição do arranjo, restando ainda espaço livre no seu início. Uma nova inclusão no final da lista não seria possível, a menos que todos os nodos fossem deslocados para baixo, de modo a disponibilizar espaço no final da lista. Uma forma de simplificar este gerenciamento é fazendo uma `ocupação circular` do arranjo: o novo nodo, no final da lista, seria incluído na primeira posição do arranjo, sinalizando através do indicador de final de lista (`FL`) o índice desta posição.

No caso de ocupação circular de um arranjo, o índice de final da lista poderá ser menor do que o índice de início da lista. A Figura 3.16 ilustra este tipo de ocupação do arranjo `LL`, representando uma lista de `6` nodos, que inicia no índice `9` (primeiro nodo da lista) e termina no índice `4` (último nodo da lista).

A ocupação circular do arranjo altera os algoritmos que implementam as operações sobre esta lista. A seguir são apresentados os algoritmos que implementam as operações básicas, considerando a possibilidade de ocupação circular do espaço disponível no arranjo, para listas sem descritor.

3.3.1 criação de uma lista linear vazia

A possibilidade de ocupar circularmente o arranjo não vai influir no procedimento de inicialização da lista (Algoritmo 3.1).

3.3.2 inserção de um novo nodo

A possibilidade de ocupar circularmente o espaço disponível para a lista influi diretamente nos procedimentos que implementam esta operação. Todas as

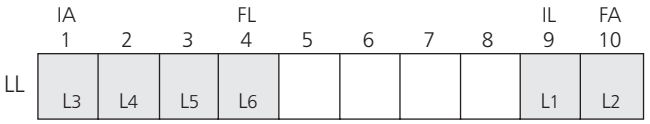


Figura 3.16 Ocupação circular do arranjo.

configurações devem ser analisadas, para contemplar a ocupação circular do espaço disponível.

No caso de inserção de um novo nodo no início de uma lista, caso haja espaço para tal, nunca será necessário o deslocamento de nodos: se a lista iniciar na primeira posição disponível da área reservada para a lista, o nodo será inserido na última posição disponível. Quando a inserção for no final da lista, se a lista terminar na última posição do arranjo, o nodo será inserido na primeira posição. Os índices `IL` e `FL` informam, respectivamente, onde inicia e onde termina a lista após a inserção.

O deslocamento de nodos é inevitável na inserção em uma posição no meio da lista. Esta situação é ilustrada na Figura 3.17, com a inserção de um novo nodo na quinta posição da lista, sendo os nodos acima desta posição deslocados uma posição para cima.

A seguir é apresentado um algoritmo que executa esta operação. O algoritmo deve receber a posição (ordem na lista) em que o nodo deve ser inserido (`K`), o valor que vai compor o campo de informação do novo nodo (`InfoNodo`), além das informações sobre nome do arranjo e índices dos limites da área disponível para a lista (`IA` e `FA`) e dos limites da lista (`IL` e `FL`). Ao final de sua execução, o parâmetro `Sucesso` retorna verdadeiro se a inserção foi executada com sucesso, caso contrário retorna falso.

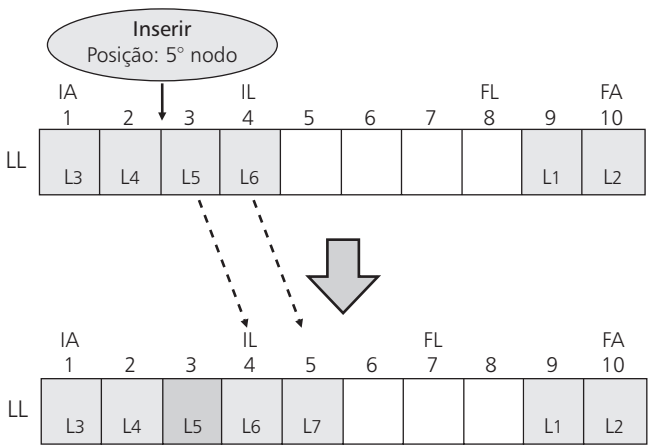


Figura 3.17 Inserção no meio de uma lista, com ocupação circular do arranjo.

O algoritmo inicialmente verifica se é possível inserir o novo nodo, calculando o número de nodos que a lista apresenta (N). Com base neste valor, verifica se ainda existe espaço livre no arranjo, e se a ordem na lista onde o nodo deve ser inserido é válida. No caso de ser possível inserir o nodo, o algoritmo analisa diversas posições diferentes de inserção (primeiro nodo da lista, no início da lista, no final, ou em uma posição intermediária) e, para cada uma delas, calcula a posição de inserção (Pos), atualizando os índices de início e de final da lista. Finalmente, o novo nodo é inserido na posição Pos calculada. Deslocamentos para abrir espaço para o novo nodo são necessários somente quando a inserção é realizada no meio da lista.

Algoritmo 3.15 - InserirLLCirArrPosK

```
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             K (inteiro)
             InfoNodo (TipoNodo)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variáveis auxiliares: Ind, N, Pos (inteiro)
início
  se IL = 0 {LISTA VAZIA}
  então N ← 0
  senão se IL ≤ FL
        então N ← FL-IL+1
        senão N ← (FA-IL+1) + (FL-IA+1)
  se (N = FA-IA+1) ou (K > N+1) ou (K < 1) ou (N=0 e K≠1)
  então Sucesso ← falso
  senão início
        se (N = 0)
        então início {PRIMEIRO NODO}
              Pos ← IA
              IL ← FL ← IA
              fim
        senão início
              se (IL+K-1 ≤ FA)
              então Pos ← IL+K-1
              senão Pos ← (IA-1) + (K - (FA-IL+1))
              se K = 1 {INSERÇÃO NO INÍCIO}
              então se IL > IA
                    então início
                          Pos ← IL-1
                          IL ← IL-1
```

```
                    fim
               senão início
                    IL ← FA
                    Pos ← FA
                    fim
          senão se K = N+1 {INSERÇÃO NO FINAL}
               então se Pos = FL+1
                    então FL ← FL+1
                    senão FL ← IA
               senão {INSERÇÃO NO MEIO}
                    se (IL ≤ FL) e (FL < FA)
                        ou (IL > FL) e (Pos < IL)
                    então início
                         para Ind de FL incr-1 até Pos faça
                           LL[Ind+1] ← LL[Ind]
                         FL ← FL+1
                         fim
                    senão início
                         se FL ≠ FA
                         então para Ind de FL incr-1 até IA
                              faça LL[Ind+1] ← LL[Ind]
                         LL[IA] ← LL[FA]
                         para Ind de FA-1 incr-1 até Pos
                             faça LL[Ind+1] ← LL[Ind]
                         se FL = FA
                         então FL ← IA
                         senão FL ← FL+1
                         fim
          fim
     LL[Pos] ← InfoNodo
     Sucesso ← verdadeiro
     fim
fim
```

3.3.3 remoção de um nodo

A ocupação circular do arranjo influi na remoção, na medida em que ao efetuar a remoção, o restante da lista deve ser atualizado. Um algoritmo correspondente à operação de remoção do "k-ésimo" nodo de uma lista circular é apresentado a seguir. Ele deve receber a ordem do nodo a ser removido (K), o nome do arranjo e os índices de controle da lista e do espaço a ser ocupado no arranjo. A remoção somente poderá ser realizada caso esta ordem esteja compreendida dentro da lista.

O algoritmo inicia calculando o número de nodos da lista (N). Com base neste valor, verifica se a ordem do nodo a ser removido é válida. Se a ordem estiver correta, calcula a posição do nodo a ser removido (Pos). Em seguida, analisa as diferentes posições do nodo a ser removido (o único nodo, o primeiro nodo, o último nodo, ou um nodo no meio da lista), fazendo o deslocamento dos demais nodos para manter a contigüidade da lista, e atualizando o indicador de final de lista.

Algoritmo 3.16 - RemoverKLLCirArr

```
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             K (inteiro)
             InfoNodo (TipoNodo)
   Saídas: LL (TipoLista)
           IL, FL (inteiro)
           Sucesso (lógico)
   Variáveis auxiliares: Ind, N, Pos (inteiro)
início
  se IL = 0 {LISTA VAZIA}
  então N ← 0
  senão se IL ≤ FL
        então N ← FL-IL+1
        senão N ← (FA-IL+1) + (FL-IA+1)
  se (N = 0) ou (K ≤ 0) ou (K > N)
  então Sucesso ← falso
  senão início
        se N = 1 {REMOÇÃO DO ÚNICO}
        então IL ← FL ← 0
        senão se K = 1 {REMOÇÃO DO PRIMEIRO}
              então se IL < FA
                    então IL ← IL+1
                    senão IL ← IA
              senão se K = N {REMOÇÃO DO ÚLTIMO}
                    então se FL = IA
                          então FL ← FA
                          senão FL ← FL-1
                    senão início {REMOÇÃO NO MEIO}
                          se (IL+K-1 ≤ FA)
                          então Pos ← IL+K-1
                          senão Pos ← IA + (K - (FA-IL+1)-1)
                          se (IL < FL) ou ((IL>FL) e (K>=FA-IL+2))
                          então início
                                para Ind de Pos até FL-1
```

```
                                    faça LL[Ind] ← LL[Ind+1]
                               FL ← FL-1
                               fim
                          senão início
                               para Ind de Pos+1 até FA
                                    faça LL[Ind-1] ← LL[Ind]
                               LL[FA] ← LL[IA]
                               se FL ≠ IA
                               então para Ind de IA+1 até FL
                                       faça LL[Ind-1] ← LL[Ind]
                               se FL = IA
                               então FL ← FA
                               senão FL ← FL-1
                               fim
                          fim
        Sucesso ← verdadeiro
        fim
fim
```

3.3.4 acesso a um nodo

A seguir serão analisadas as duas formas básicas para acessar um nodo de uma lista: através da posição relativa do nodo na lista ou através do conteúdo de algum campo do nodo.

acesso a nodo identificado por sua ordem na lista

O algoritmo que faz este acesso necessita receber as informações que dizem respeito ao arranjo (seu nome e os índices de seu início e final da área disponível para a lista), as informações relativas à lista (seu início e seu final) e a ordem na lista do nodo a ser acessado (`K`). A operação somente poderá ser executada se a ordem do nodo a ser acessado corresponder a um nodo da lista. Ao final da execução, o algoritmo informa se a operação teve `Sucesso` e, caso tenha encontrado o nodo buscado, retorna o seu campo de informação através do parâmetro `InfoNodo`.

Inicialmente é calculado o número de nodos da lista (`N`), para verificar se o valor recebido para a ordem do nodo a ser acessado é válido (`K`). Caso seja válido, o algoritmo calcula a posição em que se localiza o nodo (`Pos`), considerando a ocupação circular do arranjo. O acesso ao nodo é feito diretamente pela sua posição.

```
Algoritmo 3.17 - AcessarKLLCirArr
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             K (inteiro)
   Saídas: InfoNodo (TipoNodo)
           Sucesso (lógico)
   Variáveis auxiliares: N, Pos (inteiro)
início
  se IL = 0 {LISTA VAZIA}
  então N ← 0
  senão se IL ≤ FL
        então N ← FL-IL+1
        senão N ← (FA-IL+1) + (FL-IA+1)
  se (N = 0) ou (K ≤ 0) ou (K > N)
  então Sucesso ← falso
  senão início
        se (IL+K-1 ≤ FA)
        então Pos ← IL+K-1
        senão Pos ← IA + (K - (FA-IL+1)-1)
        InfoNodo ← LL[Pos]
        Sucesso ← verdadeiro
        fim
fim
```

acesso a nodo identificado através de seu conteúdo

A identificação de um nodo a partir de seu conteúdo é feita percorrendo a lista, a partir de seu início, até que o nodo buscado seja encontrado ou até o final da lista.

A seguir é apresentado um algoritmo que localiza o nodo que contém um determinado valor `ValBuscado` em uma lista implementada sobre um arranjo, considerando que o espaço disponível deste arranjo é utilizado de maneira circular. É utilizado novamente o tipo de nodo definido na seção 3.1.3, com dois campos de informação, `Valor` (onde está o valor buscado) e `Info` (restante das informações no nodo). O algoritmo devolve a `Posição` (o índice no arranjo) do nodo para o qual o campo `Valor` é igual ao valor buscado. Caso esta informação não seja encontrada no arranjo, devolve `Posição` igual a zero, o que limita a utilização desta função para os casos em que o espaço disponível para esta lista no arranjo não inclua o índice nulo. O algoritmo deve receber, ainda, o nome do arranjo que implementa a lista, e os índices de início e de final da lista e da área disponível sobre o arranjo.

Inicialmente é calculado o número de nodos da lista (N), para verificar se a lista não está vazia. Caso isto não ocorra, passa a ser executada a busca. A busca é feita em todos os nodos, a partir do primeiro da lista. Quando é alcançado o nodo que está no final da área do arranjo (FA), o próximo a ser examinado é aquele que está no início da área (IA). Este algoritmo não considera ordenação nos valores contidos no campo Valor dos nodos.

Algoritmo 3.18 - PosValLLCirArr

```
   Entradas: LL (TipoLista)
             IA, FA, IL, FL (inteiro)
             ValBuscado (TipoValor)
   Saída: Posição (inteiro)
   Variáveis auxiliares:
             N, I (inteiro)
             Achou, Sair (lógico)
início
  Achou ← falso
  Posição ← 0
  se IL = 0 {LISTA VAZIA}
  então N ← 0
  senão se IL ≤ FL
        então N ← FL-IL+1
        senão N ← (FA-IL+1) + (FL-IA+1)
  se N ≠ 0
  então início
        I ← IL
        Sair ← falso
        enquanto (não Achou) e (não Sair)
        faça se LL[I].Valor = Val
             então início
                   Posição ← I
                   Achou ← verdadeiro
                   fim
             senão se I = FL
                   então Sair ← verdadeiro
                   senão se I ≠ FA
                         então I ← I+1
                         senão I ← IA
        fim
fim
```

3.4 ⇢ listas lineares encadeadas

Nas seções anteriores foi visto que se pode utilizar um arranjo para representar um conjunto de dados contíguos. O arranjo é uma das formas de representação de listas que aproveita a seqüencialidade da memória, ou seja, ocupa um espaço contíguo na memória e permite acessar qualquer um de seus elementos. Entretanto, não é uma estrutura muito flexível, pois é necessário fazer uma estimativa do número máximo de nodos da lista. Uma alternativa é usar uma estrutura de dados que cresça conforme seja necessário inserir novos nodos e que igualmente diminua quando nodos anteriormente inseridos são excluídos. Essas estruturas de dados são denominadas `estruturas dinâmicas` e armazenam cada nodo da lista por alocação dinâmica de memória.

Uma forma de implementar estruturas dinâmicas é através do encadeamento, onde os nodos são ligados entre si para indicar a ordem existente entre eles. Assim, a ordem dos nodos é definida por uma informação contida no próprio nodo, que informa qual o próximo nodo da lista. Esta informação está contida em um campo denominado `campo de elo`. Os nodos de uma lista encadeada podem estar alocados em quaisquer posições na memória, contíguas ou não, uma vez que a ordem é fornecida explicitamente através do campo de elo, sendo totalmente independente de sua posição física. Dessa forma, a contigüidade de uma lista linear encadeada é lógica.

A forma mais comum de implementar listas encadeadas é utilizando, no campo de elo, diretamente o endereço físico do próximo nodo da lista. As linguagens de programação disponibilizam variáveis que podem armazenar endereços físicos, geralmente denominadas de ponteiros (apontadores). O programa de aplicação não tem acesso direto ao endereço contido em uma variável do tipo ponteiro, mas permite que este endereço seja testado e utilizado para alcançar o próximo nodo da lista. Assim, o espaço total de memória gasto pela estrutura é proporcional ao número de nodos armazenados na lista.

A Figura 3.18 apresenta graficamente uma lista linear encadeada de quatro nodos, na qual o ponteiro `PtLista` contém o endereço do primeiro nodo da lista. Em cada nodo, o campo `Elo` "aponta" para o endereço do próximo nodo (indicado graficamente através de uma seta). O último nodo da lista apresenta, através de uma linha inclinada no campo `Elo`, um endereço nulo representando o final da lista, ou seja, indicando que o campo de elo não está apontando para outro nodo.

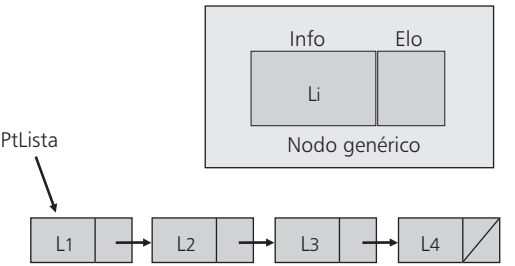


Figura 3.18 Lista encadeada.

Uma lista linear implementada através de encadeamento deve apresentar obrigatoriamente:

- um ponteiro para o primeiro nodo da lista, através do qual a lista pode ser acessada. Através deste ponteiro é feito todo o acesso à lista. Pode ser uma variável simples do tipo ponteiro, ou um campo (do tipo ponteiro) de alguma variável complexa;
- encadeamento entre os nodos, através de algum campo de elo; e
- uma indicação de final da lista. Este endereço nulo deverá ser passível de teste, para que os algoritmos possam detectar quando o final da lista foi alcançado.

A seguir são apresentados os algoritmos relativos às operações básicas que podem ser executadas sobre listas lineares, quando implementadas através de encadeamento. Os seguintes tipos de dados são utilizados nos algoritmos apresentados:

```
TipoPtNodo = ↑TipoNodo
TipoNodo = registro
              Info: TipoInfoNodo
              Elo : TipoPtNodo
           fim registro
```

3.4.1 criação de uma lista linear encadeada

Antes de iniciar a construção da lista, o ponteiro que indica seu primeiro elemento deve ser inicializado com um endereço nulo, indicando que a lista está vazia. No procedimento que realiza esta inicialização, apresentado a seguir, este ponteiro é denominado `PtLista`.

```
Algoritmo 3.19 – InicializarLLEnc
  Entradas: –
  Saída: PtLista (TipoPtNodo)
início
  PtLista ← nulo
fim
```

3.4.2 inserção de um novo nodo

Para inserir um nodo em uma lista encadeada deve-se, inicialmente, alocar o novo nodo e preenchê-lo com o valor correspondente. Caso não se consiga alocar um novo nodo por falta de espaço físico, isto deve ser informado ao usuário. Em seguida, o novo nodo deve ser inserido na posição solicitada na lista, o que requer tão somente a adequação dos campos de elo dos nodos que vão ficar antes e depois deste novo nodo: o campo de elo do nodo anterior deverá apontar para o novo nodo, e o campo de elo do novo nodo deverá conter o endereço do próximo na lista. Nunca haverá necessidade de deslocar nodos de sua posição física para efetivar a inserção do novo nodo, como freqüentemente acontecia no caso da implementação através de contigüidade física visto anteriormente.

Em seguida são analisadas as três posições em que o nodo pode ser inserido: no início da lista, no final, e em uma posição intermediária.

Inserção no início da lista encadeada. Para inserir o novo nodo no início da lista deve-se, após alocar o nodo e preenchê-lo com o valor correspondente, apontar seu campo de elo para o endereço daquele que era o primeiro e atualizar o ponteiro de início da lista para o novo nodo. Caso a lista esteja vazia, este passará a ser seu único nodo.

A Figura 3.19 mostra a inserção de um novo nodo no início de uma lista que tinha três nodos. O novo nodo passará a ocupar a primeira posição, sendo agora este o nodo indicado pelo ponteiro da lista `PtLista`. O campo de elo do nodo inserido apontará para aquele que era antes o primeiro nodo da lista, que agora passa a ser o segundo.

A seguir é apresentado um algoritmo que realiza a inserção de um novo nodo no início de uma lista linear encadeada. Ele recebe o endereço do primeiro nodo da lista (`PtLista`) e os `Dados` com os quais deverá ser preenchido o campo de informações do novo nodo. O parâmetro `Sucesso` retorna falso caso não seja

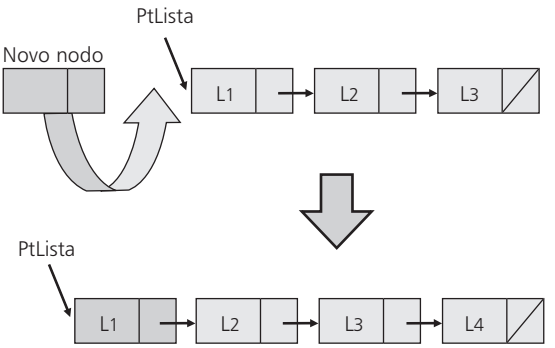


Figura 3.19 Inserção no início de uma lista encadeada.

possível alocar o novo nodo. Caso a inserção seja realizada, o ponteiro para o início da lista (`PtLista`) é atualizado para o novo nodo inserido.

```
Algoritmo 3.20 - InserirIniLLEnc
   Entradas: PtLista (TipoPtNodo)
             Dados (TipoInfoNodo)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variável auxiliar: Pt (TipoPtNodo)
início
  alocar(Pt)
  se Pt = nulo
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        Pt↑.Info ← Dados
        Pt↑.Elo ← PtLista
        PtLista ← Pt
        fim
fim
```

Inserção no final da lista encadeada. A inserção de um novo nodo no final da lista requer somente encadear aquele que era o último nodo da lista com o novo nodo, ou seja, fazer com que seu campo de elo aponte para o endereço onde foi alocado o novo nodo, conforme mostrado na Figura 3.20. O ponteiro que guarda o início da lista somente será afetado no caso em que a lista era vazia, quando então apontará para este novo nodo.

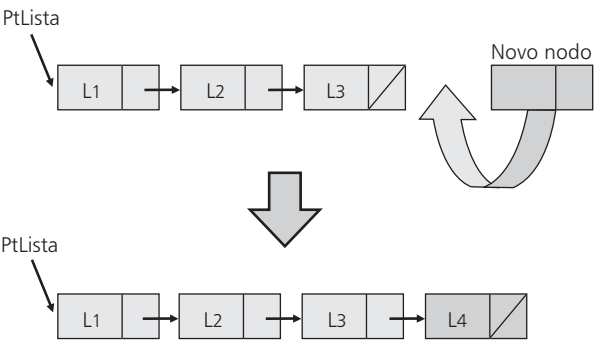


Figura 3.20 Inserção no final de uma lista encadeada.

Para alcançar o último nodo da lista, é necessário percorrê-la, a partir de seu primeiro nodo. A necessidade de percorrer uma lista encadeada aparece em muitas aplicações, com o objetivo de procurar um determinado nodo ou de realizar alguma operação em alguns ou todos os nodos (procurar alguma informação, imprimir conteúdo, alterar valor, etc.). Isto pode ser feito com o auxílio de uma variável auxiliar, do tipo ponteiro, que inicialmente assume o endereço do primeiro nodo da lista, avançando progressivamente para os próximos nodos através dos campos de elo, até alcançar o final da lista, indicado pelo campo de elo nulo.

O algoritmo apresentado a seguir realiza a inserção no final da lista, retornando `Sucesso` com valor falso somente no caso de não existir espaço físico para alocar o novo nodo. É utilizada uma variável do tipo ponteiro (`P1`) para guardar o endereço do novo nodo, e outra (`P2`) para percorrer a lista até o último nodo, identificado por ter o campo de elo nulo. Caso a lista esteja inicialmente vazia, o novo nodo será indicado pelo ponteiro da lista (`PtLista`).

```
Algoritmo 3.21 - InserirFimLLEnc
   Entradas: PtLista (TipoPtNodo)
             Dados (TipoInfoNodo)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variáveis auxiliares: P1, P2 (TipoPtNodo)
início
   alocar(P1)
   se P1 = nulo
   então Sucesso ← falso
```

```
   senão início
         Sucesso ← verdadeiro
         P1↑.Info ← Dados
         P1↑.Elo ← nulo
         se PtLista = nulo
         então  PtLista ← P1
         senão início
               P2 ← PtLista
               enquanto P2↑.Elo ≠ nulo
               faça P2 ← P2↑.Elo
               P2↑.Elo ← P1
               fim
         fim
fim
```

Inserção no meio da lista encadeada. A inserção de um novo nodo no meio de uma lista encadeada é feita adequando os campos de elo dos nodos anterior e posterior ao novo nodo. A Figura 3.21 mostra como é feita a inserção de um nodo na terceira posição de uma lista que originalmente tinha três nodos.

A seguir é apresentado um algoritmo que executa esta operação. Ele recebe, como parâmetro, a ordem que o novo nodo deverá ocupar na lista (K), além do endereço do início da lista e dos dados a serem inseridos no novo nodo. O parâmetro `Sucesso` retorna falso quando não existir espaço físico para

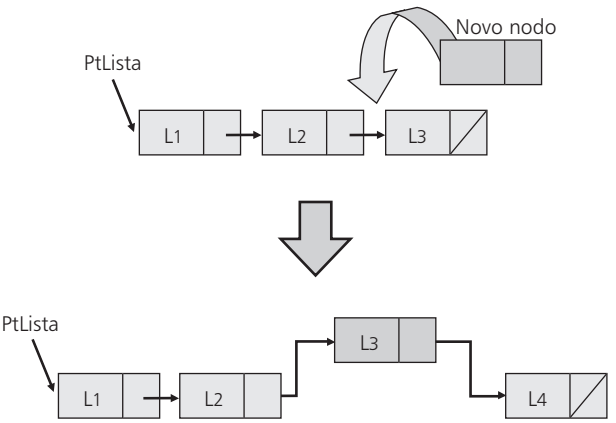


Figura 3.21 Inserção no meio de uma lista encadeada.

alocar o novo nodo, ou quando a posição requerida para inserção não for compatível com o número de nodos da lista. No caso da inserção ter sucesso, o ponteiro `PtAnt` percorre a lista até o nodo anterior à posição de inserção, sendo feito o ajuste de seu campo de elo para o novo nodo (`PtNovo`), e do campo de elo do novo nodo para aquele que era apontado pelo nodo anterior (`PtAnt`). Ressaltamos que esse algoritmo pode ser utilizado para a inserção de um nodo em qualquer posição da lista, incluindo a primeira e a última posições.

Algoritmo 3.22 - InserirKLLEnc

```
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
             Dados (TipoInfoNodo)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variáveis auxiliares: PAnt, PtNovo (TipoPtNodo)
início
  alocar(PtNovo)
  se PtNovo = nulo
  então Sucesso ← falso
  senão se ((PtLista = nulo) e (K ≠ 1)) ou (K < 1)
        então início
              liberar(PtNovo)
              Sucesso ← falso
              fim
        senão se K = 1
              então início
                    PtNovo↑.Info ← Dados
                    PtNovo↑.Elo ← PtLista
                    PtLista ← PtNovo
                    Sucesso ← verdadeiro
                    fim
              senão início
                    PtAnt ← PtLista
                    enquanto (PtAnt↑.Elo ≠ nulo) e (K > 2)
                    faça início
                         PtAnt ← PtAnt↑.Elo
                         K ← K - 1
                         fim
                    se K > 2
                    então início
                          liberar(PtNovo)
                          Sucesso ← falso
```

```
                fim
            senão início
                PtNovo↑.Info ← Dados
                PtNovo↑.Elo ← PtAnt↑.Elo
                PtAnt↑.Elo ← PtNovo
                Sucesso ← verdadeiro
                fim
            fim
fim
```

3.4.3 remoção de um nodo

A remoção de um nodo de uma lista linear encadeada é feita simplesmente mudando o encadeamento dos nodos anterior e posterior ao nodo a ser removido: o nodo imediatamente anterior apontará para aquele que era o seguinte do nodo excluído da lista. Caso o nodo liberado seja o primeiro, o endereço do segundo deverá ser copiado para o ponteiro de início da lista. Caso seja o último, o anterior deverá ficar com campo de elo nulo. Após este encadeamento, que garante a continuidade da lista, a posição ocupada pelo nodo removido é liberada. A Figura 3.22 mostra a remoção do terceiro nodo (K = 3) de uma lista de quatro nodos. Para implementar esta operação, a lista deve ser percorrida a partir do seu primeiro nodo, indicado pelo ponteiro `PtLista`, com a finalidade de localizar o nodo a ser removido.

O próximo algoritmo remove um nodo identificado pela sua posição na lista (K). `Sucesso` retorna falso caso a ordem solicitada seja menor do que o primeiro nodo da lista. A partir do endereço do primeiro nodo da lista, esta é

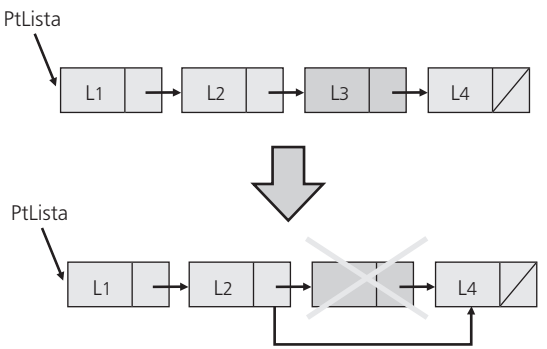


Figura 3.22 Remoção de um nodo de uma lista linear encadeada.

percorrida até que seja alcançado o nodo de ordem K, ou que seja alcançado o final da lista – neste último caso, a ordem solicitada está fora da lista, retornando Sucesso falso. Uma vez localizado o nodo a ser removido, o nodo anterior a ele (cujo endereço foi guardado durante o percurso da lista, no ponteiro PtAnt) é encadeado com o nodo seguinte àquele que foi excluído. Caso o nodo a ser removido seja o primeiro, o endereço de acesso à lista é atualizado.

```
Algoritmo 3.23 - RemoverKLLEnc
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variáveis auxiliares: PtAnt, PtK (TipoPtNodo)
início
  se K < 1
  então Sucesso ← falso
  senão início
        PtK ← PtLista
        PtAnt ← nulo
        enquanto (PtK ≠ nulo) e (K > 1)
        faça início
             K ← K-1
             PtAnt ← PtK
             PtK ← PtK↑.Elo
             fim
        se PtK = nulo
        então Sucesso ← falso
        senão início
              se PtK = PtLista
              então PtLista ← PtLista↑.Elo
              senão  PtAnt↑.Elo ← PtK↑.Elo
              liberar(PtK)
              Sucesso ← verdadeiro
              fim
        fim
fim
```

3.4.4 acesso a um nodo

O acesso a um determinado nodo de uma lista encadeada requer que a lista seja percorrida, a partir de seu primeiro nodo, até o nodo buscado. Este nodo

pode ser identificado através de alguma informação nele contida, ou pela sua ordem na lista. Diferentemente do caso de alocação de uma lista sobre um arranjo, não existe, no caso de lista encadeada, a possibilidade de acessar diretamente algum nodo, sem que ele seja alcançado percorrendo a lista a partir de seu primeiro nodo.

O algoritmo a seguir tem por finalidade localizar um determinado nodo de uma lista linear encadeada, identificado pela sua ordem na lista (K), devolvendo o endereço físico deste nodo (PtK) ao programa que o acionou. Caso a posição do nodo não seja compatível com o tamanho da lista, ou a lista esteja vazia, PtK retorna o endereço nulo.

```
Algoritmo 3.24 - AcessarKLLEnc
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
   Saída: PtK (TipoPtNodo)
início
  se (K < 1) ou (PtLista = nulo)
  então PtK ← nulo
  senão início
        PtK ← PtLista
        enquanto (PtK ≠ nulo) e (K > 1)
        faça início
             K ← K-1
             PtK ← PtK↑.Elo
             fim
        se K > 1
        então PtK ← nulo
        fim
fim
```

3.4.5 destruição de uma lista linear encadeada

Caso uma lista linear encadeada não seja mais necessária durante a execução de uma aplicação, as posições ocupadas por ela devem ser liberadas. Isto é feito percorrendo a lista a partir de seu primeiro nodo, liberando cada uma das posições ocupadas pelos nodos. No final, a variável do tipo ponteiro que faz referência à lista (PtLista) deverá conter o endereço nulo. O procedimento que realiza esta operação é apresentado a seguir.

```
Algoritmo 3.25 - DestruirLLEnc
   Entrada: PtLista (TipoPtNodo)
```

```
   Saída: PtLista (TipoPtNodo)
   Variável auxiliar: Pt (TipoPtNodo)
begin
   enquanto PtLista ≠ nulo
   faça início
        Pt ← PtLista
        PtLista ← PtLista↑.Elo
        liberar(Pt)
        fim
   liberar(PtLista)
fim
```

3.5 → lista encadeada circular

Quando uma lista linear encadeada apresenta um elo ligando o último nodo ao primeiro, ela se torna uma lista circular (Figura 3.23). Neste caso, qualquer nodo pode ser utilizado para fazer acesso à lista, pois toda ela pode ser percorrida a partir de qualquer nodo. Mesmo sendo circulares, estas listas também apresentam um ponteiro para fazer referência à lista (`PtLista`), sendo o nodo acessado em primeiro lugar identificado como o primeiro da lista.

As operações apresentadas para listas simplesmente encadeadas podem ser adaptadas para listas circulares. A **criação da lista** não é alterada, uma vez que somente o ponteiro da lista é inicializado. A lista é vazia quando o ponteiro para o seu início é nulo. Quando a lista apresenta um só nodo, seu campo de elo aponta para ele mesmo. A operação de **destruição da lista** é praticamente a mesma, mudando somente a identificação do último nodo a ser liberado, que é aquele que contém o endereço do primeiro em seu campo de elo. Já as operações de inserção, remoção e busca apresentam algumas alterações relevantes, que serão analisadas a seguir.

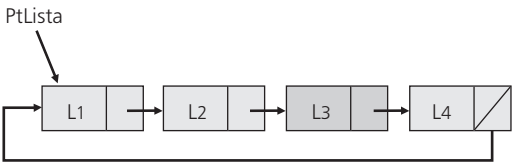


Figura 3.23 Lista encadeada circular.

3.5.1 inserção de um novo nodo

Quando a lista é circular, o nodo a ser inserido sempre apresentará um endereço no seu campo de elo, encadeando-o com o seguinte. No caso de ser o único nodo da lista, seu próprio endereço será colocado em seu campo de elo.

O algoritmo que executa esta operação recebe, como parâmetro, a ordem que o novo nodo deverá ocupar na lista (K), além do endereço do início da lista e dos dados a serem inseridos no novo nodo. O retorno do parâmetro Sucesso será falso quando não existir espaço físico para alocar o novo nodo, ou caso a posição requerida para a inserção não seja compatível com o número de nodos da lista. A inserção de um nodo como primeiro da lista recebe tratamento diferenciado de qualquer outra posição, sendo seu endereço colocado no ponteiro da lista (PtLista). A inserção requer que seja localizada a posição de inserção, o que é feito por um ponteiro percorrendo a lista até o nodo anterior à posição de inserção, realizando em seguida os encadeamentos necessários.

```
Algoritmo 3.26 - InserirKLLEncCir
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
             Dados (TipoInfoNodo)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variáveis auxiliares: PtAnt, PtNovo (TipoPtNodo)
início
  alocar(PtNovo)
  se PtNovo = nulo
  então Sucesso ← falso
  senão se ((PtLista = nulo) e (K ≠ 1)) ou (K < 1)
        então início
              liberar(PtNovo)
              Sucesso ← falso
              fim
        senão início
              Sucesso ← verdadeiro
              PtNovo↑.Info ← Dados
              se K = 1 {INSERE COMO PRMEIRO}
              então início
                    se PtLista = nulo
                    então PtNovo↑.Elo ← PtNovo
                    senão início
```

```
                        PtAnt ← PtLista
                        enquanto PtAnt↑.Elo ≠ PtLista
                        faça PtAnt ← PtAnt↑.Elo
                        PtNovo↑.Elo ← PtLista
                        PtAnt↑.Elo ← PtNovo
                        fim
                  PtLista ← PtNovo
                  fim
            senão início
                  PtAnt ← PtLista
                  enquanto (PtAnt↑.Elo ≠ PtLista) e (K > 2)
                  faça início
                       PtAnt ← PtAnt↑.Elo
                       K ← K - 1
                       fim
                  se K > 2  {ORDEM FORA DA LISTA}
                  então início
                        liberar(PtNovo)
                        Sucesso ← falso
                        fim
                  senão início
                        PtNovo↑.Info ← Dados  {INSERE NO MEIO}
                        PtNovo↑.Elo ← PtAnt↑.Elo
                        PtAnt↑.Elo ← PtNovo
                        fim
                  fim
            fim
fim
```

3.5.2 remoção de um nodo

A remoção de um nodo de uma lista encadeada circular requer a localização do nodo imediatamente anterior a ele, devendo este ser encadeado ao seguinte na lista, sendo depois liberada a posição que o nodo ocupava. Tratamento diferenciado deve ser dado ao caso de remoção do primeiro nodo, devendo ser atualizado o ponteiro da lista.

O algoritmo a seguir remove um nodo identificado pela sua ordem na lista (K). `Sucesso` retorna falso caso a lista esteja vazia, ou a ordem solicitada não esteja compreendida na lista. A lista é percorrida a partir do endereço do primeiro nodo da lista até que o nodo a ser removido seja alcançado. O ponteiro `PtAnt` guardará o endereço do nodo anterior para que seja encadeado ao

nodo seguinte. Caso o nodo a ser removido seja o primeiro, o endereço de acesso à lista deve ser atualizado.

Algoritmo 3.27 - RemoverKLLEncCir

```
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variáveis auxiliares: PtAnt, PtK (TipoPtNodo)
início
  se (K < 1) ou (PtLista = nulo)
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        se K = 1
        então se PtLista↑.Elo = PtLista
              então início
                    liberar(PtLista)
                    PtLista ← nulo
                    fim
              senão início
                    PtAnt ← PtLista
                    enquanto PtAnt↑.Elo ≠ PtLista
                    faça PtAnt ← PtAnt↑.Elo
                    PtAnt↑.Elo ← PtLista↑.Elo
                    liberar(PtLista)
                    PtLista ← PtAnt↑.Elo
                    fim
        senão início
              PtAnt ← PtLista
              enquanto (PtAnt↑.Elo ≠ PtLista) e (K > 2)
              faça início
                   PtAnt ← PtAnt↑.Elo
                   K ← K - 1
                   fim
              se PtAnt↑.Elo = PtLista  {ORDEM FORA DA LISTA}
              então Sucesso ← falso
              senão início
                    PtK ← PtAnt↑.Elo
                    PtAnt↑.Elo ← PtK↑.Elo
                    liberar(PtK)
                    fim
              fim
        fim
fim
```

3.5.3 acesso a um nodo

A principal alteração introduzida pelo fato de a lista ser encadeada circular consiste na maneira como a lista é percorrida. Quando a lista não era circular, seu último nodo apresentava campo de elo nulo – ao encontrar este valor era identificado o final da lista. No caso de lista circular nunca será encontrado um campo de elo nulo. O processo de percurso da lista deve ser suspenso quando for alcançado novamente o seu primeiro nodo.

A título de ilustração é apresentado a seguir um algoritmo que percorre uma lista circular com o objetivo de realizar uma mesma operação em todos os nodos: exibir o conteúdo de seu campo de informação. É utilizado o mesmo tipo de nodo das listas encadeadas anteriormente apresentadas.

```
Algoritmo 3.28 - ImprimirLLEncCir
   Entrada: PtLista (TipoPtNodo)
   Saídas: -
   Variável auxiliar: PtAux (TipoPtNodo)
início
  se PtLista = nulo
  então escrever(´Lista vazia!´)
  senão início
        PtAux ← PtLista
        repita
           escrever(PtAux↑.Info)
           PtAux ← PtAux↑.Elo
        até que PtAux = PtLista
        fim
fim
```

3.6 → listas lineares duplamente encadeadas

Uma lista duplamente encadeada é aquela que possui duas referências ao invés de uma, permitindo que a lista seja percorrida nos dois sentidos – do início para o final, e do final para o início. Para que isto seja possível, cada nodo apresenta dois campos de elo, um que armazena o endereço do nodo imediatamente anterior na lista (`Anterior`) e o outro que armazena o endereço do nodo seguinte (`Próximo`), conforme mostrado na Figura 3.24. O primeiro nodo da lista terá sempre o apontador para o nodo anterior no endereço nulo, e o último nodo terá endereço nulo no campo que indica o próximo nodo da lista.

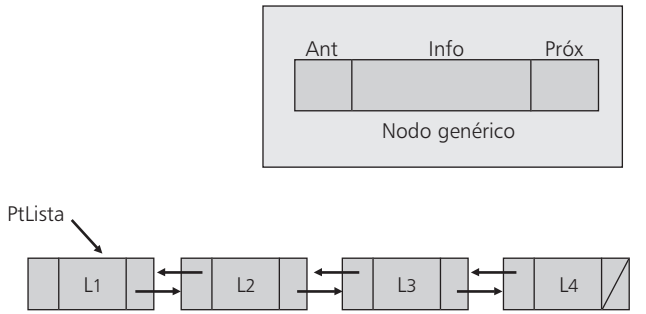


Figura 3.24 Lista duplamente encadeada.

A implementação das operações fica facilitada quando é utilizado o duplo encadeamento, pelo fato de se poder percorrer a lista em qualquer direção, a qualquer momento. Apesar desta forma de representação de lista apresentar um nodo mais complexo, com mais informações armazenadas, a sua facilidade de manipulação justifica sua grande utilização.

A seguir são apresentados os algoritmos relativos às operações básicas que podem ser executadas em listas duplamente encadeadas, sendo utilizado o seguinte tipo de dados para os nodos:

```
TipoNodo = registro
              Ant: TipoPtNodo
              Info: TipoInfoNodo
              Prox: TipoPtNodo
           fim registro
```

Das operações básicas, a operação de **criação da lista duplamente encadeada** é exatamente igual à da lista simplesmente encadeada, pois envolve somente a alocação e a inicialização do ponteiro que faz referência para o início da lista (`PtLista`). Muda somente o tipo de nodo utilizado. A operação de **destruição da lista** também não é alterada. Já as operações de inserção, remoção e acesso devem ser adaptadas, sendo mostradas a seguir.

3.6.1 inserção de um novo nodo

Na inserção de um novo nodo em uma lista duplamente encadeada é necessário atualizar os campos de elo dos dois sentidos de percurso. A Figura 3.25

mostra graficamente o que deve ser feito para inserir o novo nodo na lista: (1) no nodo que ficará antes do novo nodo, alterar o apontador para o próximo da lista; (2) no nodo que ficará depois dele, alterar o campo que aponta para o anterior na lista; e (3) no novo nodo, atualizar os dois campos de apontadores, para o próximo e para o anterior.

O algoritmo apresentado a seguir percorre a lista até o nodo após o qual o novo nodo deve ser inserido, utilizando para isso um ponteiro auxiliar (`PtAnt`). O novo nodo, alocado em uma posição indicada por `PtNovo`, é então encadeado nos dois sentidos, com o nodo apontado por `PtAnt` e com aquele que era o seguinte, cujo endereço está no campo `Prox` de `PtAnt`. `Sucesso` retorna falso nos casos em que a inserção não é possível: (1) caso não exista espaço para alocar o novo nodo; (2) quando a lista estiver vazia e a posição solicitada for diferente da primeira; ou (3) quando a ordem solicitada não for compatível com o tamanho da lista.

```
Algoritmo 3.29 - InserirKLLDupEnc
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
             Dados (TipoInfoNodo)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
```

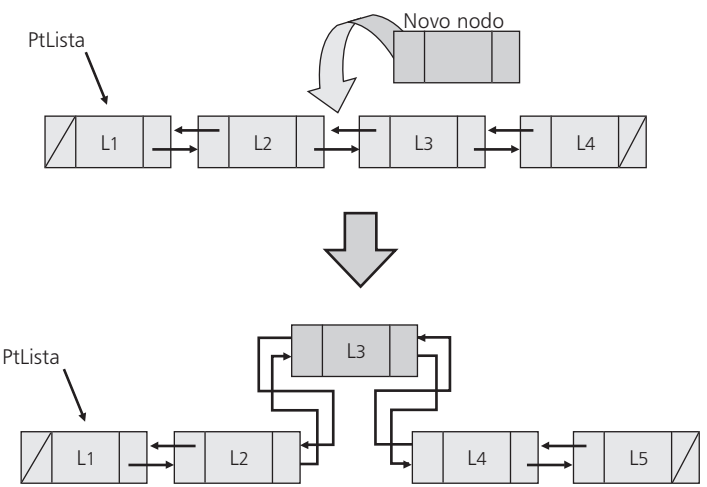


Figura 3.25 Inserção de um novo nodo em uma lista duplamente encadeada.

```
   Variáveis auxiliares: PtAnt, PtNovo (TipoPtNodo)
início
  alocar(PtNovo)
  se PtNovo = nulo
  então Sucesso ← falso
  senão se ((PtLista = nulo) e (K ≠ 1)) ou (K < 1)
        então início
              liberar(PtNovo)
              Sucesso ← falso
              fim
        senão se K = 1
              então início
                    PtNovo↑.Info ← Dados
                    PtNovo↑.Prox ← PtLista
                    se PtLista <> nulo
                    então PtLista↑.Ant ← PtNovo
                    PtNovo↑.Ant ← nulo
                    PtLista ← PtNovo
                    Sucesso ← verdadeiro
                    fim
              senão início
                    PtAnt ← PtLista
                    enquanto (PtAnt↑.Prox ≠ nulo) e (K > 2)
                    faça início
                         PtAnt ← PtAnt↑.Prox
                         K ← K - 1
                         fim
                    se K > 2
                    então início
                          liberar(PtNovo)
                          Sucesso ← falso
                          fim
                    senão início
                          PtNovo↑.Info ← Dados
                          PtNovo↑.Prox ← PtAnt↑.Prox
                          PtNovo↑.Ant ← PtAnt
                          PtAnt↑.Prox ← PtNovo
                          if PtNovo↑.Prox <> nulo
                          then PtNovo↑.Prox↑.Ant ← PtNovo
                          Sucesso ← verdadeiro
                          fim
                    fim
fim
```

3.6.2 remoção de um nodo

Todo cuidado deve ser tomado para que o duplo encadeamento não seja perdido quando um nodo for removido da lista. A Figura 3.26 mostra graficamente como deve ser realizada a operação de remoção de um nodo, quando este estiver em uma posição intermediária da lista – os nodos anterior e posterior a ele são encadeados, nos dois sentidos, sendo depois liberada a posição de memória ocupada pelo nodo a ser removido. Cuidados especiais devem ser tomados quando o nodo a ser removido estiver em uma das extremidades da lista.

A implementação desta operação, mostrada a seguir, é semelhante à remoção de uma lista simplesmente encadeada: inicialmente é localizado o nodo que deve ser removido, percorrendo a lista a partir de seu primeiro nodo, com o ponteiro `PtK`; em seguida, o nodo anterior a este é encadeado com o que vem a seguir, sendo ele então removido. Caso o nodo removido seja o primeiro da lista, é atualizado o ponteiro da lista para o nodo seguinte. A remoção não será realizada no caso da ordem do nodo solicitado não estar compreendida na lista, ou da lista estar vazia.

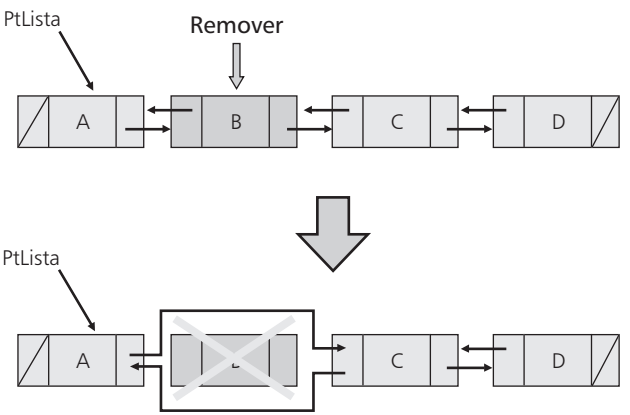


Figura 3.26 Remoção de um nodo de uma lista duplamente encadeada.

```
Algoritmo 3.30 - RemoverKLLDupEnc
   Entradas: PtLista (TipoPtNodo)
             K (inteiro)
   Saídas: PtLista (TipoPtNodo)
           Sucesso (lógico)
   Variável auxiliar: PtK (TipoPtNodo)
início
  se ((PtLista = nulo) e (K ≠ 1)) ou (K < 1)
  então Sucesso ← falso
  senão início
        PtK ← PtLista
        enquanto (PtK ≠ nulo) e (K > 1)
        faça início
             K ← K-1
             PtK ← PtK↑.Prox
             fim
        se PtK = nulo
        então Sucesso ← falso
        senão início
              Sucesso ← verdadeiro
              se PtK = PtLista
              então início
                    PtLista↑.Ant ← nulo
                    PtLista ← PtLista↑.Prox
                    fim
              senão início
                    PtK↑.Ant↑.Prox ← PtK↑.Prox
                    se PtK↑.Prox <> nulo
                    então PtK↑.Prox↑.Ant ← PtK↑.Ant
                    liberar(PtK)
                    fim
              fim
        fim
fim
```

3.6.3 acesso à lista duplamente encadeada

O acesso aos nodos de uma lista duplamente encadeada será feito sempre a partir do seu primeiro nodo, podendo depois o percurso ser realizado nos dois sentidos. Caso a lista seja percorrida a partir do seu início, o acesso a um determinado nodo é igual ao que foi visto para a lista simplesmente encadeada.

Para ilustrar a vantagem de utilizar o duplo encadeamento, é mostrada a seguir uma aplicação que utiliza este tipo de lista encadeada para exibir o conteúdo dos campos de informação de cada nodo, do final para o início – primeiro é exibido o campo de informação do último nodo, depois o do penúltimo nodo, e assim por diante. Para isto, um ponteiro auxiliar (PtAux) é posicionado inicialmente no início da lista, percorrendo-a até alcançar o último nodo. Utilizando, então, o encadeamento para os nodos anteriores, o mesmo ponteiro percorre novamente a lista, imprimindo os campos de informação de cada nodo.

```
Algoritmo 3.31 - ImprimirLLDupEncInv
   Entrada: PtLista (TipoPtNodo)
   Saídas: -
   Variável auxiliar: PtAux (TipoPtNodo)
início
  se PtLista = nulo
  então escrever(´Lista vazia !´)
  senão início
        PtAux ← PtLista
        enquanto PtAux↑.Prox ≠ nulo
        faça PtAux←PtAux↑.Prox
        enquanto PtAux ≠ PtLista
        faça início
             escrever(PtAux↑.Info)
             PtAux ← PtAux↑.Ant
             fim
        escrever(PtAux↑.Info)
        fim
fim
```

3.6.4 lista duplamente encadeada, com descritor

Nos casos em que o final da lista é acessado seguidamente, a utilização de um descritor facilita a sua manipulação, pois nele pode ser armazenado diretamente o endereço do último nodo. A existência do duplo encadeamento permite que, uma vez acessado diretamente esse último nodo, os demais nodos da lista também possam ser acessados. A última operação apresentada, por exemplo, seria simplificada caso o acesso ao último nodo fosse direto. Assim, o descritor reúne, em um único elemento, as referências para o início e para o final da lista.

A Figura 3.27 ilustra a utilização de um descritor com três campos: o primeiro indicando o endereço do primeiro nodo da lista; o segundo, o número total de nodos da lista (na figura, 4 nodos); e o terceiro, o endereço do último nodo.

O tipo deste descritor é:

```
TipoDecrLDE = registro
                    Prim : TipoPtNodo
                    N : inteiro
                    Ult : TipoPtNodo
                fim registro
```

Os algoritmos para inserção, remoção e busca em uma lista duplamente encadeada com descritor são semelhantes aos algoritmos para listas duplamente encadeadas, bastando somente fazer o acesso através do descritor e atualizar o descritor a cada modificação efetuada na lista.

Para exemplificar a implementação de uma operação em uma lista linear duplamente encadeada com descritor, a seguir é mostrado um algoritmo que insere um nodo no final desta lista, pegando o endereço do último nodo diretamente no descritor. O procedimento supõe que já foi alocado o descritor da lista, do tipo acima definido, sendo conhecido o seu endereço `PtDescrLDE`. Supõe ainda que sempre seja possível alocar um novo nodo.

```
Algoritmo 3.32 - InserirFimLLDupEncDesc
   Entradas: PtDescr (TipoPtDescrLDE)
             Valor (TipoInfoNodo)
```

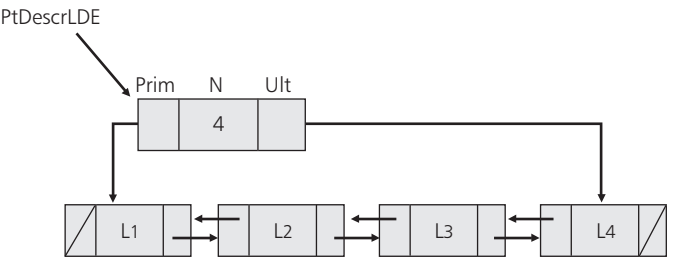


Figura 3.27 Lista duplamente encadeada, com descritor.

```
   Saída: PtDescr (TipoPtDescrLDE)
   Variáveis auxiliares: PtAux, PtUlt (TipoPtNodo)
início
  alocar(PtAux)
  PtAux↑.Info ← Valor;
  PtAux↑.Prox ← nulo
  PtDescr↑.N ← PtDescr↑.N + 1
  se PtDescr↑.Ult = nulo
  então início
        PtDescr↑.Prim ← PtDescr↑.Ult ← PtAux
        PtAux↑.Ant ← nulo
        fim
  senão início
        PtUlt ← PtDescr↑.Ult
        PtUlt↑.Prox ← PtAux
        PtAux↑.Ant ← PtUlt
        PtDescr↑.Ult ← PtAux
        fim
fim
```

3.7 → lista duplamente encadeada circular

Uma lista duplamente encadeada pode ser, também, circular. Neste caso, o primeiro nodo da lista tem como antecessor o último nodo, e o último, por sua vez, tem o primeiro como seguinte. Assim, a lista pode ser percorrida em qualquer sentido, a partir de qualquer nodo – é o caso mais geral de lista linear encadeada, que proporciona o acesso mais simplificado a todos os seus nodos.

Na Figura 3.28 é mostrada uma lista duplamente encadeada circular, na qual o indicador de início da lista está apontando para o nodo L1. O acesso à lista é feito sempre através de seu primeiro nodo. Caso se conheça *a priori* o número de nodos da lista, é possível escolher qual o melhor sentido de percurso

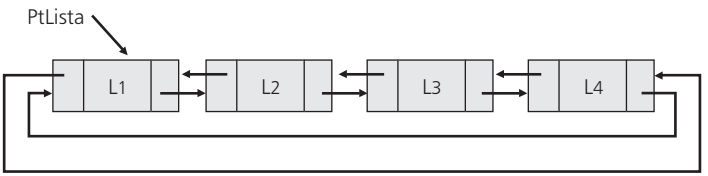


Figura 3.28 Lista duplamente encadeada circular.

a ser adotado, de modo a chegar mais rápido ao nodo que se quer acessar. Por exemplo, caso a lista tenha 20 nodos, se o nodo requerido é o de ordem 18, é mais apropriado percorrê-la de trás para frente.

Quanto à implementação das operações sobre este tipo de lista, a operação de **criação da lista circular** é similar à de criação de uma lista encadeada qualquer, bastando inicializar o ponteiro da lista em um endereço nulo, representando a lista vazia. A forma de **acessar os nodos da lista** é pouco alterada quando a lista duplamente encadeada é circular, mudando somente a identificação do final da lista, que é quando se alcança novamente o seu primeiro nodo. O mesmo acontece com a operação de **destruição da lista**. As maiores alterações introduzidas pelo fato de a lista ser circular são relacionadas às operações de inserção e remoção de nodos, apresentadas a seguir.

3.7.1 inserção de um novo nodo

Na operação de inserção de um novo nodo neste tipo de lista, todo cuidado deve ser tomado para que o encadeamento entre o primeiro e o último nodos, nos dois sentidos, não seja perdido. As operações que inserem ou removem nodos no meio da lista não são alteradas pelo fato de a lista ser circular. Entretanto, caso uma destas operações envolva um dos nodos extremos da lista, o encadeamento entre eles deve ser atualizado.

Para exemplificar, a seguir é mostrada a operação de inserção no final da lista. Supondo que a lista já tenha sido criada, a operação terá sucesso sempre que houver espaço físico para alocar o novo nodo. Caso a lista esteja inicialmente vazia, o novo nodo será também o primeiro nodo da lista. Quando a lista já tiver algum nodo, o encadeamento do novo nodo na posição de último é feito simplesmente através das informações contidas nos campos de elo do primeiro nodo, e daquele que era o último. A Figura 3.29 ilustra esta situação, incluindo um novo nodo no final da lista. O encadeamento circular facilita a implementação desta operação, pois não é preciso percorrer toda a lista para alcançar seu final.

```
Algoritmo 3.33 - InserirFimLLDupEncCir
   Entradas: PtDescr (TipoPtDescrLDE)
             Dados (TipoInfoNodo)
   Saídas: PtDescr (TipoPtDescrLDE)
           Sucesso (lógico)
```

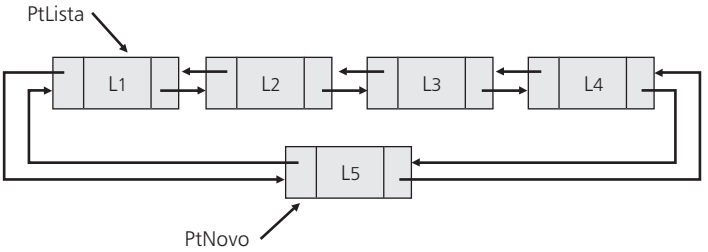


Figura 3.29 Inserção no final de uma lista duplamente encadeada circular.

```
   Variável auxiliar: PtNovo (TipoPtNodo)
início
  alocar(PtNovo)
  se PtNovo = nulo
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        PtNovo↑.Info ← Dados
        se PtLista = nulo
        então início
              PtNovo↑.Prox ← PtNovo
              PtNovo↑.Ant ← PtNovo
              PtLista ← PtNovo
              fim
        senão início
              Ptnovo↑.Ant ← PtLista↑.Ant
              PtNovo↑.Prox ← PtLista
              PtLista↑.Ant↑.Prox ← PtNovo
              PtLista↑.Ant ← PtNovo
              fim
        fim
fim
```

3.7.2 remoção de um novo nodo

O fato de a lista duplamente encadeada ser circular afeta a operação de remoção de um nodo somente quando o nodo a ser removido for um dos extremos da lista. Na remoção do nodo de uma das extremidades deve ser mantido o encadeamento circular. A seguir é apresentado um algoritmo que remove o último nodo de uma lista duplamente encadeada circular. Como o endereço do último nodo já está contido no campo `Ant` do primeiro nodo da

lista, a implementação da operação é imediata, bastando atualizar os encadeamentos para que o penúltimo nodo passe a ser o último, liberando em seguida a posição ocupada pelo nodo a excluir. Supõe-se que a lista já foi criada. Caso a lista esteja vazia, Sucesso retornará falso. Se a lista apresentar somente um nodo, resultará vazia.

```
Algoritmo 3.34 - RemoverUltLLDupEncCir
     (var PtLista: TipoPtNodo): lógico;
   Entradas: PtDescr (TipoPtDescrLDE)
   Saídas: PtDescr (TipoPtDescrLDE)
           Sucesso (lógico)
   Variável auxiliar: PtAux (TipoPtNodo)
início
  se PtLista = nulo
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        PtAux ← PtLista↑.Ant
        se PtLista↑.Prox = PtLista
        então PtLista ← nulo
        senão início
              PtLista↑.Ant ← PtAux↑.Ant
              PtAux↑.Ant↑.Prox ← PtLista
              fim
        liberar(PtAux)
        fim
fim
```

3.7.3 lista duplamente encadeada circular, com descritor

No caso de ser requerido o acesso a um determinado nodo de uma lista duplamente encadeada circular, identificando este nodo através de sua ordem na lista, o tempo de execução pode ser otimizado pelo fato de a lista poder ser percorrida nos dois sentidos. Identificando o sentido apropriado, pode-se garantir que o percurso até alcançar este nodo seja o menor possível. Para isso, é necessário que seja conhecido o número total de nodos da lista, que pode estar contido em um descritor desta lista.

O descritor definido na Seção 3.6.4 para uma lista duplamente encadeada apresentava, além de um campo contendo o número total de nodos da lista (N), outros dois campos com os endereços do primeiro e do último nodos da lista. No caso aqui analisado, sendo a lista circular, somente o primeiro nodo

da lista é necessário, pois o último pode ser diretamente acessado a partir deste. No exemplo a seguir será utilizado o seguinte descritor (Figura 3.30):

```
TipoPtDescrLDEC = registro
                      Prim : TipoPtNodo
                      N : inteiro
                  fim registro
```

O algoritmo a seguir acessa o nodo de ordem K da lista, devolvendo o conteúdo de seu campo de informação (Valor). Sucesso retorna falso caso a lista esteja vazia (quando o campo Prim de seu descritor for um endereço nulo), ou quando a ordem do nodo solicitado estiver fora da lista, o que é informado diretamente pelo descritor, através do campo que contém o número de nodos (N). Uma vez acessado o primeiro nodo da lista, um ponteiro auxiliar (PtAux) percorre a lista até o nodo desejado, no sentido mais adequado.

```
Algoritmo 3.35 - AcessarKLLDupEncCirDesc
   Entradas: PtDescrLDEC (TipoPtDescrLDEC)
             K (inteiro)
   Saídas: Valor (TipoInfoNodo)
           Sucesso (lógico)
   Variável auxiliar: PtAux (TipoPtNodo)
início
  se (PtDescrLDEC↑.Prim = nulo) ou (K < 1) ou (K > PtDescrLDEC↑.N)
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        PtAux ← PtDecrLDEC↑.Prim
        se K ≤ PtDescrLista↑.N / 2
```

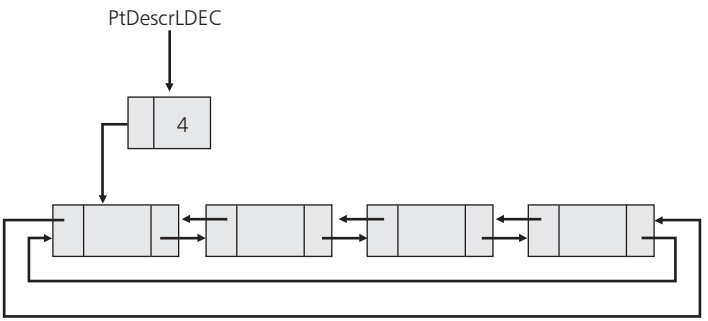


Figura 3.30 Lista duplamente encadeada circular, com descritor.

```
        então {PERCORRE DO INÍCIO PARA O FINAL}
              enquanto K > 1
              faça início
                   PtAux ← PtAux↑.Prox
                   K ← K – 1
                   fim
        senão início {PERCORRE DO FINAL PARA O INÍCIO}
              PtAux ← PtAux↑.Ant
              enquanto PtDecrLista↑.N – K  > 0
              faça início
                   PtAux ← PtAux↑.Ant
                   K ← K + 1
                   fim
              fim
        Valor ← PtAux↑.Info
        fim
fim
```

3.8 ⋯→ considerações gerais

Não existe, *a priori*, uma forma considerada a melhor para implementar listas lineares, seja através de contigüidade física ou por encadeamento, com encadeamento simples ou duplo, com ou sem descritor. Cada uma delas apresenta vantagens e desvantagens, devendo ser escolhida aquela que for mais adequada à aplicação específica que estiver sendo implementada.

A implementação de listas lineares por contigüidade física (sobre arranjos) apresenta como principal vantagem o acesso direto a qualquer elemento da lista, feito através do índice no arranjo. O tempo para acessar qualquer nodo da lista é constante, dependendo somente do conhecimento prévio do índice a ser acessado.

Entretanto, este tipo de implementação apresenta algumas desvantagens. A principal delas é que a implementação por contigüidade física apresenta quase sempre a necessidade de deslocamento (movimentação de dados) quando um elemento é inserido ou excluído da lista. Além disso, o tamanho da lista deve ser pré-estimado, ou seja, o tamanho sempre será limitado ao tamanho definido para o arranjo. Sendo assim, as listas lineares implementadas por contigüidade física são indicadas para aplicações que manipulam listas pequenas, cujo tamanho máximo possa ser previamente definido, e que apresentem como opera-

ções mais freqüentes consulta, inserção e remoção no final da lista, evitando deste modo a necessidade de deslocamento de nodos. Uma alternativa para evitar o deslocamento constante dos nodos é implementar a ocupação da lista no arranjo de forma circular: o novo nodo, no final lista, pode ser incluído na primeira posição do arranjo quando sucessivas inclusões e remoções levarem a uma situação em que o final da lista esteja na última posição do arranjo.

A implementação de listas através de encadeamento gera uma estrutura mais flexível, própria para aplicações onde as listas são dinâmicas, com grande número de inserções e de remoções de nodos. Uma grande vantagem nestes casos é não ser necessário dimensionar o número de nodos para a lista, que vão sendo alocados dinamicamente ao longo da execução da aplicação. Deste modo, a única limitação à construção da lista é o tamanho disponível da memória do computador que implementa a aplicação.

Entretanto, os algoritmos de busca em listas lineares podem ser considerados pouco eficientes neste tipo de implementação, pois a busca sempre deve iniciar pelo primeiro elemento que, neste caso, contém o menor valor da lista. Uma simples modificação na estrutura física da lista pode ser de grande auxílio: quando o último nodo da lista tem como próximo o primeiro nodo, formando assim uma lista circular. Diversas aplicações que necessitam representar conjuntos cíclicos se beneficiam com a implementação de listas circulares. Por exemplo, arestas que delimitam uma face podem ser agrupadas em uma lista circular.

A principal desvantagem da implementação de listas através de encadeamento é que o tempo de acesso pode ser bem maior do que no caso da implementação por contigüidade física, uma vez que geralmente é necessário percorrer a lista para encontrar um determinado nodo ou posição de inserção. Esta limitação é parcialmente contornada através da definição de listas duplamente encadeadas, que permitem seu percurso nos dois sentidos. Particularmente as listas duplamente encadeadas, embora facilitem a forma de percorrer a lista, requerem que o espaço físico utilizado seja maior devido à utilização de dois campos de elo em cada nodo. Em aplicações onde o número de nodos é muito elevado, este acréscimo de memória ocupada pode ser relevante.

A utilização de um descritor que contenha algumas informações sobre a lista, seja na implementação por contigüidade física ou através de encadeamento, apresenta vantagens em diversas aplicações, desde que as informações nele contidas sejam adequadas. Embora aumente um pouco o tempo de acesso

à lista, pois este sempre deve ser feito via descritor, sua utilização afasta do usuário os detalhes da implementação da lista, fazendo com que a manipulação da lista seja feita em um nível mais abstrato.

3.9 → exercícios

exercícios com listas lineares – contigüidade física

exercício 1 Considere uma lista de valores inteiros e implemente um algoritmo que receba como parâmetro dois valores (`n1` e `n2`) e uma lista, e insira o valor `n2` após o nodo que contém o valor `n1`.

exercício 2 O algoritmo que remove um nodo com base em sua ordem na lista (Algoritmo 3.6) pode ser otimizado da seguinte forma: se o nodo a ser removido estiver localizado na primeira metade da lista, o deslocamento deve ser feito para baixo; caso contrário, o deslocamento deve ser feito para cima. Implemente, de forma otimizada, um algoritmo que receba como parâmetro uma lista e um valor e retorne a lista resultante da exclusão do elemento cuja posição na lista foi indicada pelo parâmetro valor.

exercício 3 Construa um algoritmo que receba como parâmetros uma lista e um valor, valor este que representa a posição de um nodo na lista. O algoritmo deverá retornar as informações contidas neste nodo e a lista resultante da exclusão deste nodo.

exercício 4 O Algoritmo 3.9 implementa a busca de um nodo através de algum valor de seu campo de informação, considerando uma lista ordenada em ordem crescente de valores. Implemente uma função que receba como parâmetro uma lista com os valores ordenados em ordem decrescente, e retorne a posição do nodo na lista.

exercício 5 A seção 3.2 define um tipo de descritor, denominado `TipoDescr`, que contém cinco campos, com as seguintes informações: `IA` – índice de início da área disponível para a lista; `IL` – índice do primeiro nodo da lista; `N` – comprimento da lista; `FL` – índice do último nodo da lista; e `FA` – índice do final da área disponível para a lista. Construa um algoritmo que receba como parâmetro o descritor da lista, um valor que representa uma posição na lista e um valor para ser inserido na lista; e retorne a lista resultante da inclusão do novo nodo duas posições acima do valor passado como parâmetro.

exercício 6 Considerando a lista do exercício anterior, acessada através de um descritor do tipo `TipoDescr`, implemente uma função que receba como parâmetro o descritor da lista e retorne o número de nodos que a lista apresenta no momento.

exercício 7 Defina um descritor adequado para acessar uma lista linear implementada sobre um arranjo, para uma aplicação que acesse seguidamente os nodos que apresentam o maior e o menor valor em seu campo de aplicação. Em seguida, implemente um algoritmo que receba como parâmetro o descritor da lista e um valor e inclua um novo nodo no final desta lista, atualizando o descritor, caso seja necessário.

exercício 8 Implemente um algoritmo que receba como parâmetro uma lista implementada sobre um arranjo com ocupação circular, e um valor. O algoritmo deve retornar a lista resultante da remoção do nodo que contém este valor.

exercício 9 Implemente uma função que receba como parâmetro uma lista circular, implementada sobre um arranjo com ocupação circular, e retorne o número de nodos que a lista apresenta no momento.

exercícios com listas lineares encadeadas

exercício 10 Construa um algoritmo que receba, como parâmetro, o endereço do primeiro nodo de uma lista encadeada e um valor. O algoritmo deve retornar: o número total de nodos da lista; o número de nodos da lista que possuem em seu conteúdo o valor passado como parâmetro e sua respectiva posição na lista; e o número de nodos que possuem em seu conteúdo valores maiores do que o valor passado como parâmetro.

exercício 11 Implemente um algoritmo que receba como parâmetro um ponteiro para o primeiro nodo de uma lista encadeada e dois valores, e retorne a lista resultante da troca de todas as ocorrências do primeiro valor pelo segundo, ambos passados como parâmetros, e o número total de trocas efetuadas.

exercício 12 Implemente um algoritmo que receba como parâmetro o primeiro nodo de uma lista simplesmente encadeada e mostre o conteúdo de todos os nodos da lista, de trás para frente. Note que a lista não é duplamente encadeada. Implemente esse algoritmo de forma recursiva e não recursiva, e depois compare as duas implementações.

exercício 13 Construa um algoritmo que receba como parâmetro duas listas encadeadas ordenadas e retorne a lista resultante da combinação das duas, sendo que a lista resultante também deve estar ordenada.

exercício 14 Implemente uma função que receba como parâmetro duas listas encadeadas e retorne um valor lógico que indique se as duas listas são idênticas.

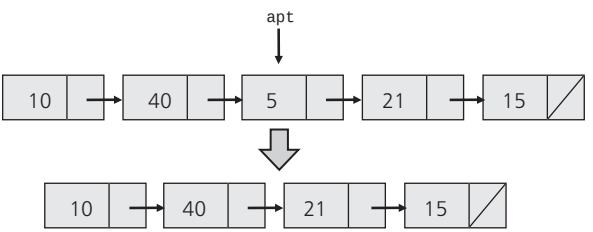
exercício 15 Construa um algoritmo que construa uma lista encadeada reversa, de modo que os dados fornecidos em primeiro lugar sejam os do último nodo, e os últimos dados, os do primeiro.

exercício 16 Polinômios podem ser representados por meio de listas encadeadas, cujos nodos são registros com três campos: coeficiente, expoente e uma referência ao nodo seguinte. Implemente uma função que receba como parâmetro $X \in R$ como parâmetro, e retorne o resultado do cálculo de $p(x)$.

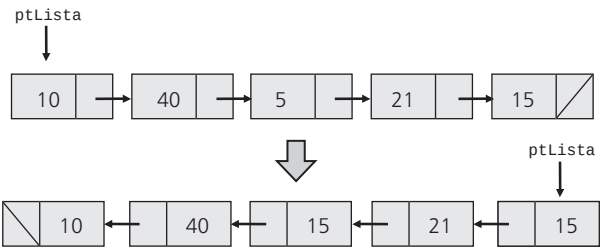
exercício 17 Construa um algoritmo que receba como parâmetro duas listas e um número inteiro `N` e retorne a lista resultante da inserção da segunda lista na primeira, seqüencialmente, a partir da posição `N` da primeira lista.

exercício 18 Considere a existência de duas listas ordenadas, a primeira (`L1`) cujos nodos são palavras e a segunda (`L2`) que contém um número em cada nodo. Implemente um algoritmo que remova da lista `L1` os nodos cujas posições estão na lista `L2`. As posições na lista `L2` variam de `1` a `N`. Caso algum nodo da segunda lista apresente um valor de posição não válido para a lista `L1`, a remoção correspondente não é realizada.

exercício 19 Considere um único ponteiro (`apt`) para um nodo de uma lista simplesmente encadeada. Implemente um algoritmo que receba como parâmetro o ponteiro `apt` e retorne a lista resultante da exclusão do nodo apontado por `apt`, conforme ilustrado na figura a seguir:



exercício 20 Implemente um algoritmo que receba como parâmetro uma lista encadeada e retorne a lista resultante da inversão do encadeamento de seus nodos. A inversão deve ser realizada de forma a não alterar a ordem física dos elementos na lista, conforme mostrado na figura a seguir:



exercício 21 Considere duas listas encadeadas circulares, não vazias. Os ponteiros para o primeiro elemento de cada lista são `PtLista1` e `PtLista2`, respectivamente. Analise o seguinte trecho de código:

```
Apt ← PtLista1↑.Prox
PtLista1↑.Prox ← PtLista2↑.Prox
PtLista2↑.Prox ← Apt
PtLista1 ← PtLista2
PtLista2 ← nulo
```

- Qual o resultado da execução deste trecho de código?
- Qual o resultado da execução deste trecho de código caso `PtLlista1` e `PtLista2` apontassem para dois elementos distintos da mesma lista circular?

exercício 22 Implemente uma função que receba como parâmetro o endereço de acesso a uma lista encadeada circular e um número inteiro `N` e retorne o número de nodos da lista que possuem um campo de informação com valores maiores do que o valor `N`.

exercício 23 Implemente um algoritmo que receba como parâmetro um ponteiro para um elemento de uma lista circular encadeada e um valor e retorne a lista resultante da remoção de todos os nodos que apresentarem em seu conteúdo o valor passado como parâmetro.

exercícios com listas lineares duplamente encadeadas

exercício 24 Implemente uma função que receba, como parâmetro, o ponteiro para uma lista simplesmente encadeada e retorne o ponteiro para uma lista duplamente encadeada, resultante da cópia dos valores da lista simplesmente encadeada.

exercício 25 Construa um algoritmo que receba como parâmetros uma lista duplamente encadeada e dois valores (original e novo). O algoritmo deve percorrer a lista comparando os campos de informação dos nodos e trocando todas as ocorrências do valor original pelo valor novo.

exercício 26 Implemente um algoritmo que calcule o maior elemento, o menor elemento e a média aritmética dos elementos de uma lista duplamente encadeada, com descritor.

exercício 27 Implemente a operação que insere um novo nodo em qualquer posição de uma lista duplamente encadeada circular.

exercício 28 Implemente um algoritmo que retorne o valor associado a um determinado nodo de uma lista duplamente encadeada circular. O nodo a ser consultado deve ser identificado por sua ordem, e deve ser acessado de forma otimizada, ou seja, no sentido mais próximo da posição em que se encontra.

exercícios de aplicações

exercício 29 Uma maneira usual de representar conjuntos é listando seus elementos. Implemente uma aplicação que ofereça as operações usuais de conjuntos (união, intersecção e diferença), considerando que cada um dos conjuntos é representado por uma lista encadeada.

exercício 30 A lenda conta que Josephus não teria sobrevivido para tornar-se famoso se não fosse o seu talento matemático. Durante a guerra entre judeus e romanos, ele estava entre um bando de 41 judeus rebeldes encurralados pelos romanos em uma caverna. Não havia esperança de vencer os inimigos sem reforços e existia um único cavalo para escapar. Os rebeldes fizeram um pacto para determinar qual deles escaparia em busca

de ajuda. Eles formaram um círculo e, começando a partir de um deles, começaram a contar no sentido horário em torno do círculo. Quando a contagem alcançava o número 3, aquele rebelde era removido do círculo e a contagem recomeçava a partir do próximo soldado, até que o último soldado restasse no círculo. Este seria o soldado que tomaria o cavalo e fugiria em busca de ajuda. Mas Josephus, com medo da morte iminente, calculou rapidamente onde ele deveria estar neste círculo para que pudesse ser ele o último soldado restante no círculo. Implemente uma aplicação para solucionar o problema de Josephus: dado um número de soldados `N` e um número `D` ≤ `N`, que estabelece o número fixo de remoção de elementos, determine a ordem em que os soldados são eliminados do círculo e qual deles escapará.

exercício 31 Suponha que uma fábrica de potes plásticos com tampas coloridas (vermelha-V, verde-G, azul-B). O funcionário da linha de montagem monta 100 potes por dia. No início da jornada ele recebe uma ordem de montagem, onde consta o número de potes a montar de cada cor. Outro funcionário monta *kits* com potes de três cores diferentes. Especifique as estruturas de dados para os tipos de pote e *kit*. Implemente uma aplicação que represente o processo de montagem dos *kits*.

exercício 32 Uma frase pode ser representada por uma lista linear encadeada, sendo que o campo de informação de cada nodo da lista contém um único caractere ou símbolo. Implemente uma aplicação para realizar as operações descritas a seguir, considerando uma lista que inicia no endereço `PtFrase`:

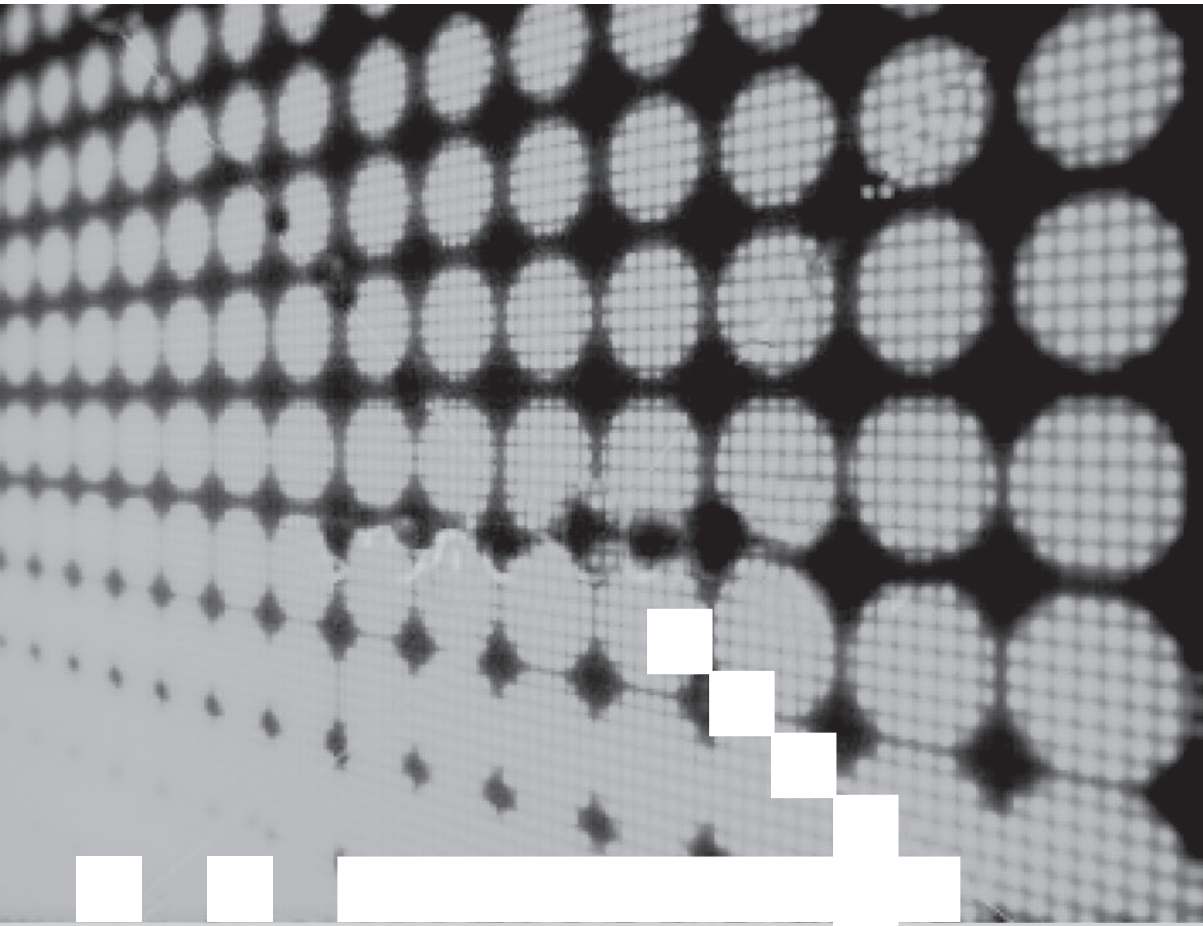
- converta a lista encadeada em uma lista armazenada sobre um arranjo `FRASE`, cujos elementos são caracteres;
- implemente uma função que recebe como parâmetro dois caracteres, `I1` e `I2`, e retorne o ponteiro para uma nova lista, representando a palavra que inicia com o caractere `I1` e termina com o caractere `I2`.

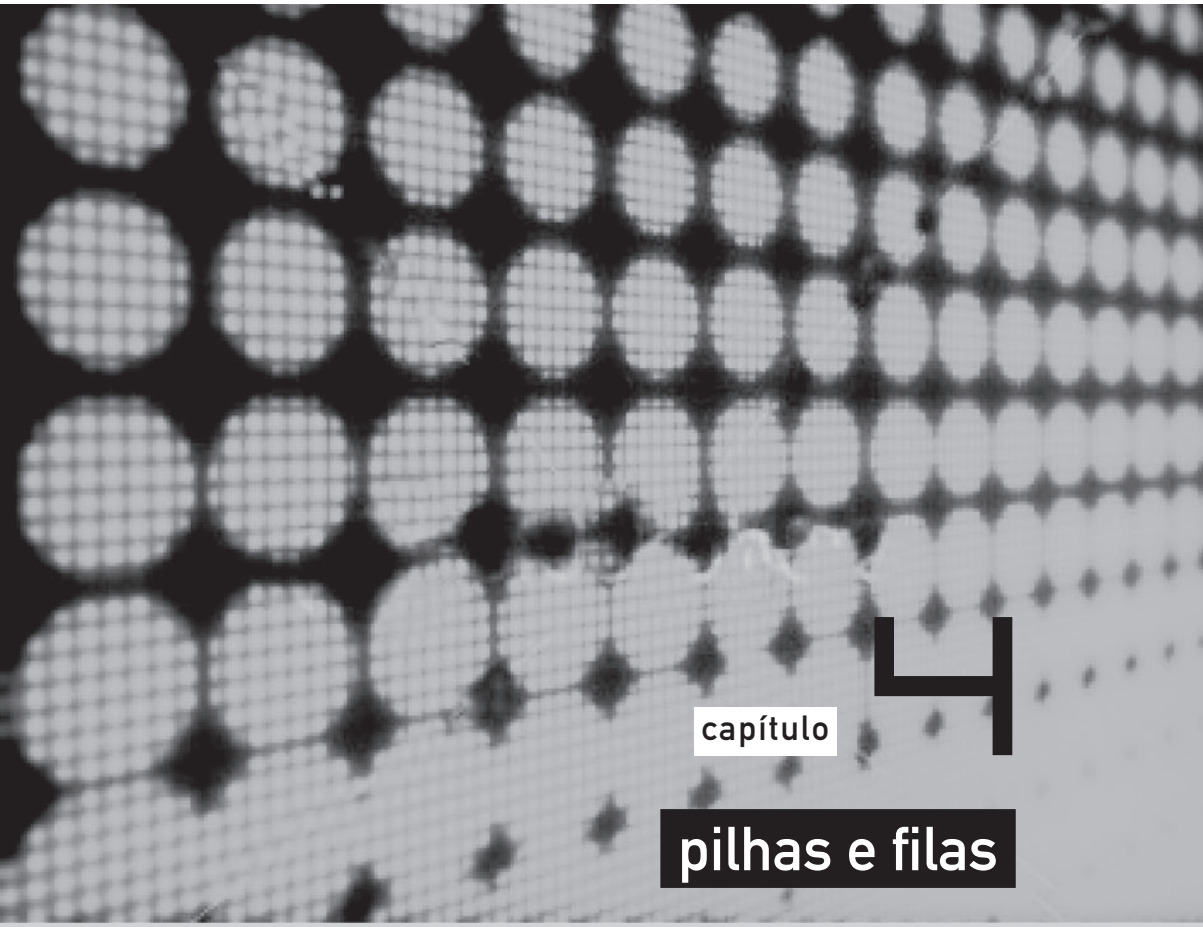
exercício 33 Considerando duas listas, cada uma armazenando uma frase na forma descrita no exercício anterior, acessadas pelos ponteiros `PtFrase1` e `PtFrase2`, implemente os seguintes algoritmos:

- uma função que recebe como parâmetro o ponteiro para as duas frases e devolve um inteiro indicando a posição de início da primeira ocorrência da segunda frase dentro da primeira frase. Se a segunda frase não estiver contida na primeira, então deve ser devolvido o valor `-1`;
- um procedimento que recebe como parâmetro dois inteiros, `I1` e `I2`, e substitua os elementos da lista apontada por `PtL2` por `I2` elementos da lista apontada por `PtL1`, começando da posição `I1`. A lista apontada por `PtL1` não deve ser modificada;
- uma função que devolva `-1` se a frase apontada por `PtL1` for menor do que a frase apontada por `PtL2`, `0` se forem do mesmo tamanho e `1` se a frase apontada por `PtL1` for maior.

exercício 34 Os dados de um grupo de atletas foram organizados em uma lista linear simplesmente encadeada. O campo de informação de cada elemento da lista apresenta o nome e a altura de um atleta. O endereço do primeiro elemento da lista está em um descritor, cujo endereço está na variável-ponteiro `ATLETAS`. A lista está organizada em ordem alfabética de atletas. Implemente uma aplicação para:

- gerar uma segunda lista, duplamente encadeada, cujo endereço está na variável-ponteiro `ALTURAS`. Nesta segunda lista deverão estar as mesmas informações da primeira lista, porém organizadas por ordem decrescente de alturas;
- excluir das duas listas o nodo correspondente a um determinado atleta (nome passado como parâmetro).





capítulo 4

pilhas e filas

Determinadas listas lineares apresentam alguma disciplina restrita de organização e de acesso a seus nodos, que deve ser obedecida por todas as operações realizadas sobre os nodos destas listas. São listas especiais nas quais não são permitidas operações sobre quaisquer nodos, mas somente sobre aqueles definidos pela organização da lista.

As duas principais restrições apresentadas para listas são:

- LIFO (*Last In First Out*) – dentre os nodos da lista, o primeiro nodo a ser retirado deve ser o último nodo que foi inserido;
- FIFO (*First In First Out*) – primeiro nodo a ser retirado deve ser o primeiro que foi inserido.

As listas que respeitam a restrição LIFO são denominadas Pilhas; já aquelas que obedecem a restrição FIFO, são denominadas Filas. Essas estruturas de dados, embora simples, se sobressaem devido à grande utilidade que apresentam, modelando diversas aplicações práticas, tanto em computação quanto em outros domínios. O critério LIFO, por exemplo, corresponde a uma pilha de pratos, onde um novo prato é sempre colocado no topo da pilha, devendo o primeiro prato a ser retirado ser o do topo da pilha. Um exemplo do critério FIFO é o funcionamento de uma fila de banco, onde as pessoas são atendidas na ordem que entram na fila: o primeiro a chegar será o primeiro a ser atendido. Pilhas e Filas estão, portanto, entre as estruturas de dados que são freqüentemente implementadas e fazem parte de um vasto conjunto de aplicações que também incluem outras estruturas de dados mais sofisticadas.

Neste capítulo são discutidos os algoritmos que lidam com pilhas, filas e filas duplas. Para cada uma destas estruturas são vistas inicialmente as operações básicas considerando a implementação por contigüidade física e, em seguida, as mesmas operações considerando implementação por encadeamento.

4.1 → pilhas

Pilhas são listas nas quais o acesso somente pode ser feito em uma das extremidades, denominada de topo da pilha. Todas as consultas, alterações, inclusões e remoções de nodos somente podem ser realizadas sobre um nodo, que é aquele que está na extremidade considerada o topo da pilha. Esta disciplina de acesso é também conhecida como LIFO – *Last In First Out* – o primeiro nodo a ser retirado deve ser o último nodo que foi incluído na pilha (Figura 4.1).

Pilhas são estruturas de dados fundamentais, sendo usadas em muitas aplicações em computação. Por exemplo, editores de texto oferecem um mecanismo de reversão de operações ("*undo*") que cancela operações recentes e reverte um documento ao estado anterior à operação. Esse mecanismo é implementado mantendo as alterações do texto em uma pilha. Navegadores para Internet também armazenam os endereços mais recentemente visitados em uma estrutura de dados do tipo pilha. Cada vez que o navegador visita um novo *site*, o endereço é armazenado na pilha de endereços. Usando a operação de retorno ("*back*"), o navegador permite que o usuário retorne ao último *site* visitado, retirando o seu endereço da pilha.

As operações que podem ser realizadas sobre uma pilha são limitadas pela disciplina de acesso que apresentam. Assim, somente as seguintes operações podem ser executadas:

- criar a pilha vazia;
- inserir um novo nodo no topo da pilha (operação comumente denominada PUSH);
- excluir o nodo do topo da pilha (operação comumente denominada POP);
- consultar e/ou modificar o nodo que está no topo da pilha;
- destruir a pilha, liberando as posições reservadas para ela.

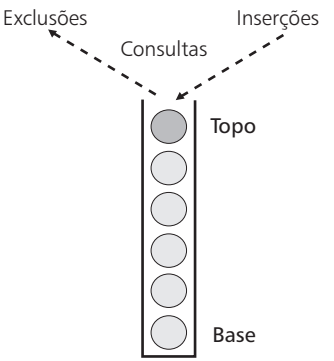


Figura 4.1 Pilha.

Estas operações são analisadas a seguir, considerando as duas possíveis formas de representação física – por contigüidade física e por encadeamento.

4.1.1 pilhas implementadas por contigüidade física

Para manipular uma pilha implementada sobre um arranjo é necessário que se conheça três valores:

- o índice do arranjo a partir do qual a pilha pode ser implementada, geralmente denominado de `Base` da pilha;
- o índice ocupado pelo elemento que está no `Topo` da pilha, e que é o único elemento que pode ser manipulado; e
- o índice do último elemento do arranjo que poderá ser utilizado por esta pilha, aqui denominado de `Lim` (limite da pilha).

Como exemplo de manipulação desta estrutura de dados, consideremos uma pilha implementada sobre um arranjo denominado `PILHA`, a partir do índice "1", com espaço para armazenar no máximo 6 nodos. Nessa pilha são armazenados valores inteiros. A seguinte seqüência de operações é apresentada graficamente na Figura 4.2:

1 inicializar a pilha – no exemplo, isto é indicado por `Topo` valendo `0` (figura 4.2a);
2 inserir um novo nodo com valor `3` (figura 4.2b);

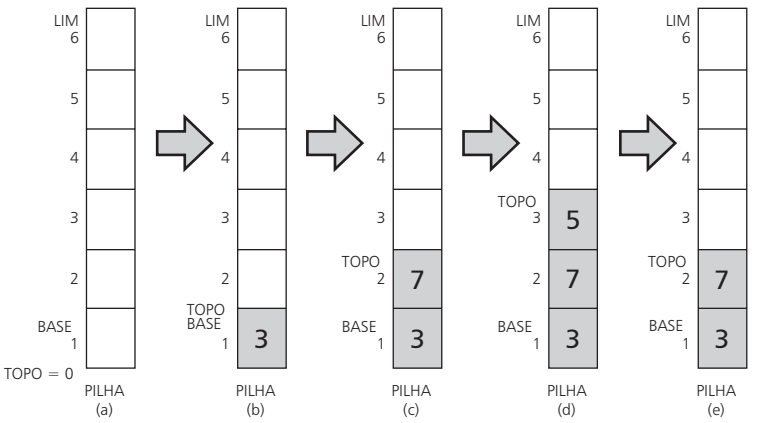


Figura 4.2 Exemplo de manipulação de uma pilha.

3 inserir um novo nodo com valor 7 (figura 4.2c);
4 inserir um novo nodo com valor 5 (figura 4.2d);
5 remover um nodo (figura 4.2e).

Após a execução destas operações, caso a pilha fosse consultada, o valor retornado seria 7, uma vez que este é o valor do nodo que está no topo da pilha.

A seguir são apresentados os algoritmos relativos às operações básicas sobre uma pilha, considerando este tipo de implementação. Será utilizado o seguinte tipo de dado:

```
TipoPilha = arranjo [1.. N] de TipoNodo
```

A operação de destruição da pilha não será apresentada, uma vez que envolve somente o espaço reservado para a pilha no arranjo, o que deve ser controlado diretamente pela aplicação.

■ criação da pilha

A criação de uma pilha resultará em uma pilha vazia, devolvendo ao usuário as informações necessárias para seu posterior acesso. Deverá ser utilizada alguma convenção para indicar que a pilha está vazia. Optamos pela seguinte estratégia: a indicação de que a pilha está vazia é feita quando o índice de topo da pilha estiver indicando uma unidade a menos do que o índice de sua base. Os algoritmos que manipulam a pilha (inserção, remoção e consulta) utilizam esta estratégia para reconhecer a pilha vazia. Caso outra forma de representação da pilha vazia seja utilizada, os algoritmos deverão ser a ela adaptados.

Assim, considerando a pilha implementada sobre o arranjo `Pilha` da Figura 4.2, as variáveis de controle da pilha são inicializadas em:

```
Base ← 1
Topo ← Base – 1
Lim ← 6
```

A aplicação deve conhecer os limites do arranjo reservados para a pilha (`Base` e `Lim`). Um algoritmo que inicializa uma pilha através da estratégia acima citada deve receber, da aplicação, o índice da base da pilha, inicializando o `Topo` com uma unidade a menos, como apresentado a seguir.

```
Algoritmo 4.1 – InicializarPilhaArr
   Entrada: Base (inteiro)
   Saída: Topo (inteiro)
início
  Topo ← Base – 1
fim
```

■ inserção de um nodo na pilha

Um novo nodo só pode ser inserido no topo da pilha, ou seja, logo acima da posição ocupada pelo topo da pilha. Esta operação é também conhecida como operação PUSH. Para que isso seja possível, é necessário que haja espaço para este novo nodo no espaço reservado para a pilha sobre o arranjo, o que é informado pelo valor do limite da pilha, `Lim`. Caso a inserção seja realizada, o valor do topo da pilha é incrementado, sendo este agora o nodo do topo e, conseqüentemente, o único ao qual se tem acesso.

O algoritmo apresentado a seguir insere um novo nodo no topo da pilha. Inicia verificando se há espaço no arranjo para a inserção e, em caso de haver espaço, passa o `Topo` para a próxima posição do arranjo, inserindo o valor recebido (`Valor`) nesta posição. O parâmetro `Sucesso` retorna falso no caso de não haver mais espaço para inserir um novo nodo.

```
Algoritmo 4.2 – InserirPilhaArr
   Entradas: Pilha (TipoPilha)
             Lim (inteiro)
             Topo (inteiro)
             Valor (TipoNodo)
   Saídas: Pilha (TipoPilha)
           Topo (inteiro)
           Sucesso (lógico)
início
  se Topo < Lim
  então início
        Topo ← Topo + 1
        Pilha[Topo] ← Valor
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

■ remoção de um nodo da pilha

Somente o nodo do topo de uma pilha pode ser removido. Esta operação, denominada simplesmente de remoção da pilha, é também conhecida por operação POP. A operação pode ser realizada se a pilha apresentar pelo menos um nodo, diminuindo o valor do índice do seu topo.

No algoritmo apresentado a seguir é removido o nodo do topo da pilha. Se a pilha estiver vazia no início da operação, `Sucesso` retorna falso. Como muitas vezes o nodo removido é utilizado pela aplicação, este algoritmo também retorna o valor do nodo que foi removido no parâmetro `ValorRemovido`.

```
Algoritmo 4.3 – RemoverPilhaArr
   Entradas: Pilha (TipoPilha)
             Topo (inteiro)
             Base (inteiro)
   Saídas: Pilha (TipoPilha)
           Topo (inteiro)
           Sucesso (lógico)
           ValorRemovido (TipoNodo)
início
  se Topo ≥ Base
  então início
        ValorRemovido ← Pilha[Topo]
        Topo ← Topo – 1
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

■ acesso à pilha

Consultas e/ou alterações somente podem ser realizadas sobre o nodo que está no topo da pilha. Para acessar algum outro nodo da pilha é necessário antes remover os nodos que estão acima dele, a partir dos que estão no topo da pilha.

A seguir é apresentado um algoritmo que consulta o nodo do topo da pilha, devolvendo seu valor através do parâmetro `Valor`. Caso a pilha esteja vazia, o parâmetro `Sucesso` retorna falso, informando que a operação não pode

ser realizada. Como esta operação somente consulta o nodo do topo da pilha sem removê-lo, não são alterados os valores do arranjo que implementa a pilha, nem suas variáveis de controle (`Base`, `Topo` e `Lim`).

```
Algoritmo 4.4 – ConsultarPilhaArr
   Entradas: Pilha (TipoPilha)
             Base (inteiro)
             Topo (inteiro)
   Saídas: Valor (TipoNodo)
           Sucesso(lógico)
início
  se Topo ≥ Base
  então início
        Valor ← Pilha[Topo]
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

4.1.2 pilhas implementadas por encadeamento

Pilhas também podem ser implementadas encadeando seus nodos, através da utilização de ponteiros. Nesse caso, cada nodo da pilha terá pelo menos dois campos: um campo onde são armazenadas as informações relativas a este nodo, aqui denominado de `Info`, e um campo que serve para encadear os nodos da pilha, que denominamos de `Elo`. Assim, o tipo definido para os nodos da pilha encadeada será o seguinte:

```
TipoNodo = registro
             Info: TipoInfo
             Elo: TipoPtNodo
           fim registro
```

A Figura 4.3 mostra uma pilha implementada por encadeamento. O endereço do topo da pilha (`PtPilha`) deve ser conhecido, sendo este o único nodo acessado devido à disciplina imposta por esta estrutura de dados. A base da pilha será o último nodo, apresentando no campo `Elo` um endereço nulo.

A seguir são apresentados algoritmos para as operações básicas sobre pilhas, quando implementadas por encadeamento.

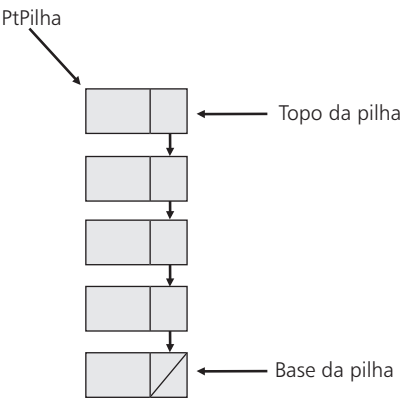


Figura 4.3 Pilha implementada por encadeamento.

■ criação de pilha encadeada

A criação de uma pilha encadeada é feita atribuindo um endereço nulo à variável-ponteiro que guardará o endereço do topo da pilha, de acordo com a convenção utilizada neste texto para representar uma pilha vazia.

O algoritmo para criação da pilha, apresentado a seguir, simplesmente coloca o endereço nulo na variável que irá apontar para o topo da pilha (`PtPilha`).

```
Algoritmo 4.5 – CriarPilhaEnc
   Entradas: -
   Saída: PtPilha (TipoPtNodo)
início
    PtPilha ← nulo
fim
```

■ inserção de um nodo em pilha encadeada

Como a inserção é feita sempre no topo da pilha, o novo nodo deve ser encadeado com aquele que estava no topo da pilha, passando a ser o novo topo da pilha encadeada. Essa operação é equivalente à operação de inserção no início de uma lista simplesmente encadeada (Algoritmo 3.20).

O algoritmo a seguir realiza as seguintes operações: (1) aloca o novo nodo, preenchendo-o com o `Valor` recebido para seu campo de informação; (2) en-

cadeia este novo nodo com aquele que estava no topo da pilha no momento de ser executada esta operação; e (3) direciona a variável que apontava para o topo da pilha para este novo nodo, que agora passa a ser o seu topo. É feita a suposição de que a alocação de novos nodos sempre possa ser realizada.

```
Algoritmo 4.6 – InserirPilhaEnc
   Entradas: PtPilha (TipoPtNodo)
          Valor (TipoInfo)
   Saída: PtPilha (TipoPtNodo)
   Variável local: PtNovo (TipoPtNodo)
início
  alocar(PtNovo)
  PtNovo↑.Info ← Valor
  PtNovo↑.Elo ← PtPilha
  PtPilha ← PtNovo
fim
```

■ remoção de um nodo em pilha encadeada

A remoção do nodo do topo da pilha é executada fazendo a variável-ponteiro que indicava o topo da pilha apontar para o nodo que estava na segunda posição na pilha, liberando em seguida o espaço que estava sendo ocupado pelo nodo que foi removido. Essa operação é equivalente à operação de remoção do primeiro nodo de uma lista encadeada (Algoritmo 3.23, com `K=1`).

O algoritmo a seguir implementa esta operação, retornando `Sucesso` verdadeiro caso a pilha não esteja inicialmente vazia, ou seja, caso exista um nodo no topo da pilha para ser removido. Caso seja removido o último nodo da pilha, o ponteiro de acesso passará a ter o endereço `nulo`.

```
Algoritmo 4.7 – RemoverPilhaEnc
   Entrada: PtPilha (TipoPtNodo)
   Saídas: PtPilha (TipoPtNodo)
          Sucesso (lógico)
   Variável local:  PtAux (TipoPtNodo)
início
  se PtPilha ≠ nulo
  então início
        PtAux ← PtPilha
        PtPilha ← PtPilha↑.Elo
        liberar(PtAux)
```

```
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

■ acesso à pilha encadeada

O acesso ao nodo que está no topo da pilha fica muito simples quando esta é implementada através de encadeamento, pois a variável-ponteiro de acesso à pilha já contém o endereço do seu topo, único nodo da pilha que pode ser acessado para consulta ou alteração. O algoritmo a seguir executa uma consulta à pilha, devolvendo o valor contido no campo de informação do nodo que está no topo através do parâmetro `Valor`. O parâmetro lógico `Sucesso` indica se a operação foi executada adequadamente, ou seja, retorna falso caso a pilha não tenha algum nodo para ser consultado.

```
Algoritmo 4.8 – ConsultarPilhaEnc
   Entrada: PtPilha (TipoPtNodo)
   Saídas: Valor (TipoInfo)
           Sucesso (lógico)
início
   se PtPilha = nulo
   então Sucesso ← falso
   senão início
         Sucesso ← verdadeiro
         Valor ← PtPilha↑.Info
         fim
fim
```

Uma operação que combina consulta e remoção é muito utilizada em aplicações que utilizam pilhas para armazenar valores que posteriormente serão utilizados, na ordem inversa da qual foram armazenados. Isto é facilmente obtido empilhando estes valores (inserindo-os na pilha), para depois desempilhá-los. A segunda parte ("desempilhar") combina as operações de consulta (fornece o valor que está no topo da pilha para que seja utilizada pela aplicação) e de remoção do topo (pois este não é mais necessário).

O algoritmo a seguir implementa esta operação. O valor do nodo que estava no topo da pilha é devolvido através do parâmetro `Valor`, sendo em seguida

este nodo liberado. O ponteiro para o topo passa a apontar para o nodo seguinte. Caso o nodo desempilhado seja o último da pilha, o ponteiro da pilha passará a conter o endereço nulo.

```
Algoritmo 4.9 – Desempilhar
   Entrada: PtPilha (TipoPtNodo)
   Saídas: PtPilha (TipoPtNodo)
           Valor (TipoInfo)
           Sucesso (lógico)
   Variável local: PtAux (TipoPtNodo)
início
   se PtPilha = nulo
   então Sucesso ← falso
   senão início
         Sucesso ← verdadeiro
         Valor ← PtPilha↑.Info
         PtAux ← PtPilha
         PtPilha ← PtPilha↑.Elo
         liberar(PtAux)
         fim
fim
```

■ destruição de uma pilha encadeada

Quando uma pilha não é mais necessária à aplicação, é interessante que as posições ocupadas por ela sejam liberadas. Isto é feito percorrendo a pilha, a partir de seu topo, liberando cada uma das posições acessadas. No final da operação, o endereço do topo da pilha será nulo. O algoritmo a seguir implementa esta operação, que é equivalente à operação de destruição de uma lista encadeada (Algoritmo 3.25).

```
Algoritmo 4.10 – DestruirPilhaEnc
   Entrada: PtPilha (TipoPtNodo)
   Saída: PtPilha (TipoPtNodo)
   Variável local: PtAux (TipoPtNodo)
início
   enquanto PtPilha ≠ nulo
   faça início
        PtAux ← PtPilha
        PtPilha ← PtPilha↑.Elo
        liberar(PtAux)
        fim
fim
```

4.2 ⋯→ filas

Filas são listas nas quais somente podem ser acessados os dois nodos que estão nas duas extremidades da lista. As inclusões de novos nodos são sempre efetuadas no final da lista, e as operações de consulta, alteração de informações e de remoção, no seu início. Esta disciplina de acesso recebe o nome de FIFO – *First In First Out* – primeiro nodo a ser retirado deve ser o primeiro que foi incluído (Figura 4.4).

Existem várias aplicações na área de computação para as estruturas de dados do tipo fila. Por exemplo, a estrutura apropriada para o gerenciamento de impressões em sistemas multiusuários, onde somente uma requisição pode ser atendida de cada vez, é uma fila, na qual as primeiras requisições para impressão são também as primeiras a serem executadas.

As operações que podem ser realizadas sobre uma fila são:

- criar a fila vazia;
- inserir um novo nodo no final da fila;
- excluir o nodo que está no início da fila;
- consultar e/ou modificar o nodo que está no início da fila;
- destruir a fila, ou seja, liberar o espaço que estava reservado para a fila.

A seguir são analisadas estas operações considerando novamente as duas possíveis formas de representação física – por contigüidade física e por encadeamento.

4.2.1 filas implementadas por contigüidade física

A disciplina de fila impõe que todas as inserções de novos nodos sejam feitas no final da fila, e que todas as retiradas ocorram no início. Como na implementação de uma fila sobre um arranjo todos os nodos poderiam ser acessados a qualquer momento, é necessário que sejam conhecidos os índices

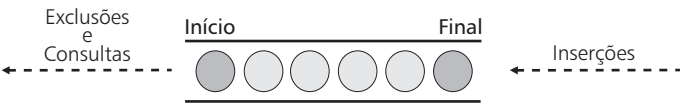


Figura 4.4 Fila.

que identificam as extremidades da fila, de modo que as operações sejam realizadas somente nestes elementos. Neste texto são utilizados os seguintes nomes para estes índices:

- `LI` e `LS` – índices que delimitam o início e o final do espaço reservado no arranjo para a fila;
- `IF` – início da fila, ou seja, índice do primeiro nodo da fila, de onde são retirados os nodos e no qual são feitas as operações de consulta e de alteração;
- `FF` – fim da fila, ou seja, índice do último nodo da fila, atrás do qual são inseridos os novos nodos.

Como exemplo de manipulação desta estrutura de dados, consideremos uma fila implementada sobre um arranjo denominado FILA, a partir do índice 1, com espaço para armazenar no máximo 6 nodos. Nesta fila são armazenados valores inteiros. A seguinte seqüência de operações é apresentada graficamente na Figura 4.5:

1 inicializar a fila, aqui representado pelo valor zero para o índice de início da fila (`IF`) (Figura 4.5a);

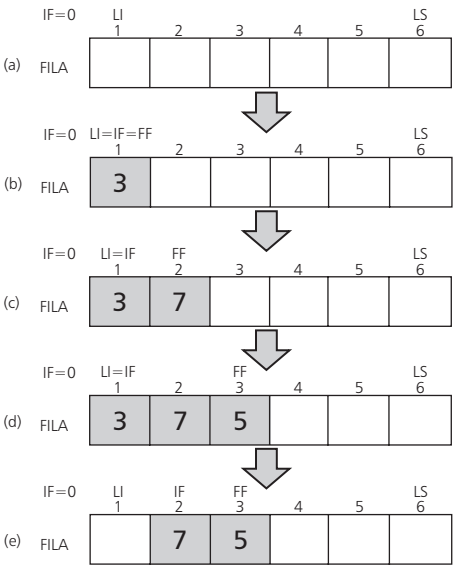


Figura 4.5 Exemplo de manipulação de uma fila.

2 inserir um novo nodo com valor 3 (Figura 4.5b);
3 inserir um novo nodo com valor 7 (Figura 4.5c);
4 inserir um novo nodo com valor 5 (Figura 4.5d);
5 remover um nodo (Figura 4.5e).

Caso seja realizada uma consulta a esta fila após estas operações, o valor retornado será 7, que é o valor contido no nodo que está no início da fila.

Neste exemplo pode-se observar que a área disponível para a fila vai sendo preenchida, avançando o início da fila ao longo do arranjo à medida que os nodos vão sendo removidos da fila. Chegará um momento em que não será mais possível inserir novos nodos no final da fila, por falta de espaço na área disponível para o arranjo. Entretanto, no início desta área provavelmente restarão posições que poderiam ser ocupadas pelos novos nodos. Por isso, sugere-se que na implementação de filas sobre arranjos seja utilizada a estratégia de listas circulares – quando não existir mais espaço no final da área disponível no arranjo para inserir o novo nodo, é utilizado o espaço que sobrou no início, conforme apresentado na Seção 3.3. Na Figura 4.6 é ilustrada a situação em que uma fila de quatro nodos inicia no elemento de índice 4 e termina na primeira posição do arranjo.

Embora a forma mais indicada para implementar filas sobre arranjos seja fazendo a ocupação circular da área disponível, alternativamente poderia ser feito o rearranjo deste espaço, conforme mostrado no capítulo 3.

A seguir são apresentados algoritmos que implementam as operações básicas que podem ser executadas sobre uma fila. Será utilizado o seguinte tipo de dado:

```
TipoFila = arranjo[1.. N] de TipoNodo
```

O arranjo no qual a fila é implementada é denominado neste texto de `Fila`, sendo utilizada a estratégia de ocupação circular do espaço disponível para a fila em todas as operações. A alocação de espaço sobre o arranjo que pode

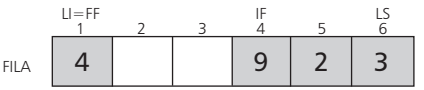


Figura 4.6 Ocupação circular do arranjo pela fila.

ser ocupado pela fila fica a cargo da aplicação, que deve informar os índices de início e de final desta área (LI e LS).

■ criação de uma fila

Uma vez alocada a área destinada para a fila, a operação de criação desta fila inicializa as variáveis de início e de final da fila. Deve ser definida uma estratégia para reconhecer quando a fila estiver vazia. Neste texto adota-se que a fila vazia é representada pelos valores dos índices de início e de final imediatamente anteriores ao primeiro do espaço disponível para a fila.

A seguir é apresentado um algoritmo para realizar a operação de inicializar (criar) uma fila. O índice do início da área disponível para a fila (LI) é recebido como parâmetro. Ao final de sua execução, os indicadores de início e final da fila (IF e FF) são devolvidos inicializados, usando como convenção que o início e o final da fila, uma unidade a menos do que o índice da primeira posição que pode ser ocupada pela fila no arranjo.

```
Algoritmo 4.11 – InicializarFilaArr
   Entrada: LI (inteiro)
   Saídas: IF, FF (inteiros)
início
  IF ← FF ← LI – 1
fim
```

■ inserção de um nodo em uma fila

Novos nodos são inseridos sempre no final da fila. Para realizar a inserção é necessário verificar se existe espaço disponível para o novo nodo. Três situações podem ser identificadas:

- existe espaço atrás do último nodo, estando o índice de início da fila antes do de final da fila, isto é, FF < LS e IF < FF;
- a fila ocupa o espaço até o final, mas há espaço livre no início da área disponível, isto é, FF = LS e LI < IF; e
- a fila ocupa o espaço até o final e continua no início do arranjo, restando espaço livre no meio, isto é, FF < IF – 1.

Uma vez verificada a existência de espaço para a inserção, o novo nodo pode ser inserido. O índice de final da fila é atualizado e o valor para o campo de

informação, passado como parâmetro, é inserido na posição correspondente. O algoritmo apresentado a seguir implementa esta operação, retornando Sucesso falso caso não exista espaço para a inserção do novo nodo.

```
Algoritmo 4.12 – InserirFilaArr
   Entradas:  Fila (TipoFila)
              LI, LS, IF, FF (inteiros)
              Info (TipoNodo)
   Saídas: IF, FF (inteiros)
           Sucesso (lógico)
início
  se (FF ≠  IF – 1) e ((IF ≠ LI)  ou  (FF ≠ LS))
  então início
        se  IF = LI – 1
        então IF ← FF ← LI  {INSERÇÃO DO PRIMEIRO NODO}
        senão se FF = L
              então FF ← LI  {INSERÇÃO NO INÍCIO}
              senão FF ← FF + 1  {INSERÇÃO NO MEIO OU ATRÁS}
        FILA[FF] ← Info
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

■ remoção de um nodo de uma fila

Somente o nodo que está no início da fila pode ser removido (excluído da fila) devido à disciplina imposta para esta estrutura de dados. Para efetuar a remoção do nodo é necessário somente avançar o índice do início da fila para o próximo nodo, que passará a ser o primeiro. Deve-se tomar cuidado ao avançar este índice quando estiver sendo adotada a utilização circular do espaço do arranjo destinado à fila.

A seguir é apresentado um algoritmo que realiza esta operação. O parâmetro lógico Sucesso retorna falso caso a fila esteja vazia. Se existir algum nodo na fila, o índice de início da fila (IF) avança para o próximo nodo. Se o nodo removido estiver na última posição da área disponível (IF = LS), o próximo nodo estará no início da área disponível (em LI). Caso o nodo removido seja o único da fila (IF = FF), os índices de início e de final da fila são colocados no valor padronizado para fila vazia (aqui adotado como uma unidade inferior ao índice de início da área disponível). Somente o valor do índice de início da fila é atualizado – nenhuma alteração é feita no arranjo.

```
Algoritmo 4.13 – RemoverFilaArr
   Entradas:  LI, LS, IF, FF (inteiros)
   Saídas: IF, FF (inteiros)
           Sucesso (lógico)
início
  se IF ≠ LI – 1
  então início
        se IF = FF
        então IF ← FF ← LI – 1  {FILA FICA VAZIA}
        senão se IF = LS
              então IF ← LI
              senão IF ← IF + 1
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

■ acesso a uma fila

Somente o nodo que está no início da fila pode ser acessado, tanto para consulta como para alteração. O algoritmo a seguir realiza uma consulta a uma fila, retornando o valor contido no campo de informação do nodo que está no início da fila através do parâmetro `Info`. Caso a fila esteja vazia, o parâmetro `Sucesso` retorna falso.

```
Algoritmo 4.14 – ConsultarFilaArr
   Entradas:  Fila (TipoFila)
              LI, IF (inteiros)
   Saídas: Info (TipoNodo)
           Sucesso (lógico)
início
  se  IF ≠ LI – 1
  então início
        Info ← Fila[IF]
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

4.2.2 filas implementadas por encadeamento

Quando uma fila é implementada através de encadeamento, cada nodo deverá ter pelo menos um campo de informação (aqui denominado `Info`) e um campo de elo que o encadeia com o próximo nodo da fila (aqui denominado `Elo`). Os tipo aqui definido para os nodos da pilha encadeada será portanto o seguinte:

```
TipoNodo = registro
             Info: TipoInfo
             Elo: TipoPtNodo
           fim registro
```

Devido à disciplina das filas, não é adequado conhecer somente o endereço do primeiro nodo da fila, pois como novos nodos sempre devem ser inseridos no final, será necessário percorrer toda a fila a cada inserção de um nodo. A forma mais adequada de implementar filas por encadeamento é utilizando um descritor para acessar a fila, no qual são armazenados os endereços do primeiro nodo (para operações de remoção e consulta) e do último nodo (para operações de inserção). Nos algoritmos mostrados a seguir é utilizada esta estratégia, sendo usado um descritor com dois campos (Figura 4.7):

`Prim` – campo que contém o endereço do primeiro nodo da fila; e
`Ult` – que contém o endereço do último nodo.

Os tipos de dados correspondentes a este descritor e do ponteiro para este descritor são os seguintes:

```
TipoDFila = registro
              Prim: TipoPtNodo
```

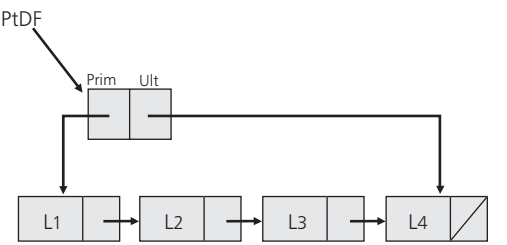


Figura 4.7 Descritor para uma fila.

```
            Ult: TipoPtNodo
          fim registro
TipoPtDFila = ↑TipoDFila
```

■ criação da fila encadeada

A criação de uma fila encadeada, endereçada por um descritor, consiste na alocação deste descritor, inicializando os endereços do primeiro e do último nodo em um endereço nulo. O algoritmo a seguir executa esta operação, retornando o endereço do descritor alocado.

```
Algoritmo 4.15 – CriarFilaEnc
   Entradas: -
   Saída: PtDFila (TipoPtDFila)
início
  alocar(PtDFila)
  PtDFila↑.Prim ← nulo
  PtDFila↑.Ult ← nulo
fim
```

■ inserção de um nodo na fila encadeada

A operação de inserção de um novo nodo em uma fila encadeada requer somente a alocação do novo nodo e o seu encadeamento com o último. A utilização do descritor contendo diretamente o endereço do último nodo da fila torna esta operação trivial.

A seguir é apresentado um algoritmo que realiza esta operação. O algoritmo recebe, como parâmetro, o endereço do descritor (`PtDFila`), alocado através da operação anterior. O novo nodo é alocado e encadeado com aquele que era o último, passando agora a ser o último da fila. Caso este seja o primeiro nodo alocado para a fila, seu endereço também é colocado no campo de início da fila, no descritor.

```
Algoritmo 4.16 – InserirFilaEnc
   Entradas: PtDFila (TipoPtDFila)
             Valor (TipoInfo)
   Saídas: -
   Variável auxiliar: PtNovo (TipoPtNodo)
início
  alocar(PtNovo)
```

```
  PtNovo↑.Info ← Valor
  PtNovo↑.Elo ← nulo
  se PtDFila↑.Prim = nulo
  então PtDFila↑.Prim ← PtNovo
  senão (PtDFila↑.Ult)↑.Prox ← PtNovo
  PtDFila↑.Ult ← PtNovo
fim
```

■ remoção de um nodo da fila encadeada

Somente o primeiro nodo pode ser removido de uma fila. Assim, o processo de remoção necessita apenas do endereço do primeiro nodo da fila, que está armazenado no descritor.

O algoritmo a seguir executa esta operação, retornando `Sucesso` falso caso a lista esteja vazia, ou seja, quando não existir algum nodo a ser removido. Caso exista algum nodo para ser removido, o campo de início da fila é dirigido para o segundo nodo, sendo o nodo requerido liberado. Se o nodo removido for o único da fila, tanto o campo de início como o de final da fila são atualizados para o endereço nulo.

```
Algoritmo 4.17 – RemoverFilaEnc
   Entrada: PtDFila (TipoPtDFila)
   Saída: Sucesso (lógico)
   Variável auxiliar: PtAux (TipoPtNodo)
início
  se PtDFila↑.Prim ≠ nulo
  então início
        PtAux ← PtDFila↑.Prim
        PtDFila↑.Prim ← PtAux↑.Elo
        liberar(PtAux)
        se PtDFila↑.Prim = nulo
        então PtDFila↑.Ult ← nulo
        Sucesso ← verdadeiro
        fim
   senão Sucesso ← falso
fim
```

■ acesso a uma fila encadeada

Somente o primeiro nodo de uma fila encadeada pode ser acessado para consulta ou alteração. O algoritmo a seguir devolve o valor contido no campo

de informação do primeiro nodo através do parâmetro `Valor`. O parâmetro `Sucesso` retorna falso caso a lista esteja vazia.

```
Algoritmo 4.18 – ConsultarFilaEnc
   Entrada: PtDFila (TipoPtDFila)
   Saídas: Valor (TipoInfo)
           Sucesso (lógico)
início
  se PtDFila↑.Prim ≠ nulo
  então início
        Valor ← PtDFila↑.Prim↑.Info
        Sucesso ← verdadeiro
        fim
  senão Sucesso ← falso
fim
```

■ destruição de uma fila encadeada

A destruição de uma fila encadeada é feita percorrendo-a, a partir de seu início, liberando todas as posições ocupadas. Finalmente, o descritor também é liberado. O algoritmo apresentado a seguir executa esta operação. Para caracterizar que a fila foi totalmente liberada, o endereço de seu descritor (`PtDFila`) é retornado com endereço nulo.

```
Algoritmo 4.19 – DestruirFilaEnc
   Entrada: PtDFila (TipoPtDFila)
   Saída: PtDFila (TipoPtDFila)
   Variáveis auxiliares: P1, P2 (TipoPtNodo)
início
   se PtDFila↑.Prim ≠ nulo
   então início
         P1 ← PtDFila↑.Prim
         repita
            P2 ← P1↑.Elo
            liberar(P1)
            P1 ← P2
         até P1 = nulo
         fim
   liberar(PtDFila)
   PtDFila ← nulo
fim
```

4.3 ⋯→ fila dupla – *deque*

Uma fila dupla, também conhecida como "deque", é um tipo especial de fila, na qual é permitido o acesso a qualquer uma das duas extremidades da lista, mas somente às extremidades (Figura 4.8). Inserções, alterações, remoções e consultas podem ser realizadas tanto no início quanto no final da fila dupla. Exemplos deste tipo de lista são:

- canais de navegação marítima ou fluvial;
- servidões com circulação nos dois sentidos.

As seguintes operações podem ser realizadas sobre filas duplas:

- criar a fila dupla vazia;
- inserir um novo nodo em uma das duas extremidades;
- excluir o nodo que está em uma das duas extremidades;
- consultar e/ou modificar o nodo que está em uma das duas extremidades;
- destruir a fila, liberando o espaço que estava reservado para ela.

A seguir são analisadas estas operações, considerando novamente as duas possíveis formas de representação física – por contigüidade física e por encadeamento. A extremidade na qual deve ser realizada uma determinada operação deverá ser informada pela aplicação. Duas estratégias podem ser utilizadas na implementação das operações: (1) definir uma operação diferente para cada uma das extremidades; ou (2) construir uma operação genérica, que se aplique às duas extremidades, passando a informação de qual a extremidade considerada através de um parâmetro. Para ilustrar as duas formas de definição das operações neste texto, a primeira forma será utilizada para o caso de implementação através de contigüidade física, e a segunda para o caso de filas duplas encadeadas.

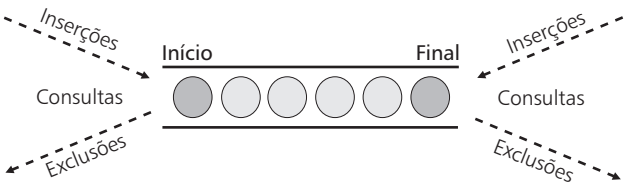


Figura 4.8 Fila dupla (*Deque*).

4.3.1 filas duplas implementadas por contigüidade física

A implementação de uma fila dupla sobre um arranjo é semelhante à implementação de uma fila simples, tendo como base os índices que controlam o início e o final da área do arranjo (`LI` e `LS`) a ser utilizada para a fila, e as extremidades desta fila (`IF` e `FF`). Para otimizar a utilização do arranjo é aconselhável que neste caso também seja utilizada a estratégia de ocupação circular do espaço disponível. Entretanto, como não se pode prever em qual extremidade as operações serão mais freqüentes, a lista dupla deve ser implementada de forma circular nos dois sentidos, tanto para a direita quanto para a esquerda. No caso da lista simples, as inclusões sendo sempre no final da fila, a ocupação circular era feita sempre continuando a fila no início do arranjo.

A seguir são analisados os algoritmos que implementam as operações básicas sobre uma fila dupla, com ocupação circular do espaço disponível. Será utilizado o seguinte tipo de dado:

```
TipoDequeArr = arranjo[1.. N] de TipoNodo
```

Nos algoritmos apresentados a seguir, a representação da fila vazia é feita através de seu índice de início valendo uma unidade a menos do que o índice de início da área disponível (`IF = LI-1`). A operação de destruição deste tipo de fila não será analisada, uma vez que deverá ser controlada diretamente pela aplicação.

Como citado anteriormente, as operações de inserção, remoção e consulta, mostradas a seguir, serão específicas para cada uma das extremidades da fila dupla, extremidade esta que deve ser definida pelo usuário.

■ criação de uma fila dupla

A operação de criação de uma fila dupla implementada sobre um arranjo é idêntica à de criação de uma fila convencional (Algoritmo 4.11), consistindo tão somente da inicialização dos indicadores de início e de final da fila, de acordo com a área a ela reservada.

O algoritmo para realizar esta operação recebe, como parâmetro, o índice do início da área disponível para a fila (`LI`). Ao final de sua execução, os indicadores de início e de final da fila (`IF` e `FF`) estarão inicializados em uma

unidade a menos do que o índice da primeira posição que pode ser ocupada pela fila no arranjo.

```
Algoritmo 4.20 – InicializarDequeArr
   Entrada: LI (inteiro)
   Saídas: IF, FF (inteiros)
início
  IF ← FF ← LI – 1
fim
```

■ inserção de um nodo em uma fila dupla

A aplicação deverá indicar em qual das extremidades o novo nodo deve ser inserido. Para realizar a inserção é necessário que a área disponível para a fila não esteja toda ocupada. Caso haja espaço para a inserção do novo nodo e considerando a ocupação circular do arranjo, as seguintes situações podem ocorrer:

- para inserção no final da fila, o nodo será inserido na primeira posição logo após o último nodo ocupado. Caso esse ocupe a última posição disponível no arranjo para esta fila, o novo nodo ocupará a primeira posição desta área disponível. Esta operação é a mesma apresentada para a fila tradicional (Algoritmo 4.12);
- para inserção no início da fila dupla, o nodo é inserido na posição imediatamente anterior ao nodo que está no início da fila. Caso esse ocupe a primeira posição disponível no arranjo, o novo nodo ocupará a última posição da área disponível. Esta operação é semelhante àquela apresentada para inserção de um nodo na primeira posição de uma lista (Algoritmo 3.15, para `K=1`).

O algoritmo apresentado a seguir implementa a operação de inserção no início de uma fila dupla, retornando `Sucesso` falso caso não exista espaço disponível para a inserção. Inicialmente o algoritmo verifica se existe espaço livre para inserir o nodo, o que é determinado pela posição ocupada pelos índices de início e de final da fila. A inserção é realizada se existir espaço, utilizando, caso seja necessário, a ocupação circular no arranjo. Se a fila estiver vazia, o novo nodo é inserido na primeira posição da área disponível, sendo atualizados os índices de início e de final da fila.

```
Algoritmo 4.21 – InserirIniDequeArr
   Entrada: Deque (TipoDequeArr)
            LI, LS, IF, FF (inteiros)
```

```
           Info (TipoNodo)
   Saídas: Deque (TipoDequeArr)
           IF, FF (inteiros)
           Sucesso (lógico)
início
  se ((FF =  IF – 1) ou ((IF = LI) e (FF = LS)))
  então Sucesso ← falso
  senão início
        Sucesso ← verdadeiro
        se IF = LI – 1  {DEQUE VAZIA}
        então IF ← FF ← LI
        senão se IF > LI
              então IF ← IF – 1
              senão IF ← LS
        Deque[IF] ← Info
        fim
fim
```

■ remoção de um nodo de uma fila dupla

Como na operação anterior, a extremidade da qual deverá ser realizada a remoção deve ser especificada pela aplicação. A remoção do nodo do início é semelhante à remoção de um nodo de uma fila simples, enquanto que a remoção do último nodo equivale à retirada de um nodo do topo de uma pilha.

Para exemplificar, a seguir é apresentado um algoritmo que implementa a remoção do último nodo de uma fila dupla implementada sobre um arranjo. Somente o índice de final da fila dupla é atualizado, sem alteração nos conteúdos do arranjo que implementa a fila. Caso a fila resulte vazia, os índices de início e de final da fila dupla são colocados em uma unidade a menos do que o índice de início da área disponível no arranjo. O parâmetro lógico `Sucesso` retorna falso somente no caso de a fila estar vazia no início da operação.

```
Algoritmo 4.22 – RemoverFimDequeArr
   Entradas: LI, LS, IF, FF (inteiros)
   Saídas: IF, FF (inteiros)
           Sucesso (lógico)
início
  se IF ≠ LI – 1
  então início
        se IF = FF {DEQUE VAI FICAR VAZIA}
        então IF ← FF ← LI  – 1
        senão se FF = LI
```

```
            então FF ← LS
            senão FF ← FF – 1
        Sucesso ← verdadeiro
        fim
   senão Sucesso ← falso
fim
```

■ acesso a uma fila dupla

Já que as duas extremidades podem ser acessadas, aumentam as possibilidades de utilização desta estrutura de dados. Uma aplicação poderá consultar especificamente uma das extremidades (especificando *a priori* qual delas), ou consultar as duas extremidades simultaneamente em busca de alguma informação.

Para ilustrar este último caso, a seguir é apresentado um algoritmo que devolve o maior dos dois valores contidos nas extremidades da fila dupla. Supondo que todos os valores contidos nos campos de informação são inteiros, todos maiores do que zero, no caso de a fila estar vazia é devolvido o valor nulo. No caso de os valores dos nodos das duas extremidades serem iguais, o algoritmo retornará o valor sem informar a igualdade.

```
Algoritmo 4.23 – ConsultarMaiorDequeArr
   Entradas: Deque (TipoDequeArr)
             LI, IF, FF (inteiros)
   Saída: MaiorValor (inteiro)
início
  se  IF = LI – 1
  então MaiorValor ← 0
  senão se Deque[IF] > Deque[FF]
        então MaiorValor ← Deque[IF]
        senão MaiorValor ← Deque[FF]
fim
```

4.3.2 filas duplas encadeadas

Na implementação de filas duplas através de encadeamento também é conveniente a utilização de um descritor que contenha o endereço do primeiro e do último nodos, permitindo acessá-los diretamente. Isto permite que o acesso ao último nodo seja otimizado para as operações de inserção, alteração e consulta. No entanto, esta representação não facilita a operação de remoção

quando o nodo a ser removido for o último, pois neste caso é necessário atualizar o campo de elo do penúltimo nodo. Este problema é solucionado quando se implementa este tipo de fila como uma lista duplamente encadeada.

A seguir são analisadas as operações básicas para listas duplas implementadas com duplo encadeamento. Os tipos utilizados para o descritor e seu apontador são os seguintes:

```
TipoDDeque = registro
                Prim: TipoPtNodo
                Ult: TipoPtNodo
             fim registo
TipoPtDDeque = ↑TipoDDeque
```

O tipo do nodo é o mesmo utilizado na seção 3.6, ou seja:

```
TipoNodo = registro
              Ant: TipoPtNodo
              Info: TipoInfoNodo
              Prox: TipoPtNodo
           fim registro
```

Diferentemente do apresentado para as listas duplas implementadas sobre arranjos, onde cada operação era específica para uma das extremidades, aqui são vistos algoritmos genéricos, que implementam as operações em qualquer uma das duas extremidades, sendo o lado informado através de um parâmetro.

■ criação de uma fila dupla encadeada

De forma semelhante à mostrada para uma fila encadeada simples, a criação de uma fila dupla encadeada endereçada por um descritor consiste na alocação deste descritor, inicializando os endereços do primeiro e do último nodo da fila dupla em um endereço nulo (`nulo`). O algoritmo a seguir executa esta operação.

Algoritmo 4.24 – CriarDequeEnc

```
   Entradas: -
   Saída: PtDDeque (TipoPtDDeque)
início
  alocar(PtDDeque)
  PtDDeque↑.Prim ← PtDDdeque↑.Ult ← nulo
fim
```

■ inserção de um nodo em fila dupla encadeada

Como um novo nodo pode ser inserido em qualquer uma das duas extremidades da fila dupla, é necessário que seja definida a posição de inserção. As inserções na frente ou no final de uma fila dupla são semelhantes à inserção no início e no final de uma lista duplamente encadeada, vistas anteriormente.

Para ilustrar a operação de inserção, a seguir é apresentado um algoritmo que insere um novo nodo na fila dupla, implementada com duplo encadeamento, sendo informado através de um parâmetro (Lado) qual a extremidade em que a inserção deve ser realizada: para a inserção no início deve ser fornecido o caractere "I"; e para a inserção no final, o caractere "F". O parâmetro Sucesso retorna falso no caso de receber outro caractere. Supõe-se que o descritor da lista dupla tenha sido previamente alocado, quando da inicialização da lista. Além de inserir o nodo na extremidade solicitada e encadeá-lo apropriadamente, o campo correspondente do descritor é atualizado.

```
Algoritmo 4.25 – InserirDequeEnc
   Entradas: PtDeque (TipoPtDeque)
             Lado (caractere)
             Valor (TipoInfoNodo)
   Saída: Sucesso (lógico)
var PtNovo (TipoPtNodo)
início
  Sucesso ← falso
  se (Lado = "I") ou (Lado = "F")
  então início
        Sucesso ← verdadeiro
        alocar(PtNovo)
        PtNovo↑.Info ← Valor
        se Lado = "I"
        então início  {INSERE NO INÍCIO}
              PtNovo↑.Ant ← nulo
              se PtDDeque↑.Prim = nulo
              então início
                    PtDDeque↑.Ult ← PtNovo
                    PtNovo↑.Prox ← nulo
                    fim
              senão início
                    PtNovo↑.Prox ← PtDDeque↑.Prim
                    (PtDDeque↑.Prim)↑.Ant ← PtNovo
                    fim
              PtDDeque↑.Prim ← PtNovo
```

```
            fim
        senão início  {INSERE NO FINAL}
            PtNovo↑.Prox ← nulo
            se PtDDeque↑.Prim = nulo
            então início
                PtNovo↑.Ant ← nulo
                PtDDeque↑.Prim ← PtNovo
                fim
            senão início
                (PtDDeque↑.Ult)↑.Prox ← PtNovo
                PtNovo↑.Ant ← PtDDeque↑.Ult
                fim
            PtDDeque↑.Ult ← PtNovo
            fim
        fim
fim
```

■ remoção de um nodo da fila dupla encadeada

Também para a operação de remoção deve ser definido qual o nodo a ser removido: o primeiro ou o último. A existência do descritor permite o acesso direto a qualquer uma das duas extremidades. Quando o último nodo for removido, é necessário atualizar o elo do penúltimo nodo, que passará a ser o último.

O algoritmo apresentado a seguir, a exemplo do anterior, recebe a informação de qual extremidade o nodo deve ser removido através de um parâmetro ("`I`" quando for o primeiro, e "`F`" para o último). O parâmetro `Sucesso` retorna falso nas seguintes condições: se a lista estiver vazia, pois não haverá nodo a ser removido; ou quando o caractere recebido for diferente dos dois acima citados. Caso a remoção possa ser realizada, é feita a liberação da posição ocupada pelo nodo, tornado nulo o endereço de campo próximo do nodo que agora passa a ser o último, e atualizado o descritor. Se a lista apresentar somente um nodo, a remoção resultará na lista vazia.

Algoritmo 4.26 – RemoverDequeEnc

```
   Entradas: PtDDeque (TipoPtDDeque)
             Lado (caractere)
   Saída: Sucesso (lógico)
var PtAux, PtAnt (TipoPtNodo);
início
  Sucesso ← falso
  se (PtDDeque↑.Prim ≠ nulo) e ((Lado = "I") ou (Lado = "F"))
```

```
  então início
        Sucesso ← verdadeiro
        se Lado = "I"
        então início  {REMOVE O PRIMEIRO}
              PtAux ← PtDDeque↑.Prim
              se PtDDeque↑.Prim = PtDDeque↑.Ult
              então PtDDeque↑.Prim ← PtDDeque↑.Ult ← nulo
              senão início
                    PtDDeque↑.Prim ← PtAux↑.Prox
                    (PtDDeque↑.Prim)↑.Ant ← nulo
                    fim
              fim
        senão início  {REMOVE O ÚLTIMO}
              PtAux ← PtDDeque↑.Ult
              se PtDDeque↑.Prim = PtDDeque↑.Ult
              então PtDDeque↑.Prim ← PtDDeque↑.Ult ← nulo
              senão início
                    PtDDeque↑.Ult ← PtAux↑.Ant
                    (PtDDeque↑.Ult)↑.Prox ← nulo
                    fim
              fim
        liberar(PtAux)
        fim
fim
```

■ acesso a uma das extremidades da fila dupla encadeada

Como somente as duas extremidades de uma fila dupla podem ser acessadas, a operação de acesso é imediata quando for utilizado o descritor que contém os endereços do primeiro e do último nodo. Para ilustrar, apresentamos a seguir um algoritmo que devolve, através do parâmetro `Valor`, o valor contido no campo de informação de uma fila dupla, sendo informada qual a extremidade a ser consultada. É utilizada a mesma convenção das operações anteriores, sendo a extremidade informada através de um parâmetro (`Lado`). O parâmetro `Sucesso` retorna falso no caso de a lista estar vazia, ou de ser fornecido um caractere inválido (diferente de "`I`" e "`F`") para definir a extremidade que deve ser consultada.

Algoritmo 4.27 – ConsultarDequeEnc

```
   Entradas: PtDDeque (TipoPtDDeque)
             Lado (caractere)
   Saídas: Valor (TipoInfoNodo)
```

```
          Sucesso (lógico)
início
  Sucesso ← falso
  se (PtDDeque↑.Prim ≠ nulo) e ((Lado = "I") ou (Lado = "F"))
  então início
        Sucesso ← verdadeiro
        se Lado = "I"
        então Valor ← (PtDDeque↑.Prim).Info
        senão Valor ← (PtDDeque↑.Ult).Info;
        fim
fim
```

4.4 ⇢ exercícios

exercícios com pilhas

exercício 1 Em algumas aplicações tem-se que trabalhar com mais de uma pilha ao mesmo tempo. Pode-se implementar essas pilhas em um mesmo arranjo, de forma que cada uma ocupe parte desse arranjo. Faça um programa que tenha duas pilhas alocadas em um mesmo arranjo, sendo que cada uma delas cresce em sentido oposto. Implemente algoritmos para as seguintes operações:

- inicialização das duas pilhas;
- verificar se as duas pilhas estão vazias;
- empilhar um elemento em cada uma das pilhas;
- desempilhar um elemento de cada uma das pilhas;
- imprimir o conteúdo do nodo no topo das duas pilhas.

exercício 2 Faça um algoritmo que leia um conjunto de valores inteiros e armazene estes valores em duas pilhas, uma para os valores positivos lidos e a outra, para os valores negativos. As pilhas devem ser implementadas sobre um mesmo arranjo `PILHAS`, a partir de posições fornecidas como parâmetros. Em caso de *overflow* de alguma das pilhas, deve ser emitida uma mensagem.

exercício 3 Considere um arranjo unidimensional de `N` elementos inteiros, utilizado para implementar duas pilhas `A` e `B`. As bases das pilhas estarão uma em cada extremidade do arranjo. Cada uma das pilhas apresenta um índice próprio indicando o topo. Escreva:

- um algoritmo para incluir um elemento no topo da pilha B, atualizando o indicador de topo desta pilha. Este algoritmo receberá, como parâmetros, o valor do elemento a ser incluído na pilha B, os índices correspondentes ao topo das duas pilhas e o número de elementos do arranjo. Deverá testar se existe uma célula livre para incluir o elemento (se as duas pilhas não estão ocupando todo o arranjo) e, caso isto ocorrer, não efetuar a inclusão e enviar uma mensagem avisando;
- um algoritmo que calcule a quantidade de elementos da pilha A que também aparecem na pilha B. Elementos repetidos na mesma pilha devem ser contados apenas uma vez.

exercício 4 Implemente um algoritmo que receba, como parâmetro, o endereço do nodo no topo de uma pilha encadeada e retorne o endereço do topo de uma cópia dessa pilha. Ao final da execução, a pilha deve estar no mesmo estado em que estava no início.

exercício 5 Implemente um algoritmo que teste se duas pilhas são ou não iguais. O algoritmo deve retornar o valor `1` se as duas pilhas forem iguais, e `0` caso contrário. Ao final da execução, as duas pilhas devem estar no mesmo estado em que estavam no início.

exercícios com filas

exercício 6 No caso de duas filas serem implementadas sobre um mesmo arranjo, cada uma com seu espaço disponível definido, implemente algoritmos para as seguintes operações:

- inicializar uma das duas filas;
- verificar se uma das filas está vazia;
- inserir um novo nodo em uma das filas;
- remover um nodo de cada uma das filas;
- informar o conteúdo do primeiro nodo de cada uma das filas.

exercício 7 Dadas duas filas implementadas sobre dois arranjos, escreva um algoritmo que devolva o maior dos dois elementos na frente destas filas, removendo este elemento da fila correspondente. O algoritmo deverá receber, como parâmetros, os índices de início e de final de cada uma das filas, e atualizar o índice de início da fila da qual foi removido o elemento.

exercício 8 Implemente um algoritmo que receba três filas, `F`, `F_Impares` e `F_Pares`, e separe todos os valores guardados em `F` de tal forma que os valores pares são movidos para a fila `F_Pares` e os valores ímpares são movidos para `F_Impares`.

exercício 9 Escreva um algoritmo que recebe duas filas encadeadas, sendo que cada uma delas contém valores numéricos ordenados. O algoritmo deverá formar uma terceira fila encadeada, também ordenada, na qual estarão os valores armazenados nas filas originais.

exercício 10 Implemente as operações básicas sobre filas encadeadas, considerando que não é utilizado um descritor para a fila, ou seja, tem-se somente o endereço do primeiro nodo da fila.

exercícios com filas duplas

exercício 11 O Algoritmo 4.21 insere um novo nodo na extremidade da frente de uma fila dupla. Construa outro algoritmo que insira um novo nodo na outra extremidade, ou seja, no final da fila dupla.

exercício 12 O Algoritmo 4.22 remove o nodo que está no final de uma fila dupla. Construa outro algoritmo que remova o nodo que está na frente desta mesma fila dupla.

exercício 13 Considerando uma fila dupla implementada por contigüidade física com ocupação circular do espaço disponível, construa algoritmos para as seguintes operações:

- inserir um novo nodo em qualquer uma das duas extremidades da fila dupla. A extremidade onde deve ser realizada a inserção é passada como parâmetro. O algoritmo deverá informar caso não haja espaço para a inserção;
- remover um nodo de qualquer uma das extremidades, sendo esta extremidade passada como parâmetro. O algoritmo deverá ter um parâmetro lógico que indica se a remoção foi realizada com sucesso;
- informar o valor contido em uma das extremidades da fila dupla. A extremidade considerada deve ser passada como parâmetro.

exercício 14 Adapte os algoritmos de inclusão e de remoção de um novo nodo, tanto aqueles apresentados no texto (seção 4.3.1) como os que foram

construídos para o exercício anterior, para o caso de não ser utilizada a estratégia de ocupação circular do espaço disponível.

exercício 15 Reescreva as operações definidas para filas duplas encadeadas, na seção 4.3.2, para o caso em que não seja utilizado um descritor para a fila. Considere que somente o endereço do primeiro nodo da fila dupla é conhecido, e que não existe duplo encadeamento entre os nodos.

exercício 16 Reescreva as operações definidas para filas duplas encadeadas, para o caso em que não seja utilizado um descritor para a fila, mas a fila seja duplamente encadeada. Somente o endereço do primeiro nodo da fila deverá ser conhecido.

exercícios com pilhas, filas e filas duplas

exercício 17 Como você implementaria uma pilha de filas? Escreva rotinas para implementar as operações corretas para essa estrutura de dados.

exercício 18 Como você implementaria uma fila de pilhas? Escreva rotinas para implementar as operações corretas para essa estrutura de dados.

exercício 19 Dada uma fila onde cada nodo contém um número, escreva um algoritmo que coloque os nodos desta fila em ordem crescente, usando duas pilhas como únicas variáveis auxiliares.

exercício 20 Considere:

1. uma pilha implementada sobre um arranjo `P(20)` a partir da primeira posição. O topo da pilha está armazenado na variável `TOPO`; e
2. uma fila implementada sobre outro arranjo `F(20)`, a partir da posição 1. O início e o final da fila estão armazenados, respectivamente, nas variáveis `INICIO` e `FIM`. É utilizada a otimização de espaço do arranjo (quando a fila chegar ao final do arranjo, se ainda existir espaço disponível no início do arranjo, este será utilizado).

Escreva:

- um algoritmo que retire um elemento do topo da pilha e o coloque na fila;
- um algoritmo que retire um elemento da fila e o coloque na pilha.

exercícios de aplicações

exercício 21 Um estacionamento é composto de um único beco que guarda no máximo 10 carros. Existe apenas uma entrada/saída no estacionamento, em uma extremidade do beco. Se chegar um cliente para retirar um carro que não seja o mais próximo da saída, todos os carros bloqueando seu caminho sairão do estacionamento e os outros carros voltarão a ocupar a seqüência inicial. Escreva um programa que controle um grupo de carros deste estacionamento. O programa deve imprimir uma mensagem sempre que um carro chegar ou sair. Quando um carro chegar, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não houver vaga, o carro partirá sem entrar no estacionamento. Quando um carro sair do estacionamento, a mensagem deverá incluir o número de vezes em que o carro foi manobrado para fora e para dentro do estacionamento para permitir que outros carros saíssem.

exercício 22 Uma empilhadeira carrega caixas de 3, 5 e 7 toneladas. Há três pilhas A, B e C. A pilha A é onde se encontram todas as caixas que chegam no depósito. Com um detalhe: caixas maiores não podem ser empilhadas sobre caixas menores. Elabore um algoritmo que efetue o controle das caixas, de forma que caso uma caixa de maior peso do que uma que já está em A deva ser empilhada, todas as caixas que estão em A são movidas para as pilhas auxiliares B (contendo somente caixa de 5 toneladas) e C (contendo somente caixas de 3 toneladas) até que se possa empilhar a nova caixa. Depois, todas as caixas são movidas de volta para a pilha A.

exercício 23 Imagine um colecionador de vinhos que compra vinhos recentes e os guarda em uma adega para envelhecerem, e que a cada ocasião especial abre sempre sua última aquisição (para poupar os mais antigos). Construa um programa que:

- permita incluir novos vinhos na adega;
- informe qual vinho deve ser aberto em uma ocasião especial;
- relacione as cinco aquisições mais antigas.

As informações básicas que o registro de vinhos deve conter são: nome do produto e safra.

exercício 24 O problema das Torres de Hanói é bastante estudado em computação. O problema consiste em N discos de diferentes diâmetros e três estacas: A, B e C. Inicialmente os discos estão encaixados na estaca A onde o

menor está sempre em cima do maior disco. O objetivo é deslocar os discos para uma estaca `C`, usando a estaca `B` como auxiliar. Somente o primeiro disco de cada estaca pode ser deslocado. Construa a resolução desse exercício considerando `N = 4`, ou seja, para quatro discos.

exercício 25 Uma pilha pode ser usada para rastrear os tipos de escopos encontrados em uma expressão matemática e verificar se o uso deles está correto. Os delimitadores de escopo podem ser os parênteses, os colchetes e as chaves. Escreva um programa que leia uma expressão matemática e verifique se os escopos estão posicionados de forma correta.

exercício 26 Em um banco, deseja-se saber qual é o tempo médio que uma pessoa fica na fila em um dia de grande movimento (por exemplo, dia de pagamento). Para isso, são distribuídos 100 bilhetes. Ao entrar na fila única dos caixas, a pessoa recebe um bilhete onde está marcada a hora de entrada. Ao ser atendida pelo caixa, nesse mesmo bilhete é marcada a hora de atendimento. Ao acabar de recolher os 100 bilhetes, eles são computados, calculando-se o tempo médio de espera de uma pessoa na fila do caixa nesse período de amostragem. Faça um programa que simule esse ambiente. Para isso, construa um algoritmo que gerencie a entrada do cliente na fila, outro que gerencie a saída do cliente da fila, e um terceiro algoritmo para o cálculo do tempo médio de espera na fila.

exercício 27 Construa um programa que administre as filas de reservas de filmes em uma locadora, sendo que para cada filme existem sete filas (uma para cada dia da semana). O usuário é quem determina qual é o dia da semana de sua preferência para alugar um determinado filme. O programa deve permitir inclusões nas filas em qualquer ocasião e remoções das mesmas apenas nos respectivos dias – quando o cliente é comunicado por telefone da disponibilidade do filme e quando é confirmada sua locação (caso não seja locado, o próximo da fila será procurado).

exercício 28 O estacionamento Central possui uma única alameda que guarda até 10 carros. Os carros entram pela extremidade sul do estacionamento e saem pela extremidade norte. Se chegar um cliente para retirar um carro que não esteja estacionado na posição do extremo norte, todos os carros ao norte serão deslocados para fora, o carro sairá do estacionamento e os outros carros voltarão à mesma ordem em que se encontravam inicialmente. Sempre que um carro deixa o estacionamento, todos os carros ao sul são deslocados

para frente, de modo que o tempo inteiro todos os espaços vazios estão na parte sul do estacionamento. Escreva um programa que leia um grupo de linhas de entrada. Cada entrada contém um `C` (de chegada) e um `P` (de partida), além da placa do carro. O programa deve imprimir uma mensagem cada vez que o carro chegar ou partir. Quando o carro chegar, a mensagem deverá especificar se existe ou não vaga para o carro. Se não existir vaga, o carro esperará pela vaga ou até que uma linha de partida seja lida para o carro. Quando houver espaço disponível, outra mensagem deverá ser impressa. Quando o carro partir, a mensagem deverá incluir o número de vezes que o carro foi deslocado dentro do estacionamento, incluindo a própria partida mas não a chegada. Esse número será zero se o carro for embora a partir da linha de espera. Presume-se que os carros sairão e partirão na ordem especificada pela entrada.

exercício 29 Em um ambiente computacional em rede existe uma impressora que é compartilhada por todos. Para gerenciar o uso dessa impressora, o sistema operacional dispõe de quatro filas circulares, a saber:

- `fila de entrada` – recebe as solicitações de impressão de origem diversa;
- `fila 0` – recebe todas as solicitações de impressão da fila de entrada com prioridade máxima;
- `fila 1` – recebe as solicitações de impressão da fila de entrada com prioridade normal;
- `fila 2` – recebe as solicitações de impressão da fila de entrada com prioridade baixa.

Cada nodo destas filas é formado por um registro, com dois campos: `Prioridade` e `Identificação`. As filas estão implementadas em quatro arranjos: `FE(50)`, `F0(15)`, `F1(15)` e `F2(15)`, respectivamente. A cada 0,5 segundos o sistema operacional deve descarregar a fila de entrada, distribuindo as solicitações entre as filas com prioridade. Escreva um algoritmo que receba estas filas e esvazie a fila de entrada, distribuindo as solicitações de impressão em suas respectivas filas de prioridades.

exercício 30 Escreva um programa para simular um sistema de computador multiusuário, como segue: cada usuário tem um `ID` exclusivo e deseja executar uma operação no computador. Entretanto, somente uma transação pode ser processada por vez. Cada linha de entrada representa um único usuário

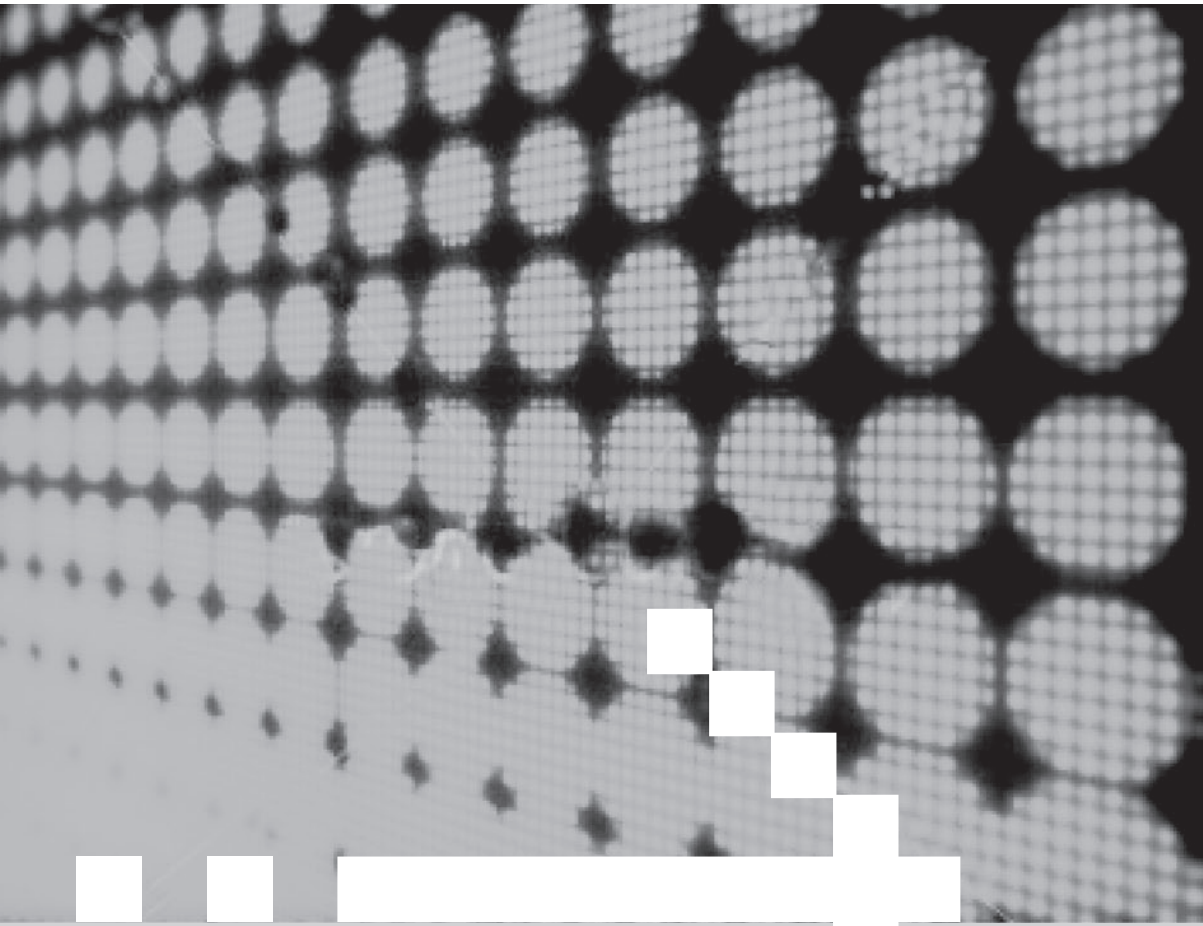
e contém o `ID` do usuário seguido de uma hora de início e a duração de sua transação. O programa deve simular o sistema e imprimir uma mensagem contendo o `ID` do usuário e a hora em que a transação começou e terminou. No final da simulação, ele deve imprimir o tempo médio geral de espera para uma transação (o tempo de espera de uma transação é o intervalo de tempo entre a hora em que a transação foi solicitada e a hora em que ela foi iniciada).

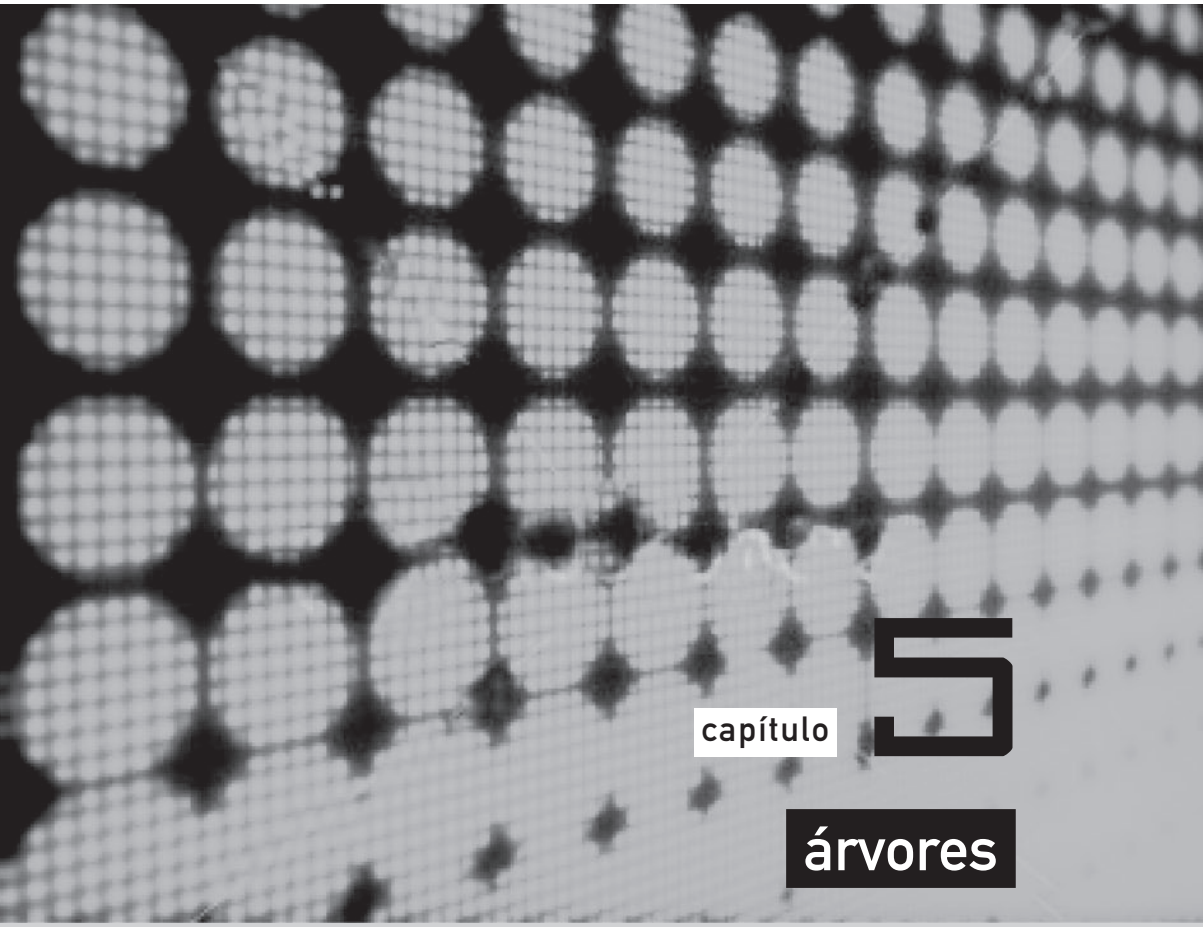
exercício 31 Simule uma agência de banco que possua três caixas atendendo uma fila única de clientes, de forma a determinar o tempo médio de espera do cliente na fila, para cada transação realizada (conforme tabela abaixo). À medida que qualquer um dos caixas fique livre, o primeiro cliente da fila o utiliza. Quando o cliente entra na fila, o horário é anotado. Quando ele sai, verifica-se o tempo que ele aguardou. O tempo que o cliente vai demorar no caixa vai depender da transação a ser realizada. Na simulação, essa transação deverá ser aleatória e escolhida na tabela abaixo. Use um cronômetro para simular o tempo. Quando terminar o expediente (a ser definido pelo usuário e controlado pelo cronômetro), o processo de atendimento é imediatamente interrompido. Além de mostrar o tempo médio de espera do cliente na fila, mostre quantas vezes cada transação foi feita, quantos clientes foram atendidos e quantos clientes não foram atendidos (os que estavam na fila na hora que terminou o expediente).

Transação	Código	Tempo
Saldo	0	10
Saque	1	20
Aplicação	2	30
Extrato semanal	3	40
Extrato mensal	4	50
Pagamento de conta em dinheiro	5	60
Pagamento de conta em cheque	6	70
Pagamento de conta com saque	7	80

exercício 32 Implemente uma aplicação para gerenciar o uso de 3 impressoras que estão ligadas a um computador. Sabe-se que o computador fica enviando partes do documento para a impressora e não o documento inteiro (somente se ele for pequeno). Por exemplo, se quisermos imprimir 5 arquivos (`Arq1`, `Arq2`, `Arq3`, `Arq4` e `Arq5`), o computador pega parte do `Arq1` e envia para a `impressora-01`, parte do `Arq2` e envia para a `impressora-02` e parte

do `Arq3` e envia para a `impressora-03`. Faça isso com os outros arquivos até que eles tenham sido completamente impressos. À medida que um arquivo termine de ser impresso, pegue o próximo arquivo da fila e inicie a sua impressão. Tome cuidado para que um arquivo não comece a ser impresso em uma impressora e termine em outra. Considere que o computador mande 1kb de texto por vez para a impressora e cada impressora gaste 3 segundos para imprimir 1kb de texto. No final, mostre quantos arquivos foram impressos, quanto tempo (em segundos) cada uma das impressoras funcionou, quanto (em kb) cada uma imprimiu e qual o tempo médio de impressão dos arquivos na fila.





capítulo 5

árvores

As árvores são uma das estruturas mais importantes da área de computação, com utilização em muitas aplicações do mundo real. Neste tipo de estrutura, os relacionamentos lógicos entre os dados representam alguma dependência de hierarquia ou composição entre os nodos, formando uma hierarquia de subordinação.

5.1 → conceitos básicos

Antes de iniciar a análise desta estrutura de dados, nesta seção são apresentados alguns conceitos básicos necessários à compreensão do restante do texto.

Uma árvore é geralmente representada graficamente como mostrado na Figura 5.1. As linhas que unem dois nodos representam os relacionamentos lógicos e as dependências de subordinação existentes entre eles. Na figura pode-se observar que o nodo A se relaciona somente com os nodos B, C e D, e não com os demais. Por sua vez, o nodo B se relaciona com o A e também com o E. Intuitivamente, observa-se que os relacionamentos do nodo B com A e com E são diferentes – existe uma hierarquia que faz com que o relacionamento de A para B seja o mesmo que o de B para E, e que o relacionamento de E para B seja o mesmo que o de B para A. A hierarquia de subordinação mostra que um subconjunto dos nodos é subordinado a outro nodo. Por exemplo, o subconjunto de nodos H, I e J está subordinado ao nodo D.

Relacionamentos de subordinação, formando hierarquias, podem apresentar diferentes significados, como:

- hierarquias de especialização, representando classes e subclasses, conforme mostrado na Figura 5.2, na qual um veículo (classe) pode ser espe-

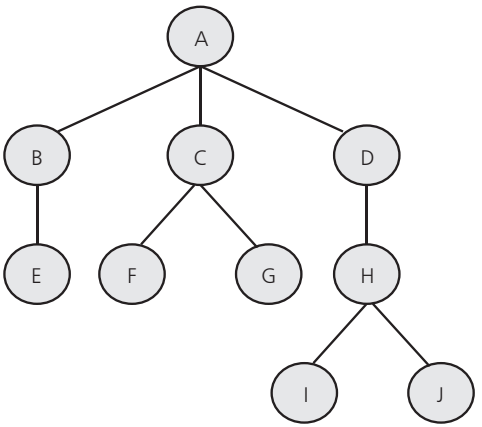


Figura 5.1 Representação gráfica de uma árvore.

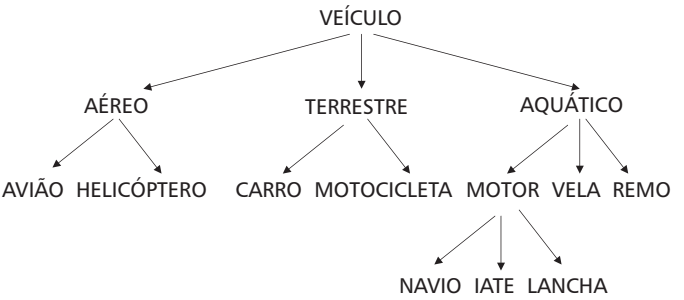


Figura 5.2 Hierarquia de especialização.

cializado em aéreo, terrestre ou aquático (subclasses). Cada uma dessas classes pode, por sua vez, ser especializada em outras categorias;

- hierarquias de composição, conforme mostrado na Figura 5.3. Neste exemplo, o nodo que representa um carro é composto por três partes: chassis, motor e rodas;
- hierarquias de dependência, como na Figura 5.4, que representa o organograma de uma empresa.

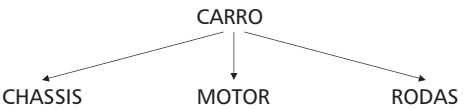


Figura 5.3 Hierarquia de composição.

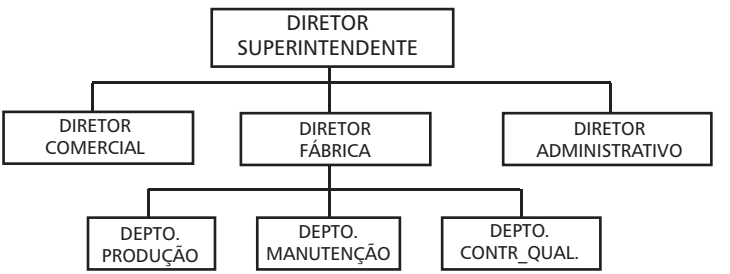


Figura 5.4 Hierarquia de dependência.

Existem outras formas de representar graficamente uma árvore, além daquela apresentada na Figura 5.1. A árvore representada na Figura 5.5a também pode ser representada através das seguintes notações:

`diagramas de inclusão`, onde cada nodo é representado por um círculo, sendo os nodos subordinados representados dentro dos seus hierarquicamente superiores (Figura 5.5b);

`diagramas de barras`, nos quais, de forma semelhante a uma linguagem indentada, os nodos são rotulados e representados por uma linha, com indentação. A posição de cada nodo representa a sua posição na hierarquia (Figura 5.5c);

`representação aninhada`, na qual são utilizados parêntesis para aninhar os nodos subordinados a um determinado nodo (Figura 5.5d);

`numeração por níveis`, que utiliza uma numeração para cada nodo. O nodo inicial de mais alta hierarquia recebe o número `1`. Os demais nodos recebem números seqüenciais, sempre precedidos pelo número do nodo ao qual são subordinados (Figura 5.5e).

5.1.1 terminologia

A terminologia utilizada para referenciar conceitos envolvidos nas estruturas de dados denominadas de árvores não é padronizada, sendo utilizados no-

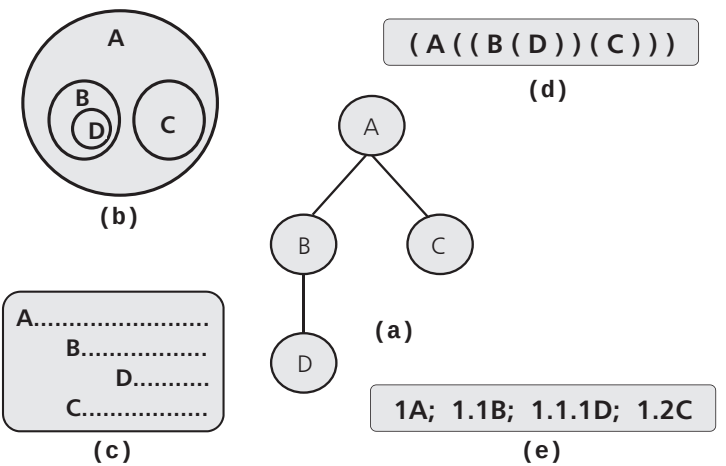


Figura 5.5 Outras formas de representar uma árvore.

mes diferentes para os mesmos conceitos em diferentes publicações. A seguir é apresentada a terminologia empregada neste texto.

Raiz. É um nodo diferenciado, presente em todas as árvores, ao qual são subordinados todos os outros nodos da árvore. O acesso a todos os nodos da árvore é sempre feito a partir de sua raiz. Na Figura 5.6, o nodo denominado de raiz é aquele identificado como A.

Nodos descendentes. São os nodos que apresentam alguma relação de dependência com um nodo mais acima na hierarquia representada pela árvore. Na figura 5.6, os nodos B e C são descendentes diretos da raiz (A), enquanto que o nodo identificado como D é seu descendente indireto.

Uma forma usual de indicar os graus de dependência entre os nodos, provavelmente originária de árvores genealógicas, é o de chamar os descendentes diretos de um nodo de filhos (filhas ou sucessores) deste nodo, e o nodo em questão, de pai (mãe, ascendente ou antecessor) de seus descendentes diretos. Assim, todos os descendentes diretos de um nodo são denominados de irmãos (irmãs) entre si.

Subárvore. É um conjunto de nodos, sendo todos eles subordinados a um único nodo, externo a esta subárvore. Na figura 5.7 são identificadas as seguintes subárvores da raiz A: B – E, C – F – G, e D – H – I – J.

Grau de um nodo. Denomina-se de grau de um nodo ao número de subárvores que são subordinadas diretamente a este nodo, ou seja, à quantidade

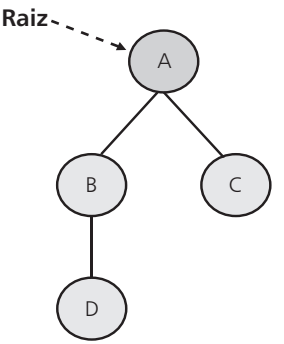


Figura 5.6 Raiz de uma árvore e seus descendentes.

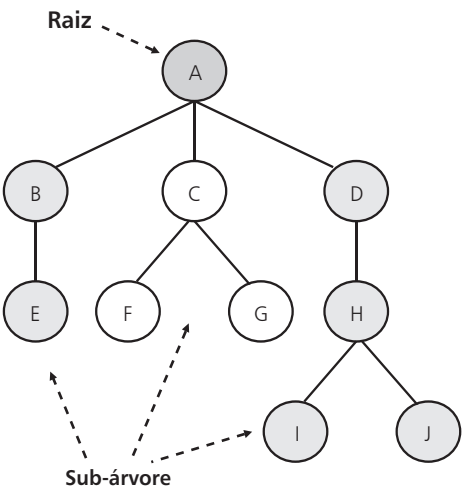


Figura 5.7 Subárvores da raiz A.

de subárvores para as quais este nodo é a raiz. Às vezes é também denominado de `grau de saída do nodo`, representando o número de descendentes que este nodo apresenta. Na árvore da Figura 5.7, por exemplo, a raiz (nodo identificado como `A`) tem grau 3, pois apresenta 3 subárvores. Nesta mesma figura, o nodo identificado com `C` tem grau 2, enquanto que o nodo `F` tem grau zero.

`Grau de uma árvore.` O `grau de uma árvore` é o maior valor dentre os graus de todos os seus nodos. O grau da árvore representada na Figura 5.7 é `3`, uma vez que este é o maior grau de algum nodo desta árvore, no caso, do nodo `A.`

`Folha ou terminal (externo).` Os nodos de grau zero (que não apresentam descendentes) são denominados `folhas` ou `terminais`. Na árvore da Figura 5.7, as folhas são os nodos `E`, `F`, `G`, `I` e `J`.

`Nodo de derivação (interno)`. Os nodos de grau maior do que zero e que, conseqüentemente, apresentam alguma subárvore, são denominados `nodos de derivação` ou `nodos internos`. Na árvore da Figura 5.7, os nodos `A`, `B`, `C`, `D` e `H` são nodos de derivação.

Nível de um nodo. O nível de um nodo corresponde ao número de ligações entre este nodo e a raiz da árvore, acrescido de uma unidade. A raiz de uma árvore tem sempre nível 1. Os nodos ligados diretamente a ela (B, C e D na árvore da Figura 5.7) apresentam nível 2. Os nodos identificados como I e J na Figura 5.7 apresentam nível 4.

Caminho. Um caminho consiste em uma seqüência de nodos consecutivos distintos entre dois nodos. Na Figura 5.7, o caminho entre os nodos A e J é formado pela seqüência de nodos A, D, H e J.

Comprimento do caminho. Dado um determinado caminho entre dois nodos, o comprimento deste caminho é determinado pelo número de níveis entre estes dois nodos, diminuído de uma unidade. O comprimento do caminho entre os nodos A e J da Figura 5.7 é 3.

Altura ou profundidade. A altura de um dado nodo é o número de nodos do maior caminho deste nodo até um de seus descendentes-folha. A altura ou profundidade da árvore é igual ao maior nível de seus nodos. Todos os nodos-folha possuem altura 1. A altura da árvore da Figura 5.7 é 4, indicando que existe um ou mais nodos a uma distância de 3 níveis da raiz.

Floresta. É formada por um conjunto de zero ou mais árvores disjuntas. A Figura 5.8 mostra um exemplo de uma floresta composta por três árvores.

Árvore ordenada. Uma árvore é dita ordenada quando a ordem de suas subárvores é relevante para a aplicação que está sendo representada através desta estrutura de dados. Por exemplo, na Figura 5.9 são apresentadas duas

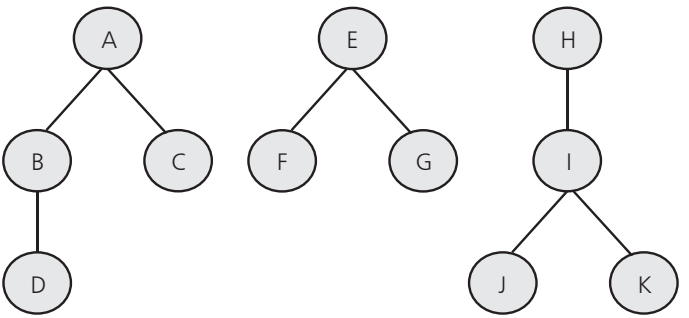


Figura 5.8 Exemplo de floresta.

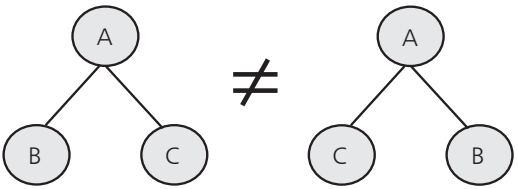


Figura 5.9 Árvores ordenadas.

árvores, cada uma com três nodos. Nas duas, a raiz é identificada por A. Além disso, nas duas árvores a raiz apresenta dois nodos descendentes, com as mesmas identificações (B e C). Caso estas duas árvores sejam "ordenadas", elas são diferentes entre si, pois os dois descendentes estão em ordem diferente. Se a ordem entre os descendentes não é relevante para a aplicação, então as duas árvores são iguais.

Árvore binária / n-ária. Uma árvore é dita binária quando apresentar no máximo grau 2 em cada nodo. De modo semelhante, uma árvore é denominada n-ária quando apresentar no máximo grau "n" em cada nodo.

Árvores isomorfas. Duas árvores não ordenadas são isomorfas quando é possível que se tornem coincidentes através de uma permutação na ordem das subárvores de seus nodos. Por exemplo, caso as duas árvores da Figura 5.9 não sejam ordenadas, elas são isomorfas.

Árvore balanceada. Uma árvore balanceada é aquela na qual existe uma distribuição equilibrada entre os nodos da árvore, ou seja, existe uma diferença mínima entre todas as folhas e a raiz. Uma árvore cheia ou completamente balanceada é aquela em que todas as folhas estão a uma distância igual da raiz. Na Figura 5.8, somente a árvore da esquerda não é balanceada.

5.1.2 operações sobre árvores

Com base na terminologia apresentada, uma árvore é definida formalmente como sendo o conjunto finito T de zero ou mais nodos, podendo ocorrer duas situações:

(1) caso o número de nodos da árvore seja diferente de zero,

- existe sempre um nodo denominado raiz da árvore;

- os demais nodos formam `m > 0` conjuntos disjuntos S_1, S_2,..., S_m, onde cada um destes conjuntos é, por sua vez, uma árvore, sendo as árvores S_i denominadas `subárvores`;

(2) caso o número de nodos da árvore seja zero, a árvore é denominada `vazia`.

Árvores, como quaisquer estruturas de dados, requerem a definição das operações que podem ser realizadas sobre elas. As operações básicas sobre árvores são as seguintes:

`criação de uma árvore` – operação através da qual são alocadas as variáveis necessárias para a definição da árvore, e inicializadas suas variáveis de controle. As demais operações ficam habilitadas somente depois da execução desta operação;
`inserção de um novo nodo em uma determinada posição` – é a maneira de formar a árvore, inserindo os nodos um a um. O novo nodo pode ser inserido como a raiz, como uma folha ou em alguma posição intermediária da árvore;
`exclusão de um nodo` – utilizada para excluir um nodo de uma árvore. Esta operação, quando não se realiza sobre uma folha, implica na reorganização da árvore;
`acesso a um nodo` – para consulta ou atualização dos valores internos de um nodo;
`destruição de uma árvore` – operação que é executada quando uma árvore existente não é mais necessária.

Além das operações básicas para manutenção de uma árvore, podem ser identificadas outras que são muito utilizadas na manipulação desta estrutura de dados devido à sua forma particular de estruturação. Exemplos de outras operações que poderiam ser definidas:

`pai` – dado um determinado nodo de uma árvore, esta operação retorna o endereço do nodo imediatamente superior a ele na hierarquia, usualmente denominado de seu "pai";
`tamanho` – operação que retorna o número total de nodos de uma árvore;
`altura` – retorna a altura da árvore em questão.

Neste capítulo não são detalhados os algoritmos que implementam as operações básicas para árvores, devido à complexidade que apresentam para ár-

vores gerais. A implementação destas operações será vista somente para as árvores binárias, apresentadas no capítulo 6.

Também as árvores podem ser implementadas de duas maneiras: através de contigüidade física na memória, ou utilizando encadeamento. As duas alternativas são analisadas a seguir.

5.2 → árvores implementadas através de contigüidade física

A representação de uma árvore através de contigüidade física não é intuitiva como era no caso das listas lineares. Deve ser estabelecida alguma estratégia para a representação das hierarquias de subordinação entre os diferentes nodos, uma vez que esta não poderá ser representada simplesmente através da organização linear das posições de um arranjo. Além das informações contidas nos nodos, o arranjo que implementar uma árvore deverá conter informações complementares, tais como número de sucessores de cada nodo, para garantir sua fiel representação.

As operações executadas sobre as árvores devem ser implementadas conhecendo e respeitando a estratégia de representação estabelecida na representação física. Esta estratégia deve também ser conhecida pela aplicação que irá realizar operações sobre a árvore. Tratando-se de árvores, é importante que se possa acessar, através de operações, os descendentes diretos de qualquer nodo, o que pode não ser trivial tratando-se de contigüidade física – os descendentes diretos podem estar bastante afastados fisicamente de seu antecessor.

As operações básicas sobre árvores implementadas através de contigüidade física são usualmente realizadas da seguinte forma:

`inserção ou remoção de um nodo` – estas duas operações implicam quase que via de regra em deslocamento de informações ao longo do arranjo utilizado para implementar a árvore. Na inserção, uma vez localizada a posição onde o novo nodo deve ser inserido, os nodos representados a partir desta posição devem ser avançados de uma posição, para dar lugar ao novo nodo. Na remoção de um nodo, os nodos seguintes devem ser deslocados para ocupar a posição que ficou vaga. Além disso, as informações complementares contidas nos nodos deverão ser, a cada vez, atualizadas;

`acesso a um nodo` – deve ser feito respeitando a hierarquia da árvore, percorrendo-a de acordo com a estratégia utilizada na representação, com auxílio das informações complementares armazenadas para cada nodo.

Nas subseções a seguir são apresentadas duas estratégias diferentes para implementação de árvores através de contigüidade física, lembrando não serem estas as únicas formas para este tipo de implementação.

5.2.1 implementação por níveis

A implementação por níveis inicia com a representação da raiz da árvore na primeira posição a ser ocupada na memória. Nas posições seguintes são armazenados todos os sucessores diretos da raiz, em alguma ordem pré-estabelecida. Em seguida são representados os sucessores de cada um dos nodos anteriores, de acordo com a mesma ordem utilizada no nível anterior, e assim por diante.

A Figura 5.10b mostra a implementação por níveis da árvore da Figura 5.10a. A primeira posição traz o nodo raiz, representado por A. Para que a representação seja fiel é necessário informar, em cada nodo, o número de descendentes que ele apresenta. Esta informação pode, por exemplo, ser armazenada

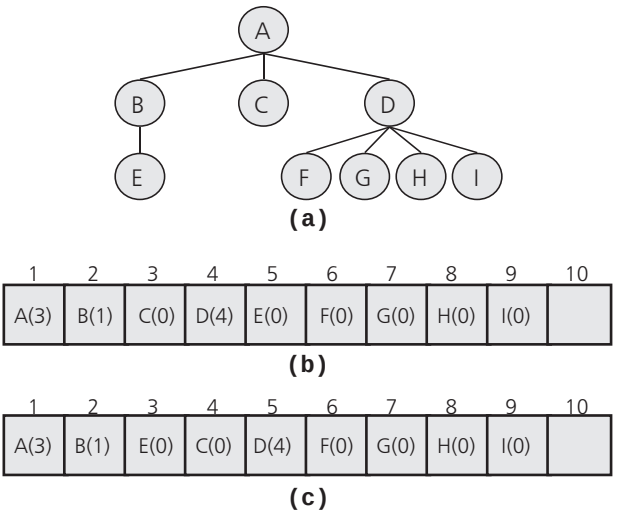


Figura 5.10 Árvores implementadas através de contigüidade física.

em algum campo especial junto às informações correspondentes ao nodo. Assim, no exemplo mostrado deve ser armazenado, além das informações correspondentes ao nodo raiz A, o seu número de descendentes (3). Logo após a representação do nodo raiz são representados os seus descendentes diretos, em alguma ordem pré-definida (no exemplo, da esquerda para a direita), sendo a cada um associado o seu número de descendentes. Assim, na segunda posição é representado o nodo B com um descendente, seguindo os nodos C (que não tem descendente) e D (com quatro descendentes). A seguir são representados os descendentes dos nodos B, C e D, nesta ordem, e assim por diante, até que todos os nodos sejam representados.

5.2.2 implementação por profundidade

Esta outra forma de implementação através de contigüidade física inicia também pela representação da raiz na primeira posição a ser ocupada na memória. Em seguida é representado o primeiro sucessor da raiz, seguido de todos os seus sucessores. Depois é representado o segundo sucessor da raiz, seguido de seus respectivos sucessores. Isto é repetido até que sejam representados todos os sucessores da raiz.

A Figura 5.10c apresenta esta forma de implementação para a mesma árvore representada na Figura 5.10a. Observa-se que a representação desta árvore nesse caso é diferente: iniciando pelo nodo raiz A, em seguida é representado o seu primeiro descendente B. A seguir é representado o descendente do nodo B, que é o nodo E. Esgotados os descendentes de B, é representado o segundo descendente da raiz, o nodo C, que não apresenta descendentes. Segue o terceiro descendente da raiz, D, seguido de seus descendentes, F, G, H e I. A cada um dos nodos deve ser associado o seu número de descendentes, para que a representação da árvore seja absolutamente fiel.

5.2.3 vantagens e desvantagens da implementação por contigüidade física

A implementação de árvores por contigüidade física não constitui, em geral, uma boa solução para a representação física de árvores. A razão principal é a dificuldade que geralmente se tem de seguir a hierarquia implícita nestas estruturas ao manipular a árvore. Além disso, a inserção e remoção de nodos é demorada, implicando quase sempre em completas reorganizações da estrutura. Entretanto, esta forma de representação geralmente é eficiente em

termos de espaço ocupado, principalmente quando o grau dos nodos não varia muito.

A implementação fica simplificada caso exista alguma limitação no número de descendentes de cada nodo, como no caso de árvores binárias ou ternárias. Por exemplo, considerando uma árvore ternária, como cada nodo só pode apresentar três descendentes, não é necessário armazenar o número de descendentes diretos de cada nodo – são reservadas, no arranjo, três posições para os descendentes de cada nodo. Deve ser utilizada alguma convenção para informar se algum descendente não existe. Geralmente será bastante alto o número de elementos que informam somente que o nodo anterior não apresenta descendente direto. Além disso, a forma de identificar os descendentes diretos de um nodo é complexa, pois estes poderão estar muito afastados de seu antecessor. A Figura 5.11 apresenta uma árvore ternária e uma possível representação por níveis desta árvore, utilizando o símbolo λ para representar a inexistência de um descendente. Esta representação está considerando a ordenação da árvore ao representar os descendentes: por exemplo, considerando o nodo B, a posição 5 do arranjo informa que B não apresenta descendente à esquerda; a posição 6 contém o descendente central; e a posição 7 informa que este nodo não tem descendente à direita.

Essa forma de implementação é apropriada em dois casos:

quando os nodos são processados exatamente na ordem em que foram armazenados, e não seguindo a hierarquia representada pela árvore; e

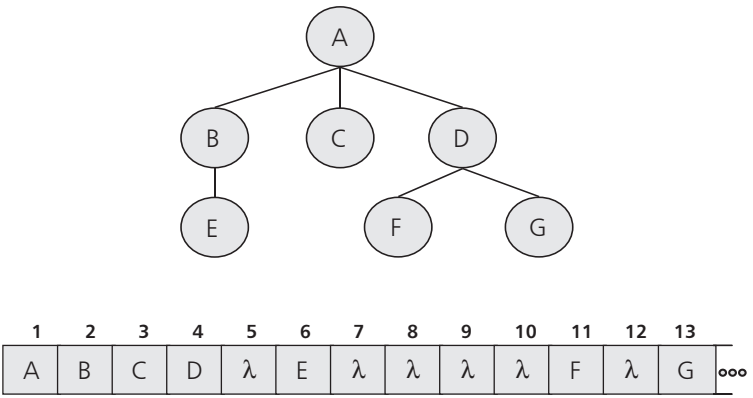


Figura 5.11 Árvore ternária implementada através de contigüidade física.

quando, excetuando a raiz, os demais nodos têm graus iguais ou muito semelhantes. Neste caso, a dificuldade de acesso à hierarquia é um pouco minorada pela simetria da árvore, pois os algoritmos que implementam as operações podem encontrar, com um pouco mais de facilidade, os descendentes de cada nodo. Além disso, não haverá desperdício de espaço ocupado para representar informações relativas à inexistência de nodos.

Esta forma de implementação constitui uma alternativa importante para o arquivamento permanente de árvores. Neste caso, a estrutura representada por encadeamento durante a manipulação é convertida para contigüidade física ao ser armazenada. A operação inversa é feita quando a estrutura é recuperada para manipulação.

5.3 → árvores implementadas por encadeamento

Nesta forma de implementação, cada nodo da árvore é representado por uma variável que deve apresentar, além das informações relativas a este nodo, os endereços dos nodos que são seus descendentes diretos. O acesso aos nodos de uma árvore implementada por encadeamento se dá sempre através do endereço da raiz, sendo os demais nodos alcançados somente através dos endereços contidos nos campos de elo de cada nodo. Deste modo, a hierarquia de subordinação, implícita nas árvores, fica perfeitamente representada.

Na implementação física, todos os nodos da árvore devem apresentar a mesma estrutura. Por isso, ao definir a estrutura para implementar uma determinada árvore por encadeamento deve ser reservado um espaço para armazenar

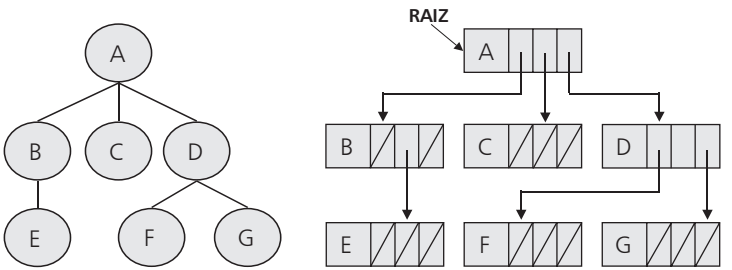


Figura 5.12 Árvore implementada através de encadeamento.

tantos elos para descendentes quanto o valor do grau máximo que aparece na árvore, ou seja, cada nodo terá tantos campos de elo quanto o maior número de descendentes presente em algum nodo da árvore.

A Figura 5.12 representa a implementação através de encadeamento da árvore apresentada na Figura 5.11, lembrando que a árvore é ordenada por níveis. A estrutura escolhida para armazenar cada nodo prevê a possibilidade de qualquer nodo ter até três descendentes (número de descendentes da raiz), visto que apresenta três campos de elo. O acesso a esta estrutura é feito através do endereço `RAIZ`. Nota-se que neste exemplo todos os campos estão sendo utilizados somente na representação da raiz. Nos demais nodos, que apresentam menor número de descendentes (ou nenhum, como é o caso das folhas da árvore), vários campos de elo ficam ociosos. Este problema pode ser significativo caso algum nodo apresente grau bem mais elevado que os demais.

5.3.1 operações básicas

As operações básicas sobre árvores, quando implementadas através de encadeamento, são realizadas da seguinte forma:

`criação da árvore` – consiste somente em inicializar no endereço nulo o ponteiro que vai trazer o endereço da raiz da árvore;

`inserção de um novo nodo` – o novo nodo é geralmente inserido como uma folha. Depois de alocado o novo nodo, seu endereço é colocado em um dos campos de elo do nodo que será seu pai (encadeamento). Caso o nodo inserido seja o primeiro da árvore, ou seja, sua raiz, não haverá encadeamento com antecessor e seu endereço será o de acesso a toda a árvore;

`remoção de um nodo` – o nodo é liberado, e no campo de elo de seu pai é colocado o endereço nulo. Esta operação é simples se o nodo a remover for uma folha. Caso seja um nodo que possui descendentes, alguma estratégia deve ser definida (remover também todos os descendentes, ou reorganizar a árvore);

`acesso a um nodo` – a árvore é sempre acessada através da raiz, devendo ser percorrida através dos campos de elo, até que o nodo desejado seja encontrado. Este percurso deve ser feito com alguma disciplina, uma vez que o acesso é sempre do pai para o filho (não é possível, a partir de um nodo filho, acessar seu pai). Durante o percurso muitas vezes é necessário,

cada vez que se vai acessar um filho, guardar o endereço do pai para, posteriormente, poder acessar seus outros filhos;

`destruição da árvore` – a árvore é percorrida a partir de sua raiz, sendo todos os nodos liberados.

5.3.2 vantagens e desvantagens da implementação por encadeamento

Constata-se que a implementação de árvores por encadeamento é bastante intuitiva. Por este motivo, no restante deste texto somente será considerada esta forma de implementação para árvores.

Árvores cujos nodos têm grau variado apresentam geralmente muitos campos de elo ociosos. Este problema pode ser significativo caso algum nodo apresente grau bem mais elevado que os demais.

O acesso às informações na árvore pode ser dificultado devido à necessidade de acessar qualquer nodo sempre através da raiz. Este problema é minimizado pela definição de disciplinas adequadas para percorrer os nodos da árvore. Por outro lado, as operações de inserção e remoção de nodos são muito simplificadas, consistindo basicamente na atualização de endereços nos campos de elo de alguns nodos.

Devido à sua poderosa capacidade de representar dados de aplicações, diversos tipos particulares de árvores foram desenvolvidos. No capítulo que segue é analisado um tipo específico de árvore, denominado `árvore binária`, muito utilizado na prática. Para estas são vistos os algoritmos das operações básicas de manipulação, considerando sempre a implementação através de encadeamento.

5.4 → exercícios

exercícios com conceitos de árvores

exercício 1 Considere a árvore a seguir, representada através de parênteses aninhados:

```
( A (B) ( C (D (G) (H)) (E) (F (I)) ) )
```

Represente esta mesma árvore através de:

- diagrama de inclusão;
- representação hierárquica;
- numeração por níveis.

exercício 2 Para a mesma árvore do exercício anterior, responda às perguntas a seguir.

- Quantas subárvores esta árvore contém?
- Quais os nodos-folha?
- Qual o grau de cada nodo?
- Qual o grau da árvore?
- Liste os ancestrais dos nodos B, G e I.
- Identifique as relações de parentesco entre os nodos.
- Liste os nodos de quem C é ancestral próprio.
- Liste os nodos de quem D é descendente próprio.
- Dê o nível e a altura do nodo F.
- Dê o nível e a altura do nodo A.
- Qual a altura da árvore?

exercício 3 Para uma árvore de grau "d", determine:

- o grau dos nodos internos da árvore;
- o grau dos nodos-folhas;
- o número de nodos da árvore caso o grau seja "d" e a altura "h";
- a altura da árvore, se o grau for "d" e o número de nodos "n".

exercícios com árvores implementadas sobre arranjos

exercício 4 Considere uma árvore cheia, de grau `x`.

- Para alocação seqüencial, por níveis, se um nodo estiver armazenado na posição `i` de um arranjo, em quais posições estarão seus `x` filhos?
- Considerando esta alocação seqüencial, quais as conseqüências de inserções e eliminações na árvore?

exercício 5 Considere uma árvore armazenada em um arranjo denominado `ARVORE`. Cada elemento do arranjo é um registro com dois campos: `INFO`

(com a informação contida no nodo) e DESC (com o número de descendentes deste nodo). Considere que não existe limitação do número de descendentes que um nodo pode apresentar. Escolha uma estratégia de armazenamento e, com base nesta estratégia, escreva algoritmos para:

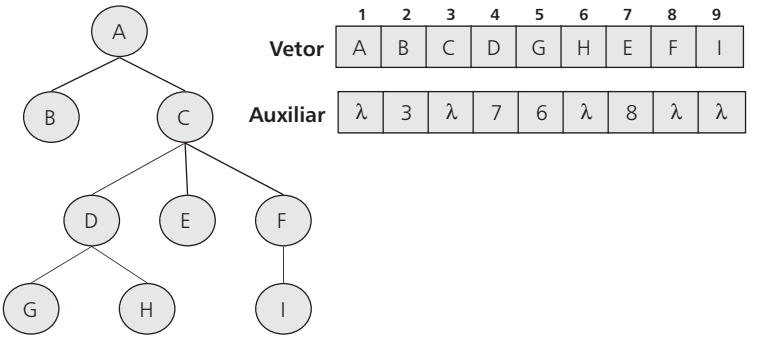
- acrescentar um novo nodo como descendente direto de um determinado nodo da árvore;
- remover um determinado nodo da árvore, identificado pelo conteúdo de seu campo de informação;
- procurar um determinado nodo através de seu campo de informação, retornando o número de descendentes que apresenta;
- informar os conteúdos dos campos de informação dos descendentes de um determinado nodo;
- alterar o campo de informação de um determinado nodo.

exercício 6 Considere uma árvore ternária implementada sobre um arranjo, usando a estratégia da implementação por níveis. Escreva algoritmos para executar as operações básicas sobre esta árvore (inserir um nodo, remover um nodo e acessar um determinado nodo). Repita este exercício considerando a implementação por profundidade.

exercício 7 Para a mesma árvore ternária do exercício anterior, ainda implementada sobre um arranjo, escreva uma função que devolva o número de nodos que a árvore apresenta.

exercício 8 Tendo como entrada uma árvore ternária (Avr), implementada por níveis conforme ilustrada na figura 5.11, e um valor (V), escrever um algoritmo que localize esse valor na árvore. Se encontrado, calcule o número de nodos das subárvores de V, recursivamente para cada subárvore a partir de V.

exercício 9 Considere uma estrutura de armazenamento para árvores composta dos dois arranjos mostrados na figura abaixo. Um deles (Vetor) contém os nodos, armazenados em uma determinada ordem. O outro, um arranjo auxiliar, indica, para cada nodo V, o índice do nodo que segue imediatamente à subárvore da raiz V, se existir. Caso não exista, essa informação é λ. Para a árvore representada na figura a seguir, a estrutura de armazenamento seria a seguinte:



Especifique uma função para determinar os níveis dos nodos da árvore.

exercício 10 As informações relativas a uma determinada aplicação estão armazenadas em uma estrutura de árvore qualquer, utilizando para isto um arranjo A de uma dimensão, de 50 elementos. Cada elemento deste arranjo representa um nodo da árvore, sendo composto por um registro com 3 campos – o nome do item considerado, seu valor, e o número de descendentes deste item:

```
Tipo T = registro
            Nome: string
            Valor: real
            NrDesc: integer
         fim registro
```

Com base nesta estrutura:

- escreva um algoritmo que recebe este arranjo e um nome, e que imprime os nomes e valores dos descendentes do nodo que contém o nome recebido. Caso o nome não seja encontrado em algum nodo da árvore, o algoritmo deverá escrever uma mensagem;
- escreva um algoritmo para inserir um novo nodo na árvore. Este algoritmo deve apresentar quatro parâmetros: o arranjo A que armazena a árvore, o nome `NOVO_NOME` e o valor `NOVO_VALOR` do nodo que deve ser incluído na árvore, e o valor `VALOR_PAI` correspondente ao nodo que será o antecessor (pai) do novo nodo. Considere que sempre haverá espaço para alocar um novo nodo.

exercícios com árvores implementadas através de encadeamento

exercício 11 A partir dos dois arranjos utilizados na representação de uma árvore no exercício anterior (exercício 9), escreva um algoritmo que gere a mesma árvore implementada através de encadeamento. Suponha que cada nodo pode ter no máximo quatro descendentes.

exercício 12 Considerando uma árvore que pode ter no máximo quatro descendentes por nodo, implementada por encadeamento, defina um tipo para seus nodos e escolha uma forma de organizar os nodos (por níveis ou por profundidade). Depois, escreva um algoritmo que recebe o endereço da raiz desta árvore e que imprime todos os valores contidos em cada nodo, seguidos dos valores de seus descendentes.

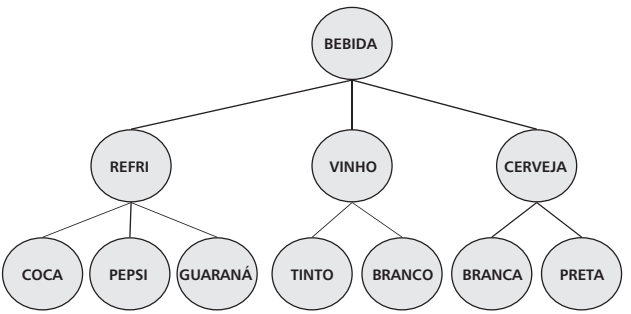
exercício 13 Considerando a árvore definida no exercício anterior, escreva um algoritmo que retorne a quantidade de nodos que tenham somente um filho.

exercício 14 Construa um algoritmo que crie uma cópia de uma árvore com número variável de filhos.

exercício 15 Escreva uma função que recebe o endereço da raiz de uma árvore armazenada sore um arranjo e um valor de código e retorna, se o código existir, uma lista encadeada contendo a cópia de todos os nodos dos filhos do nodo que contem o código, ou o endereço nulo caso esse código não exista na árvore.

exercícios de aplicações

exercício 16 Considere a árvore ternária abaixo, que apresenta categorias de bebidas:



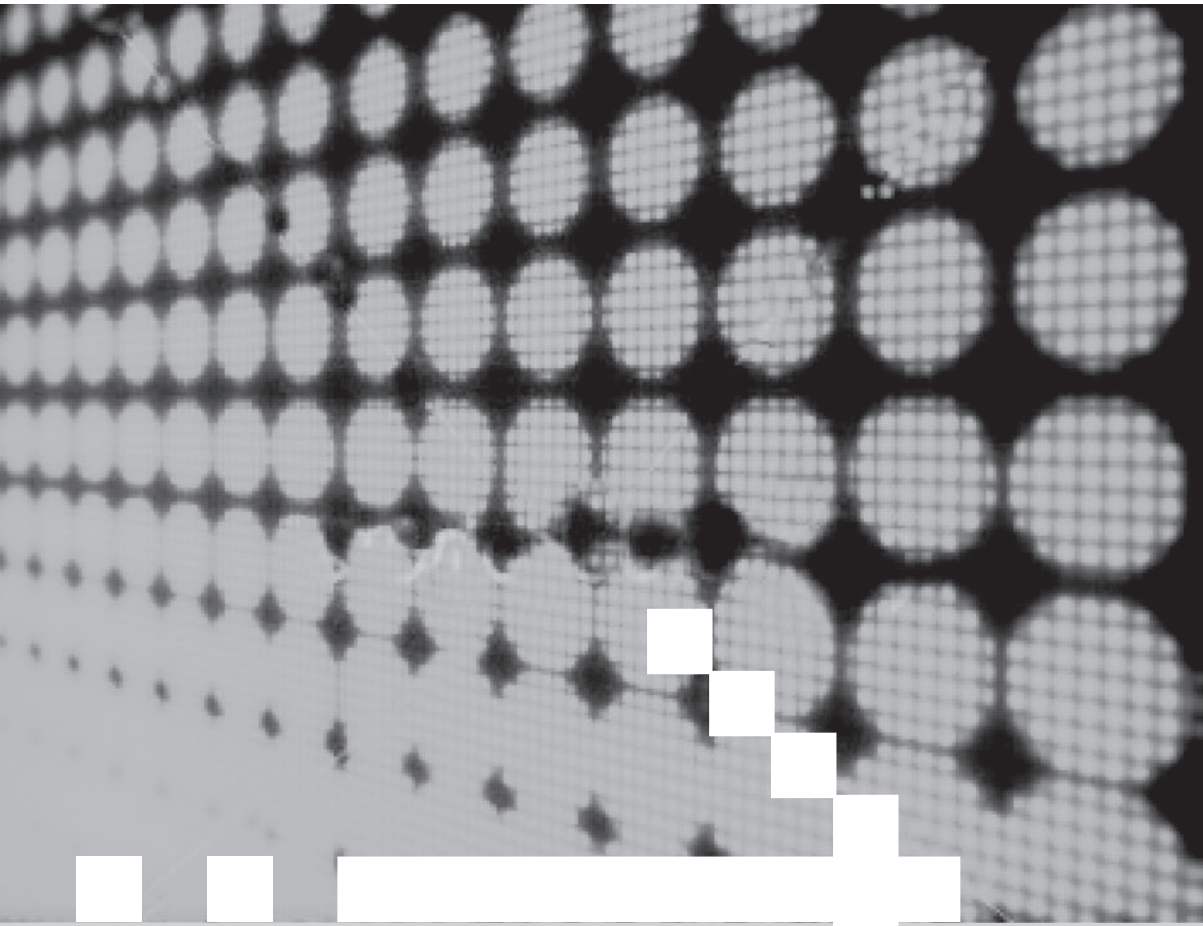
Em cada nodo, além do nome da bebida, deve ser armazenado seu preço e o número de unidades disponíveis para venda em um estabelecimento comercial que utiliza esta estrutura.

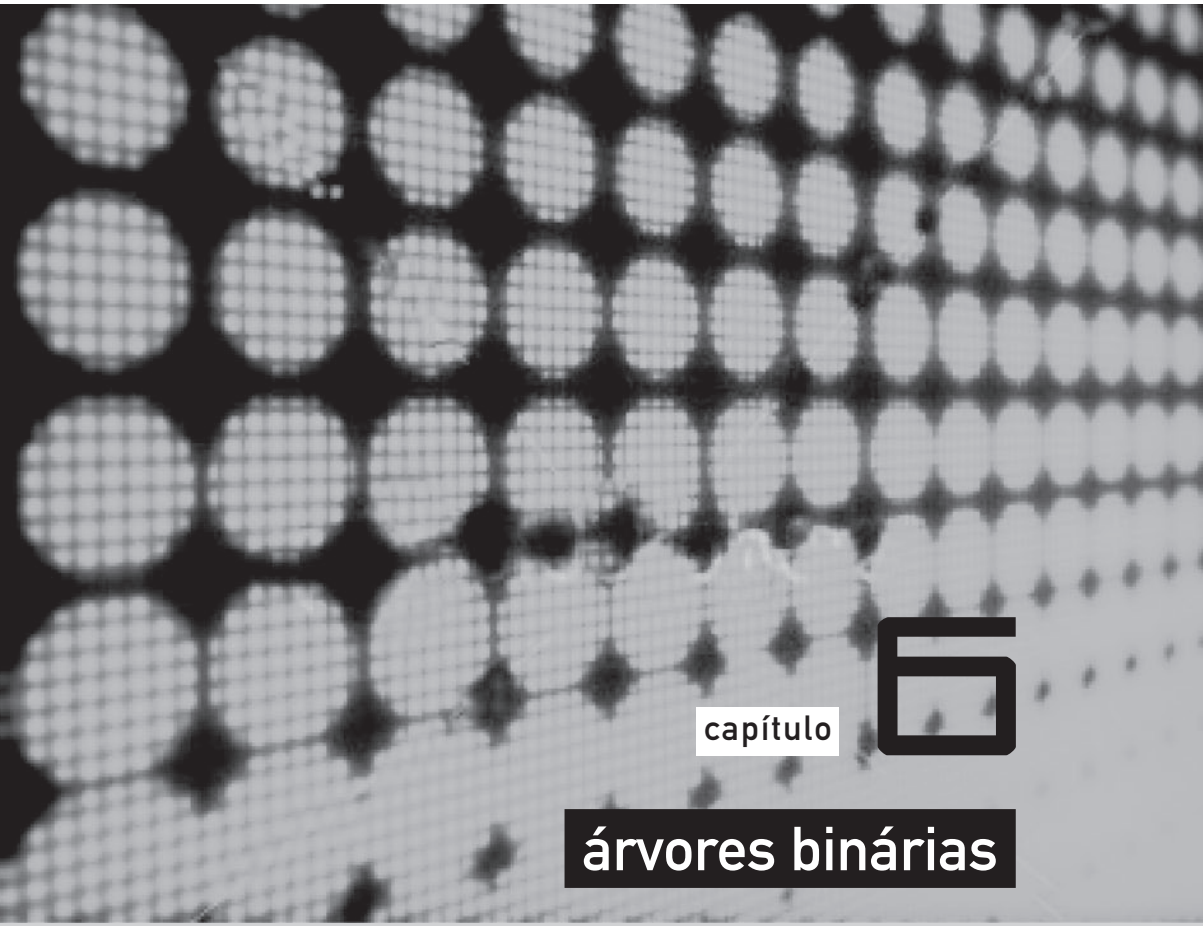
- Implemente esta árvore sobre um arranjo, por níveis.
- Escreva um procedimento que liste o nome e o preço do tipo de bebida fornecido como parâmetro (refrigerante, vinho ou cerveja).
- Escreva uma função que informe o número de unidades disponíveis para venda de uma determinada bebida, fornecida como parâmetro.

exercício 17 Repita o exercício anterior, implementando a árvore através de encadeamento.

exercício 18 Escreva um programa que permita montar e exibir a árvore genealógica de uma pessoa.

exercício 19 As informações relativas a uma determinada aplicação estão armazenadas em uma estrutura de árvore qualquer, como a da Figura 5.12. Mostre uma possível implementação desta árvore através de alocação seqüencial (utilizando arranjos) e outra através de encadeamento (com ponteiros). Defina os tipos de variáveis que seriam utilizados nas duas implementações.





capítulo 6

árvores binárias

Uma árvore binária apresenta o grau de cada nodo sempre menor ou igual a 2 (ou seja, cada nodo apresenta no máximo dois descendentes), podendo ou não ser ordenada. No caso de ser ordenada, as subárvores de cada nodo são identificadas por sua posição, sendo uma denominada de subárvore esquerda e a outra de subárvore direita.

A Figura 6.1 apresenta três tipos especiais de árvores binárias:

`árvore estritamente binária` (Figura 6.1a) – toda a árvore em que cada nodo tem grau `0` ou `2`, ou seja, quando todo nodo apresenta `0` ou `2` filhos;

`árvore binária completa` (Figura 6.1b) – é uma árvore estritamente binária na qual todo nodo que apresente alguma subárvore vazia está localizado no último (folha) ou no penúltimo nível desta árvore;

`árvore binária cheia` (Figura 6.1c) – quando todas as folhas estão à mesma profundidade, e todos os nodos internos têm grau `2`. Uma árvore cheia é completa e estritamente binária.

A limitação do número de descendentes faz com que a implementação de uma árvore binária se torne bastante simples. Na implementação por contigüidade física não é necessário representar em cada nodo o seu número de descendentes, bastando reservar sempre espaço para dois descendentes. É necessário adotar alguma convenção para indicar os descendentes que não existem – por exemplo, algum valor que não faça parte do conjunto de valores armazenado na árvore. A Figura 6.2 apresenta um exemplo da implementação de uma árvore binária (Figura 6.2a) sobre um arranjo (Figura 6.2b), utilizando a representação por níveis (Seção 5.2.1). É utilizado o símbolo "λ"

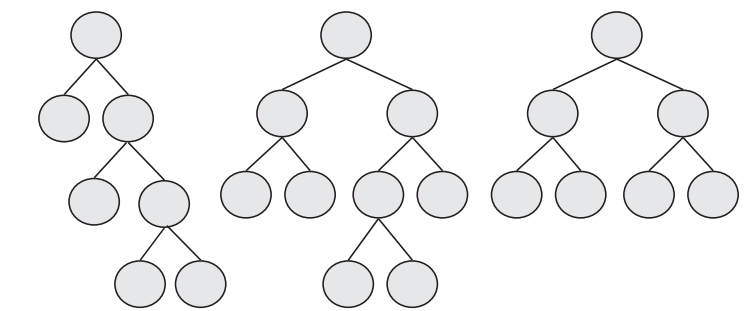


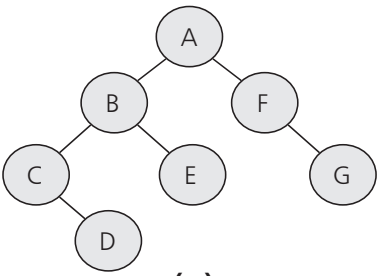
Figura 6.1 Tipos de árvores binárias.

para representar a inexistência de um determinado descendente, como no caso do elemento de índice 6 do arranjo, onde seria representado o descendente à esquerda do nodo F.

Na implementação de árvores binárias por encadeamento, a estrutura de um nodo apresenta sempre dois campos de elo: um para o descendente à esquerda, e o outro para o da direita. A Figura 6.3 mostra a implementação da árvore da Figura 6.2a através de encadeamento.

Na análise das operações básicas a serem aplicadas em árvores binárias, feita a partir da Seção 6.2, será considerada somente a implementação através de encadeamento, por ser esta a forma mais intuitiva para este tipo de estrutura de dados.

Antes de analisar as operações sobre árvores binárias, na próxima seção é apresentada a forma de transformar uma árvore n-ária em uma binária equivalente, permitindo que, para qualquer tipo de árvore, sejam utilizadas as operações que seguem.



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	F	C	E	λ	G	λ	D	λ	λ	λ	λ

(b)

Figura 6.2 Árvore binária implementada através de contigüidade física.

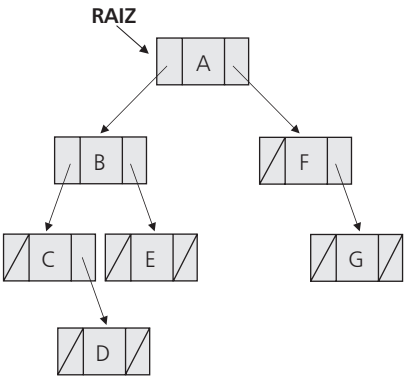


Figura 6.3 Árvore binária implementada através de encadeamento.

6.1 ⇢ transformação de árvore n-ária em binária

Mesmo que uma aplicação necessite estruturar seus dados através de uma árvore de grau mais elevado, é sempre possível transformar esta representação em uma árvore binária equivalente. O objetivo desta transformação é, além da facilidade de representação da árvore binária, possibilitar o uso dos algoritmos correspondentes às operações sobre árvores binárias, já desenvolvidos e muito utilizados. A árvore binária gerada terá sempre o mesmo número de nodos da árvore original. O processo de transformação é mostrado a seguir, passo a passo, tomando como base a árvore da Figura 6.4:

a raiz das duas árvores (da árvore n-ária que se quer transformar, e da binária equivalente) é a mesma (nodo `A`);
o primeiro descendente direto da raiz na árvore n-ária passa a ser o descendente à esquerda da raiz na árvore binária (`B`);
o segundo descendente da raiz (`C`), irmão de `B`, passa a ser o descendente à direita do anterior (`B`);
os demais descendentes da raiz passam a ser descendentes à direita sempre do irmão anterior. No exemplo, `D` passa a ser o descendente à direita de `C` e, caso houvesse outro irmão de `B`, este seria o descendente à direita de `D`;
o processo executado para a raiz é repetido para cada nodo da árvore original.

Resumindo, na árvore binária equivalente cada nodo apresentará seu primeiro filho como descendente à esquerda, e o irmão seguinte como descendente à

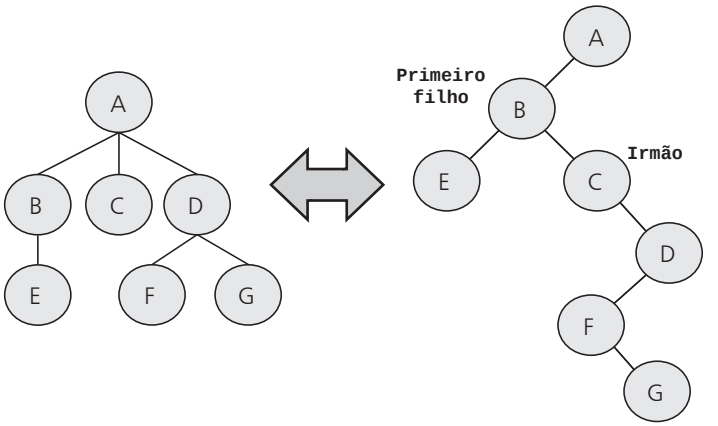


Figura 6.4 Representação de uma árvore n-ária através de uma binária.

direita. Caso a árvore original não seja ordenada, não é relevante a ordem em que são considerados os nodos de um mesmo nível durante a transformação.

Também é possível representar uma floresta de árvores n-árias através de uma única árvore binária. Para isso, o seguinte processo deve ser seguido:

considerar as raízes de todas as árvores da floresta como nodos irmãos (escolhendo uma ordem aleatoriamente);
para cada uma das árvores da floresta, efetuar a transformação antes apresentada, representando-a como uma árvore binária a partir da raiz já inserida na árvore resultante desta transformação.

A Figura 6.5 mostra como duas árvores n-árias podem ser representadas através de uma única árvore binária.

Cabe ressaltar que a árvore binária equivalente deve ser interpretada adequadamente, lembrando o significado dos descendentes. Uma vez executados os processamentos requeridos na árvore equivalente, deverá ser possível voltar à representação original da árvore. O processo de reconversão da árvore binária na n-ária correspondente é mostrado a seguir, passo a passo, tomando como base novamente as árvores da Figura 6.4:

a raiz das duas árvores (da árvore binária que se quer transformar, e da n-ária equivalente) é a mesma (nodo A);

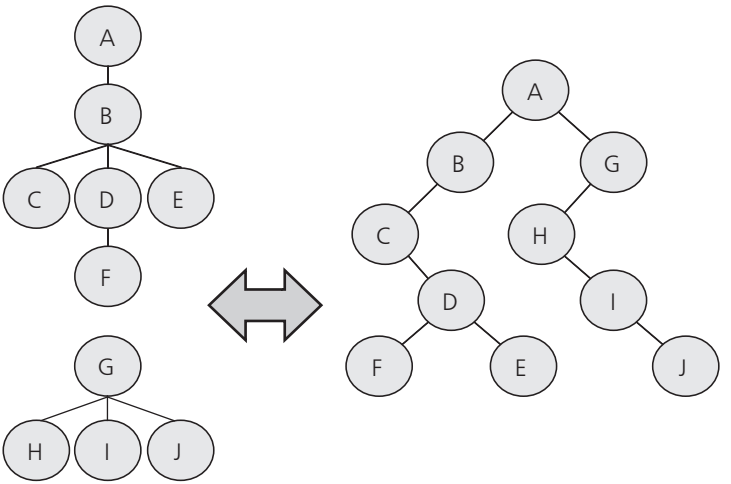


Figura 6.5 Representação de uma floresta de árvores n-árias através de uma binária.

o descendente à esquerda da raiz na árvore binária passa a ser o descendente à esquerda da raiz na árvore n-ária;
o descendente à direta da raiz na árvore binária passa a ser irmão à direita do seu nodo pai na árvore n-ária. No exemplo, o nodo C passa a ser irmão à direita do nodo B e o nodo D passa a ser irmão à direita do nodo C;
o processo executado para a raiz é repetido para cada nodo da árvore original.

No caso em que a raiz da árvore binária possui uma subárvore à direita, esta subárvore será a raiz de uma nova árvore n-ária, resultando em uma floresta.

6.2 → operações sobre árvores binárias

Embora a implementação de árvores binárias possa ser realizada através de contigüidade física, somente a implementação através de encadeamento será considerada neste texto, devido à natureza intuitiva que apresenta para este tipo de estrutura de dados.

Nos algoritmos das operações para árvores binárias serão considerados os seguintes tipos de dados:

```
TipoPtNodo = ↑TipoNodo
TipoNodo = registro
             Info: TipoInfo
             Esq, Dir: TipoPtNodo
           fim registro
TipoArvore = TipoPtNodo
```

6.2.1 criação de uma árvore vazia

Criar uma árvore binária implementada por encadeamento significa tão somente inicializar o ponteiro que irá guardar o endereço de seu nodo raiz. O valor deste ponteiro é inicializado no endereço nulo, indicando que esta árvore está vazia, ou seja, que sua raiz ainda não foi alocada. O nodo raiz somente será alocado fisicamente no momento em que for inserido o primeiro nodo na árvore. O procedimento a seguir implementa esta operação.

```
Algoritmo 6.1 – CriarArvore
   Entradas: –
   Saída: Arv (TipoArvore)
início
    Arv ← nulo
fim
```

6.2.2 inserção de um novo nodo

O primeiro nodo a ser inserido em qualquer árvore, seja ela binária ou não, é aquele que representará a sua raiz. Todas as demais inserções de nodos serão feitas como descendentes de algum nodo já definido.

O processo de inserção deve levar em consideração se a árvore binária é ou não ordenada. Caso não seja ordenada, a inserção somente vai depender da limitação de descendentes do nodo-pai. No caso de ser ordenada, a aplicação deve informar a posição em que o novo nodo deve ser inserido, se como descendente à esquerda ou à direita. A inserção somente será realizada se este descendente específico ainda não estiver presente. Os algoritmos que serão apresentados neste capítulo consideram **árvores binárias ordenadas**.

As operações de inserção da raiz e dos demais nodos são executadas de forma diferente, analisadas separadamente, a seguir.

Para inserir o nodo raiz é necessário que a árvore tenha sido inicializada, e que esteja vazia. A operação aloca o novo nodo, preenche seu campo de informação com o conteúdo fornecido pela aplicação, e informa a esta o endereço do nodo alocado. Toda a manipulação posterior desta árvore será feita através do endereço da raiz.

O algoritmo apresentado a seguir, após alocar o novo nodo, inicializa os ponteiros para seus descendentes no endereço nulo, preenche o campo de informação com o conteúdo recebido (`Valor`) e devolve ao programa de aplicação o endereço do nodo alocado (`Arv`).

```
Algoritmo 6.2 – AlocarRaiz
   Entrada: Valor (TipoInfo)
   Saída: Arv (TipoArvore)
   Variável auxiliar: PtRaiz (TipoPtNodo)
início
  alocar(PtRaiz)
  PtRaiz↑.Esq ← PtRaiz↑.Dir ← nulo
  PtRaiz↑.Info ← Valor
  Arv ← PtRaiz
fim
```

Uma vez tendo sido definida a raiz, um novo nodo pode ser inserido como descendente à esquerda ou à direita de outro nodo já existente (nodo-pai). Para tal, é necessário inicialmente identificar o endereço do nodo-pai. Isto pode ser feito percorrendo a árvore à procura de alguma informação contida no campo de informação dos nodos, ou recebendo diretamente da aplicação o endereço buscado.

A inserção do novo nodo somente poderá ser realizada caso o nodo identificado como nodo-pai ainda não possua o descendente específico – isto é, caso seja solicitado inserir um descendente à esquerda, o nodo–pai não poderá apresentar descendente à esquerda. A inserção é simples: basta alocar o novo nodo e encadeá-lo a seu pai. Ressaltamos que está sendo considerada somente a possibilidade de alocar nodos-folha.

Como exemplo desta operação, o algoritmo a seguir insere um filho à esquerda de um determinado nodo, identificado pelo conteúdo de seu campo de informação (`ValorPai`). São também fornecidos o endereço da raiz da árvore (`Arv`) e a informação a ser inserida no novo nodo (`ValorFilho`). O parâmetro

Sucesso retornará falso no caso do nodo-pai não ser encontrado, ou quando este já apresentar um descendente à esquerda.

O algoritmo utiliza uma função denominada Localizar que, dado o endereço da raiz de uma árvore (Arv), devolve o endereço do nodo cujo campo de informação é igual a ValorPai. Caso este nodo não seja encontrado, a função devolve o endereço nulo. A função Localizar será detalhada mais adiante (Algoritmo 6.6). Os algoritmos para percorrer uma árvore binária para procurar um determinado nodo serão apresentados na Seção 6.2.4.1.

```
Algoritmo 6.3 – InserirFilhoEsq
   Entradas: Arv (TipoArvore)
             ValorPai, ValorFilho (TipoInfo)
   Saídas: Arv (TipoArvore)
           Sucesso (lógico)
   Variáveis auxiliares: Pai, Novo (TipoPtNodo)
início
    Sucesso ← falso
    Pai ← Localizar(Arv, ValorPai)
    se (Pai ≠ nulo) e (Pai↑.Esq = nulo)
    então início
          Sucesso ← verdadeiro
          alocar(Novo)
          Novo↑.Dir ← Novo↑.Esq ← nulo
          Novo↑.Info ← ValorFilho
          Pai↑.Esq ← Novo
          fim
fim
```

6.2.3 remoção de um nodo

A remoção de um nodo de uma árvore é geralmente complexa, implicando quase sempre em uma reorganização de parte desta árvore. A remoção de um nodo pode ser feita de duas formas:

a remoção lógica – o nodo não é fisicamente excluído. Alguma informação especial contida no próprio nodo informa que este não é mais válido. A estrutura da árvore é integralmente preservada. Esta não é a solução ideal no caso de a aplicação remover muitos nodos, pois todos ficarão alocados, ocupando espaço e fazendo com que pesquisas se tornem mais demoradas, uma vez que todos os nodos removidos também são analisados;

b `remoção física` – o nodo é fisicamente excluído. Se o nodo considerado é uma folha, ele é simplesmente liberado, devendo ser atualizado o elo de seu nodo-pai. Entretanto, caso se trate de um nodo interno, a árvore deverá ser reorganizada.

A seguir é mostrado um algoritmo que remove um nodo-folha de uma árvore binária. O nodo a ser removido é identificado pelo `Valor` de seu campo de informação. Caso o nodo não seja encontrado, ou que não seja uma folha, o algoritmo retorna `Sucesso` falso. Se for o único nodo da árvore (sua raiz), retorna a árvore vazia. O algoritmo utiliza uma função `LocalizarPai` que percorre a árvore em busca do nodo desejado, e retorna o endereço de seu nodo-pai. Caso o nodo não seja encontrado, retorna endereço nulo.

Algoritmo 6.4 – RemoverFolha

```
   Entradas: Arv (TipoArvore)
             Valor (TipoInfo)
   Saídas: Arv (TipoArvore)
           Sucesso (lógico)
   Variável auxiliar: Pai, Nodo (TipoPtNodo)
início
    se Arv↑.Info = Valor  {NODO É A RAIZ}
    então se (Arv↑.Dir = nulo) e (Arv↑.Esq = nulo)
          então início
                liberar(Arv)
                Arv ← nulo
                Sucesso ← verdadeiro
                fim
          senão Sucesso ← falso
    senão início
          Pai ← LocalizarPai(Arv, Valor)
          se (Pai ≠ nulo)
          então início
                se (Pai↑.Esq ≠ nulo) e (Pai↑.Esq↑.Info = Valor)
                então início  {FILHO A ESQUERDA}
                      Nodo ← Pai↑.Esq
                      se (Nodo↑.Esq = nulo) e (Nodo↑.Dir = nulo)
                      então início  {É FOLHA}
                            Sucesso ← verdadeiro
                            liberar(Nodo)
                            Pai↑.Esq ← nulo
                            fim
                      senão Sucesso ← falso
```

```
                fim
            senão início  {FILHO A DIREITA}
                Nodo ← Pai↑.Dir
                se (Nodo↑.Esq = nulo) e (Nodo↑.Dir = nulo)
                então início  {É FOLHA}
                    Sucesso ← verdadeiro
                    liberar(Nodo)
                    Pai↑.Dir ← nulo
                    fim
                senão Sucesso ← falso
                fim
            fim
        senão Sucesso ← falso
        fim
fim
```

6.2.4 acesso aos nodos

Quando se quer acessar todos os nodos de uma árvore, obrigatoriamente se passará mais de uma vez pelo mesmo nodo. O acesso à arvore é sempre realizado através da raiz. Se esta raiz apresenta dois descendentes, o acesso a cada um deles é feito a partir dela, implicando que a raiz seja acessada duas vezes. Geralmente se necessita acessar todos os nodos de uma árvore com o objetivo de realizar a mesma operação sobre todos eles (por exemplo, procurar um determinado nodo, imprimir os campos de informação de todos os nodos, etc.). À realização desta operação sobre um nodo denominamos de `visita ao nodo`.

Mesmo que se acesse um nodo diversas vezes durante o percurso da árvore, a operação pretendida (visita) deve ser realizada somente uma vez sobre esse nodo. É necessário definir uma estratégia de percurso desta árvore para que isso seja possível.

Denomina-se **`caminhamento`** de uma árvore o método de percurso sistemático de todos os nodos da árvore, de modo que cada nodo seja visitado exatamente uma vez. Um `caminhamento completo` define uma seqüência de nodos, de maneira que cada nodo da árvore passa a ter um nodo seguinte, ou um nodo anterior, ou ambos (exceto árvores de um só nodo). Todos os nodos da árvore devem aparecer nesta seqüência, cada um apenas uma vez. A ordem em que os nodos aparecem no caminhamento completo representa a ordem em que será feita a visita aos nodos.

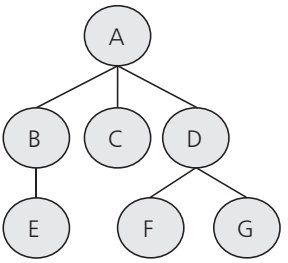
O objetivo buscado – acessar todos os nodos da árvore – pode ser alcançado através de diferentes caminhamentos. A seqüência de nodos obtida será diferente conforme o caminhamento utilizado. Por exemplo, na Figura 6.6 são apresentados dois possíveis caminhamentos (caminhamento 1 por níveis e caminhamento 2 por profundidade) que poderiam ter sido utilizados para percorrer uma determinada árvore – além deste, outros ainda podem ser definidos.

Como uma árvore binária apresenta no máximo dois descendentes em cada nodo, ela pode ser vista como composta por sua raiz, com uma subárvore à esquerda desta raiz e outra subárvore à sua direita (Figura 6.7). Os caminhamentos na árvore binária são definidos pela ordem em que são visitados estes três elementos – a raiz e suas duas subárvores. Cada subárvore é também uma árvore binária, com uma raiz com suas duas subárvores. Visitar uma subárvore significa visitar todos os nodos desta subárvore, utilizando para estas visitas o mesmo caminhamento.

Os principais caminhamentos para árvores binárias são apresentados a seguir, juntamente com a ordem em que devem ser realizadas as operações correspondentes sobre a árvore.

- `Caminhamento prefixado à esquerda`
  1. visitar a raiz
  2. percorrer e visitar a subárvore da esquerda
  3. percorrer e visitar a subárvore da direita

- `Caminhamento prefixado à direita`
  1. visitar a raiz
  2. percorrer e visitar a subárvore da direita
  3. percorrer e visitar a subárvore da esquerda



Caminhamento 1:
A - B - C - D - E - F - G

Caminhamento 2:
A - B - E - C - D - F - G

Figura 6.6 Caminhamentos para percorrer uma árvore binária.

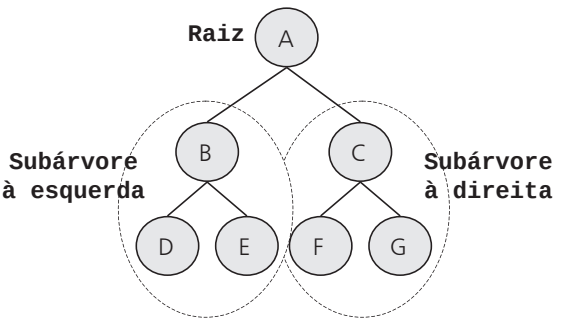


Figura 6.7 Árvore binária: raiz e subárvores.

- `Caminhamento pós-fixado à esquerda`
  1. percorrer e visitar a subárvore da esquerda
  2. percorrer e visitar a subárvore da direita
  3. visitar a raiz

- `Caminhamento pós-fixado à direita`
  1. percorrer e visitar a subárvore da direita
  2. percorrer e visitar a subárvore da esquerda
  3. visitar a raiz

- `Caminhamento central à esquerda`
  1. percorrer e visitar a subárvore da esquerda
  2. visitar a raiz
  3. percorrer e visitar a subárvore da direita

- `Caminhamento central à direita`
  1. percorrer e visitar a subárvore da direita
  2. visitar a raiz
  3. percorrer e visitar a subárvore da esquerda

Como exemplo, considerando a árvore apresentada na Figura 6.7, a visita aos seus nodos, de acordo com o caminhamento utilizado, terá a seguinte seqüência:

- caminhamento prefixado à esquerda → a – b – d – e – c – f – g
- caminhamento prefixado à direita → a – c – g – f – b – e – d
- caminhamento central à esquerda → d – b – e – a – f – c – g

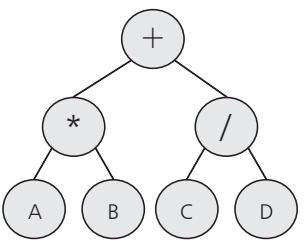
- caminhamento central à direita → g – c – f – a – e – b – d
- caminhamento pós-fixado à esquerda → d – e – b – f – g – c – a
- caminhamento pós-fixado à direita → g – f – c – e – d – b – a

Um exemplo de aplicação para um determinado caminhamento é a representação de uma expressão aritmética em uma árvore binária, com os operandos contidos nas folhas, e os operadores em nodos internos. Caso esta árvore seja percorrida com o caminhamento correto, no caso o `central à esquerda`, a prioridade dos operadores estará garantida. A Figura 6.8 apresenta a representação de uma expressão aritmética através de uma árvore binária, e a seqüência de operações realizadas através deste caminhamento que garante a avaliação desta expressão de acordo com a prioridade dos operadores aritméticos.

A seguir são apresentados alguns algoritmos de percurso sobre árvores binárias.

■ percurso através de diferentes caminhamentos

Um algoritmo que percorre uma árvore binária recebe somente o endereço da raiz da árvore, a partir da qual todos os outros nodos são acessados. Para cada nodo acessado, suas duas subárvores devem ser percorridas, de acordo com a ordem escolhida para o percurso. Assim, ao se alcançar um determinado nodo, passa-se a percorrer uma de suas subárvores, guardando de alguma forma o endereço do nodo inicial para depois utilizá-lo para percorrer a segunda subárvore. Isto geralmente é feito com o auxílio de uma `pilha`.



(A * B) + (C / D)

Figura 6.8 Árvore binária representando uma expressão aritmética.

Os algoritmos que correspondem aos três caminhamentos, apresentados a seguir, utilizam uma pilha para guardar informações. Para a pilha são utilizadas as seguintes representações simbólicas:

a pilha é representada por uma variável `Pilha`, sem entrar em detalhes de sua implementação, que pode ser sobre um arranjo ou através de encadeamento;
a pilha vazia é simbolicamente representada por [ ];
a pilha guardará endereços de nodos;
o operador "⇐" representa as operações de inserir um endereço no topo da pilha (`pilha ⇐ endereço`) ou remover o endereço do topo da pilha (`ponteiro ⇐ pilha`);
o procedimento `visitar(PtNodo)` simboliza a execução da operação que se quer realizar sobre cada nodo, podendo esta ser a pesquisa do valor contido no nodo, a alteração do valor de seu campo de informação, etc.

caminhamento prefixado

Quando uma árvore binária é percorrida através do caminhamento prefixado, o primeiro nodo a ser visitado é a raiz. Depois de feita a visita à raiz, são visitados todos os nodos de suas subárvores, na ordem requerida – por exemplo, se for caminhamento prefixado à esquerda, são visitados primeiro os nodos da subárvore da esquerda, depois os da direita.

Para implementar o caminhamento prefixado à esquerda, uma vez feita a visita à raiz, antes de descer pela subárvore da esquerda deve ser guardado o endereço da raiz da subárvore da direita, para que esta seja depois percorrida. O algoritmo que implementa este caminhamento inicia colocando o endereço da raiz da árvore na pilha. Em seguida são repetidos os passos a seguir, até que a pilha fique vazia:

```
1. retirar o endereço do topo da pilha;
2. se o endereço que estava no topo da pilha não for nulo:
   2.1 fazer a visita ao nodo correspondente a este endereço;
   2.2 colocar na pilha, nesta ordem, seus descendentes da direita
       (guardando este endereço para mais tarde), e da esquerda (que
       vai ficar no topo, para ser pesquisado).
```

O algoritmo que implementa estes passos recebe somente o endereço da raiz da árvore. O algoritmo de `Visitar` não é detalhado, pois vai depender da operação que se quer realizar sobre cada nodo.

```
Algoritmo 6.5 – PrefixadoEsq
   Entrada: Arv (TipoArvore)
   Saídas: –
   Variáveis auxiliares:
      PtAux (TipoPtNodo)
      Pilha (TipoPilha)
início
  Pilha ⇐ Arv
  enquanto Pilha ≠ [ ]
  faça início
       PtAux ⇐ Pilha
       se (PtAux ≠ nulo)
       então início
             Visitar(PtAux)
             Pilha ⇐ PtAux↑.Dir
             Pilha ⇐ PtAux↑.Esq
             fim
       fim
fim
```

O algoritmo que implementa o caminhamento prefixado à direita é semelhante a este apresentado, invertendo somente a ordem em que os descendentes de um nodo são inseridos na pilha: deve ser inserido primeiramente o endereço do descendente da esquerda, e depois o da direita.

Como exemplo de percurso de uma árvore através de caminhamento prefixado à esquerda, a seguir é mostrado o algoritmo da função `Localizar` utilizada no Algoritmo 6.3. Esta função percorre a árvore em busca de um nodo com um determinado `Valor` em seu campo de informação. Neste exemplo, o percurso poderia ser realizado em qualquer ordem, tendo sido aqui empregado o caminhamento prefixado à esquerda. Considera-se a implementação da pilha sobre um arranjo `Pilha`, cujos elementos são endereços para nodos da árvore, ou seja, cada elemento da pilha é do tipo `TipoPtNodo`. A função devolve o endereço do nodo onde foi encontrado o `Valor` ou, caso não encontre, o endereço nulo. São utilizadas, nesta função, as operações para pilhas implementadas sobre arranjos, definidas no capítulo 4.

```
Algoritmo 6.6 – Localizar (Função)
   Entradas: Arv (TipoArvore)
             Valor (TipoInfo)
   Retorno: (TipoPtNodo)
   Variáveis auxiliares:
```

```
        PtAux (TipoPtNodo)
        Achou, InsOK, OK (lógico)
        Pilha (TipoPilha)
início
  InicializarPilhaArr(Base, Topo)  {VARIÁVEIS GLOBAIS}
  Achou ← falso
  InserirPilhaArr(Pilha, Lim, Topo, Arv, InsOK) {INSERE RAIZ NA PILHA}
  enquanto (Topo ≥ 0) e (não Achou) e (InsOK)
  faça início
       RemoverPilhaArr(Pilha, Topo, Base, OK, PtAux)
       se (PtAux ≠ nulo) e (OK)
       então se PtAux↑.Info = Valor
             então início
                   Achou ← verdadeiro
                   Localizar ← PtAux
                   fim
             senão início
                  InserirPilhaArr(Pilha, Lim, Topo, PtAux↑.Dir, InsOK)
                  InserirPilhaArr(Pilha, Lim, Topo, PtAux↑.Esq, InsOK)
                   fim
       fim
  se (não Achou)
  então Localizar ← nulo
fim
```

caminhamento pós-fixado

Para mostrar como a disciplina escolhida para percorrer uma árvore altera os algoritmos que implementam operações sobre esta árvore, faremos agora a mesma operação anteriormente apresentada, alterando somente a forma de caminhamento para pós-fixado. Neste caso, a raiz deverá ser o último nodo a ser visitado – inicialmente são visitados todos os nodos da subárvore da esquerda, depois todos os da subárvore da direita e, por último, a raiz. Como o acesso é através da raiz, seu endereço deve ser guardado. Mas, como este endereço vai ser acessado duas vezes antes de ser feita a visita a este nodo (uma vez para cada subárvore), deve ser guardada alguma indicação suplementar para indicar se este acesso é o primeiro ou o segundo.

O algoritmo a seguir utiliza novamente uma pilha para guardar os endereços dos nodos. Nesta pilha serão colocados os endereços dos nodos da árvore, de duas formas: ou um endereço simples (quando um nodo é acessado pela primeira vez) ou um endereço "marcado" (alguma indicação que

sinalize que o nodo está sendo acessado pela segunda vez). Quando um endereço é retirado da pilha, se for um endereço simples, o mesmo é novamente retornado à pilha, desta vez marcado, acrescido dos endereços de suas subárvores. Quando um endereço marcado é removido da pilha, isto indica que este nodo pode agora ser visitado, pois suas duas subárvores já foram visitadas.

Esta estratégia é utilizada no algoritmo apresentado a seguir que implementa o caminhamento pós-fixado à esquerda. O algoritmo recebe somente o endereço da árvore, e a percorre realizando uma visita a cada nodo, na ordem definida pelo caminhamento em questão. A operação executada durante a visita a um nodo não é especificada, estando representada no algoritmo através de uma chamada a um procedimento denominado `Visitar`. Também não estão detalhados os algoritmos das funções `Marcado` (função lógica que verifica se um determinado endereço da pilha está marcado) e `Marcar` (função que marca um endereço a ser inserido na pilha).

```
Algoritmo 6.7 – Pós-FixadoEsq
   Entrada: Arv (TipoArvore)
   Saídas: –
início
   Pilha ⇐ Arv
   enquanto Pilha ≠ [ ]
   faça início
        PtAux ⇐ Pilha;
        se (PtAux ≠  nulo)
        então se Marcado(PtAux)
              então Visitar(PtAux)
              senão início
                    Pilha ⇐ Marcar(PtAux)
                    Pilha ⇐ PtAux↑.Dir
                    Pilha ⇐ PtAux↑.Esq
                    fim
        fim
fim
```

caminhamento central

A forma de percorrer uma árvore através de caminhamento central, tanto à esquerda como à direita, também pode ser implementada utilizando uma pilha, de modo semelhante ao do caminhamento pós-fixado à esquerda

apresentado anteriormente. A única diferença consiste na ordem em que os endereços são colocados na pilha, para que a raiz seja visitada antes de sua subárvore da direita.

operações de acesso implementadas através de algoritmos recursivos

Uma árvore binária é uma estrutura implicitamente recursiva: é formada por uma raiz, com uma subárvore à esquerda e outra subárvore à direita. Cada uma destas subárvores, por sua vez, têm a mesma estrutura: uma raiz e duas subárvores, uma à esquerda e a outra à direita. Isso faz com que se torne natural a utilização de procedimentos recursivos para percorrer árvores binárias. Os algoritmos recursivos são muito mais simples, uma vez que a pilha, implementada explicitamente nos algoritmos anteriores, agora é tratada diretamente pelas chamadas recursivas.

Por exemplo, a implementação do percurso prefixado à esquerda para percorrer uma árvore binária é iniciada fazendo a visita à raiz. Em seguida deve ser percorrida toda a subárvore à esquerda através do mesmo caminhamento, o que é feito através de uma chamada recursiva para a subárvore da esquerda, entregando-lhe o endereço da raiz desta subárvore. Durante a execução desta chamada recursiva, outras chamadas recursivas serão feitas, uma para cada subárvore desta. Quando terminar a execução da primeira chamada recursiva, toda a subárvore da esquerda foi percorrida. É feita, então, outra chamada recursiva, entregando agora o endereço da raiz da subárvore da direita, chamada esta que vai percorrer esta segunda subárvore. As chamadas recursivas param depois de chegar em uma folha, pois vão receber endereços nulos para as subárvores.

A seguir é mostrado o algoritmo que percorre uma árvore binária através de caminhamento prefixado à esquerda, utilizando recursividade. A operação é realizada sempre que o endereço recebido seja válido (diferente de nulo), efetuando a visita ao nodo recebido e fazendo as chamadas recursivas para as suas subárvores. Quando uma chamada recebe um endereço nulo, nada é executado.

```
Algoritmo 6.8 – PrefixadoEsqRec
   Entrada: Arv (TipoArvore)
   Saídas: –
início
   se Arv ≠ nulo
```

```
   então início
         Visitar(Arv)
         PrefixadoEsqRec(Arv↑.Esq))
         PrefixadoEsqRec(Arv↑.Dir)
         fim
fim
```

Os algoritmos recursivos para percorrer árvores binárias através dos outros tipos de caminhamentos são muito semelhantes a este mostrado acima, adequando somente a ordem das chamadas recursivas e da visita à raiz. A seguir são mostrados os algoritmos para os caminhamentos pós-fixado e central à esquerda.

Algoritmo 6.9 – Pós-FixadoEsqRec

```
   Entrada: Arv (TipoArvore)
   Saídas: –
início
   se Arv ≠ nulo
   então início
         Pós-fixadoEsqRec(Arv↑.Esq))
         Pós-fixadoEsqRec(Arv↑.Dir)
         Visitar(Arv)
         fim
fim
```

Algoritmo 6.10 – CentralEsqRec

```
   Entrada: Arv (TipoArvore)
   Saídas: –
início
   se Arv ≠ nulo
   então início
         CentralEsqRec(Arv↑.Esq))
         Visitar(Arv)
         CentralEsqRec(Arv↑.Dir)
         fim
fim
```

6.2.5 destruição de uma árvore

Quando uma árvore alocada durante uma aplicação não for mais necessária, todas as posições ocupadas por seus nodos podem ser liberadas. Isto pode ser feito percorrendo toda a árvore através de um dos caminhamentos pós-fixados, liberando cada um dos nodos alocados (a visita ao nodo é a sua liberação). É

necessário que o caminhamento de percurso seja pós-fixado para que a liberação de um nodo somente seja feita quando todos os seus descendentes foram liberados. O último nodo a ser liberado será a raiz. A operação de destruição de uma árvore deverá retornar nulo o endereço de acesso a esta árvore.

6.3 → exemplos de aplicações que utilizam árvores binárias

Nesta seção são apresentados alguns exemplos de aplicações de árvores binárias, utilizando sempre as soluções recursivas, mais apropriadas às árvores. Fica a sugestão de adaptar os algoritmos apresentados para o caso de não se querer utilizar a recursividade (para tornar o algoritmo mais eficiente, ou por não dispor deste recurso na linguagem de programação utilizada na implementação).

6.3.1 construção de uma árvore

A operação de construção de uma árvore depende da ordem em que são fornecidos os valores para seus nodos. Deve ser escolhido um caminhamento adequado a esta ordem. A árvore gerada estará diretamente relacionada ao caminhamento escolhido – isto é, deverá ser percorrida posteriormente através do caminhamento para o qual foi gerada. Por exemplo, a Figura 6.9 mostra a ordem em que devem ser fornecidos os valores para que seja construída a árvore binária apresentada, considerando o caminhamento prefixado à esquerda: o primeiro valor obtido será o da raiz, o próximo será o de seu filho à esquerda, e assim por diante. Os asteriscos representam algum dado que informe que não existe o descendente buscado.

O algoritmo a seguir mostra, em passos gerais, como pode ser gerada esta árvore a partir dos dados obtidos:

1. alocar o nodo correspondente à raiz;
2. preencher o campo de informação da raiz com o primeiro valor obtido;
3. construir a subárvore da esquerda (chamada recursiva), e colocar o endereço da raiz desta subárvore no elo à esquerda do nodo raiz;
4. construir a subárvore da direita, e colocar o endereço de sua raiz no elo à direita da raiz da árvore que está sendo montada.

O próximo algoritmo constrói uma árvore ou uma subárvore, dependendo do momento em que é chamado. O endereço do nodo alocado é devolvido

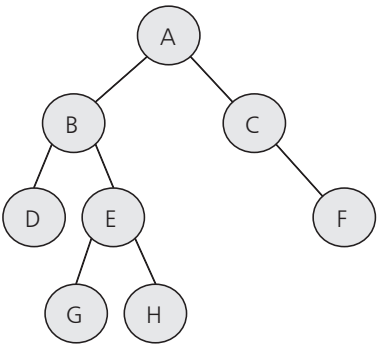


Figura 6.9 Ordem de valores para gerar árvore com caminhamento prefixado à esquerda.

através do parâmetro `Arv`. As chamadas recursivas solicitam a construção das subárvores, sendo os endereços das raízes destas subárvores colocados nos campos de elo esquerdo e direito do nodo considerado. O algoritmo utiliza uma função denominada `ObtémInfo` que obtém a informação a ser inserida no campo de informação do novo nodo. Quando esta função devolver o caractere "*" é sinal de que foi alcançado o final de um ramo da árvore, sendo inserido o endereço nulo como endereço da subárvore construída.

```
Algoritmo 6.11 – ConstruirArv
   Entradas: –
   Saída: Arv (TipoArvore)
   Variável auxiliar: InfoProx (TipoInfo)
início
   InfoProx ← ObtémInfo
   se InfoProx = "*"
   então Arv ← nulo
   senão início
         alocar(Arv)
         Arv↑.Info ← InfoProx
         ConstruirArv(Arv↑.Esq)
         ConstruirArv(Arv↑.Dir)
         fim
fim
```

Ressaltamos que este algoritmo somente poderá ser utilizado caso os valores sejam fornecidos de acordo com o caminhamento prefixado à esquerda. Se os valores forem informados em outra ordem, o algoritmo para a construção da árvore deverá ser adaptado.

6.3.2 montagem de uma lista a partir de uma árvore

A aplicação a seguir mostra como pode ser montada uma lista linear simplesmente encadeada a partir das informações contidas nos nodos de uma árvore binária. Os nodos da lista montada terão dois campos: um campo de informação idêntico ao campo de informação dos nodos da árvore, e outro com o elo para o próximo nodo da lista. A lista é montada a partir do último nodo, que vai conter a informação da raiz da árvore, alocando e encadeando os novos nodos na frente da lista, como mostrado na Figura 6.10. Se a lista montada for percorrida a partir de seu início, os nodos encontrados correspondem a percorrer a árvore através do caminhamento pós-fixado à direita.

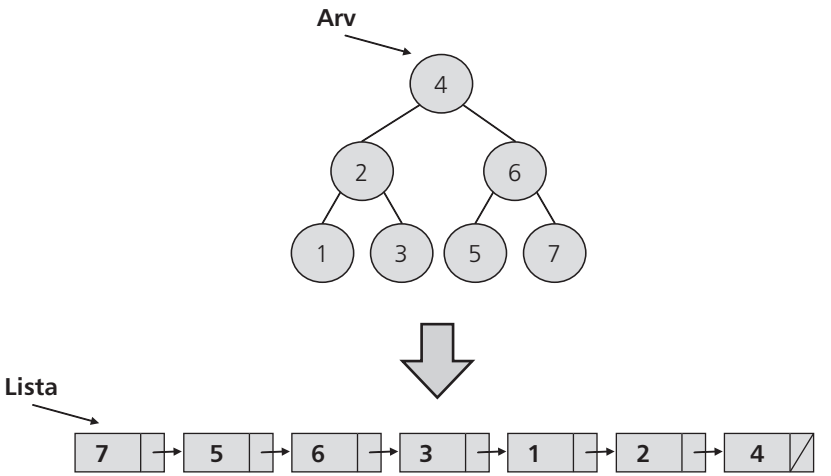


Figura 6.10 Lista linear montada a partir de árvore binária.

O algoritmo a seguir implementa esta operação. Ele recebe o endereço da raiz da árvore e devolve, através de outro parâmetro, o endereço do nodo inicial da lista. Caso o endereço recebido como raiz da árvore seja nulo, o algoritmo devolve o endereço nulo para o início da lista. O algoritmo é recursivo: aloca um novo nodo para a lista e o encadeia com aquele que era o primeiro da lista, passando este a ser o primeiro; em seguida, faz uma chamada recursiva para cada um de seus descendentes (esquerdo e direito). Ao final de cada uma das chamadas recursivas estará montada a lista correspondente a toda a subárvore.

```
Algoritmo 6.12 – FazerListaDeArv
   Entrada: Arv (TipoArvore)
   Saída: Lista (TipoPtLista)
   Variável auxiliar: PtListaAux (TipoPtLista)
início
se  Arv ≠ nulo
então início
      alocar(PtListaAux)
      PtListaAux↑.Info ← Arv↑.Info
      se Lista = nulo
      então PtListaAux↑.Prox ← nulo
      senão PtListaAux↑.Prox ← Lista
      Lista := PtListaAux
      FazerListaDeArv (Arv↑.Esq, Lista)
      FazerListaDeArv(Arv↑.Dir, Lista)
      fim
fim
```

6.3.3 cálculo do valor de uma expressão aritmética

Como mostrado na Figura 6.8, a representação de uma expressão aritmética através de uma árvore binária permite sua avaliação de acordo com a prioridade dos operadores. Esta forma de árvore é gerada pelo analisador sintático de um compilador ao fazer o reconhecimento da expressão. A seguir é apresentada uma função que avalia uma expressão aritmética a partir de sua representação em árvore binária.

A árvore gerada apresenta dois tipos de nodos, com semânticas diferentes: alguns representam um operando (valores numéricos ou valores de variáveis), outros um operador. Para diferenciar os nodos, o algoritmo a seguir utiliza

um campo especial em cada nodo, denominado `Tipo`, que informa o tipo de informação que o nodo contém. O valor de `Tipo = 0` corresponde a um operando, e `Tipo = 1`, a um operador. Além disso, operandos e operadores têm tipos diferentes, devendo ser armazenados em campos diferentes, aqui denominados de `Valor` (numérico, para o operando), e `Oper` (do tipo caractere, para o operador). Os nodos desta árvore têm, portanto, a seguinte estrutura:

```
TipoNodoEA = registro
                Tipo: inteiro
                Oper: caractere
                Valor: real
                Esq, Dir: ↑TipoNodoEA
             fim registro
TipoPtNodoEA = ↑TipoNodoEA
```

É utilizado o caminhamento central à esquerda para processar a expressão aritmética. A operação a ser realizada pela função depende do tipo do nodo: caso seja um operando, o resultado é o valor contido no nodo; caso seja um operador, o resultado da função é dado pelo resultado da aplicação deste operador ao resultado obtido nas subárvores da esquerda e da direita do nodo em questão, resultado este obtido através de chamadas recursivas, fornecendo a raiz da subárvore correspondente. A função devolve o valor zero caso receba um endereço nulo para a raiz da árvore ou quando o valor correspondente ao `TIPO` for inválido.

Algoritmo 6.13 – ValorEA (Função)

```
   Entrada: Arv (TipoArvore)
   Retorno: (real)
início
   se (Arv = nulo) ou ((Arv↑.Tipo ≠ 0) e (Arv↑.Tipo ≠ 1))
   então ValorEA ← 0
   senão se Arv↑.Tipo = 0           {OPERANDO}
         então ValorEA ← Arv↑.Oper
         senão caso Arv↑.Oper seja  {OPERADOR}
                "+": ValorEA ← ValorEA(Arv↑.Esq) + ValorEA(Arv↑.Dir)
                "-": ValorEA ← ValorEA(Arv↑.Esq) – ValorEA(Arv↑.Dir)
                "*": ValorEA ← ValorEA(Arv↑.Esq) * ValorEA(Arv↑.Dir)
                "/": ValorEA ← ValorEA(Arv↑.Esq) / ValorEA(Arv↑.Dir)
               fim caso
fim
```

6.4 → árvores binárias de pesquisa

Árvores binárias que apresentam uma relação de ordem nos campos de informação de seus nodos são denominadas `Árvores Binárias de Pesquisa` (ABP). A ordem é definida pelo valor contido em um determinado campo de informação dos nodos, aqui denominado `chave`.

A ordem entre os nodos de uma ABP é a seguinte: para qualquer nodo de uma ABP, todos os nodos de sua subárvore da esquerda contém valores de chave menores do que a do nodo considerado, e todos os nodos da subárvore da direita apresentam valores de chave maiores ou iguais à chave do nodo considerado. A possibilidade de uma ABP apresentar chaves duplicadas depende da aplicação que estiver sendo representada: algumas aplicações requerem que as chaves sejam únicas, enquanto que outras permitem chaves repetidas.

A Figura 6.11 apresenta um exemplo de ABP na qual estão representados somente os valores das chaves – outros valores de informação, não representados na figura, podem estar contidos nos nodos. Neste exemplo, os nodos da árvore estão em ordem crescente caso a árvore seja percorrida através de caminhamento central à esquerda, e em ordem decrescente caso a árvore seja percorrida através de caminhamento central à direita.

Devido à ordem que apresentam, estas árvores são eficientes para pesquisar valores, ou seja, procurar as informações associadas a uma determinada chave – daí o nome dado a este tipo de árvore. No exemplo da figura 6.11, sabe-se que para qualquer nodo, todos os nodos de sua subárvore à esquerda têm chaves menores, e todos os nodos de sua subárvore à direita têm chaves

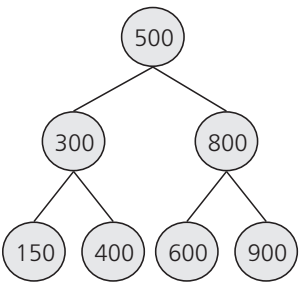


Figura 6.11 Exemplo de árvore binária de pesquisa.

maiores. Portanto, para encontrar um nodo com uma determinada chave nesta ABP, inicia-se comparando a chave buscada com a da raiz. Caso seja diferente, o caminho a seguir será definido verificando se a chave procurada é maior ou menor do que a chave do nodo considerado. Isto permite que a pesquisa siga em somente um dos ramos, sendo o outro ramo imediatamente descartado, diminuindo consideravelmente o número de nodos a analisar e, conseqüentemente, o tempo gasto na pesquisa. Por exemplo, na Figura 6.11, caso seja buscado o nodo cuja chave vale `600`, é efetuado o seguinte procedimento:

1. compara-se o valor buscado (`600`) com a chave da raiz (`500`). Como são diferentes e sendo o valor buscado maior do que a chave da raiz, a pesquisa segue pela subárvore da direita;
2. o segundo nodo analisado é, então, a raiz da subárvore da direita, cuja chave é `800`. Neste caso, sendo ainda diferente da chave buscada, e sendo esta menor do que a deste nodo, a pesquisa segue pela subárvore da esquerda;
3. o próximo nodo analisado, raiz da subárvore da esquerda do anterior, possui a chave igual à buscada.

Vemos, portanto, que o nodo buscado foi encontrado analisando somente três nodos da árvore, bem menos do que se a pesquisa fosse realizada sem levar em consideração a ordem em que se encontram as chaves dos nodos. Isto significa que, para qualquer pesquisa, o número máximo de comparações efetuadas é igual ao número de níveis da árvore.

As operações básicas sobre ABPs são analisadas a seguir. Tratando-se de um caso particular de árvore binária, as operações de criação e destruição da árvore são as mesmas apresentadas anteriormente. Serão vistas aqui, portanto, as operações de inserção de um novo nodo (respeitando a ordem), remoção de um nodo, e busca (pesquisa) de um determinado nodo.

Os tipos de dados utilizados nos algoritmos para ABP são os seguintes:

```
TipoPtNodoABP = ↑TipoNodoABP
TipoNodoABP = registro
                Info: TipoInfo
                Chave: inteiro
                Esq, Dir: TipoPtNodoABP
              fim registro
```

6.4.1 inserção de um novo nodo

A inserção de um novo nodo em uma ABP deve respeitar a ordem apresentada pelas chaves dos nodos, ou seja, a árvore resultante deve continuar ordenada após a inserção. Uma forma de garantir esta ordem é inserindo sempre os novos nodos como folhas. A posição onde um novo nodo vai ser inserido é definida por sua chave. Por exemplo, a Figura 6.12 mostra as posições onde devem ser inseridos dois novos nodos na ABP da Figura 6.11, um com valor de chave `380` e o outro com `750`.

A seguir é mostrada uma função que insere um nodo em uma ABP. Considera-se que não existam chaves duplicadas. Para inserir um novo nodo é feita uma pesquisa a partir da raiz da árvore, comparando a chave de cada nodo com a chave do nodo a ser inserido – se a chave do novo nodo for maior, o nodo deverá ser inserido na subárvore da direita do nodo em questão; caso contrário, será inserido na sua subárvore da esquerda. Como as chaves dos nodos são únicas, a nova inserção não deverá ser realizada caso seja encontrado algum nodo com a chave igual ao nodo que se quer inserir. A inserção será efetivada no momento em que se encontrar uma folha da árvore, quando então o novo nodo será inserido como folha da anterior.

Por exemplo, para se encontrar a posição onde deve ser inserido o nodo com chave `380` no exemplo da Figura 6.12 é feita, inicialmente, a comparação deste valor com a chave da raiz da árvore. A chave desta raiz sendo `500`, a

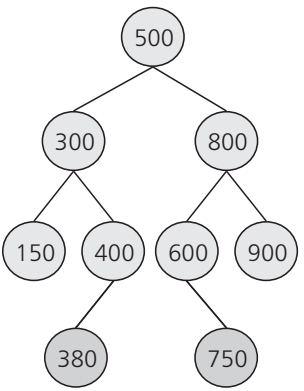


Figura 6.12 Inserção de dois novos nodos na ABP.

inserção deverá ser realizada na subárvore da esquerda, pois 380 é menor do que 500. A chave da raiz desta subárvore é 300, indicando que a inserção deverá ser realizada na subárvore da direita deste nodo. Nesta, a chave da raiz é 400, indicando a inserção na subárvore da esquerda desta; entretanto, como esta subárvore não existe, o novo nodo é inserido como folha à esquerda do nodo de chave 400.

O algoritmo apresentado a seguir considera que o novo nodo já foi alocado e preenchido, sendo seu endereço fornecido através do parâmetro Novo. O objetivo é, portanto, achar a posição em que este nodo deve ser inserido, de modo a preservar a característica de ordenamento da árvore, e fazer o encadeamento. A operação é realizada por uma função lógica. Como as chaves devem ser únicas, a função retorna falso caso já exista um nodo com esta chave na árvore, devendo então a aplicação remover o nodo anteriormente alocado. A pesquisa é realizada por uma função (BuscarABP) que será apresentada na Seção 6.4.3 (Algoritmo 6.16), que procura um nodo com uma determinada chave em uma ABP, retornando seu endereço, ou o endereço nulo caso não a encontre. Caso a chave não seja encontrada na árvore, a função InserirABP passa a procurar onde o novo nodo deve ser inserido. Caso seja o primeiro nodo a ser inserido na árvore (quando o endereço recebido para Raiz for nulo), o novo nodo passará a ser a raiz. Se a árvore já possuir nodos, a função passará a efetuar as comparações, descendo pela subárvore adequada até chegar em uma folha, quando então o novo nodo é encadeado.

```
Algoritmo 6.14 – InserirABP (Função)
   Entradas: Raiz (TipoPtNodoABP)
             Novo (TipoPtNodoABP)   {NOVO NODO, JÁ ALOCADO}
   Saídas: Raiz (TipoPtNodoABP)
   Retorno: (lógico)
   Variáveis auxiliares:
      PAux (TipoPtNodoABP)
      Inseriu (lógico)
início
   se BuscarABP(Raiz, Novo↑.Chave) ≠ nulo
   então InserirABP ← falso {CHAVE JÁ EXISTE NA ÁRVORE}
   senão início
         InserirABP ← verdadeiro
         se Raiz = nulo
         então Raiz ← Novo  {INSERIR PRIMEIRO NODO – RAIZ}
         senão início  {BUSCA POSIÇÃO E INSERE}
               PAux ← Raiz
```

```
                Inseriu ← falso
                enquanto não Inseriu
                faça se Novo↑.Chave > PAux↑.Chave
                     então se PAux↑.Dir = nulo
                           então início
                                 PAux↑.Dir ← Novo
                                 Inseriu ← verdadeiro
                                 fim
                           senão PAux ← PAux↑.Dir
                     senão se PAux↑.Esq = nulo
                           então início
                                 PAux↑.Esq ← Novo
                                 Inseriu ← verdadeiro
                                 fim
                           senão Paux ← PAux↑.Esq
                fim
          fim
fim
```

Note que a ordem em que os nodos são inseridos em uma ABP é importante para a configuração resultante. Na Figura 6.13 são apresentadas duas ABPs diferentes, geradas a partir dos mesmos dados, fornecidos em ordem diferente.

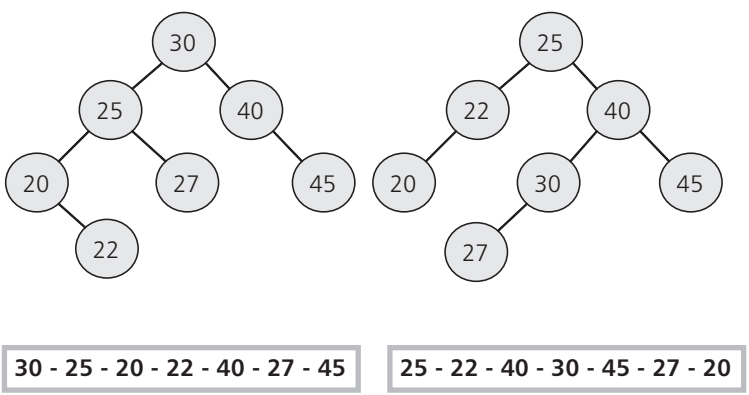


Figura 6.13 Ordem de inserção dos nodos em uma ABP e árvores resultantes.

6.4.2 remoção de nodo

Como visto na Seção 6.2.3, a operação de remoção de um nodo de uma árvore é complexa, implicando quase sempre em uma reorganização de parte desta árvore. No caso de uma ABP, esta reorganização deve ser feita de modo a preservar a ordem das chaves.

Quando for realizada uma `remoção lógica` de um nodo (quando o nodo permanece alocado, com alguma informação de que ele não é mais válido), é necessário que a chave deste nodo permaneça acessível, podendo ser utilizada nas buscas para identificar qual a subárvore que deve ser seguida a partir deste nodo.

A seguir é analisada a alternativa de `remoção física` de nodos. Dependendo da posição do nodo a ser removido na árvore, três situações podem ocorrer, implicando em diferentes formas de fazer a remoção, analisadas a seguir.

`Remoção de uma folha`. Quando o nodo a ser removido for uma folha, ele é simplesmente liberado, bastando atualizar o campo de elo de seu ascendente direto para o endereço nulo. A remoção de uma folha é o caso mais simples, pois não requer que a árvore seja reorganizada. A Figura 6.14 mostra uma ABP e a remoção da folha de chave `60` desta árvore.

`Remoção de nodo de derivação com somente uma subárvore`. Para que um nodo interno seja fisicamente removido é necessário que outro nodo ocupe seu lugar na árvore, sempre respeitando a ordem numérica das chaves. Caso o nodo a ser removido possua somente uma subárvore, isto pode ser feito subindo de um nível a raiz desta subárvore, de modo que ela ocupe a posição que o nodo removido ocupava. Assim fica preservada a ordem das chaves da

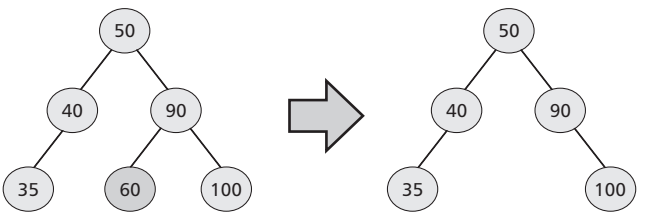


Figura 6.14 Remoção de uma folha em uma ABP.

ABP. A Figura 6.15 mostra a remoção do nodo de derivação de chave 45, cuja posição será ocupada pela raiz de sua subárvore, de chave 20.

Remoção de nodo de derivação com duas subárvores. É a situação mais complexa, na qual a árvore necessita ser reestruturada. Duas estratégias podem ser adotadas: (1) substituir o nodo a ser removido por aquele que possua a maior chave da sua subárvore da esquerda; ou (2) substituir o nodo por aquele que apresentar a menor chave da sua subárvore da direita. Uma vez identificado o nodo que irá ocupar a posição daquele que será removido, dois casos podem ocorrer, analisados a seguir.

Caso 1. Se o nodo que irá substituir o removido for uma **folha**, é feita simplesmente a troca de sua posição na árvore. Exemplificando esta situação, na Figura 6.16 é removido o nodo de chave 40 e, em seu lugar, é colocado o nodo de chave 30, maior chave de sua subárvore da esquerda.

Caso 2. Se o nodo a ser colocado na posição do removido for um **nodo de derivação**, a situação é diferente. Por exemplo, na Figura 6.17, para remover o nodo de chave 85, caso seja usada a opção de substituí-lo pelo nodo de maior chave de sua subárvore da esquerda, este será o nodo de chave 80. Se o nodo 80 tivesse algum descendente à direita, este teria necessariamente uma chave maior do que o valor 80. Devido à ordenação da ABP, o nodo que vai ser utilizado na substituição sempre terá somente uma subárvore.

A substituição do nodo a ser removido da árvore pelo nodo encontrado deve ser feita em dois passos: (1) inicialmente é removido de sua posição o nodo que vai ser utilizado na substituição (no exemplo, o nodo de chave 80), utilizando a operação vista no item anterior, ou seja, subindo um nível a raiz de

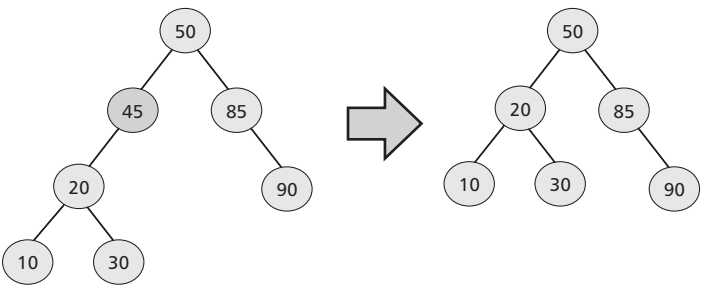


Figura 6.15 Remoção de nodo de derivação com uma subárvore, em ABP.

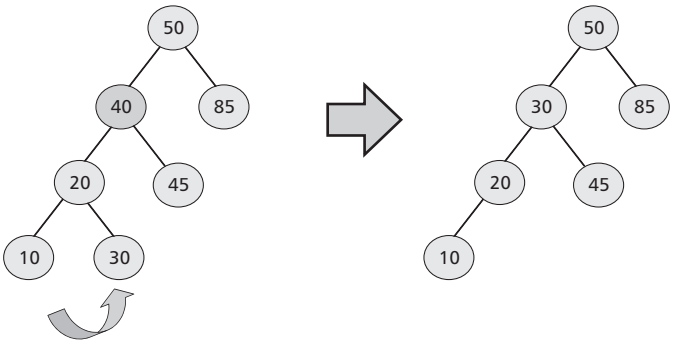


Figura 6.16 Remoção de nodo de derivação com duas subárvores, em ABP – caso 1.

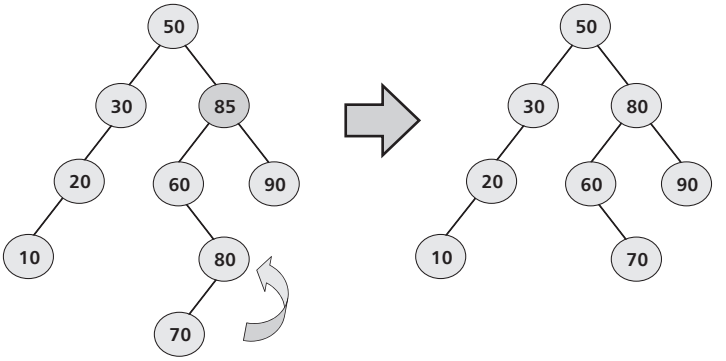


Figura 6.17 Remoção de nodo de derivação com duas subárvores, em ABP – caso 2.

sua subárvore; (2) em seguida, este nodo é encadeado de modo a ocupar a posição do nodo a ser removido. A Figura 6.17 mostra esta operação: inicialmente o nodo de chave `80` é substituído pela raiz de sua subárvore, de chave `70`, para depois ocupar a posição do nodo removido (`85`).

O algoritmo `RemoverABP` apresentado a seguir remove um nodo, identificado por sua `ChaveRem`, de uma ABP. Supõe-se que esta chave esteja presente na árvore. É utilizado um procedimento que é chamado recursivamente até encontrar o nodo buscado, sendo que a cada chamada é passada a `Raiz` da subárvore que deve ser percorrida na busca do nodo. Além desses parâmetros, o procedimento recebe ainda o endereço do antecessor da raiz de cada subárvore analisada (`Pai`), para permitir acertar seu encadeamento quando o

nodo for removido. Na primeira chamada do procedimento deve ser fornecido o mesmo endereço para os três parâmetros `Arv`, `Pai` e `Raiz`. Se o nodo a ser removido for a raiz da árvore, o tratamento é diferenciado, o novo endereço da raiz da árvore (`Arv`) deve ser devolvido ao programa de aplicação, não sendo necessário ajustar o encadeamento do antecessor do nodo removido. Caso o nodo não seja encontrado na ABP, é devolvido o endereço nulo.

Algoritmo 6.15 – RemoverABP

```
   Entradas: Arv, Pai, Raiz (TipoPtNodoABP)
             ChaveRem (inteiro)
   Saída: Arv (TipoPtNodoABP)
   Variável auxiliar: Anterior (TipoPtNodoABP)
início
   se Raiz ≠ nulo
   então se Raiz↑.Chave ≠ ChaveRem
         então se Raiz↑.Chave > ChaveRem
               então RemoverABP(Arv, Raiz, Raiz↑.Esq, ChaveRem)
               senão RemoverABP(Arv, Raiz, Raiz↑.Dir, ChaveRem)
         senão início  {VAI REMOVER O NODO DE ENDEREÇO RAIZ}
               se (Raiz↑.Esq = nulo) e (Raiz↑.Dir = nulo)
               então {REMOVER FOLHA}
                     início
                     se Raiz = Arv
                     então Arv ← nulo
                     senão se Pai↑.Dir = Raiz
                           então Pai↑.Dir ← nulo
                           senão Pai↑.Esq ← nulo
                     fim
               senão se (Raiz↑.Esq = nulo) ou (Raiz↑.Dir = nulo)
                     então {NODO TEM SOMENTE UMA SUBÁRVORE}
                           se Raiz↑.Esq = nulo
                           então {SUBIR SUBÁRVORE DIREITA}
                                 se Raiz ≠ Arv
                                 então se Pai↑.Esq = Raiz
                                       então Pai↑.Esq ← Raiz↑.Dir
                                       senão Pai↑.Dir ← Raiz↑.Dir
                                 senão Arv ← Raiz↑.Dir
                           senão {SUBIR SUBÁRVORE ESQUERDA}
                                 se Raiz ≠ Arv
                                 então se Pai↑.Esq = Raiz
                                       então Pai↑.Esq ← Raiz↑.Esq
                                       senão Pai↑.Dir ← Raiz↑.Esq
                                 senão Arv ← Raiz↑.Esq
```

```
                    senão {NODO TEM DUAS SUBÁRVORES}
                          início
                          Nodo ← Raiz↑.Esq
                          Anterior ← Raiz
                          enquanto Nodo↑.Dir ≠ nulo
                          faça início
                               Anterior ← Nodo
                               Nodo ← Nodo↑.Dir
                               fim
                          {NODO VAI SUBSTITUIR RAIZ}
                          se (Nodo↑.Esq = nulo)
                          então Anterior↑.Dir ← nulo {É UMA FOLHA}
                          senão Anterior↑.Dir ← Nodo↑.Esq
                          se Raiz ≠ Arv
                          então se Pai↑.Esq = Raiz
                                then Pai↑.Esq ← Nodo
                                senão Pai↑.Dir ← Nodo
                          senão Arv ← Nodo
                          Nodo↑.Esq ← Raiz↑.Esq
                          Nodo↑.Dir ← Raiz↑.Dir
                          fim
               liberar(Raiz)
               Raiz ← nulo
               fim
fim
```

6.4.3 acesso a um nodo

Como já mencionado, as ABPs são estruturadas de maneira a tornar mais rápida a busca a um determinado nodo, com base em sua chave. A busca de uma determinada chave em uma ABP é realizada da seguinte forma:

1. o valor procurado é comparado com a chave do nodo-raiz. Se for igual, cessa a busca;
2. caso o valor procurado seja maior do que a chave do nodo-raiz, toda a subárvore à esquerda é eliminada da busca, sendo o processo de busca repetido para a raiz da subárvore da direita;
3. caso o valor procurado seja menor do que a chave do nodo-raiz, a busca segue na subárvore da esquerda.

A função a seguir implementa esta busca, recebendo o endereço da raiz da árvore (`Arv`) e o valor de chave a ser buscado (`ChaveBuscada`). A função de-

volve o endereço do nodo que apresentar esta chave. Caso não seja encontrado algum nodo com esta chave, a função retorna endereço `nulo`.

```
Algoritmo 6.16 – BuscarABP (Função)
   Entradas: Arv (TipoPtNodoABP)
             ChaveBuscada (inteiro)
   Retorno: (TipoPtNodoABP)
   Variáveis auxiliares:
       Achou (lógico)
       PtNodo (TipoPtNodoABP)
início
  PtNodo ← Arv
  Achou ← falso
  enquanto (não Achou) e (PtNodo ≠ nulo)
  faça se PtNodo↑.Chave = ChaveBuscada
       então Achou ← verdadeiro
       senão se PtNodo↑.Chave > ChaveBuscada
             então PtNodo ← PtNodo↑.Esq
             senão PtNodo ← PtNodo↑.Dir
  BuscarABP ← PtNodo
fim
```

Outra forma de implementar esta mesma função é utilizando a natureza implicitamente recursiva das árvores binárias, conforme mostrado na função a seguir.

```
Algoritmo 6.17 – BuscarABPRec (Função)
   Entradas: Raiz (TipoPtNodoABP)
             ChaveBuscada (inteiro)
   Retorno: (TipoPtNodoABP)
início
  se Raiz = nulo
  então BuscarABPRec ← nulo
  senão se Raiz↑.Chave = ChaveBuscada
        então BuscarABPRec ← Raiz
        senão se Raiz↑.Chave > ChaveBuscada
             então BuscarABPRec ← BuscarABPRec(Raiz↑.Esq, ChaveBuscada)
             senão BuscarABPRec ← BuscarABPRec(Raiz↑.Dir, ChaveBuscada)
fim
```

6.5 ⇢ árvores balanceadas

Considerando somente a operação de busca de informações, o tempo de execução para encontrar uma informação depende diretamente da distância do nodo que contém esta informação até a raiz da árvore. Árvores balanceadas têm por objetivo otimizar as operações de consulta, diminuindo o número médio de comparações efetuadas até que um determinado nodo seja encontrado. Duas formas de balanceamento podem ser utilizadas:

`balanceamento por altura`, também denominado `balanceamento uniforme`, no qual todas as folhas ficam à mesma distância da raiz, ou a uma distância muito semelhante. A ABP mostrada na Figura 6.11 é completamente balanceada por altura;

`balanceamento por freqüência`, também denominado `balanceamento não uniforme`, no qual a distribuição dos nodos leva em conta a freqüência de acessos feitos a cada nodo, sendo aqueles mais acessados dispostos mais próximos à raiz da árvore.

Estes dois tipos de balanceamento serão analisados a seguir para árvores binárias.

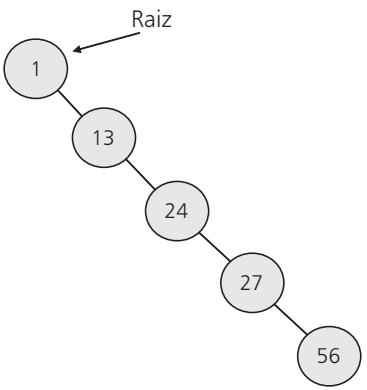


Figura 6.18 ABP desbalanceada.

6.5.1 árvores balanceadas por altura – AVL

No balanceamento por altura, somente a distância do nodo até a raiz é considerada. Quanto mais balanceada for uma árvore, mais rapidamente serão acessados todos os seus nodos. Uma ABP (Seção 6.4) construída por uma aplicação através de sucessivas inserções de novos nodos, pode resultar muito desbalanceada por altura, levando a buscas nada eficientes das informações. Por exemplo, a Figura 6.18 mostra a ABP resultante da inserção de nodos quando as chaves são fornecidas na seguinte ordem: 1 – 13 – 24 – 27 – 56.

Exigir que uma árvore seja completamente balanceada por altura restringe a possibilidade de sua utilização porque apresenta uma manutenção muito dispendiosa, exigindo reorganizações freqüentes para correções dos desvios na estrutura da árvore. Pode-se definir um grau de balanceamento para garantir uma determinada eficiência nas buscas, ou seja, um fator que limita o máximo desbalanceamento que a árvore pode apresentar deixando a manutenção da estrutura da árvore menos onerosa. Árvores que apresentam esta característica são denominadas `árvores altamente balanceadas` com `fator de balanceamento FB(k)`, ou `k-balanceadas`. Uma árvore possui `FB(k)` se, para qualquer nodo, as alturas de suas subárvores não diferem por mais de `k` unidades. Assim, árvores `FB(1)` são aquelas para as quais as alturas das subárvores de qualquer nodo não diferem de mais de uma unidade; nas árvores `FB(2),` as alturas das subárvores variam no máximo de duas unidades; e assim por diante.

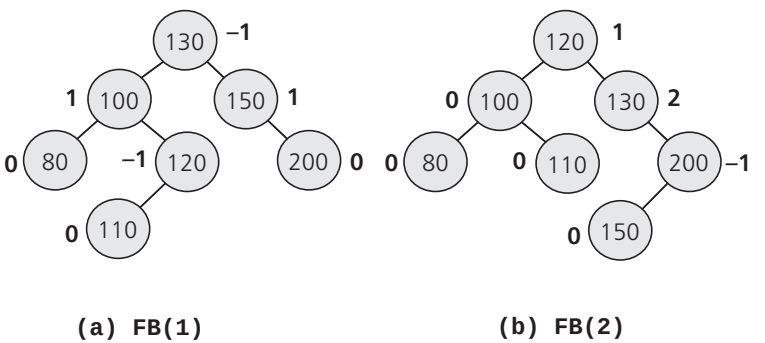


Figura 6.19 ABP com fator de balanceamento FB(k) representado em cada nodo.

Para garantir o balanceamento desejado de uma árvore, deve-se calcular o FB para todos os seus nodos a cada inserção ou remoção realizada. Nos algoritmos mostrados a seguir é utilizada a seguinte fórmula para calcular este fator:

```
FB = (altura subárvore direita) menos (altura subárvore esquerda)
```

A ordem escolhida entre as subárvores (aqui direita menos a esquerda) não é relevante, mas deverá ser a mesma em toda a aplicação. Um exemplo de árvore FB(1) é apresentado na figura 6.19a, e de uma árvore FB(2) na figura 6.19b. Junto a cada nodo é mostrado o seu FB. Nota-se que em alguns nodos o FB fica negativo, devido à ordem em que é calculado. Por exemplo, na árvore (a) desta figura, o FB da raiz (nodo 130) é –1, pois a altura da subárvore à direita é 2 (distância entre os nodos 130 e 200), e da à esquerda é 3 (distância entre os nodos 130 e 110). A informação representada, na realidade, consiste no módulo do FB, não sendo considerado o sinal negativo. O FB da árvore toda será o maior valor, em módulo, encontrado em algum nodo.

Árvores AVL. ABPs que apresentam FB(1) para balanceamento por altura são conhecidas como Árvores AVL. Isto significa que, em todo nodo, a diferença entre as alturas de suas subárvores não excede a uma unidade O nome AVL vem de *Adelson-Velskii e Landis,* que introduziram, em 1962, uma estrutura de árvore binária balanceada com respeito às alturas das subárvores.

É bom lembrar que a árvore deixa de ser AVL caso em algum de seus nodos não se verificar a propriedade FB(1). Aplicações que utilizem árvores AVL deverão tomar muito cuidado no momento de inserir ou remover nodos, para que o balanceamento por altura não seja perdido.

Muitas vezes é necessário reestruturar a árvore para manter seu balanceamento. Toda vez que for efetuada uma operação de inserção de um novo nodo em uma árvore AVL, ou que seja removido um nodo, deve ser feita uma verificação de toda a árvore, recalculando os fatores de balanceamento de cada nodo. Costuma-se incluir em cada nodo da árvore AVL um campo que indica o seu fator de balanceamento, para tornar mais eficiente este processo de verificação após uma inserção ou remoção. Caso em algum nodo seja detectado um fator diferente de |1| (valor em módulo, sem considerar eventuais sinais negativos), a árvore deve ser reestruturada para que volte a ser uma árvore AVL. Ao reestruturar uma árvore AVL é importante que seja preservada a ordem das chaves, utilizada para a busca de informações.

operações de rotação

A reestruturação de uma árvore AVL é feita através de uma `operação de rotação` em torno do nodo onde foi constatado o fator `FB` maior do que uma unidade (valor absoluto). A seguir são analisadas quatro operações de rotação, denominadas simples à esquerda, simples à direita, dupla à esquerda e dupla à direita. Nos algoritmos relativos a estas operações é utilizado o seguinte tipo de nodo:

```
TipoPtNodoAVL = ↑TipoNodoAVL
TipoNodoAVL = registro
                FB: inteiro     {fator de balanceamento do nodo}
                Chave: inteiro
                Info: TipoInfo
                Esq, Dir: TipoPtNodoAVL
              fim registro
```

rotação simples à direita

A reestruturação de uma árvore AVL deve ser feita através deste tipo de rotação sempre que:

o nodo desbalanceado apresentar fator negativo;
a raiz da subárvore da esquerda deste nodo também apresentar fator negativo.

A Figura 6.20 mostra como é efetuada esta rotação. Nesta figura, `T1`, `T2` e `T3` são subárvores binárias, vazias ou não; o nodo `P` representa o nodo que apresentou fator maior do que |1|, raiz desta rotação, e o nodo `U`, a raiz de sua subárvore da esquerda. A rotação simples à direita consiste em uma rotação de `U` sobre `P`, sendo feita da seguinte maneira:

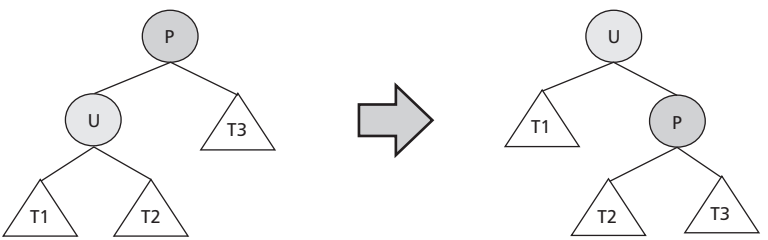


Figura 6.20 Rotação simples à direita em árvore AVL.

o nodo U vai ocupar a posição antes ocupada pelo nodo P;
o nodo P passa a ser o descendente da direita do nodo U;
a subárvore da direita do nodo P passa a ser a subárvore da esquerda do nodo U.

A Figura 6.21 apresenta um exemplo de rotação simples à direita. A inserção de um novo nodo, com chave 4, na árvore AVL original (a) fez com que o nodo de chave 42 passasse a apresentar fator –2 (b). A rotação à direita em torno do nodo de chave 42 faz com que a árvore volte a ser AVL (c).

O algoritmo a seguir executa a rotação à direita, recebendo o endereço do nodo para o qual foi detectado o fator maior do que |1| (Pt), e devolvendo o endereço do nodo que ocupa o seu lugar depois da rotação.

```
Algoritmo 6.18 – RotaçãoDireitaAVL
   Entrada: Pt (TipoPtNodoAVL)
   Retorno: Pt (TipoPtNodoAVL)
   Variável auxiliar: Ptu (TipoPtNodoAVL)
início
   Ptu ← Pt↑.Esq
   Pt↑.Esq ← Ptu↑.Dir
```

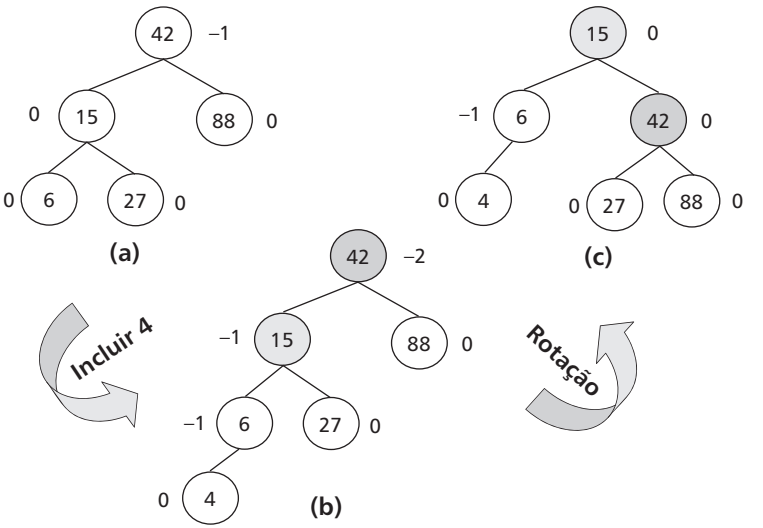


Figura 6.21 Exemplo de rotação simples à direita.

```
   Ptu↑.Dir ← Pt
   Pt↑.FB ← 0
   Pt ← Ptu
fim
```

rotação simples à esquerda

Este tipo de rotação deve ser utilizado sempre que:

o nodo desbalanceado apresentar fator positivo;
a raiz da subárvore da esquerda deste nodo também apresentar fator positivo.

A rotação simples à esquerda é mostrada na Figura 6.22, onde o nodo P representa o nodo que apresentou fator maior do que |1|, raiz desta rotação, e o nodo Z, a raiz de sua subárvore da direita. A rotação simples à esquerda consiste em uma rotação de Z sobre P, sendo feita da seguinte maneira:

o nodo Z vai ocupar a posição antes ocupada pelo nodo P;
o nodo P passa a ser o descendente da esquerda do nodo Z;
a subárvore da esquerda do nodo Z passa a ser a subárvore da direita do nodo P.

A Figura 6.23 apresenta um exemplo de rotação simples à esquerda. A inserção de um novo nodo, com chave 90, na árvore AVL original (a) fez com que o nodo de chave 42 passasse a apresentar fator 2 (b). Após a rotação à esquerda este nodo passa a apresentar fator 0, e a árvore volta a ser AVL (c).

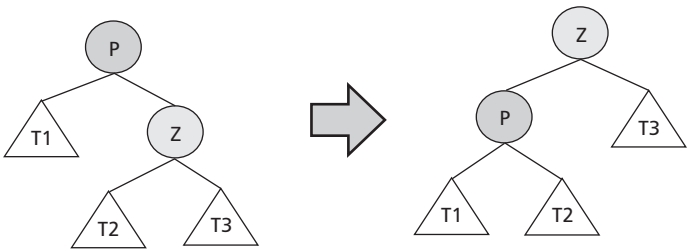


Figura 6.22 Rotação simples à esquerda.

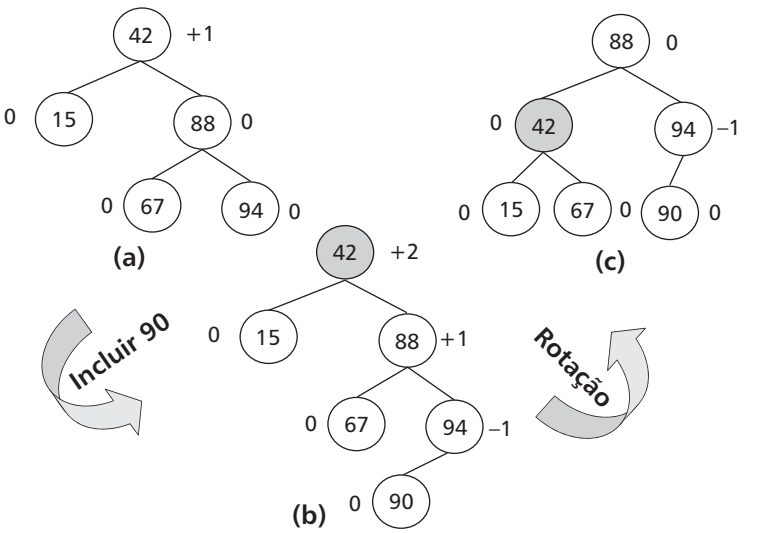


Figura 6.23 Exemplo de rotação simples à esquerda.

O algoritmo a seguir executa a rotação à esquerda, recebendo o endereço do nodo para o qual foi detectado o fator maior do que |1| (Pt), e devolvendo o endereço do nodo que ocupa o seu lugar depois da rotação.

```
Algoritmo 6.19 – RotaçãoEsquerdaAVL
   Entrada: Pt (TipoPtNodoAVL)
   Retorno: Pt (TipoPtNodoAVL)
   Variável auxiliar: Ptu (TipoPtNodoAVL)
início
   Ptu ← Pt↑.Dir
   Pt↑.Dir ← Ptu↑.Esq
   Ptu↑.Esq ← Pt
   Pt↑.FB ← 0
   Pt ← Ptu
fim
```

rotação dupla à direita

A reestruturação de uma árvore AVL deve ser feita através deste tipo de rotação sempre que:

um nodo apresentar fator negativo;
a raiz da subárvore da esquerda deste nodo apresentar fator positivo.

Esta forma de rotação é mostrada na Figura 6.24, onde o nodo P representa o nodo que apresentou fator maior do que 1; seu descendente à esquerda é o nodo U; e o descendente à direita de U é o nodo V. A operação é composta por duas rotações, executadas em seqüência uma da outra: inicia por uma rotação simples à esquerda, seguida de uma rotação simples à direita.

A Figura 6.25 resume a operação de rotação dupla à direita, que consiste de uma rotação de V sobre P e depois sobre V, sendo feita da seguinte maneira:

o nodo V vai ocupar a posição antes ocupada pelo nodo P;
o nodo U passa a ser o descendente da esquerda do nodo V;
o nodo P passa a ser o descendente da direita do nodo V;
a subárvore da esquerda do nodo V passa a ser a subárvore da direita do nodo U;
a subárvore da direita do nodo V passa a ser a subárvore da esquerda do nodo P.

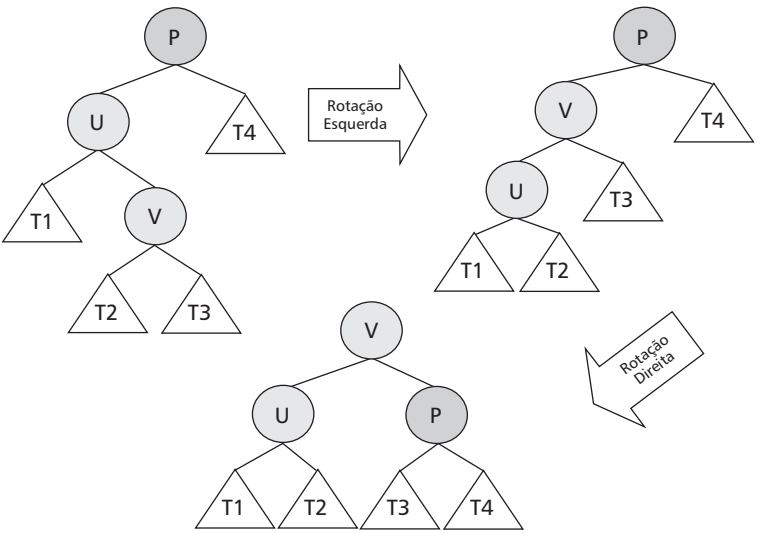


Figura 6.24 Rotação dupla à direita – seqüência de rotações.

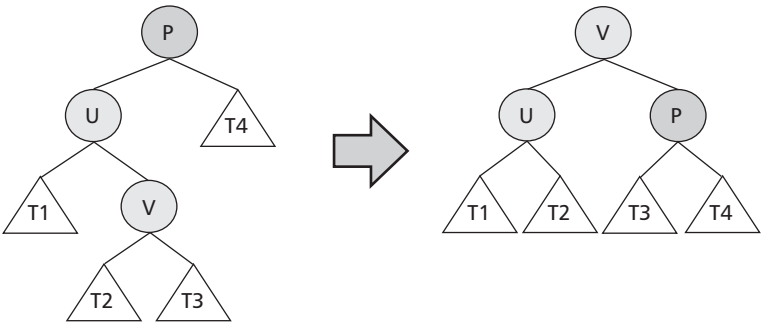


Figura 6.25 Rotação dupla à direita.

Um exemplo de rotação dupla à direita é apresentado na Figura 6.26. A inserção de um novo nodo, com chave `100`, na árvore AVL original (a) fez com que o nodo de chave `110` passe a apresentar fator `–2` (b). A rotação dupla à direita faz com que a árvore volte a ser AVL (c).

O algoritmo a seguir executa a rotação dupla à direita. Ele recebe o endereço do nodo para o qual foi detectado o fator maior do que |1|, e devolve o endereço do nodo que ocupa o seu lugar depois da rotação.

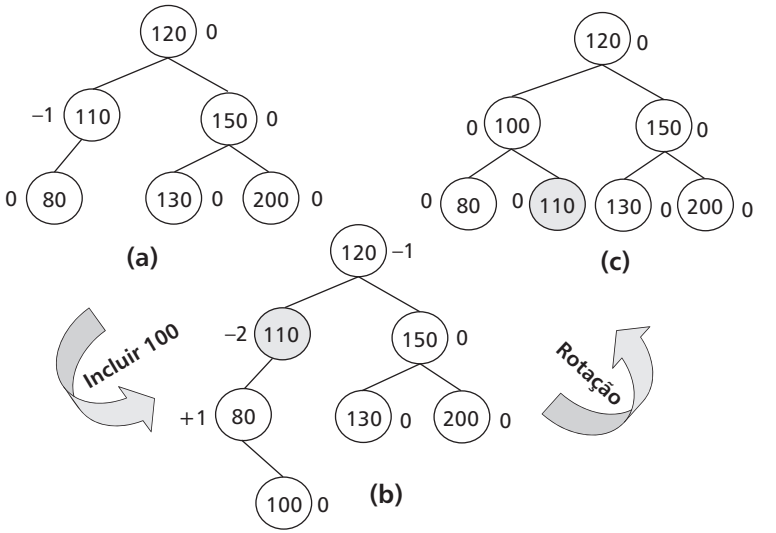


Figura 6.26 Exemplo de rotação dupla à direita.

```
Algoritmo 6.20 – RotaçãoDuplaDireitaAVL
   Entrada: Pt (TipoPtNodoAVL)
   Retorno: Pt (TipoPtNodoAVL)
   Variáveis auxiliares:
       Ptu, Ptv (TipoPtNodoAVL)
início
   Ptu ← Pt↑.Esq
   Ptv ← Ptu↑.Dir
   Ptu↑.Dir ← Ptv↑.Esq
   Ptv↑.Esq ← Ptu
   Pt↑.Esq ← Ptv↑.Dir
   Ptv↑.Dir ← Pt
   se Ptv↑.FB = 1
   então Pt↑.FB ← –1
   senão Pt↑.FB ← 0
   se Ptv↑.FB = –1
   então Ptu↑.FB ← 1
   senão Ptu↑.FB ← 0
   Pt ← Ptv
fim
```

rotação dupla à esquerda

A rotação dupla à esquerda é empregada quando a árvore AVL apresentar:

um nodo com fator positivo;
a raiz da subárvore da direita deste nodo com fator negativo.

Esta forma de rotação é mostrada na Figura 6.27, onde o nodo P representa o nodo que apresentou fator maior do que |1|; seu descendente à direita é o nodo Z; e o descendente à esquerda de Z é o nodo Y. Como no caso anterior, esta operação é composta por duas rotações executadas em seqüência uma da outra: inicia por uma rotação simples à direita, e é seguida de uma rotação simples à esquerda.

A Figura 6.28 mostra o resultado da operação de rotação dupla à esquerda, que consiste de uma rotação de Y sobre P e depois sobre Z, sendo executada da seguinte maneira:

o nodo Y vai ocupar a posição antes ocupada pelo nodo P;
o nodo P passa a ser o descendente da esquerda do nodo Y;
o nodo Z passa a ser o descendente da direita do nodo Y;

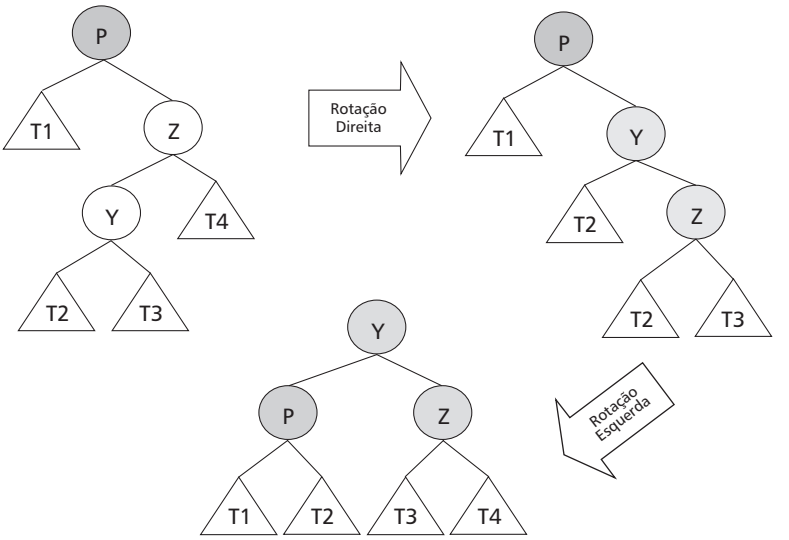


Figura 6.27 Rotação dupla à esquerda – seqüência de rotações.

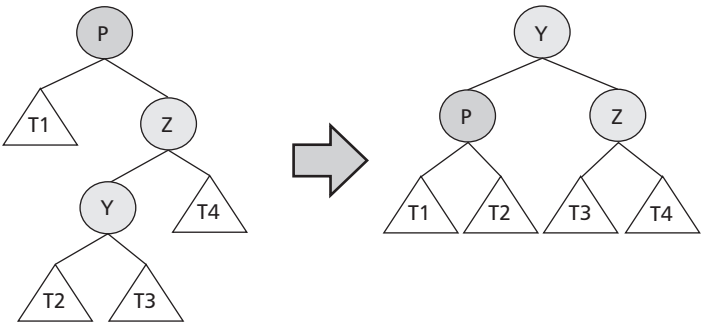


Figura 6.28 Rotação dupla à esquerda.

a subárvore da esquerda do nodo Y passa a ser a subárvore da direita do nodo P;
a subárvore da direita do nodo Y passa a ser a subárvore da esquerda do nodo Z.

A Figura 6.29 apresenta um exemplo de rotação dupla à esquerda. A inserção de um novo nodo, com chave `150`, na árvore AVL original (a) fez com que o

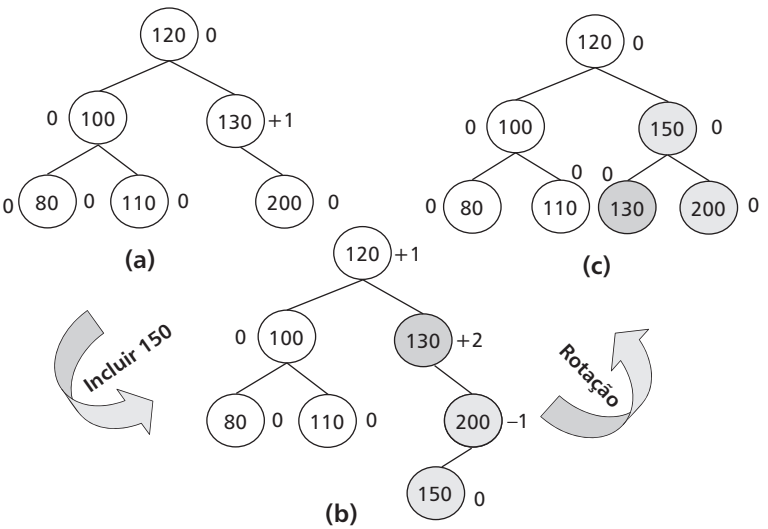


Figura 6.29 Exemplo de rotação dupla à esquerda.

nodo de chave `130` passe a apresentar fator 2 (b). A rotação dupla à direita faz com que a árvore volte a ser AVL (c).

O algoritmo a seguir executa a rotação dupla à esquerda. Ele recebe o endereço do nodo para o qual foi detectado o fator maior do que |1|, e devolve o endereço do nodo que ocupa o seu lugar depois da rotação.

```
Algoritmo 6.21 – RotaçãoDuplaEsquerdaAVL
   Entrada: Pt (TipoPtNodoAVL)
   Retorno: Pt (TipoPtNodoAVL)
   Variáveis auxiliares:
       Pty, Ptz (TipoPtNodoAVL)
início
   Ptz ← Pt↑.Dir
   Pty ← Ptz↑.Esq
   Ptz↑.Esq ← Pty↑.Dir
   Pty↑.Dir ← Ptz
   Pt↑.Dir ← Pty↑.Esq
   Pty↑.Esq ← Pt
   se Pty↑.FB = –1
   então Pt↑.FB ← 1
   senão Pt↑.FB ← 0
```

```
   se Pty↑.FB = 1
   então Ptz↑.FB ← –1
   senão Ptz↑.FB ← 0
   Pt ← Ptv
fim
```

■ inserção de um novo nodo

Com base no que foi visto anteriormente, podemos agora definir um algoritmo para inserir um novo nodo em uma árvore AVL. Devem ser fornecidos, além do endereço do nodo-raiz da árvore (A), a chave do novo nodo (ValorChave) e o campo de informação que o nodo deve apresentar (Dado). O algoritmo InserirAVL, apresentado a seguir, insere o novo nodo na posição definida pela chave fornecida, sempre como uma folha. Em seguida, ele verifica o fator de balanceamento dos nodos antecessores, e efetua as rotações necessárias para que a árvore continue sendo AVL. Os algoritmos BalancearEsq e BalancearDir, utilizados no balanceamento, não são aqui apresentados, ficando como exercícios sugeridos. O parâmetro OK informa se a inserção foi realizada com sucesso.

O algoritmo não analisa se a chave já existe na árvore, como foi feito no caso da ABP. Ele inicia procurando a localização em que o nodo deve ser inserido. Caso nesta pesquisa seja encontrado um nodo que já possua a chave do novo nodo, o procedimento é interrompido sem que a inserção seja realizada. Diferentemente do que é feito para ABP, aqui o novo nodo somente é alocado quando for encontrada sua posição.

```
Algoritmo 6.22 – InserirAVL
   Entradas: A (TipoPtNodoAVL)
             ValorChave (inteiro)
             Dado (TipoInfo)
             OK (lógico)
   Retorno: A (TipoPtNodoAVL)
início
   se A = nulo
   então início  {INSERÇÃO }
         alocar(A)
         A↑.Esq ← A↑.Dir ← nulo
         A↑.Chave ← ValorChave
         A↑.Info ← Dado
         A↑.FB ← 0
         OK ← verdadeiro
```

```
        fim
   senão início
        se ValorChave = A↑.Chave
        então OK ← falso
        senão se ValorChave < A↑.Chave
              então início
                    InserirAVL(A↑.Esq, ValorChave, Dado, OK)
                    se OK
                    então caso A↑.FB seja
                              1: A↑.FB ← 0
                              0: A↑.FB ← -1
                             -1: BalancearEsq(A)
                          fim caso
                    fim
              senão início
                    InserirAVL(A↑.Dir, ValorChave, Dado, OK)
                    se OK
                    então caso A↑.FB seja
                             -1: A↑.FB ← 0
                              0: A↑.FB ← 1
                              1: BalancearDir(A, OK)
                           fim caso
                    fim
        fim
fim
```

6.5.2 árvores balanceadas por freqüência

Uma árvore binária organizada por freqüência tem por objetivo agilizar as consultas com base no conhecimento do número de acesso aos nodos. Para isso é necessário armazenar, ao longo da aplicação, o número de acessos que foram feitos a cada nodo. De posse desta informação, a árvore é organizada de modo que as informações acessadas mais freqüentemente sejam colocadas mais próximas à raiz, permitindo que sua recuperação seja mais rápida.

Como exemplo de árvore balanceada por freqüência, vamos analisar um tipo de árvore binária que apresenta as seguintes características:

as informações estão somente nas folhas;
existe em cada folha uma informação adicional (w_i) que informa a freqüência de acesso a esta folha (índice de acesso).

Denominando o comprimento do caminho de uma folha até a raiz de l_i, para que seja balanceada por freqüência, uma árvore de m folhas deve apresentar o *caminho ponderado mínimo*, sendo este definido pelo valor mínimo obtido para o somatório abaixo:

$$\sum_{i=1}^{m} w_i l_i$$

A Figura 6.30 apresenta duas formas de organizar quatro informações em uma árvore binária deste tipo, ou seja, colocando as informações somente nas folhas. É conhecido o índice de freqüência de acesso a cada uma destas informações, representado na figura pelos valores 2, 3, 4 e 11 – o valor com índice de acesso 11 é o mais acessado, e aquele que tem o índice de acesso 2 é o que apresenta menos acessos na aplicação. Abaixo de cada árvore é apresentada a forma de calcular o somatório acima. Vemos que a árvore da esquerda apresenta um valor bem menor para este somatório. Podemos observar também que na árvore da esquerda, o valor 11, que representa a folha mais acessada, está a apenas um nível da raiz, enquanto os nodos que apresentam os menores valores (2 e 3) estão a três níveis de distância.

A construção de uma árvore binária com caminho ponderado mínimo pode ser feita através do Algoritmo de Huffman, apresentado a seguir, em passos

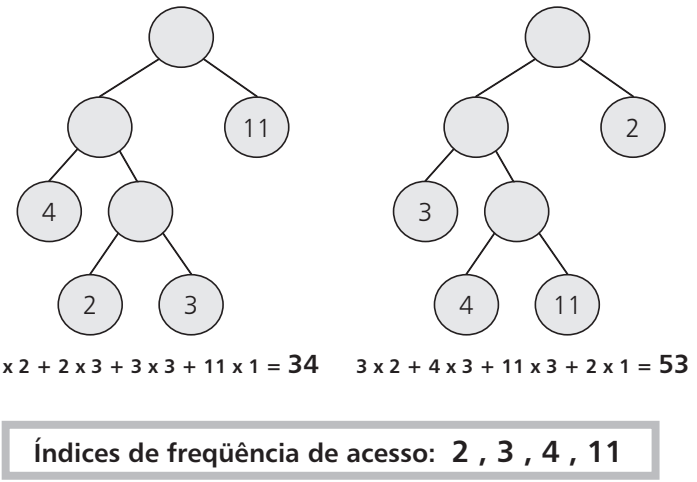


Figura 6.30 Caminho ponderado mínimo em árvores binárias.

gerais. Através deste algoritmo a árvore é construída a partir de suas folhas, onde estão as informações.

Algoritmo de Huffman

1. Construir um nodo para cada informação, associando a esta informação sua freqüência de acesso.
2. Procurar os dois menores valores contidos no conjunto de freqüências de acesso a cada informação.
3. Substituir estes dois valores pela sua soma, formando um novo nodo com este valor, e sendo este nodo a raiz dos dois valores anteriormente encontrados.
4. Repetir os passos 2 e 3 até que o conjunto de freqüências de acesso seja reduzido a um só valor.

Na Figura 6.31 é apresentado um exemplo de construção de uma árvore binária com caminho ponderado mínimo, através desse algoritmo.

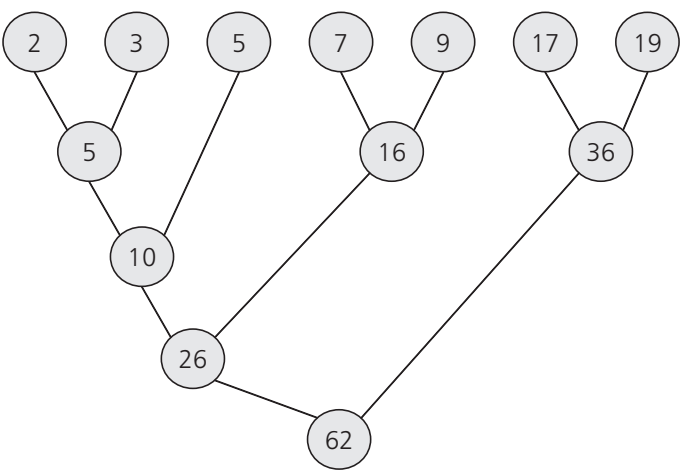


Figura 6.31 Construção de árvore com caminho ponderado mínimo pelo algoritmo de Huffman.

5.6 → exercícios

exercícios com árvores binárias

exercício 1 Considere a árvore binária da figura 6.2, armazenada por níveis em um arranjo. Implemente algoritmos para:

- inserir um novo nodo como descendente à esquerda do nodo cujo campo de informação for igual a um valor passado como parâmetro;
- listar os campos de informação dos nodos-folha;
- remover todos os nodos descendentes à direita, com seus descendentes.

exercício 2 Repita o exercício anterior, considerando que a árvore binária foi representada em profundidade.

exercício 3 Construa um algoritmo para transformar uma árvore n-ária, implementada por encadeamento, em uma árvore binária equivalente. Defina um formato para o nodo e limite em cinco o número de descendentes que cada nodo pode ter. Teste o algoritmo transformando a árvore do Exercício 16 do capítulo anterior em uma árvore binária equivalente.

exercício 4 Construa o algoritmo inverso ao do exercício anterior: dada a árvore binária, o algoritmo deve construir a árvore n-ária equivalente. Utilize o mesmo formato para o nodo desta última.

exercício 5 O Algoritmo 6.6 apresenta a função LOCALIZAR que utiliza uma pilha para guardar informações durante a busca que realiza na árvore. O algoritmo apresentado implementa esta pilha sobre um arranjo (alocação seqüencial). Reescreva esta função implementando a pilha através de encadeamento.

exercício 6 O Algoritmo 6.7 (PÓS-FIXADOESQ) percorre uma árvore binária através de caminhamento pós-fixado à esquerda. Pede-se:

- modifique este algoritmo para que realize o percurso da árvore através de caminhamento central à direita;
- implemente o algoritmo, escolhendo uma forma de marcar as informações inseridas na pilha, e optando por implementar a pilha ou através de encadeamento, ou por contigüidade física.

exercício 7 Na Seção 6.3.1 é mostrado como pode ser construída uma árvore, sendo fornecidos os valores de acordo com o caminhamento prefixado. Construa um algoritmo para construir uma árvore binária por níveis, ou seja, primeiro seria fornecido o valor para a raiz, depois os valores dos dois filhos da raiz e assim por diante. Neste caso, a natureza recursiva da árvore não poderá ser utilizada.

exercício 8 Construa um algoritmo que faça uma cópia de uma árvore existente. O algoritmo deverá receber o endereço da raiz da árvore original, e devolver o endereço da raiz da árvore gerada. Caso receba um endereço nulo, deve devolver um endereço nulo, indicando que, como não havia árvore para copiar, nenhuma árvore nova foi gerada.

exercício 9 Construa uma função recursiva `Iguais` (`Arv1`, `Arv2`), para determinar se as duas árvores binárias `Arv1` e `Arv2` são iguais, tanto em estrutura como em conteúdo de seus nodos. A função deve retornar verdadeira caso as árvores sejam iguais, e falsa, caso contrário.

exercício 10 Construa um algoritmo que insere um novo nodo em uma árvore binária não-ordenada, como descendente de um determinado nodo identificado por seu campo de informação.

exercício 11 Implemente um procedimento que insira um novo nodo em uma árvore binária ordenada, devendo este ser a folha mais à esquerda da árvore.

exercício 12 Na Seção 6.2.2 é apresentada uma função que insere um descendente à esquerda de um nodo identificado pelo seu conteúdo. Adapte esta função para inserir um descendente à direita.

exercício 13 Seja uma árvore binária `A` onde cada nodo possui, além dos ponteiros para os nodos-filhos, um campo denominado `Info`, contendo a identificação do nodo. Especifique um programa que percorra essa árvore binária e escreva, ao percorrê-la, a identificação do nodo visitado e de seu nodo-pai. Identifique o tipo de caminhamento que você escolheu para percorrer a árvore.

exercício 14 Escreva um algoritmo que recebe, como parâmetro, o do endereço da raiz de uma árvore binária e um valor que identifique um nodo, e

que imprime o conteúdo dos campos de informação deste nodo e dos seus descendentes diretos (filhos).

exercício 15 Considere a existência de duas árvores binárias, cujos endereços das raízes estão nos ponteiros `a1` e `a2`. Diz-se que `a1` ⊆ `a2` quando `a1` é equivalente a `a1` ou quando existe pelo menos uma subárvore de `a2` que seja equivalente a `a1`. Escreva uma função ou procedimento que testa se uma árvore binária `a1` está contida em outra árvore binária `a2`.

exercício 16 Tendo como entrada uma árvore binária (`Arv`) e um valor (`V`), escreva um algoritmo que localize esse valor na árvore. Se encontrado, calcule o número de nodos das subárvores de `V`, recursivamente para cada subárvore a partir de `V`. Identifique o tipo de caminhamento que você escolheu para percorrer a árvore.

exercício 17 Dada uma árvore binária, construa uma função que informe se nesta árvore existem nodos com informações duplicadas (iguais).

exercício 18 Construa um algoritmo que, dada uma lista encadeada, monte uma árvore com as informações contidas nesta lista. Identifique em que caminhamento esta árvore deve ser percorrida para que os nodos sejam acessados na mesma ordem em que estão na lista inicial.

exercícios com árvores binárias de pesquisa (ABP)

exercício 19 Escreva uma função recursiva que receba uma árvore e verifique se ela é ou não uma ABP, retornando `Verdadeiro` (1) ou `Falso` (0) conforme o caso.

exercício 20 Dadas duas ABPs `A` e `B`, desenvolva um algoritmo recursivo que retorne a ABP `C`, que representa a união entre `A` e `B`.

exercício 21 Dadas duas ABPs `A` e `B`, desenvolva um algoritmo recursivo que retorne a ABP `C`, que representa a intersecção entre `A` e `B`.

exercício 22 Considere uma ABP cujos nodos apresentam a estrutura utilizada na seção 6.4. O endereço da raiz desta árvore está na variável-apontador `RAIZ`. Escreva um algoritmo que recebe `RAIZ` e imprime o campo de informação das folhas da árvore, em ordem decrescente de valores de chaves.

exercício 23 Considerando a ABP apresentada na Figura 6.11, procure identificar a existência ou não de ordem (e caso exista, qual) para esta mesma árvore, caso seja percorrida com caminhamento (1) central à direita, e (2) prefixado à esquerda.

exercício 24 Suponha uma série de dados organizados em uma ABP. Escreva:

- a função `SomaNodos(Chave)` para determinar quantos nodos têm chave superior à fornecida;
- o algoritmo `ExcluiABPFolha` que exclui um nodo da árvore, mas somente se este for uma folha.

exercício 25 Dada uma ABP, construa um algoritmo que gere uma lista duplamente encadeada (não circular) com os dados contidos no campo de informação da árvore, respeitando a ordem crescente dos valores contidos nos seus nodos.

exercício 26 Dada uma lista encadeada, monte uma ABP com os valores contidos nos nodos da lista.

exercícios com árvores balanceadas e AVL

exercício 27 Verifique se as árvores 2-balanceadas, ou seja, que apresentam `FB(2)`, são árvores AVL.

exercício 28 Desenvolva um algoritmo para inserir nodos em uma árvore 2-balanceada.

exercício 29 Um certo professor Amongus afirma que a ordem pela qual um conjunto fixo de elementos é inserido em uma árvore AVL não importa – irá resultar sempre na mesma árvore AVL. O professor Amongus está correto? Justifique sua resposta através de um pequeno exemplo que prove se o professor Amongus está certo ou errado.

exercício 30 Reescreva os algoritmos vistos para rotações da Seção 6.5.1.1., considerando que a diferença entre as alturas é calculada diminuindo a altura da subárvore da direita da altura da árvore da esquerda (contrário do apresentado no texto).

exercício 31 Construa os algoritmos `BalancearEsq` e `BalancearDir` utilizados no Algoritmo 6.22.

exercício 32 Escreva um algoritmo que verifica se uma determinada árvore binária é ou não AVL.

exercício 33 Construa um algoritmo para remover uma folha de uma árvore AVL, cuidando para que ela continue sendo AVL.

exercício 34 Construir (fazer o desenho) uma árvore binária com comprimento de caminho ponderado mínimo, utilizando o algoritmo de Huffman. Os valores contidos nas folhas desta árvore devem ser os seguintes:

```
3 – 5 – 7 – 11 – 14 – 24 – 29 – 35
```

exercícios de aplicações

exercício 35 Considere as informações relativas aos medicamentos de uma farmácia organizadas em uma árvore binária. O endereço da raiz desta árvore é `FARM`, e os nodos apresentam dois campos de informação, além dos endereços de seus dois descendentes:

```
TipoNodo = registro
             Nome: string;
             Preço: real;
             Esq, Dir: TipoPtNodo
           fim registro
```

Construa, com e sem recursividade, os seguintes algoritmos:

- um algoritmo para imprimir todos os nomes de todos os medicamentos da farmácia, percorrendo a árvore em caminhamento central à direita;
- uma função para devolver o `Preço` de um medicamento cujo `Nome` é passado como parâmetro. Caso não encontre este nome em algum nodo da árvore, a função deve devolver o valor zero.

exercício 36 Considere as informações relativas aos produtos de uma loja organizadas em uma árvore binária cujos nodos apresentam os seguintes campos: `ESQ` (ponteiro para o descendente da esquerda), `COD` (inteiro), `NOME` (string), `VALOR` (valor numérico real) e `DIR` (ponteiro para o descendente da

direita). O endereço da raiz desta árvore está na variável PROD. Para esta árvore, construa:

- uma função que conte quantos produtos tem VALOR superior a um valor N passado como parâmetro;
- um algoritmo que liste os códigos e os valores de todos os produtos da árvore;
- uma função que informe o código e o valor de um produto identificado por seu código;
- um algoritmo que altere o campo VALOR do determinado produto para um novo valor fornecido como parâmetro. O produto a ser modificado é identificado pelo seu campo COD, também passado como parâmetro.

exercício 37 Considere que uma empresa organize suas vendas em uma árvore binária cujos nodos são formados por:

NOME	CÓDIGO	DATA	PRODUTO	VALOR	ESQ	DIR

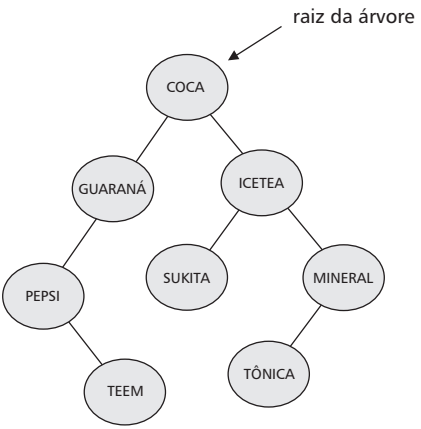
A raiz da árvore está no ponteiro RAIZ. Escreva algoritmos para:

- percorrer a árvore e somar o total de vendas da empresa, ou seja, somar todos os campos VALOR;
- listar os dados dos nodos de datas posteriores a 01/01/2007;
- eliminar o nodo relativo a um determinado código COD. Para evitar a reorganização da árvore, substitua o código por um valor que indique que o nodo está livre. Tome cuidado para zerar o valor contido no campo VALOR;
- inserir um novo produto na árvore. Para isto, inicialmente percorra a árvore verificando se existe algum nodo alocado e não utilizado (resultante da remoção de algum produto). Caso não seja encontrado um nodo livre, aloque o novo nodo como uma folha.

exercício 38 Uma expressão aritmética em notação polonesa é definida recursivamente da seguinte maneira: "*Uma expressão, em notação polonesa, consiste em um operando ou, então, em um operador seguido por duas expressões em notação polonesa.*" Considerando esta definição, toda expressão aritmética pode ser escrita de forma não ambígua em notação polonesa, dispensando-se o uso de parênteses. Escreva um algoritmo para transformar uma dada expressão aritmética na sua correspondente em notação polonesa.

exercício 39 Dada uma árvore binária que representa uma expressão aritmética, considerando apenas operações binárias, gerar a mesma expressão em notação completamente parentizada.

exercício 40 Considere a árvore binária abaixo:



Implemente esta árvore usando ponteiros. Além do nome de um refrigerante, cada nodo tem seu preço de venda e o número de unidades em estoque. Em seguida, implemente algoritmos para executar as seguintes operações:

- imprimir todos os dados da árvore, usando cada um dos caminhamentos. Teste com e sem recursividade;
- alterar o preço de um refrigerante cujo nome é passado como parâmetro. O novo valor também deve ser passado como parâmetro;
- informar quantos refrigerantes tem mais de 20 unidades;
- informar os descendentes (todos) de um determinado refrigerante (nome passado como parâmetro);
- informar qual o nodo *pai* de um determinado nodo da árvore (com exceção da raiz).

exercício 41 Implemente uma aplicação para identificar os erros ortográficos mais freqüentes em um dado texto. Devem ser fornecidos:

- arquivo texto referente ao dicionário;
- arquivo texto referente ao texto a ser analisado;
- uma constante K que indicará os erros mais freqüentes.

A aplicação deverá listar os k erros mais freqüentes, por ordem decrescente de freqüência e, dentro desta, por ordem lexicográfica. A listagem deverá ter o seguinte formato (exemplo com K=4):

Número (K)	Erro	Freqüência	Ocorre nas linhas
1	Alunso	10	10–30–50–55–78-92*-100*-125
2	Uiversidade	7	9-10*-88*-200
3	Anoo	5	2-30*-95-160
4	Imformática	5	10-55-80-100-120

O símbolo *, após um número de linha, indica que existe mais do que uma ocorrência do erro nessa linha.

exercício 42 Uma lista encadeada armazena a quantidade de unidades de cada produto disponíveis em uma loja. Cada nodo da lista é constituído de NOME (nome do produto), COD (código do produto) e NR (número de itens disponíveis para venda) e ELO (campo de elo para o próximo nodo da lista). O endereço da lista está na variável-apontador LISTA. Os nodos não apresentam ordenação. Pede-se:

- construa um algoritmo que organize as informações desta lista em uma ABP, ordenada de acordo com o campo COD. O procedimento deverá ter dois parâmetros: o ponteiro LISTA e um segundo ponteiro ARVORE, onde será devolvido o endereço da raiz da árvore. A lista original não deverá ser destruída nem modificada;
- construa uma função que devolva o número de itens disponíveis de um determinado produto, buscando-o na ABP;
- compare a eficiência dos procedimentos de consulta nas duas estruturas – na lista e na árvore.

anexo

pseudolinguagem utilizada neste texto

Na apresentação dos algoritmos ao longo deste texto é utilizada uma pseudolinguagem baseada no paradigma procedimental de programação. As construções desta pseudolinguagem podem ser facilmente identificadas nas linguagens de programação usuais, como PASCAL e C. A seguir, são apresentadas resumidamente as construções da pseudolinguagem utilizada.

Estrutura básica de um algoritmo. Todo algoritmo tem um nome. Para permitir a identificação dos algoritmos a partir da `Lista de Algoritmos`, procurou-se dar a eles nomes que dêem uma indicação de sua finalidade. Além disso, a Lista de Algoritmos traz uma breve descrição da finalidade de cada um.

Logo abaixo do nome do algoritmo são listados seus parâmetros de entrada (valores que ele deve receber para poder executar as ações requeridas), os parâmetros de saída (através dos quais os valores calculados são devolvidos ao programa de aplicação) e as variáveis locais necessárias à execução. A cada parâmetro e variável definido é associado um tipo de dado.

Em seguida são apresentados os comandos que compõem cada algoritmo, delimitados pelas palavras reservadas `início` e `fim`. A separação entre comandos é definida pelo contexto e pela indentação utilizada na apresentação dos algoritmos.

Para efeitos de padronização do texto, os nomes dos algoritmos, de todas as variáveis e dos tipos de dados definidos iniciam sempre com letra maiúscula. As palavras reservadas são representadas em letra minúscula.

Tipos de dados. Os tipos de dados primitivos, usuais em linguagens de programação, são utilizados diretamente nos algoritmos (`inteiro`, `real`, `caractere`, `string`, `lógico`). A indicação do tipo de dado é feita logo após o nome do parâmetro ou variável local correspondente, entre parêntesis. A utilização de ponteiros é detalhada mais adiante.

Os valores das variáveis lógicas são representados pelas palavras reservadas `verdadeiro` e `falso`.

São também empregados tipos definidos pelo usuário, compreendendo estes os tipos estruturados básicos presentes nas linguagens de programação, como `arranjo` e `registro`. A definição de um `arranjo` requer a indicação dos limites dos índices de acesso aos elementos, e a definição do tipo destes elementos. Para um `registro` devem ser definidos os nomes de seus campos, assim como os tipos de cada um deles. Exemplos:

```
TipoLista = arranjo [1.. 10] de TipoNodo
TipoNodo = registro
             A: inteiro
             B: lógico
           fim registro
```

Referência a variáveis estruturadas. A referência a um elemento de um arranjo é feita através do nome da variável do tipo arranjo, seguido de uma expressão cujo resultado define um índice, entre colchetes. Um campo de um registro é acessado separando com um ponto o nome da variável do tipo registro do nome do campo considerado. Exemplos:

```
NomeVarArr[10]
Reg.A
```

Ponteiros para variáveis dinâmicas. A definição de uma variável do tipo ponteiro é feita indicando qual o tipo de dado para o qual este ponteiro pode ser utilizado. A declaração de um ponteiro é feita através do símbolo "↑" seguido do tipo citado. Por exemplo, a declaração:

```
TipoPtNodo = ↑inteiro
```

define uma variável do tipo `TipoPtNodo` que somente poderá "apontar" (conter endereços de) para variáveis do tipo inteiro.

É adotada a palavra reservada `nulo` para indicar que uma variável dinâmica não contém um endereço válido, ou seja, que contém um endereço nulo. Este valor (`nulo`) pode somente ser testado.

Expressões aritméticas. Expressões aritméticas são escritas utilizando valores literais, os nomes das variáveis e dos parâmetros, e operadores aritméticos ("+", "-", "*", "/", div, etc.), podendo ser utilizados parêntesis. Exemplo:

```
(A + 2) * B
```

Expressões lógicas. Expressões lógicas são compostas utilizando valores literais, os nomes das variáveis e dos parâmetros, e os operadores relacionais ("<", ">", "=", etc.) e lógicos (`e`, `ou` e `não`). São utilizados parêntesis para facilitar a leitura das expressões. Exemplo:

```
(A = B) e (X < Y) ou (Z > 10)
```

Comandos de entrada e saída. As palavras `ler` e `escrever` identificam respectivamente os comandos de entrada e de saída de um programa de aplicação. Em seguida, entre parêntesis, é definida a lista de variáveis, separadas por vírgulas, que devem ser preenchidas por leitura ou cujos valores devem ser fornecidos como saída. Tratando-se de uma pseudo-linguagem, não é considerada formatação de dados. Exemplo:

```
escrever(A)
```

Comando de atribuição. Para representar uma atribuição é utilizado o símbolo "←", representando que a expressão à direita é atribuída à variável à esquerda. Os tipos do resultado da expressão e da variável que receberá o valor devem ser compatíveis. Exemplos:

```
A ← Info
PtNodo ← nulo
```

Comando composto. Sempre que a sintaxe exigir um só comando, poderá ser utilizado um comando composto, formado de um conjunto de comandos delimitados pelas palavras reservadas `início` e `fim`.

Comandos condicional e de seleção. O comando condicional apresenta as cláusulas `se` / *`então`*, condicionando a execução de um comando à vali-

dade da expressão lógica testada. O comando de seleção dupla acrescenta a cláusula `senão`, representando desta forma a seleção entre dois comandos, um dos quais será executado. Somente um comando segue as cláusulas `então` e `senão`, podendo este ser um comando composto. Exemplo:

```
se A = B
então Sucesso ← falso
senão início
      se I = 0
      então I ← F
      X ← 0
      fim
```

Neste exemplo, a cláusula `senão` é composta por dois comandos, o comando condicional `se / então`, seguido de um comando de atribuição.

A seleção entre um de vários comandos é representada por um comando que inicia com o cabeçalho `caso` <expressão> `seja,` seguido de uma lista de comandos com rótulos. Somente um destes comandos vai ser executado – aquele cujo rótulo for igual ao valor da expressão do cabeçalho. O final deste comando é indicado pelas palavras reservadas `fim caso`. Exemplo:

```
caso A seja
   1: X ← 0
   2: início
      ler(A)
      escrever(A)
      fim
   3: X ← Z
fim caso
```

`Comandos de repetição.` São utilizados os três comandos de repetição presentes na maioria das linguagens de programação. O comando `para / faça` define um número fixo de repetições, controlado por uma variável que é inicializada com o valor fornecido, incrementada a cada repetição de acordo com o incremento definido, e testada para verificar o final das repetições. Um só comando deve ser repetido, podendo ser utilizado um comando composto. Exemplo:

```
para Ind de Fim incr –1 até Ini faça
     início
     A ← A + 1
     L[Ind+1] ← L[Ind]
     fim
```

O comando `enquanto` / `faça` repete um comando enquanto a expressão fornecida for verdadeira. Já o comando `repita` / `até` repete um conjunto de comandos até que a expressão se torne verdadeira. Exemplos:

```
enquanto I ≤ F
faça início
     ler(A)
     I ← I+1
     fim
repita
   ler(A)
   escrever(A)
até que A = 0
```

Comandos de alocação e liberação de variáveis dinâmicas. A alocação e a liberação de variáveis dinâmicas são executadas respectivamente através dos comandos `alocar` e `liberar`, ambos apresentando uma variável do tipo ponteiro como parâmetro.

Acesso a variáveis dinâmicas. Uma variável dinâmica é referenciada através do ponteiro que guarda seu endereço. Na pseudo-linguagem aqui utilizada, o acesso a uma variável dinâmica é feito através do nome do ponteiro seguido do símbolo que o identifica como ponteiro "↑". A referência pode ser feita a uma variável simples, a um elemento de um arranjo, ou a um campo de um registro.

Funções. Neste texto são apresentados diversos algoritmos representando subprogramas, sendo especificados seus parâmetros de entrada e saída. Em alguns casos é explicitamente definido que o algoritmo apresentado corresponde a uma função. As `funções` são utilizadas para calcular um determinado valor, identificado no seu cabeçalho como "retorno", sendo definido o tipo deste valor retornado.

leituras recomendadas

AHO, Alfred V. et al. Data structures and algorithms. Indianapolis: Addison Wesley, 1983.

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução a estruturas de dados:* com técnicas de programação em C. Rio de Janeiro: Campus, 2004.

COLLINS, William J. *Programação estruturada com estudos de casos em Pascal*. São Paulo: McGraw Hill,1988. /

CORMEN, Thomas H. et al. *Algoritmos:* teoria e prática. Rio de Janeiro: Campus, 2002.

DROZDEK, Adam. *Estrutura de dados e algoritmos em C++.* São Paulo: Thomson, 2002.

FORBELLONE, André L.; EBERSPÄCHER, Henri F. *Lógica de programação:* a construção de algoritmos e estrutura de dados**.** São Paulo: Makron Books, 1993.

GOODRICH, Michael T.; TAMASSIA, Roberto. *Estruturas de dados e algoritmos em Java.* 4. ed. Porto Alegre: Bookman, 2007.

GUIMARÃES, Ângelo M.; LAGES, Newton A. C. *Algoritmos e estruturas de dados*. Rio de Janeiro: LTC, 1994.

HOROWITZ, Ellis; SAHNI, Sartaj. *Fundamentos de estruturas de dados*. 2. ed. Rio de Janeiro: Campus, 1987.

KNUTH, Donald E. *Fundamental algorithms*. Indianapolis: Addinson-Wesley, 1968. (The art of computer programming, v. 1)

KNUTH, Donald E. *Sorting and searching*. Indianapolis: Addison-Wesley, 1973. (The art of computer programming, v. 3)

LAFORE, Robert. *Estruturas de dados e algoritmos em java*. Rio de Janeiro: Ciência Moderna, 2004.

LORENZI, Fabiana; MATTOS, Patrícia N.; CARVALHO, Tanisi P. *Estruturas de dados*. São Paulo: Thompson, 2006.

MORAES, Celso R. *Estruturas de dados e algoritmos:* uma abordagem didática. São Paulo: Futura, 2003.

PEREIRA, Silvio L. *Estruturas de dados fundamentais:* conceitos e aplicações. São Paulo: Érica, 2004.

PINTO, Wilson S. *Introdução ao desenvolvimento de algoritmos e estrutura de dados*. São Paulo: Érica, 1990.

SANTOS, Clesio Saraiva; EDELWEISS, Nina. *Estruturas de dados*. Porto Alegre: UFRGS, 2000. Apostila de série de livros didáticos do Instituto de Informática da UFRGS.

SZWARCFITER, Jayme Luiz; MARKENZON, Lílian. *Estruturas de dados e seus algoritmos*. Rio de Janeiro: LTC, 1994.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C.* São Paulo: Makron Books, 1995.

TERADA, Routo. *Desenvolvimento de algoritmos e estrutura de dados*. São Paulo: McGraw-Hill, 1991.

VELOSO, Paulo A. S. *Estruturação e verificação de programas com tipos de dados*. São Paulo: Edgar Blucher, 1987.

VELOSO, Paulo A. S. et al. *Estruturas de dados*. Rio de Janeiro: Campus, 1983.

VELOSO, Paulo A. S. Tipos (abstratos) de dados, programação, especificação, implementação. In: ESCOLA DE COMPUTAÇÃO, 5., 1986, Belo Horizonte.

WIRTH, Nicklaus. *Algoritmos & estruturas de dados*. Rio de Janeiro: LTC, 1999.

ZIVIANI, Nívio. *Projetos de algoritmos com implementações em Pascal e C*. São Paulo: Thomson, 2003.

índice