

tamente com os algoritmos, as estruturas de dados é de extrema importância na Ciência da Computação. Uma estrutura de dados é uma forma eficiente de armazenar e organizar os dados dentro do computador de que nossos algoritmos tirem o melhor proveito deles. Diferentes tipos de estruturas são adequados para diferentes tipos de aplicações, sendo algumas altamente especializadas para tarefas específicas. Por isso, é considerado por muitos um tópico de difícil aprendizado, seja devido à dificuldade de compreender certos conceitos (como ponteiros), seja pela falta de clareza com que a estrutura de dados é descrita.

No objetivo de simplificar o ensino da disciplina, André Backes apresenta neste livro uma nova abordagem que descompõe os conceitos de estrutura de dados por meio de diversos recursos didáticos e ilustrativos, incluindo exemplos e avisos que ressaltem os seus pontos-chave, e de suas implementações em linguagem C.

O livro traz o programa de um curso completo de estrutura de dados, tratando com simplicidade as estruturas mais complicadas até as mais básicas.

O texto para estudantes de graduação e pós-graduação em Computação, também pode ser utilizado por profissionais que trabalham com programação e profissionais de áreas não computacionais (biólogos, engenheiros, entre outros) que precisam utilizar estruturas de dados na programação de suas tarefas.



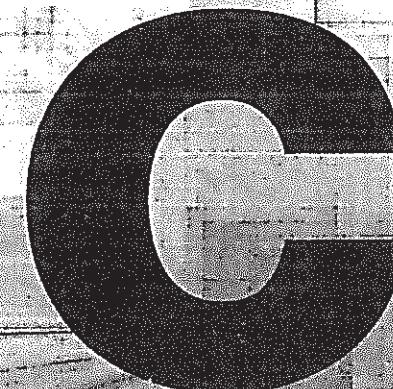
ELSEVIER

**Estrutura de dados descomplicada - em linguagem C**

ANDRÉ BACKES

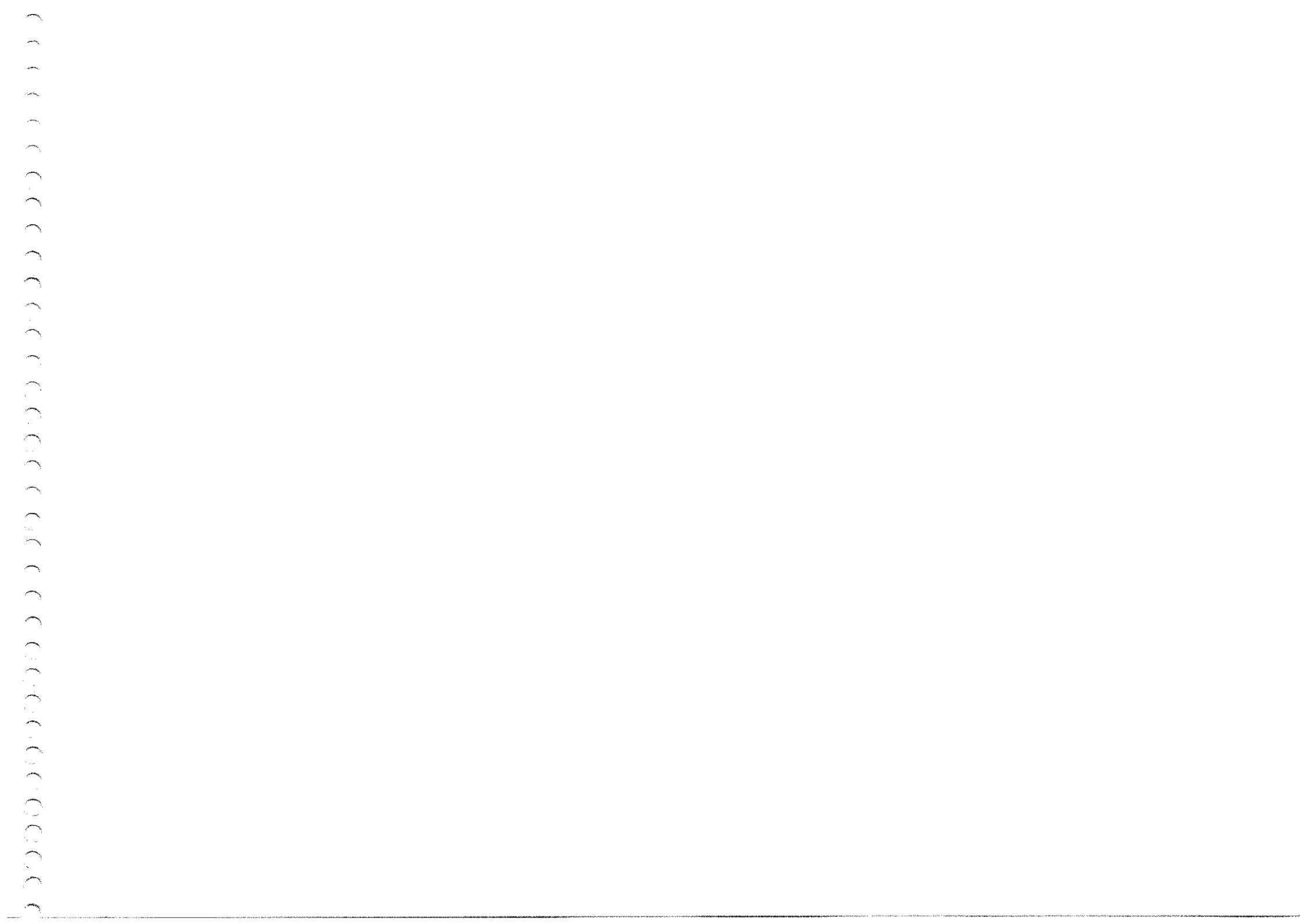
ANDRÉ BACKES

# Estrutura de dados descomplicada em linguagem



SEVIER

Material  
na WEB  
[www.elsevier.com.br](http://www.elsevier.com.br)



# **ESTRUTURA DE DADOS DESCOMPLICADA em linguagem C**

**André Ricardo Backes**

© Elsevier Editora Ltda.

os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser  
luzida ou transmitida sejam quais forem os meios empregados: eletrônicos,  
ílicos, fotográficos, gravação ou quaisquer outros.

Editor: Geisa Oliveira

Edição eletrônica: DTPhoenix Editorial

Design gráfico: Silvia Lima

Editora Ltda.

Avenida Presidente Vargas, 111 – 16º andar  
Centro de Setembro, 111 – 16º andar  
2006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Jardim Botânico, 753 – 8º andar  
01111 – Brooklin – São Paulo – SP – Brasil.

Serviço de Atendimento ao Cliente

0265340  
mento1@elsevier.com

978-85-352-8523-9  
versão digital); 978-85-352-8524-6

Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem  
existir erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses,  
contate a comunicação ao nosso Serviço de Atendimento ao Cliente, para que  
possamos esclarecer ou encaminhar a questão.

Nenhuma editora nem o autor assumem qualquer responsabilidade por eventuais danos ou  
perdas a pessoas ou bens, originados do uso desta publicação.

*Aos meus pais, que sempre me incentivaram e me  
apoiaram nos estudos, e à minha esposa Bianca,  
que preenche minha vida com alegria e felicidade.*

CIP-Brasil. Catalogação na publicação.  
Sindicato Nacional dos Editores de Livros, RJ

Backes, André Ricardo

Estrutura de dados descomplicada : em linguagem C / André Ricardo Backes.

– 1. ed. – Rio de Janeiro: Elsevier, 2016.

il.

ISBN 978-85-352-8523-9

1. C (Linguagem de programação de computador). 2. Programação  
(Computadores). I. Título.



## **AGRADECIMENTOS**

Aos amigos que fiz no convívio diário da Faculdade de Computação da Universidade Federal de Uberlândia (Facom/UFU).

Aos meus pais, que sempre me apoiaram em todas as etapas da minha vida.

Ao Prof. Dr. Odemir Martinez Bruno, pela confiança depositada em mim, orientação e amizade, sem as quais não teria me tornado um professor.

Aos meus alunos, os quais atuaram como um guia. Foi pensando em ajudá-los que me arrisquei a produzir esta obra.

A minha esposa Bianca, por me acompanhar a cada novo passo dado.

---

## O AUTOR

André Backes possui bacharelado em Informática, mestrado e doutorado em Ciências da Computação pela Universidade de São Paulo. Atualmente, é professor-adjunto da Universidade Federal de Uberlândia, revisor dos periódicos *IEEE Transactions on Image Processing* e *Pattern Recognition*. Tem experiência na área de Ciência da Computação, com ênfase em Metodologia e Técnicas da Computação, sendo seu principal tema o das áreas de processamento de imagens e visão computacional.

# SUMÁRIO

## CAPÍTULO 1

### INTRODUÇÃO

1

1.1 Algoritmos.	2
1.2 Estrutura de dados	3
1.3 Alocação de memória	3
1.3.1 Alocação estática	3
1.3.2 Alocação dinâmica	4
1.3.3 Alocação estática <i>versus</i> dinâmica	6

## CAPÍTULO 2

### ANÁLISE DA COMPLEXIDADE DE ALGORITMOS

7

2.1 Análise empírica	8
2.2 Análise matemática	8
2.2.1 Motivação	8
2.2.2 Contando instruções de um algoritmo	9
2.2.3 Custo dominante ou pior caso do algoritmo	10
2.2.4 Comportamento assintótico	11
2.2.5 A notação Grande- $O$	14
2.2.6 Os diferentes tipos de análise assintótica	15
A notação $\Omega$	16
A notação $O$	16
A notação $\Theta$	18
As notações $\sigma$ e $\omega$	19
2.2.7 Classes de problemas	19
2.3 Relações de recorrências	20
2.4 Exercícios	24

<b>CAPÍTULO 3</b>			
<b>ORDENAÇÃO E BUSCA EM ARRAYS</b>			
3.1 Definição	27	5.3 Operações básicas de uma lista	78
3.2 Algoritmos básicos de ordenação	27	5.3.1 A operação de inserção na lista	79
3.2.1 Ordenação por “bolha”: bubble sort	28	5.3.2 A operação de remoção da lista	79
3.2.2 Ordenação por seleção: selection sort	28	5.4 Lista sequencial estática	79
3.2.3 Ordenação por inserção: insertion sort	31	5.4.1 Definindo o tipo lista sequencial estática	81
3.3 Algoritmos sofisticados de ordenação	33	5.4.2 Criando e destruindo uma lista	82
3.3.1 Algoritmo merge sort	36	5.4.3 Informações básicas sobre a lista	84
3.3.2 Algoritmo quick sort	36	Tamanho da lista	84
3.3.3 Algoritmo heap sort	39	Lista cheia	85
3.4 Ordenação de um array de struct	43	Lista vazia	86
3.5 Ordenação externa	47	5.4.4 Inserindo um elemento na lista	87
3.5.1 Merge sort externo	49	Preparando a inserção na lista	87
Complexidade	50	Inserindo no início da lista	87
3.6 Busca em arrays	56	Inserindo no final da lista	88
3.6.1 Busca sequencial ou linear	57	Inserindo de forma ordenada na lista	90
3.6.2 Busca binária	57	5.4.5 Removendo um elemento da lista	91
3.6.3 Busca em um array de struct	60	Preparando a remoção da lista	91
3.7 Exercícios	61	Removendo do início da lista	91
	63	Removendo do final da lista	93
		Removendo um elemento específico da lista	94
		Otimizando a remoção da lista	96
		5.4.6 Busca por um elemento da lista	97
		Busca por posição na lista	97
		Busca por conteúdo na lista	98
		5.4.7 Análise de complexidade	99
<b>CAPÍTULO 4</b>		5.5 Lista dinâmica encadeada	100
<b>TIPO ABSTRATO DE DADOS – TAD</b>		5.5.1 Trabalhando com ponteiro para ponteiro	102
4.1 Definição	65	5.5.2 Definindo o tipo lista dinâmica encadeada	105
4.1.1 Vantagens de usar um TAD	65	5.5.3 Criando e destruindo uma lista	107
4.1.2 O tipo FILE	66	5.5.4 Informações básicas sobre a lista	109
4.1.3 Tipo opaco	67	Tamanho da lista	109
4.1.4 Operações básicas de um TAD	68	Lista cheia	110
4.2 Modularizando o programa	68	Lista vazia	110
4.3 Implementando um TAD: ponto	69	5.5.5 Inserindo um elemento na lista	111
4.4 Exercícios	70	Inserindo no início da lista	112
	74	Inserindo no final da lista	113
		Inserindo de forma ordenada na lista	114
<b>CAPÍTULO 5</b>		5.5.6 Removendo um elemento da lista	116
<b>LISTAS</b>	77	Preparando a remoção da lista	116
5.1 Definição	77		
5.2 Tipos de listas	78		

Removendo do início da lista	117	5.7.4 Inserindo um elemento na lista	164
Removendo do final da lista	118	Preparando a inserção na lista	164
Removendo um elemento específico da lista	120	Inserindo no início da lista	165
<b>5.5.7 Busca por um elemento da lista</b>	<b>121</b>	Inserindo no final da lista	166
Busca por posição na lista	121	Inserindo de forma ordenada na lista	168
Busca por conteúdo na lista	123	<b>5.7.5 Removendo um elemento da lista</b>	170
<b>5.5.8 Análise de complexidade</b>	<b>123</b>	Preparando a remoção da lista	170
<b>5.6 Lista dinâmica encadeada circular</b>	<b>124</b>	Removendo do início da lista	171
<b>5.6.1 Definindo o tipo lista dinâmica encadeada circular</b>	<b>126</b>	Removendo do final da lista	172
<b>5.6.2 Criando e destruindo uma lista</b>	<b>128</b>	Removendo um elemento específico da lista	174
<b>5.6.3 Informações básicas sobre a lista</b>	<b>130</b>	<b>5.7.6 Busca por um elemento da lista</b>	175
Tamanho da lista	131	Busca por posição na lista	176
Lista cheia	132	Busca por conteúdo na lista	177
Lista vazia	132	<b>5.7.7 Análise de complexidade</b>	177
<b>5.6.4 Inserindo um elemento na lista</b>	<b>133</b>	<b>5.8 Lista dinâmica encadeada com nó descritor</b>	178
Preparando a inserção na lista	133	<b>5.8.1 Definindo o tipo lista com nó descritor</b>	179
Inserindo no início da lista	134	<b>5.8.2 Criando e destruindo uma lista</b>	181
Inserindo no final da lista	135	<b>5.8.3 Informações básicas sobre a lista</b>	183
Inserindo de forma ordenada na lista	137	<b>5.8.4 Inserindo um elemento na lista</b>	184
<b>5.6.5 Removendo um elemento da lista</b>	<b>139</b>	Inserindo no início da lista	185
Preparando a remoção da lista	139	Inserindo no final da lista	186
Removendo do início da lista	140	<b>5.8.5 Removendo um elemento da lista</b>	187
Removendo do final da lista	142	<b>5.8.6 Análise de complexidade</b>	189
Removendo um elemento específico da lista	144	<b>5.9 Exercícios</b>	190
<b>5.6.6 Busca por um elemento da lista</b>	<b>146</b>		
Busca por posição na lista	146	<b>CAPÍTULO 6</b>	
Busca por conteúdo na lista	147		
<b>5.6.7 Análise de complexidade</b>	<b>148</b>	<b>FILAS</b>	<b>193</b>
<b>5.6.8 Aumentando o desempenho de uma lista circular</b>	<b>149</b>	6.1 Definição	193
Inserindo no início da lista	150	6.2 Tipos de filas	194
Inserindo no final da lista	152	6.3 Operações básicas de uma fila	195
Removendo do início da lista	154	6.4 Fila sequencial estática	195
<b>5.7 Lista dinâmica duplamente encadeada</b>	<b>155</b>	6.4.1 Definindo o tipo fila sequencial estática	196
<b>5.7.1 Definindo o tipo lista dinâmica duplamente encadeada</b>	<b>157</b>	6.4.2 Criando e destruindo uma fila	198
<b>5.7.2 Criando e destruindo uma lista</b>	<b>160</b>	6.4.3 Informações básicas sobre a fila	199
<b>5.7.3 Informações básicas sobre a lista</b>	<b>162</b>	Tamanho da fila	199
Tamanho da lista	162	Fila cheia	200
Lista cheia	163	Fila vazia	201
Lista vazia	164	6.4.4 Inserindo um elemento na fila	202

6.4.5 Removendo um elemento da fila	203	CAPÍTULO 8	
6.4.6 Consultando o elemento no início da fila	204	<b>PILHAS</b>	<b>241</b>
6.4.7 Análise de complexidade	205		
6.5 Fila dinâmica encadeada	205	8.1 Definição	241
6.5.1 Definindo o tipo fila dinâmica encadeada	206	8.2 Tipos de pilhas	242
6.5.2 Criando e destruindo uma fila	209	8.3 Operações básicas de uma pilha	242
6.5.3 Informações básicas sobre a fila	211	8.4 Pilha sequencial estática	243
Tamanho da fila	211	8.4.1 Definindo o tipo pilha sequencial estática	243
Fila cheia	212	8.4.2 Criando e destruindo uma pilha	245
Fila vazia	212	8.4.3 Informações básicas sobre a pilha	246
6.5.4 Inserindo um elemento na fila	213	Tamanho da pilha	247
6.5.5 Removendo um elemento da fila	215	Pilha cheia	248
6.5.6 Consultando o elemento no início da fila	216	Pilha vazia	249
6.5.7 Análise de complexidade	217	8.4.4 Inserindo um elemento na pilha	250
6.6 Criando uma fila usando uma lista	217	8.4.5 Removendo um elemento da pilha	251
6.7 Exercícios	219	8.4.6 Acessando o elemento no topo da pilha	252
		8.4.7 Análise de complexidade	253
CAPÍTULO 7		8.5 Pilha dinâmica encadeada	253
<b>FILAS DE PRIORIDADE</b>	<b>221</b>	8.5.1 Definindo o tipo pilha dinâmica encadeada	254
7.1 Definição	221	8.5.2 Criando e destruindo uma pilha	256
7.2 Operações básicas de uma fila	221	8.5.3 Informações básicas sobre a pilha	259
7.3 Implementação da fila de prioridades	222	Tamanho da pilha	259
7.4 Definindo o tipo fila de prioridades	223	Pilha cheia	260
7.5 Criando e destruindo uma fila	225	Pilha vazia	260
7.6 Informações básicas sobre a fila	226	8.5.4 Inserindo um elemento na pilha	261
7.6.1 Tamanho da fila	226	8.5.5 Removendo um elemento da pilha	263
7.6.2 Fila cheia	227	8.5.6 Acessando o elemento no topo da pilha	264
7.6.3 Fila vazia	227	8.5.7 Análise de complexidade	265
7.7 Fila de prioridade usando um array ordenado	228	8.6 Criando uma pilha usando uma lista	265
7.7.1 Inserindo um elemento na fila de prioridade	228	8.7 Exercícios	267
7.7.2 Removendo um elemento da fila de prioridade	230		
7.7.3 Consultando o elemento no início da fila de prioridade	231	CAPÍTULO 9	
7.8 Fila de prioridade usando uma heap binária	232	<b>TABELA HASH</b>	<b>269</b>
7.8.1 Inserindo um elemento na fila de prioridade	233		
7.8.2 Removendo um elemento da fila de prioridade	235	9.1 Definição	269
7.8.3 Consultando o elemento no início da fila de prioridade	237	9.2 Aplicações	271
7.9 Exercícios	238	9.3 Criando o TAD tabela hash	271
		9.3.1 Definindo o tipo tabela hash	271
		9.3.2 Criando e destruindo uma tabela hash	273

9.4 Calculando a posição da chave: função de hashing	275	10.4.2 Grafo completo	307
9.4.1 Método da divisão	276	10.4.3 Grafo regular	307
9.4.2 Método da multiplicação	277	10.4.4 Subgrafo	308
9.4.3 Método da dobra	277	10.4.5 Grafo bipartido	308
9.4.4 Tratando uma string como chave	279	10.4.6 Grafo conexo e desconexo	308
9.5 Inserção e busca sem colisão	280	10.4.7 Grafos isomorfos	309
9.6 Hashing universal	282	10.4.8 Grafo ponderado	310
9.7 Hashing perfeito e imperfeito	282	10.4.9 Grafo hamiltoniano	310
9.8 Tratamento de colisões	283	10.4.10 Grafo euleriano	311
9.8.1 Endereçamento aberto	284	10.5 Algoritmos de busca	311
Sondagem linear	284	10.5.1 Busca em profundidade	313
Sondagem quadrática	285	10.5.2 Busca em largura	316
Dúplo hash	286	10.5.3 Menor caminho entre dois vértices	319
Inserção e busca com tratamento de colisão	287	10.6 Árvore geradora mínima	323
9.8.2 Encadeamento separado	290	10.6.1 Algoritmo de Prim	325
9.9 Exercícios	291	10.6.2 Algoritmo de Kruskal	328
CAPÍTULO 10		10.7 Exercícios	331
<b>GRAFOS</b>			
10.1 Definição	293	<b>CAPÍTULO 11</b>	
10.2 Conceitos básicos	293	<b>ÁRVORES</b>	335
10.2.1 Vértice	294	11.1 Definição	335
10.2.2 Arestas: conectando os vértices	294	11.2 Conceitos básicos	336
10.2.3 Direção das arestas: grafos e digrafos	295	11.2.1 Notação	336
10.2.4 Grau de um vértice	295	11.2.2 Grau do nó e subárvore	337
10.2.5 Laços	296	11.2.3 Altura e nível da árvore	337
10.2.6 Caminhos e ciclos	297	11.3 Tipos de árvore	338
10.2.7 Arestas múltiplas e multigrafo	297	11.4 Árvore binária	338
10.3 Representação de grafos	298	11.4.1 Tipos de árvore binária	339
10.3.1 Matriz de adjacência	299	11.4.2 Tipos de implementação	340
10.3.2 Listas de adjacência	300	Implementação usando um array (heap)	340
10.3.3 Criando o TAD grafo	301	Implementação usando acesso encadeado	341
Definindo o tipo grafo	301	11.4.3 Criando a TAD árvore binária	342
Criando e destruindo um grafo	302	Definindo o tipo árvore binária	342
Inserindo uma aresta no grafo	304	Entendendo a notação de ponteiros da árvore	344
Removendo uma aresta do grafo	305	Criando e destruindo uma árvore	345
10.4 Tipos de grafos	306	Informações básicas sobre a árvore	347
10.4.1 Grafo trivial e simples	306	Percorrendo uma árvore	351
		Inserindo ou removendo um nó da árvore	354

11.5 Árvore binária de busca	354
11.5.1 Definição	354
11.5.2 Inserindo um nó na árvore	355
11.5.3 Removendo um nó da árvore	358
11.5.4 Consultando um nó da árvore	360
11.6 Árvore binária de busca balanceada	363
11.7 Árvore AVL	364
11.7.1 Definição	364
11.7.2 Implementando uma árvore AVL	366
11.7.3 Rotações	368
Rotação LL	369
Rotação RR	371
Rotação LR	372
Rotação RL	374
Quando usar cada tipo de rotação?	376
11.7.4 Inserindo um nó na árvore	377
11.7.5 Removendo um nó da árvore	382
11.8 Árvore rubro-negra	387
11.8.1 Definição	387
11.8.2 Diferença entre as árvores AVL e rubro-negra	388
11.8.3 Árvore rubro-negra caída para a esquerda	389
11.8.4 Implementando uma árvore rubro-negra	390
11.8.5 Acessando e mudando a cor dos nós	392
11.8.6 Rotações	393
11.8.7 Movendo os nós vermelhos	395
11.8.8 Inserindo um nó na árvore	398
11.8.9 Removendo um nó da árvore	403
11.9 Exercícios	406
CAPÍTULO 12	
<b>USOS E APLICAÇÕES DAS ESTRUTURAS DE DADOS</b>	<b>409</b>

**CAPÍTULO 1**

# Introdução

Uma estrutura de dados é uma forma eficiente de armazenar e organizar os dados dentro do computador para que nossos algoritmos tirem o melhor proveito deles. Porém, muitas pessoas consideram o seu estudo de difícil aprendizado, seja pela dificuldade de compreender certos conceitos (como os ponteiros), seja pela falta de clareza com que a estrutura de dados é descrita.

Nesse sentido, pretende-se uma nova abordagem que descomplique os conceitos dos diferentes tipos de estrutura de dados por meio de lembretes e avisos que ressaltem os seus pontos-chave, além de suas respectivas implementações em linguagem C, uma das mais bem-sucedidas linguagens de alto nível já criadas, considerada uma das linguagens de programação mais utilizadas de todos os tempos.

Este livro foi desenvolvido como um curso completo de estrutura de dados. Sua elaboração se baseou nas principais estruturas de dados utilizadas pelos mais diversos cursos de graduação em Ciências de Computação, Sistemas de Informação e Engenharia de Computação. Priorizaram-se a simplicidade na abordagem dos tópicos e a implementação das estruturas. Desse modo, espera-se que o livro possa ser facilmente utilizado tanto por profissionais que trabalhem com programação quanto por aqueles de áreas não computacionais.

Quanto à estrutura, adota-se a seguinte forma: no Capítulo 1, apresenta-se uma breve descrição do que é um algoritmo, do que são estruturas de dados e das diferenças entre alocação estática e dinâmica de memória.

O Capítulo 2 mostra as definições e conceitos sobre a análise de complexidade de algoritmos, a diferença entre análise empírica e análise matemática, o custo dominante de um algoritmo, além dos diferentes tipos de análise assintótica e classes de problemas. Já no Capítulo 3, descreve-se o problema de ordenação e busca de dados em arrays. São apresentados desde algoritmos básicos até algoritmos sofisticados. Também é abordada a ideia da ordenação externa (em disco).

Trata-se da criação de novos tipos de dados pelo programador, no Capítulo 4. Neste capítulo, são apresentados os conceitos necessários para a criação de um tipo abstrato de dado e modularização de programas.

O Capítulo 5 tem como tema as listas, estruturas de dados lineares utilizadas para armazenar e organizar elementos do mesmo tipo. São apresentados as operações possíveis com estas estruturas de dados, assim como os diferentes tipos de implementação e suas diferenças.

Nos Capítulos 6 e 7, são descritas as filas e as filas de prioridade, uma variação da fila convencional. Já no Capítulo 8, trata-se da implementação das pilhas. As filas e as pilhas, assim como as listas, armazenam uma sequência de elementos. Mas, diferente das listas, os elementos da fila e da pilha obedecem a uma ordem de entrada e saída.

A definição e a implementação de uma tabela hash é o tema do Capítulo 9. Nele, trata-se ainda dos tipos de função de hashing existentes e do problema da colisão de chaves.

Na sequência, o Capítulo 10 aborda os conceitos básicos de grafos, as diferentes formas de representação, os tipos de grafos, além de exemplificar alguns algoritmos de busca em grafos.

Por fim, o Capítulo 11 apresenta o conceito de árvores, os vários tipos de árvores existentes, como funciona uma árvore binária de busca, a importância do balanceamento da árvore e como se implementa uma árvore binária balanceada.

## 1.1 ALGORITMOS

Sempre que queremos resolver um problema no computador, precisamos, antes de tudo, descrevê-lo de uma forma clara e precisa. Ou seja, precisamos escrever o seu algoritmo.



Um algoritmo pode ser definido como uma sequência simples e objetiva de **instruções** para solucionar determinado problema. Cada **instrução** é uma informação que indica ao computador uma ação básica a ser executada.

Um algoritmo é uma sequência de instruções que realizam uma tarefa específica e é, frequentemente, ilustrado como uma receita, por exemplo, de bolo:

- Aqueça o forno a 180°C
- Unte uma forma redonda
- Numa taça
  - Bara
    - 75 g de manteiga
    - 250 g de açúcar
  - até ficar cremoso
  - Junte
    - 4 ovos, um a um
    - 100 g de chocolate derretido
  - Adicione aos poucos 250 g de farinha peneirada
  - Deite a massa na forma
  - Leve ao forno durante 40 minutos



A sequência de instruções que define o algoritmo deve ser sempre **finita** e também não pode ser **ambígua**.

Isso significa que nosso algoritmo deve sempre ter um fim (ou seja, deve terminar após certo número de passos) e que cada instrução deve ser precisamente definida (ou seja, não deve

permitir mais de uma interpretação de seu significado). Além disso, os algoritmos se baseiam no uso de um conjunto de instruções bem definido, que constituem um vocabulário de símbolos limitado (por exemplo, o conjunto de palavras reservadas da linguagem).

Um algoritmo é um procedimento computacional composto de três partes:

- **Entrada de dados:** são os dados do algoritmo informados pelo usuário.
- **Processamento de dados:** são os procedimentos utilizados para chegar ao resultado. É responsável pela obtenção dos dados de saída com base nos dados de entrada.
- **Saída de dados:** são os dados já processados, apresentados ao usuário.



Muitas vezes, um mesmo problema pode ser resolvido por vários algoritmos.

Os algoritmos se diferenciam uns dos outros pela maneira como utilizam os recursos do computador. Existem algoritmos que dependem principalmente do tempo que levam para ser executados, enquanto outros dependem da quantidade de memória do computador. Um exemplo de problema em que existem vários algoritmos é a ordenação de números.

## 1.2 ESTRUTURA DE DADOS

Em computação, uma **estrutura de dados** é uma forma de armazenar e organizar os dados de modo que possam ser usados de forma eficiente. Uma **estrutura de dados** é um relacionamento lógico entre diferentes tipos de dados visando à resolução de determinado problema de forma eficiente.

As **estruturas de dados** são utilizadas em diversas áreas do conhecimento e podem possuir diferentes propósitos. Trata-se de um tema fundamental da ciência da computação, pois a organização de forma coerente dos dados permite a diminuição do custo de execução de um algoritmo em termos de tempo de execução, consumo de memória ou ambos.

## 1.3 ALOCAÇÃO DE MEMÓRIA

### 1.3.1 Alocação estática

Na alocação estática de memória, o programador não precisa se preocupar em reservar memória para seus dados. A quantidade de memória necessária para armazenar as suas variáveis é automaticamente reservada na **stack** (pilha) ou em outras seções do programa. Do mesmo modo, o programador também não tem controle sobre o tempo de vida dessas variáveis na memória.



A **stack** guarda os dados alocados dentro dos escopos de funções: variáveis, locais, parâmetros, retorno de funções e endereços de outras áreas de código. Nela, as instruções e dados vão sendo empilhados e o desempilhamento ocorre automaticamente, após a execução.

Neste tipo de alocação, os dados estão organizados sequencialmente na memória do computador (como os arrays). Além disso, a quantidade total de memória utilizada pelo programa

é previamente conhecida e não pode ser mudada (não é possível redimensionar a memória). As variáveis são alocadas considerando toda a memória que seu tipo de dado necessita e não consideram a quantidade que seria realmente necessária na execução do programa. Consequentemente, podemos ter espaços alocados na memória que não são utilizados.

A Figura 1.1 mostra um exemplo passo a passo do funcionamento da **stack**. As variáveis são empilhadas na **stack** à medida que são declaradas na **main()** (Passos 1 e 2). Ao correr uma chamada de função, cria-se uma nova região na **stack** com os parâmetros locais (Passo 3). Variáveis declaradas dentro da função são empilhadas dentro da sua região da **stack** (Passo 4). Terminada a execução da função ela é removida da **stack** (desempilhada) e seu valor retornado para uma variável dentro da **main()**. A variável declarada para receber o retorno da função é empilhada dentro da região da **stack** associada a **main()** (Passo 5).

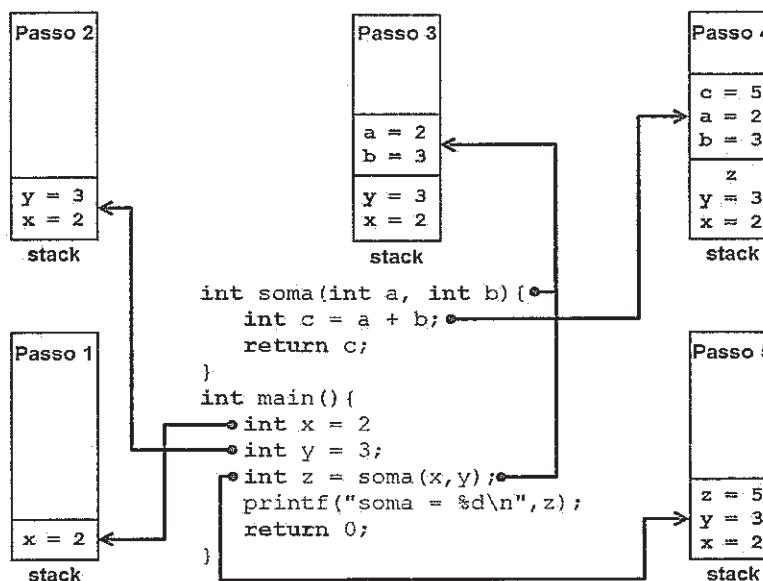


FIGURA 1.1

### 1.3.2 Alocação dinâmica

Na alocação dinâmica de memória, o programador tem total controle sobre o tamanho e o tempo de vida das posições de memória dos seus dados. Ou seja, o programador é responsável por reservar a quantidade de memória necessária para seus dados. Essa memória não é mais reservada na **stack**, mas em outra área de memória, chamada **heap**. As variáveis armazenadas na **heap** não dependem do escopo e devem ser liberadas manualmente pelo programador. Se o programador se esquecer de liberar essa memória, um vazamento de memória (memory leak) poderá ocorrer, causando uma falha no aplicativo.

A **heap** é uma seção do programa bem maior que o **stack**. É ideal para alocar muitos objetos e/ou objetos muito grandes. Nela, os dados são alocados dinamicamente por meio da função **malloc()** em C, e só podem ser acessados por ponteiros. Além de ser mais lenta que a **stack**, a remoção de objetos da **heap** é manual.

Nesta alocação, os dados não precisam estar organizados sequencialmente na memória do computador. Em geral, os blocos de memória alocados podem estar dispersos na memória do computador (estruturas encadeadas fazem melhor uso desse tipo de memória). Além disso, a quantidade total de memória utilizada pelo programa não precisa ser previamente conhecida. Os blocos de memória podem ser alocados, liberados e realocados para diferentes propósitos durante a execução do programa. O acesso aos dados agora é feito por ponteiros, que apontam para os blocos de memória alocados, e não mais por variáveis.

A Figura 1.2 mostra um exemplo passo a passo do funcionamento da **heap**. As variáveis são empilhadas na **stack** à medida que são declaradas na **main()** (Passo 1). Ao correr uma chamada da função **malloc()**, cria-se uma nova região de memória na **heap** e o ponteiro que aponta para essa região é empilhado na **stack** (Passos 2 e 3). Note que as regiões na **heap** não são escolhidas de forma sequencial. Quando a função **free()** é chamada, a região associada àquele ponteiro é liberada da **heap**, e o ponteiro ainda continua na **stack** (Passo 4). Ao final da **main()**, todas as variáveis são removidas da **stack**. Porém, uma região da **heap** continua presente, pois ela foi alocada, mas nunca liberada (Passo 5).

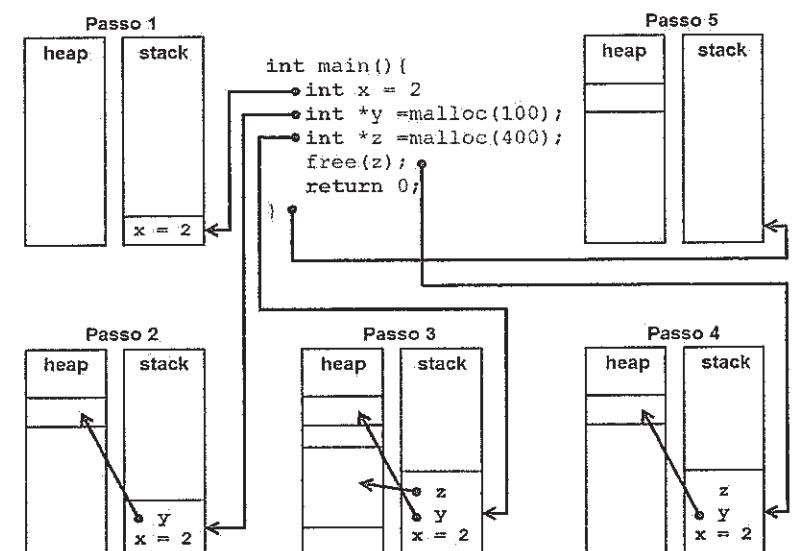


FIGURA 1.2

### 1.3.3 Alociação estática versus dinâmica

O Quadro 1.1 apresenta uma breve descrição das diferenças entre a alocação estática e a alocação dinâmica.

QUADRO 1.1

Alociação estática (stack)	Alociação dinâmica (heap)
Armazenado na memória RAM	Armazenado na memória RAM
Variáveis são liberadas automaticamente no final do escopo	Variáveis não dependem do escopo (podem ser acessadas globalmente) e devem ser liberadas manualmente
Alociação mais rápida que na heap	Alociação mais lenta que na stack
Implementado usando uma estrutura de dados do tipo pilha	Blocos de dados são alocados sob demanda
Armazena dados locais e endereços de retorno utilizados na passagem de parâmetros	Pode sofrer fragmentação após sucessivas alocações e liberações de memória
Os dados podem ser usados sem ponteiros	Os dados são acessados por ponteiros
Pode ocorrer estouro de pilha quando a pilha é muito usada	A alocação pode falhar se muita memória é solicitada
É usada quando se sabe exatamente o quanto de espaço será alocado antes do tempo de compilação e esse espaço não é muito grande	É usada quando não se sabe exatamente o quanto de espaço será alocado antes do tempo de compilação ou esse espaço é muito grande
Geralmente possui um tamanho máximo predeterminado quando o programa inicia	Responsável por vazamentos de memória

## CAPÍTULO 2

## Análise da complexidade de algoritmos

Em ciência da computação, a análise de algoritmos é a área de pesquisa cujo tema está nos algoritmos.



A análise de algoritmos busca responder à seguinte pergunta: podemos fazer um algoritmo mais eficiente?

Em programação, podemos resolver um problema de várias maneiras, isto é, é possível utilizar algoritmos diferentes para um mesmo problema.



Algoritmos diferentes, mas capazes de resolver o mesmo problema, não necessariamente o fazem com a mesma eficiência.

Essas diferenças de eficiência podem:

- Ser irrelevantes para um pequeno número de elementos processados.
- Crescer proporcionalmente com o número de elementos processados.



Para comparar a eficiência dos algoritmos foi criada uma medida chamada **complexidade computacional**.

Basicamente, a complexidade computacional indica o **custo** ao se aplicar um algoritmo, sendo:

$$\text{custo} = \text{memória} + \text{tempo}$$

em que o item **memória** indica quanto de espaço o algoritmo vai consumir e **tempo**, a duração de sua execução.

Para determinar se um algoritmo é o mais eficiente, podemos utilizar duas abordagens:

- **Análise empírica:** comparação entre os programas.
- **Análise matemática:** estudo das propriedades do algoritmo.

## 2.1 ANÁLISE EMPÍRICA

Basicamente, esse tipo de análise avalia o custo (ou a complexidade) de um algoritmo a partir da avaliação de sua execução quando implementado.



**Na análise empírica,** um algoritmo é analisado pela execução de seu programa correspondente.

A análise empírica possui uma série de vantagens. Por meio dela podemos:

- Avaliar o desempenho em determinada configuração de computador/línguagem.
- Considerar custos não aparentes. Por exemplo, o custo da alocação de memória.
- Comparar computadores.
- Comparar linguagens.



Apesar de suas vantagens, existem certas dificuldades na realização da análise empírica.

Algumas dessas dificuldades são:

- Necessidade de implementar o algoritmo. Isso depende da habilidade do programador.
- Resultado pode ser mascarado pelo hardware (computador utilizado) ou software (eventos ocorridos no momento de avaliação).
- Qual a natureza dos dados: reais, aleatórios ou perversos?



O uso de **dados aleatórios** permite avaliar o desempenho médio do algoritmo. **Dados perversos** permitem avaliar o desempenho no pior caso.

## 2.2 ANÁLISE MATEMÁTICA

### 2.2.1 Motivação

Muitas vezes, é preferível que a medição do tempo gasto por um algoritmo seja feita de maneira independente do hardware ou da linguagem usada na sua implementação. Nesse tipo de situação, convém utilizar a análise matemática do algoritmo.



A **análise matemática** permite um estudo formal de um algoritmo em termos de ideia. Ela faz uso de um computador idealizado e simplificações que buscam considerar somente os custos dominantes do algoritmo.

Nessa análise, estamos avaliando a ideia por trás do algoritmo. Para tanto, detalhes de baixo nível, como a linguagem de programação utilizada, o hardware no qual o algoritmo é executado ou o conjunto de instruções da CPU são ignorados. Esse tipo de análise permite entender como um algoritmo se comporta à medida que o conjunto de dados de entrada cresce. Assim, podemos expressar a relação entre o conjunto de dados de entrada e a quantidade de tempo necessário para processar esses dados.

### 2.2.2 Contando instruções de um algoritmo

Para entender como calcular o custo de um algoritmo, tome como exemplo o pequeno trecho de código mostrado na Figura 2.1. Este algoritmo procura o maior valor presente em um array A contendo n elementos e o armazena na variável M.

De posse deste trecho de código vamos contar quantas **instruções simples** ele executa.



**Uma instrução simples** é uma instrução que pode ser executada diretamente pelo CPU, ou algo muito perto disso.

No nosso trecho de código podemos encontrar os seguintes tipos de instruções:

- Atribuição de um valor a uma variável.
- Acesso ao valor de um determinado elemento do array.
- Comparação de dois valores.
- Incremento de um valor.
- Operações aritméticas básicas, como adição e multiplicação.

Por serem **instruções simples** todas elas possuem o mesmo custo. Além disso, vamos assumir que comandos de seleção (como o comando if) possuem custo zero, ou seja, não contam como instruções.

#### Exemplo de código

```

01 int M = A[0];
02
03 for(i = 0; i < n; i++) {
04     if(A[i] >= M) {
05         M = A[i];
06     }
07 }
```

FIGURA 2.1



O custo da linha 1 é de: 1 instrução.

Na **linha 1**, o valor da primeira posição do array é copiado para a variável M. Vamos considerar que essa tarefa requer apenas uma instrução para acessar o valor A[0] e atribuí-lo à variável M.



O custo da inicialização do laço for (linha 3) é de: 2 instruções.

No nosso algoritmo, o comando de laço **for** é utilizado para percorrer o array A. Porém, antes mesmo de percorrer o array, ele precisa ser inicializado ao custo de **uma instrução** ( $i = 0$ ). Além disso, mesmo que o array tenha tamanho zero (isto é, não possua nenhum elemento), ao menos uma comparação será executada ( $i < n$ ), o que custa mais **uma instrução**. Portanto, temos um total de duas instruções na inicialização do laço **for**. Perceba que essas instruções serão executadas antes da primeira iteração do laço **for**.



O custo para executar o comando de laço for (linha 3) é de:  $2n$  instruções.

Ao final de cada iteração do laço **for**, precisamos executar mais duas instruções: uma de incremento ( $i++$ ) e uma comparação para verificar se vamos continuar no laço **for** ( $i < n$ ). No nosso algoritmo, o comando de laço **for** será executado  $n$  vezes, que é o número de elementos no array A. Assim, essas duas instruções também serão executadas  $n$  vezes, ou seja, o seu custo será  $2n$  instruções.

Se ignorarmos os comandos contidos no corpo do laço **for**, teremos que o algoritmo precisa executar  $3 + 2n$  instruções:

- 3 instruções antes de iniciar o laço **for**.
- 2 instruções ao final de cada laço **for**, o qual é executado  $n$  vezes.

Assim, considerando um laço vazio, podemos definir uma função matemática que representa o custo do algoritmo em relação ao tamanho do array de entrada, ou  $f(n) = 2n + 3$ .

### 2.2.3 Custo dominante ou pior caso do algoritmo

Se desconsiderarmos os comandos no corpo do laço **for**, a análise do algoritmo contido na Figura 2.1 resulta que o mesmo possui um custo de  $3 + 2n$  instruções.



As instruções vistas até o momento são sempre executadas. Porém, as instruções dentro do **for** podem ou não ser executadas.

Vamos então contar as instruções restantes. Dentro do laço **for** temos um comando de seleção (**if**). Seu custo será de 1 instrução: uma única instrução será responsável pelo acesso ao valor do array e sua comparação ( $A[i] \geq M$ ).

Dentro do comando **if** temos mais uma instrução: aquela que acessa o valor do array e o atribui a outra variável ( $M = A[i]$ ). No entanto, essa instrução pode ou não ser executada e isso depende do resultado da comparação feita pelo comando **if**. Isso complica um pouco o cálculo do custo do algoritmo.



Antes, bastava saber o tamanho do array,  $n$ , para definir a função de custo  $f(n)$ . Agora, temos que considerar também o conteúdo do array.

Tome como exemplo dois arrays de mesmo tamanho

$$A1 = \{1, 2, 3, 4\}$$

$$A2 = \{4, 3, 2, 1\}$$

É fácil perceber que o array A1 irá precisar de mais instruções (o comando **if** é sempre **verdadeiro**) para achar o maior valor do que o array A2 (o comando **if** é sempre **falso**).



Ao analisarmos um algoritmo, é muito comum considerarmos o **pior caso** possível, ou seja, aquele em que o maior número de instruções é executado.

No nosso algoritmo, o **pior caso** ocorre quando o array possui valores em ordem crescente. Nesta situação, o valor de M é sempre substituído, o que resulta em um maior número de instruções. Ou seja, no **pior caso**, o laço **for** sempre executa as duas instruções. Assim, teremos que a função custo do algoritmo será  $f(n) = 2n + 2n + 3$ , ou  $f(n) = 4n + 3$ . Esta função representa o custo do algoritmo em relação ao tamanho do array ( $n$ ) de entrada no **pior caso**.

### 2.2.4 Comportamento assintótico

Vimos que o custo para o algoritmo mostrado na Figura 2.1 é dado pela função:

$$f(n) = 4n + 3$$

Essa é a função de complexidade de tempo. Ela nos dá uma ideia do custo de execução do algoritmo para um problema de tamanho  $n$ . É importante salientar que é possível criar o mesmo tipo de função para a análise do espaço gasto.



Será que todos os termos da função  $f$  são necessários para termos uma noção do custo do algoritmo?

De fato, nem todos os termos são necessários. Ou seja, podemos descartar certos termos na função e manter apenas os que nos dizem o que acontece com a função quando o tamanho dos dados de entrada ( $n$ ) cresce muito.



Se um algoritmo é mais rápido do que outro para um grande conjunto de dados de entrada, é muito provável que ele continue sendo mais rápido em um conjunto de dados menor.

Diante desse fato, podemos descartar todos os termos que crescem lentamente e manter apenas os que crescem mais rápido à medida que o valor de  $n$  se torna maior. Nossa função,  $f(n) = 4n + 3$ , possui dois termos:  $4n$  e  $3$ .



O termo  $3$  é simplesmente uma **constante de inicialização**, ou seja, não é afetado pelo valor de  $n$  e pode, portanto, ser descartado.

Como o termo  $3$  não se altera à medida que  $n$  aumenta, nossa função pode ser reduzida para  $f(n) \approx 4n$ .



Constantes que multiplicam o termo  $n$  da função também devem ser descartadas.

Ao descartarmos tais constantes, nossa função se torna muito simples,  $f(n) = n$ . Descartar esse tipo de constante faz sentido se pensarmos em diferentes linguagens de programação. Por exemplo, a seguinte linha de código em Pascal

```
M := A[i];
```

equivale ao seguinte código em linguagem C:

```
if(i >= 0 && i < n)
    M = A[i];
```

Como podemos ver, o acesso a um elemento do array em Pascal contém uma etapa de verificação para saber se aquela posição existe dentro do array, algo que não ocorre na linguagem C. Assim, ignorar essas constantes de multiplicação equivale a ignorar as particularidades de cada linguagem e compilador e analisar apenas a ideia do algoritmo.



Ao descartarmos todos os termos constantes e manter apenas o de maior crescimento, obtemos o **comportamento assintótico** do algoritmo. Chamamos de comportamento assintótico o comportamento de uma função  $f(n)$  quando  $n$  tende ao infinito.

Isso acontece porque o termo que possui o maior expoente domina o comportamento da função  $f(n)$  quando  $n$  tende ao infinito. Para entender melhor, considere duas funções de custo  $g(n) = 1000n + 500$  e  $h(n) = n^2 + n + 1$ . A Figura 2.2 mostra as curvas obtidas para essas duas funções à medida que  $n$  aumenta. Perceba que, apesar da função  $g(n)$  possuir constantes maiores multiplicando seus termos, existe um valor de  $n$  a partir do qual o resultado de  $h(n)$

é sempre maior do que  $g(n)$ , tornando os demais termos é constantes pouco importantes. Ou seja, podemos suprimir os termos menos importantes da função para considerar apenas o termo de maior grau. Assim, descreveremos a complexidade usando somente o seu custo dominante  $n$  para a função  $g(n)$  e  $n^2$  para  $h(n)$ .

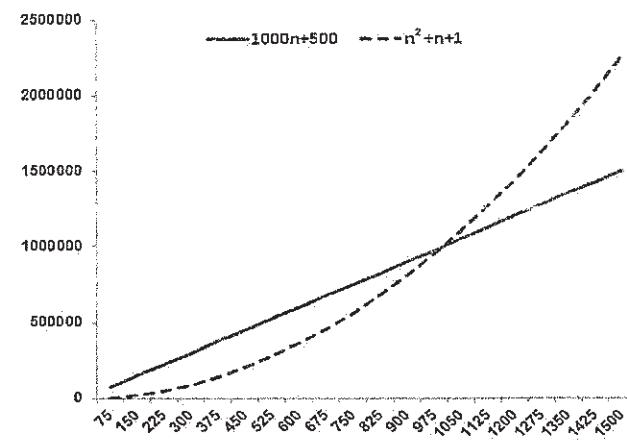


FIGURA 2.2

O Quadro 2.1 mostra alguns exemplos de função de custo junto com seus comportamentos assintóticos. Note que, se a função não possui nenhum termo multiplicado por  $n$ , seu comportamento assintótico é constante (1).

QUADRO 2.1

Função de custo	Comportamento assintótico
$f(n) = 105$	$f(n) = 1$
$f(n) = 15n + 2$	$f(n) = n$
$f(n) = n^2 + 5n + 2$	$f(n) = n^2$
$f(n) = 5n^3 + 200n^2 + 112$	$f(n) = n^3$



De modo geral, podemos obter a função de custo de um programa simples apenas contando os comandos de laços aninhados.

De fato, qualquer programa que não possua um laço será um programa com um número constante de instruções (exceto se houver recursão), ou seja,  $f(n) = 1$ . Um programa com um laço variando de 1 a  $n$  será  $f(n) = n$  (ou seja, um conjunto de instruções constantes antes e/ou depois do laço e um conjunto de instruções constante dentro do laço), dois comandos de laço aninhados terão comportamento assintótico  $f(n) = n^2$  e assim por diante.

## 2.2.5 A notação Grande-O

Dentre as várias formas de análise assintótica, a mais conhecida e utilizada é a notação Grande-O ( $O$ ). Ela representa o custo (seja de tempo ou de espaço) do nosso algoritmo no pior caso possível para todas as entradas de tamanho  $n$ .



A notação Grande-O ( $O$ ) analisa o pior caso de um algoritmo, ou seja, analisa o limite superior de entrada. Desse modo, podemos dizer que o comportamento do nosso algoritmo não pode nunca ultrapassar certo limite.

A ordenação de dados é um problema interessante para entendermos como funciona a notação Grande-O ( $O$ ), por se tratar de um problema comum em sistemas reais e por possuir uma grande variedade de algoritmos para resolvê-lo.

A Figura 2.3 apresenta a implementação de um método de ordenação: o **selection sort**. Dado um array  $V$  de tamanho  $n$ , este método procura o menor valor do array (posição  $me$ ) e o coloca na primeira posição. Em seguida, repete esse processo para a segunda posição do array, depois para a terceira etc. O processo continua até que todo o array esteja ordenado. Na sua implementação, podemos ver dois comandos de laço. Enquanto o laço externo é executado  $n$  vezes, o número de execuções do laço interno depende do valor do índice do primeiro laço. Assim, o laço interno é executado  $n-1$  vezes na primeira iteração do laço externo, depois  $n-2$  vezes e assim por diante, até que é executado apenas 1 vez.

```
Metodo selectionSort
01 void selectionSort(int *V, int n){
02     int i, j, me, troca;
03     for(i = 0; i < n-1; i++) {
04         me = i;
05         for(j = i+1; j < n; j++) {
06             if(V[j] < V[me])
07                 me = j;
08         }
09         if(i != me) {
10             troca = V[i];
11             V[i] = V[me];
12             V[me] = troca;
13         }
14     }
15 }
```

FIGURA 2.3



Para calcularmos o custo do **selection sort** temos que calcular o resultado da soma  $1 + 2 + \dots + (n - 1) + n$ . Esta soma representa o número de execuções do laço interno, algo que não é tão simples de se calcular.

Dependendo do algoritmo, calcular o seu custo exato pode ser uma tarefa muito complicada. No nosso caso, temos que  $1 + 2 + \dots + (n - 1) + n$  equivale à soma dos  $n$  termos de uma progressão aritmética,  $S(n)$ , de razão 1. Assim,

$$S(n) = 1 + 2 + \dots + (n - 1) + n$$

$$S(n) = n + (n - 1) + \dots + 2 + 1$$

$$2S(n) = (1 + n) + (2 + (n - 1)) + \dots + ((n - 1) + 2) + (n + 1)$$

Como  $1$  e  $n$ ,  $2$  e  $(n - 1)$ ... são termos equidistantes dos extremos, suas somas são iguais a  $(1 + n)$ ; logo:

$$2S(n) = (1 + n) + (2 + (n - 1)) + \dots + ((n - 1) + 2) + (n + 1)$$

$$2S(n) = n(1 + n)$$

$$S(n) = n(1 + n)/2$$

Então, temos que o número de execuções do laço interno é  $S(n) = n(1 + n)/2$ . Uma alternativa mais simples é estimar um **limite superior**. Isso pode ser feito alterando mentalmente o algoritmo e, em seguida, calculando o custo do novo algoritmo. A ideia é alterar o algoritmo para algo **menos eficiente** do que temos. Assim, saberemos que o algoritmo original é, no máximo, tão ruim ou talvez melhor que o novo algoritmo.



Uma forma de diminuirmos a eficiência do **selection sort** é trocar o laço interno (que muda de tamanho a cada execução do laço externo) por um laço que seja executado **sempre  $n$  vezes**.



Essa pequena alteração torna mais fácil descobrir o custo do algoritmo. Obviamente, ela também piora o desempenho do algoritmo, já que algumas execuções do laço interno serão inúteis. Como agora temos dois comandos de laço aninhados sendo executados  $n$  vezes cada, a função de custo do algoritmo passa a ser  $f(n) = n^2$ . Desse modo, utilizando a notação Grande-O, podemos dizer que o custo do algoritmo, no pior caso, é  $O(n^2)$ .



No nosso exemplo, a notação  $O(n^2)$  diz que o custo do algoritmo não é, assintoticamente, pior do que  $n^2$ . Em outras palavras, o custo do algoritmo original é, no máximo, tão ruim quanto  $n^2$ . Pode ser melhor, mas nunca pior.

Assim, com a notação Grande-O podemos estabelecer o **limite superior** para a complexidade real de um algoritmo. Isso significa que o nosso programa nunca vai ser mais lento do que um determinado limite.

## 2.2.6 Os diferentes tipos de análise assintótica

Existem várias formas de análise assintótica. Dentre elas, a mais conhecida e utilizada é a notação Grande-O. Ela representa a complexidade (seja de tempo ou de espaço) do nosso algoritmo no pior caso.



A notação **Grande-O** é a mais utilizada, pois é o caso de mais fácil identificação (limite superior sobre o tempo de execução do algoritmo). Para diversos algoritmos, o pior caso ocorre com frequência.

A seguir, são descritas outras formas de análise assintótica, matematicamente.

### A notação $\Omega$

A notação  $\Omega$  (lê-se grande-omega) descreve o **limite assintótico inferior** de um algoritmo. Trata-se de uma notação utilizada para analisar o **melhor caso** do algoritmo.



A notação  $\Omega(n^2)$  nos diz que o custo do algoritmo é, assintoticamente, maior ou igual a  $n^2$ . Em outras palavras, o custo do algoritmo original é, no **mínimo**, tão ruim quanto  $n^2$ .

Matematicamente, a notação  $\Omega$  é assim definida: uma função custo  $f(n)$  é  $\Omega(g(n))$  se existem duas constantes positivas  $c$  e  $m$ , tais que, para  $n \geq m$ , temos  $f(n) \geq cg(n)$ .

Em outras palavras, para todos os valores de  $n$  à direita do valor  $m$ , o resultado da nossa função custo  $f(n)$  é sempre **maior ou igual** ao valor da função usada na notação  $\Omega$ ,  $g(n)$ , multiplicada por uma constante  $c$ . Um exemplo da notação  $\Omega$  é mostrado na Figura 2.4.

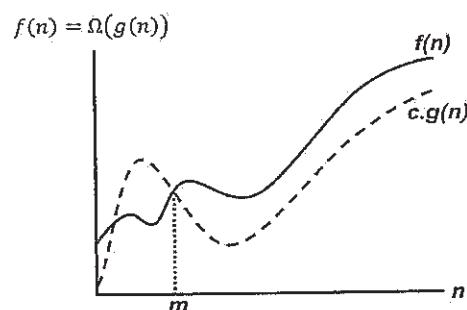


FIGURA 2.4

Um exemplo simples desse tipo de notação consiste em mostrar que a função custo do nosso algoritmo  $f(n) = 3n^3 + 2n^2$  é  $\Omega(n^3)$ . Para isso, basta considerar que  $c = 1$  e  $n \geq 0$ . Assim,  $3n^3 + 2n^2 \geq 1n^3$ .

### A notação O

A notação  $O$  (lê-se grande-o) descreve o **limite assintótico superior** de um algoritmo. Trata-se de uma notação utilizada para analisar o **pior caso** do algoritmo.



A notação  $O(n^2)$  nos diz que o custo do algoritmo é, assintoticamente, menor ou igual a  $n^2$ . Em outras palavras, o custo do algoritmo original é, no **máximo**, tão ruim quanto  $n^2$ .

Matematicamente, a notação  $O$  é assim definida: uma função custo  $f(n)$  é  $O(g(n))$  se existem duas constantes positivas  $c$  e  $m$ , tais que, para  $n \geq m$ , temos  $f(n) \leq cg(n)$ .

Em outras palavras, para todos os valores de  $n$  à direita do valor  $m$ , o resultado da nossa função custo  $f(n)$  é sempre **menor ou igual** ao valor da função usada na notação  $O$ ,  $g(n)$ , multiplicada por uma constante  $c$ . Um exemplo da notação  $O$  é mostrado na Figura 2.5.

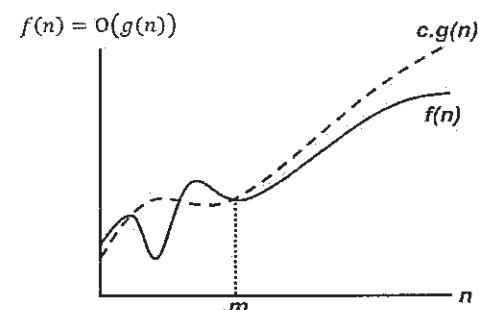


FIGURA 2.5

Um exemplo simples desse tipo de notação consiste em mostrar que a função custo do nosso algoritmo  $f(n) = 3n^3 + 2n^2$  é  $O(n^3)$ . Para isso, basta considerar que  $c = 6$  e  $n \geq 0$ . Assim,  $3n^3 + 2n^2 \leq 6n^3$ .



A notação **Grande-O** ( $O$ ) possui algumas operações, sendo a mais importante a **regra da soma**.

A **regra da soma** é muito importante na análise da complexidade de diferentes algoritmos em sequência. Basicamente, se dois algoritmos são executados em sequência, a complexidade da execução dos dois algoritmos será dada pela complexidade do maior deles, ou seja:

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

Por exemplo, se temos:

- Dois algoritmos cujos tempos de execução são  $O(n)$  e  $O(n^2)$ , a execução deles em sequência será  $O(\max(n, n^2))$ , que é  $O(n^2)$ .
- Dois algoritmos cujos tempos de execução são  $O(n)$  e  $O(n \log n)$ , a execução deles em sequência será  $O(\max(n, n \log n))$ , que é  $O(n \log n)$ .

## A notação $\Theta$

A notação  $\Theta$  (lê-se grande-theta) descreve o **limite assintótico firme** (ou estreito) de um algoritmo. Trata-se de uma notação utilizada para analisar o limite inferior e superior do algoritmo.



A notação  $\Theta(n^2)$  nos diz que o custo do algoritmo é, assintoticamente, igual a  $n^2$ . Em outras palavras, o custo do algoritmo original é  $n^2$  dentro de um fator constante acima e abaixo.

Matematicamente, a notação  $\Theta$  é assim definida: uma função custo  $f(n)$  é  $\Theta(g(n))$  se existem três constantes positivas  $c_1$ ,  $c_2$  e  $m$ , tais que, para  $n \geq m$ , temos  $c_1 g(n) \leq f(n) \leq c_2 g(n)$ .

Em outras palavras, para todos os valores de  $n$  à direita do valor  $m$ , o resultado da nossa função custo  $f(n)$  é sempre igual ao valor da função usada na notação  $\Theta$ ,  $g(n)$ , quando esta função é multiplicada por constantes  $c_1$  e  $c_2$ . Um exemplo da notação  $\Theta$  é mostrado na Figura 2.6.

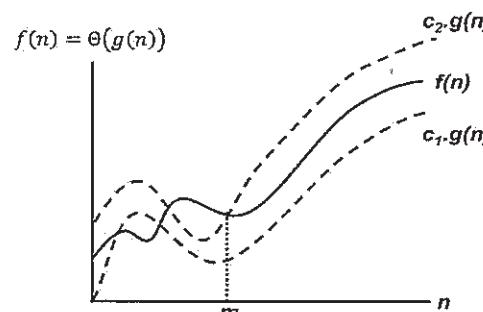


FIGURA 2.6

Um exemplo simples desse tipo de notação consiste em mostrar que a função custo do nosso algoritmo  $f(n) = \frac{1}{2}n^2 - 3n$  é  $\Theta(n^2)$ . Para tanto, iremos definir constantes positivas  $c_1$ ,  $c_2$  e  $m$ , tais que

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo  $n \geq m$ . Dividindo por  $n^2$  temos

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

A desigualdade do lado direito é considerada válida para  $n \geq 1$  quando escolhemos  $c_2 \geq 1/2$ . Já a desigualdade do lado esquerdo é considerada válida para  $n \geq 7$ , quando escolhemos  $c_1 \geq 1/14$ . Assim, basta considerar que  $c_1 \geq 1/14$ ,  $c_2 \geq 1/2$  e  $n \geq 7$  para que a função  $f(n) = \frac{1}{2}n^2 - 3n$  seja  $\Theta(n^2)$ .

## As notações $o$ e $\omega$

As notações  $o$  (lê-se pequeno-o) e  $\omega$  (lê-se pequeno-omega) são muito parecidas com as notações  $O$  e  $\Omega$ , respectivamente.



Enquanto as notações  $O$  e  $\Omega$  possuem uma relação de menor ou igual e maior ou igual, as notações  $o$  e  $\omega$  possuem uma relação de menor e maior.

As notações  $o$  e  $\omega$  não representam limites próximos da função  $f(n)$ , apenas estritamente superiores e inferiores. Ou seja:

- A notação  $o(n^2)$  nos diz que o custo do algoritmo é, assintoticamente, sempre menor do que  $n^2$ . Matematicamente, uma função custo  $f(n)$  é  $o(g(n))$  se existem duas constantes positivas  $c$  e  $m$ , tais que, para  $n \geq m$ , temos  $f(n) < cg(n)$ .
- A notação  $\omega(n^2)$  nos diz que o custo do algoritmo é, assintoticamente, sempre maior do que  $n^2$ . Matematicamente, uma função custo  $f(n)$  é  $\omega(g(n))$  se existem duas constantes positivas  $c$  e  $m$ , tais que, para  $n \geq m$ , temos  $f(n) > cg(n)$ .

## 2.2.7 Classes de problemas

A seguir, são apresentadas algumas classes de complexidade de problemas comumente usadas:

- **$O(1)$ : ordem constante.** As instruções são executadas um número fixo de vezes. Não depende do tamanho dos dados de entrada.
- **$O(\log N)$ : ordem logarítmica.** Típica de algoritmos que resolvem um problema transformando-o em problemas menores.
- **$O(N)$ : ordem linear.** Em geral, certa quantidade de operações é realizada sobre cada um dos elementos de entrada.
- **$O(N \log N)$ : ordem log linear.** Típica de algoritmos que trabalham com particionamento dos dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos.
- **$O(N^2)$ : ordem quadrática.** Normalmente, ocorre quando os dados são processados aos pares. Uma característica deste tipo de algoritmos é a presença de um aninhamento de dois comandos de repetição.
- **$O(N^3)$ : ordem cúbica.** É caracterizado pela presença de três estruturas de repetição aninhadas.
- **$O(2^N)$ : ordem exponencial.** Geralmente, ocorre quando se usa uma solução de força bruta. Não são úteis do ponto de vista prático.
- **$O(M!)$ : ordem factorial.** Geralmente, ocorre quando se usa uma solução de força bruta. Não são úteis do ponto de vista prático. Possui um comportamento muito pior que o exponencial.

Por fim, a relação entre as classes de complexidades é assim definida:

$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N^3) < O(2^N) < O(M!)$$

O Quadro 2.2 mostra uma comparação no tempo de execução de algoritmos que possuem diferentes complexidades. Neste caso, estamos considerando que nosso computador será capaz de executar 1.000.000 (um milhão) de operações por segundo.

QUADRO 2.2

	$n = 10$	$n = 20$	$n = 30$	$n = 50$	$n = 100$
$O(1)$	1,0E-05 segundos	2,0E-05 segundos	4,0E-05 segundos	5,0E-05 segundos	6,0E-05 segundos
$O(\log n)$	3,3E-05 segundos	8,6E-05 segundos	2,1E-04 segundos	2,8E-04 segundos	3,5E-04 segundos
$O(n)$	1,0E-04 segundos	4,0E-04 segundos	1,6E-03 segundos	2,5E-03 segundos	3,6E-03 segundos
$O(n^2)$	1,0E-03 segundos	8,0E-03 segundos	6,4E-02 segundos	0,13 segundos	0,22 segundos
$O(n^3)$	1,0E-03 segundos	1,0 segundo	2,8 dias	35,7 anos	365,6 séculos
$O(2^n)$	5,9E-02 segundos	58,1 minutos	3855,2 séculos	2,3E+08 séculos	1,3E+13 séculos



Na análise assintótica, as constantes de multiplicação são consideradas irrelevantes e, portanto, descartadas. Porém, elas podem ser relevantes na prática, principalmente se o tamanho da entrada é pequeno.

Considere dois algoritmos com tempo de execução  $f(n) = 10^{100}n$  e  $g(n) = 10n \log n$ . Pela análise assintótica, o primeiro é mais eficiente, já que tem complexidade  $O(n)$ , enquanto o outro é  $O(n \log n)$ . No entanto,  $10^{100}$  é um número muito grande. Alguns astrônomos consideram esse número o limite superior para a quantidade de átomos no universo observável. Neste caso,  $10n \log n > 10^{100}n$  apenas para  $n \geq 2^{100}$ . Para qualquer valor menor de  $n$  o algoritmo de complexidade  $O(n \log n)$  será melhor.

### 2.3 RELAÇÕES DE RECORRÊNCIAS

Uma função é chamada de *função recursiva* quando chama a si mesma durante a sua execução. Um exemplo clássico de função recursiva é o cálculo do fatorial de um número.



Como calcular o fatorial de 4 (definido como 4!)?

Para calcular o fatorial de 4, multiplica-se o número 4 pelo fatorial de 3 (definido como 3!). Generalizando o processo, temos que o fatorial de  $N$  é igual a  $N$  multiplicado pelo fatorial de  $(N - 1)$ , ou seja,  $N! = N * (N - 1)!$ . Este processo irá terminar quanto atingirmos o número zero. Neste caso, o valor do fatorial de 0 ( $0!$ ) é definido como sendo igual a 1. Temos então que a função fatorial é definida matematicamente como

$$0! = 1$$

$$N! = N * (N - 1)!$$

Na Figura 2.7, é possível ver a função recursiva para o cálculo do fatorial.

```
01 int factorial (int n){
02     if (n == 0)
03         return 1;
04     else
05         return n * factorial(n-1);
06 }
```

FIGURA 2.7

A definição de uma função recursiva é um exemplo de **recorrência ou relação de recorrência**, isto é, uma expressão que descreve uma função em termos de entradas menores da função. No caso do fatorial, a sua relação de recorrência é dada por

$$T(n) = T(n - 1) + n$$

Muitos algoritmos se baseiam em recorrência. Trata-se de uma ferramenta importante para a solução de problemas combinatórios. Como esses algoritmos, usualmente, não utilizam estruturas de repetição, apenas comandos condicionais, atribuições etc., podemos erroneamente imaginar que essas funções possuem complexidade  $O(1)$ . Na verdade, para saber a complexidade de um algoritmo recursivo precisamos resolver a sua relação de recorrência. Ou seja, isso significa que temos que encontrar uma **fórmula fechada** que nos dê o valor da função em termos de seu parâmetro  $n$ . Isso é geralmente obtido como uma combinação de polinômios, quocientes de polinômios, logaritmos, exponenciais etc.

Considere a seguinte relação de recorrência:

$$T(n) = T(n - 1) + 2n + 3$$

Suponha que  $n \in \{2, 3, 4, \dots\}$ . Existem inúmeras funções  $T$  que satisfazem a recorrência. Por exemplo, a Figura 2.8 mostra duas funções que satisfazem essa recorrência. Neste caso, a primeira função considera o **caso base**  $T(1) = 1$  enquanto a segunda usa  $T(1) = 5$ .

$n$	1	2	3	4	5
$T(n)$	1	8	17	28	41

$n$	1	2	3	4	5
$T(n)$	5	12	21	32	45

FIGURA 2.8



Para cada valor  $n$  e o intervalo existe uma (e apenas uma) função que tem caso base e satisfaz a recorrência.

Sendo assim, precisamos encontrar uma **fórmula fechada** para a recorrência.



Uma forma de se fazer isso é por meio da expansão da relação de recorrência até que se possa detectar um comportamento no seu caso geral.

Para entender a técnica, considere a seguinte relação de recorrência:

$$T(n) = T(n - 1) + 3$$

Essa relação de recorrência representa um algoritmo que possui três operações mais uma chamada recursiva. Para resolvemos essa recorrência devemos aplicar o termo  $T(n - 1)$  sobre a relação  $T(n)$ . Com isso, obtemos

$$T(n - 1) = T(n - 2) + 3$$

Se aplicarmos o termo  $T(n - 2)$  sobre a relação  $T(n)$ , teremos

$$T(n - 2) = T(n - 3) + 3$$

Se continuarmos esse processo, teremos a seguinte expansão:

$$\begin{aligned} T(n) &= T(n - 1) + 3 \\ T(n) &= (T(n - 2) + 3) + 3 \\ T(n) &= ((T(n - 3) + 3) + 3) + 3 \end{aligned}$$

Note que a cada passo um valor 3 é somado à expansão e o valor de  $n$  é diminuído em uma unidade. Assim, podemos resumir essa expansão usando a seguinte equação:

$$T(n) = T(n - k) + 3k$$

Resta saber quando esse processo de expansão termina, ou seja, quando ele chega ao **caso base**. Isso ocorre quando  $n - k = 1$ , ou seja,  $k = n - 1$ . Substituindo, temos

$$\begin{aligned} T(n) &= T(n - k) + 3k \\ T(n) &= T(1) + 3(n - 1) \\ T(n) &= T(1) + 3n - 3 \end{aligned}$$

Como  $T(1)$  é o **caso base**, ou seja, é onde a recursão termina, temos que o tempo de execução dele é constante, isto é,  $O(1)$ . Assim, a complexidade da recorrência é dada por  $T(n) = 3n + 3 + O(1)$ . Temos, portanto, que a complexidade é  $O(n)$ , ou seja, **linear**.

Vamos a outro exemplo. Considere agora a seguinte relação de recorrência:

$$T(n) = T(n/2) + 5$$

Esta relação de recorrência representa um algoritmo que possui cinco operações mais uma chamada recursiva que divide os dados sempre pela metade ( $n/2$ ). Diferente do exemplo anterior, a recorrência não atua sobre todos os valores de  $n$ , mas apenas nos valores de  $n$  que representem uma potência de 2. Assim, ela existe apenas para  $n \in \{2^1, 2^2, 2^3, 2^4, \dots\}$ . Considerando  $n = 2^k$ , podemos reescrever a recorrência como

$$T(2^k) = T(2^{k-1}) + 5$$

Para resolvemos essa recorrência, devemos aplicar o termo  $T(2^{k-1})$  sobre a relação  $T(2^k)$ . Com isso, obtemos

$$T(2^{k-1}) = T(2^{k-2}) + 5$$

Se aplicarmos o termo  $T(2^{k-2})$  sobre a relação  $T(2^k)$ , teremos

$$T(2^{k-2}) = T(2^{k-3}) + 5$$

Se continuarmos esse processo, teremos a seguinte expansão:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 5 \\ T(2^k) &= (T(2^{k-2}) + 5) + 5 \\ T(2^k) &= ((T(2^{k-3}) + 5) + 5) + 5 \\ &\quad \cdots \\ T(2^k) &= T(2^{k-k}) + 5k \\ T(2^k) &= T(2^0) + 5k \\ T(2^k) &= T(1) + 5k \end{aligned}$$

Perceba que a cada passo um valor 5 é somado à expansão e o valor de  $k$  é diminuído em uma unidade. Assim, podemos resumir essa expansão usando a seguinte equação, a qual já considera o seu caso base:

$$T(2^k) = T(1) + 5k$$

Temos que o tempo de execução do **caso base**,  $T(1)$ , é constante:  $O(1)$ . Desse modo, a complexidade da recorrência é dada por  $T(2^k) = 5k + O(1)$ . Devemos lembrar que substituímos  $n$  por  $2^k$  no início da expansão, de modo que  $n = 2^k$ . Aplicando o logaritmo, temos que  $k = \log_2 n$ . Substituindo, temos

$$\begin{aligned} T(2^k) &= 5k + O(1) \\ T(n) &= 5 \log_2 n + O(1) \end{aligned}$$

Como resultado, temos que a relação de recorrência tem complexidade  $O(\log_2 n)$ , ou seja, **logarítmica**.

## 2.4 EXERCÍCIOS

- 1) Na prática, que grandezas físicas influenciam a eficiência de tempo de um algoritmo?
- 2) Quando calculamos a complexidade de algoritmos não recursivos podemos nos guiar por um conjunto de regras simples de serem seguidas. Cite e descreva estas regras.
- 3) O que significa dizer que uma função  $g(n)$  é  $O(f(n))$ ?
- 4) O que significa dizer que uma função  $g(n)$  é  $\Omega(f(n))$ ?
- 5) Liste os tipos de problemas que apresentam complexidade da ordem de  $n \log n$ . Como é possível identificá-los?
- 6) Considere dois algoritmos A e B com funções de complexidade de tempo  $a(n) = n^2 - n + 500$  e  $b(n) = 47n + 47$ , respectivamente. Para quais valores de  $n$  o algoritmo A leva menos tempo para executar do que B?
- 7) Calcule a ordem de complexidade, no pior caso, das seguintes funções de custo:

- $2n + 10$
- $\frac{1}{2}n(n+1)$
- $n + \sqrt{n}$
- $n/1000$
- $\frac{1}{2}n$
- $\frac{1}{2}n - 3n$

- 8) Calcule a complexidade, no pior caso, do seguinte fragmento de código:

```
int i,j,k;
for(i=0; i < N; i++){
    for(j=0; j < N; j++){
        R[i][j] = 0;
        for(k=0; k < N; k++)
            R[i][j] += A[i][k] * B[k][j];
    }
}
```

- 9) Calcule a complexidade, no pior caso, do seguinte fragmento de código:

```
int i,j,k,s;
for(i=0; i < N-1; i++)
    for(j=i+1; j < N; j++)
        for(k=1; k < j; k++)
            s = 1;
```

- 10) Calcule a complexidade, no pior caso, do seguinte fragmento de código:

```
int i,j,s;
s = 0;
for(i=1; i < N-1; i++)
    for(j=1; j < 2*N; j++)
        s = s + 1;
```

## Ordenação e busca em arrays

### 3.1 DEFINIÇÃO

A ordenação nada mais é do que o ato de colocar um conjunto de dados em determinada ordem predefinida, como mostra o exemplo a seguir:

- 5, 2, 1, 3, 4: FORA DE ORDEM.
- 1, 2, 3, 4, 5: ORDENADO.



A ordenação permite que o acesso aos dados seja feita de forma mais eficiente.



A chave de ordenação é o "campo" do item utilizado para comparação. É por meio dele que sabemos se determinado elemento está à frente ou não de outros no conjunto ordenado.

Para realizar a ordenação, podemos usar qualquer tipo de chave, desde que exista uma regra de ordenação bem definida. Existem vários tipos possíveis de ordenação. Os tipos de ordenação mais comuns são:

Numérica: 1, 2, 3, 4, 5.

Lexicográfica (ordem alfabética): Ana, André, Bianca, Ricardo.

Além disso, independente do tipo, a ordenação pode ser:

- Crescente:
  - 1, 2, 3, 4, 5.
  - Ana, André, Bianca, Ricardo.
- Decrescente:
  - 5, 4, 3, 2, 1.
  - Ricardo, Bianca, André, Ana.



Um algoritmo de ordenação é aquele que coloca os elementos de dada sequência em certa ordem predefinida.

Existem vários algoritmos para realizar a ordenação dos dados. Eles podem ser classificados como de **ordenação interna** (in-place) ou **externa**:

- **Ordenação interna:** o conjunto de dados a ser ordenado cabe todo na memória principal. Qualquer elemento pode ser imediatamente acessado.
- **Ordenação externa:** o conjunto de dados a ser ordenado não cabe na memória principal (está armazenado em memória secundária, por exemplo, em um arquivo). Os elementos são acessados sequencialmente ou em grandes blocos.

Além disso, um algoritmo de ordenação pode ser considerado estável ou não.



Um algoritmo de ordenação é considerado estável se a ordem dos elementos com chaves iguais não muda durante a ordenação.

Imagine um conjunto de dados não ordenado com dois valores iguais, no caso, 5a e 5b:

5a, 2, 5b, 3, 4, 1: **dados não ordenados.**

Um algoritmo de ordenação será considerado **estável** se o valor 5a vier antes do valor 5b quando esse conjunto de dados for ordenado de forma crescente, ou seja, o algoritmo preserva a ordem relativa original dos valores:

1, 2, 3, 4, 5a, 5b: **ordenação estável.**

1, 2, 3, 4, 5b, 5a: **ordenação não estável.**

Nas próximas seções, iremos abordar alguns dos principais algoritmos de ordenação de dados armazenados em arrays (ordenação interna).

## 3.2 ALGORITMOS BÁSICOS DE ORDENAÇÃO

### 3.2.1 Ordenação por “bolha”: bubble sort

O algoritmo bubble sort, também conhecido como ordenação por “bolha”, é um dos algoritmos de ordenação mais conhecidos que existem. Ele tem esse nome por remeter à ideia de bolhas flutuando em um tanque de água em direção ao topo, até encontrarem o seu próprio nível (ordenação crescente).



O algoritmo bubble sort trabalha de forma a movimentar, uma posição por vez, o maior valor existente na porção não ordenada de um array para a sua respectiva posição no array ordenado. Isso é repetido até que todos os elementos estejam nas suas posições correspondentes.

A Figura 3.1 mostra a implementação do algoritmo bubble sort. O princípio de funcionamento deste algoritmo é a troca de valores em posições consecutivas de array para que, desse modo, fiquem na ordem desejada. Para entender esse processo, imagine um seguinte conjunto de valores não ordenados, como mostrado na Figura 3.2. Ao se comparar os dois primeiros valores, percebe-se que eles não estão ordenados (ordem crescente). Então, o algoritmo os troca de lugar. O processo é repetido para cada par de valores em posições consecutivas do array (linhas 5-12). Ao final do processo, teremos o maior valor na última posição do array. Falta ordenar o restante do array. Isso é feito diminuindo o valor da variável **fim** (linha 2) que está associada à última posição do array (linha 13). Em seguida, executamos todo o processo de levar o maior valor até o final do array com um comando **do-while** (linhas 3-14).

```
01 void bubbleSort(int *V, int N) {
02     int i, continua, aux, fim = N;
03     do{
04         continua = 0;
05         for(i = 0; i < fim-1; i++){
06             if(V[i] > V[i+1]){//anterior > próximo? Mudar!
07                 aux = V[i];
08                 V[i] = V[i+1];
09                 V[i+1] = aux;
10                 continua = i;
11             }
12         }
13         fim--;
14     }while(continua != 0);
15 }
```

FIGURA 3.1

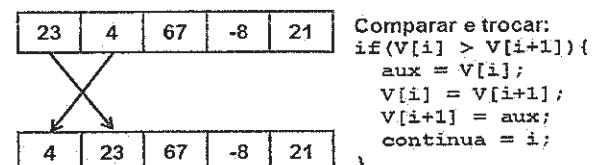


FIGURA 3.2



Perceba que o comando **do-while** termina se a variável **continua** for diferente de **ZERO**. Isso é feito para otimizar o algoritmo.

Sempre que uma troca de valores ocorrer, a variável **continua** será modificada (linha 10) em relação ao seu valor original (linha 4). Desse modo, se nenhuma troca de valores ocorrer, o algoritmo poderá ser finalizado mais cedo.

A Figura 3.3 mostra um exemplo de ordenação completa de um array em ordem crescente. Os itens coloridos são os valores comparados, enquanto os hachurados representam a porção já ordenada do array. Percebe-se que:

- Primeira iteração do comando **do-while**: encontra-se o maior valor e o movimenta até a última posição.
- Segunda iteração do comando **do-while**: encontra-se o segundo maior valor e o movimenta até a penúltima posição.
- Esse processo continua até que todo o array esteja ordenado.

Sem Ordenar					
23	4	67	-8	21	
1º Iteração do-while					
i=0	23	4	67	-8	21
v[i] > v[i+1]: Trocar					
i=1	4	23	67	-8	21
v[i] < v[i+1]: Manter					
i=2	4	23	67	-8	21
v[i] > v[i+1]: Trocar					
i=3	4	23	-8	67	21
v[i] > v[i+1]: Trocar					
Final	4	23	-8	21	67
2º Iteração do-while					
i=0	4	23	-8	21	67
v[i] < v[i+1]: Manter					
i=1	4	23	-8	21	67
v[i] > v[i+1]: Trocar					
i=2	4	-8	23	21	67
v[i] > v[i+1]: Trocar					
Final	4	-8	21	23	67
3º Iteração do-while					
i=0	4	-8	21	23	67
v[i] > v[i+1]: Trocar					
i=1	-8	4	21	23	67
v[i] < v[i+1]: Manter					
Final	-8	4	21	23	67
4º Iteração do-while					
i=0	-8	4	21	23	67
v[i] < v[i+1]: Manter					
Não houve mudanças: ordenação concluída					
Ordenado					
-8	4	21	23	67	

FIGURA 3.3



O bubble sort é um algoritmo simples e de fácil entendimento e implementação. Além disso, está entre os mais difundidos métodos de ordenação existentes. Infelizmente, não é um algoritmo eficiente, sendo estudado apenas para fins de desenvolvimento de raciocínio.

Isso ocorre porque sua eficiência diminui drasticamente à medida que o número de elementos no array aumenta. Ou seja, ele não é recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade. Considerando um array com N elementos, o tempo de execução do bubble sort é:

- $O(N)$ , melhor caso: os elementos já estão ordenados.
- $O(N^2)$ , pior caso: os elementos estão ordenados na ordem inversa.
- $O(N^2)$ , caso médio.

### 3.2.2 Ordenação por seleção: selection sort

O algoritmo selection sort, também conhecido como ordenação por “seleção”, é outro algoritmo de ordenação bastante simples. Ele tem esse nome, pois a cada passo “seleciona” o melhor elemento (maior ou menor, dependendo do tipo de ordenação) para ocupar aquela posição do array. Na prática, este algoritmo possui um desempenho quase sempre superior quando comparado com o bubble sort.



O algoritmo selection sort divide o array em duas partes: a parte ordenada, à esquerda do elemento analisado, e a parte que ainda não foi ordenada, à direita do elemento. Para cada elemento do array, começando do primeiro, o algoritmo procura na parte não ordenada (direita) o menor valor (ordenação crescente) e troca os dois valores de lugar. Em seguida, o algoritmo avança para a próxima posição do array e esse processo é feito até que todo o array esteja ordenado.

A Figura 3.4 mostra a implementação do algoritmo selection sort. O princípio de funcionamento deste algoritmo é a seleção do melhor elemento para ocupar aquela posição do array. Para entender esse processo, imagine um conjunto de valores não ordenados, como mostrado na Figura 3.5. Perceba que o valor da primeira posição do array (23) não está na sua posição correta, pois existem valores menores do que ele na porção não ordenada do array (direita). Então, o algoritmo percorre os elementos da direita procurando o índice do menor valor dentre todos (linhas 4-8). Ao chegar ao final do array, o algoritmo troca os valores do elemento atual, índice i, com o menor valor encontrado, índice menor (linhas 9-13). O processo de comparar o valor de uma posição do array com seus sucessores é repetido para cada posição do array (linhas 3-14).

**Método selection sort:**

```

01 void selectionSort(int *V, int N) {
02     int i, j, menor, troca;
03     for(i = 0; i < N-1; i++){
04         menor = i;
05         for(j = i+1; j < N; j++){
06             if(V[j] < V[menor])
07                 menor = j;
08         }
09         if(i != menor){
10             troca = V[i];
11             V[i] = V[menor];
12             V[menor] = troca;
13         }
14     }
15 }
```

FIGURA 3.4

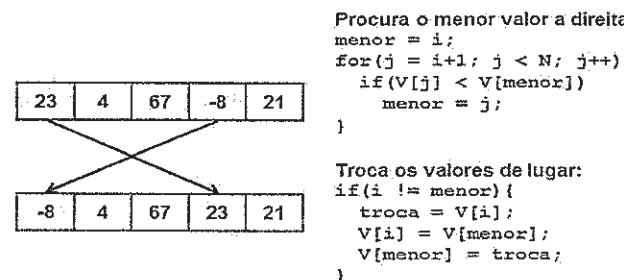


FIGURA 3.5

A Figura 3.6 mostra um exemplo de ordenação completa de um array em ordem crescente. Para cada valor da variável *i* temos o array antes e depois de ser selecionado o menor valor para ocupar aquela posição. Os itens coloridos representam o valor da posição de índice *i* e a posição com o menor valor encontrado na porção não ordenada do array (à direita de *i*).

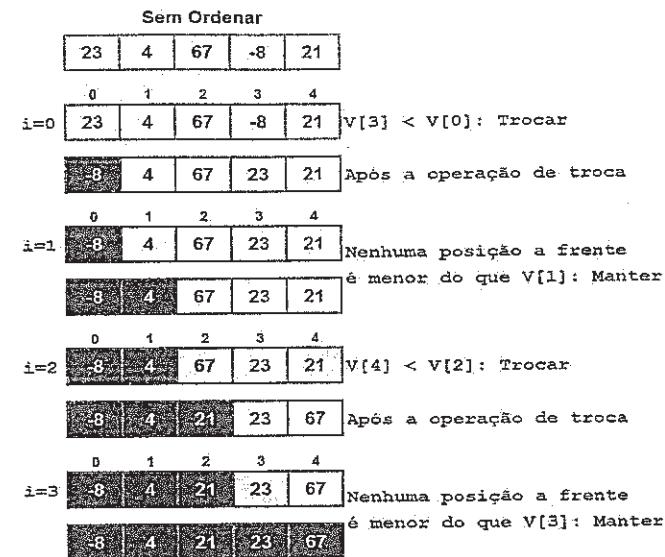


FIGURA 3.6

O selection sort, assim como o bubble sort, não é um algoritmo eficiente. Sua eficiência diminui drasticamente à medida que o número de elementos no array aumenta, não sendo recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade. Considerando um array com *N* elementos, o tempo de execução do selection sort é sempre de ordem  $O(N^2)$ . Como se pode notar, a eficiência do selection sort não depende da ordem inicial dos elementos.



Apesar de possuírem a mesma complexidade no caso médio, na prática, o selection sort quase sempre supera o desempenho do bubble sort, pois envolve um número menor de comparações.

**3.2.3 Ordenação por inserção: insertion sort**

O algoritmo insertion sort, também conhecido como ordenação por “inserção”, é outro algoritmo de ordenação bastante simples. Ele tem esse nome, pois se assemelha ao processo de ordenação de um conjunto de cartas de baralhos com as mãos: pega uma carta de cada vez e a “insere” em seu devido lugar, sempre deixando as cartas da mão em ordem. Na prática, este algoritmo possui um desempenho superior quando comparado com outros algoritmos como o bubble sort e o selection sort.



O algoritmo insertion sort percorre um array e, para cada posição  $X$ , verifica se o seu valor está na posição correta. Isso é feito andando para o começo do array, a partir da posição  $X$ , e movimentando uma posição para frente os valores que são maiores do que o valor da posição  $X$ . Desse modo, teremos uma posição livre para inserir o valor da posição  $X$  em seu devido lugar.

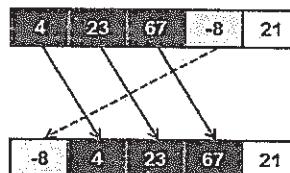
A Figura 3.7 mostra a implementação do algoritmo insertion sort. O princípio de funcionamento deste algoritmo é a inserção de um elemento do array na sua posição correta. Para entender esse processo, imagine um conjunto de valores não ordenados, como mostrado na Figura 3.8. Perceba que o valor da quarta posição do array (-8) não está na sua posição correta em relação aos seus antecessores. Então, o algoritmo movimenta os seus antecessores uma posição para frente e insere esse valor no início do array. O processo de comparar o valor de uma posição do array com seus antecessores é repetido para cada posição do array (linhas 3-8). Para cada posição, movimentam-se os valores maiores que o valor atual (linha 4) uma posição para frente no array (linhas 5-6). Ao final do processo, copia-se o valor atual para a sua posição correta, que é a posição do último valor movimentado (linha 7).

### Método insertion sort

```

01 void insertionSort(int *V, int N){
02     int i, j, atual;
03     for(i = 1; i < N; i++){
04         atual = V[i];
05         for(j = i; (j > 0) && (atual < V[j - 1]); j--)
06             V[j] = V[j - 1];
07         V[j] = atual;
08     }
09 }
```

FIGURA 3.7



Desloca os valores a esquerda e insere:  
 $\text{aux} = V[i];$   
 $\text{for}(j = i; (j > 0) \&\& (\text{aux} < V[j - 1]); j--)$   
 $V[j] = V[j - 1];$   
 $V[j] = aux;$

FIGURA 3.8

A Figura 3.9 mostra um exemplo de ordenação completa de um array em ordem crescente. Para cada valor da variável  $i$ , temos o array antes de ser ordenado e depois de ser ordenada aquela posição. Os itens coloridos são os valores verificados se estão na posição correta, enquanto os hachurados representam a porção do array movimentada durante a inserção do valor na sua posição correta.

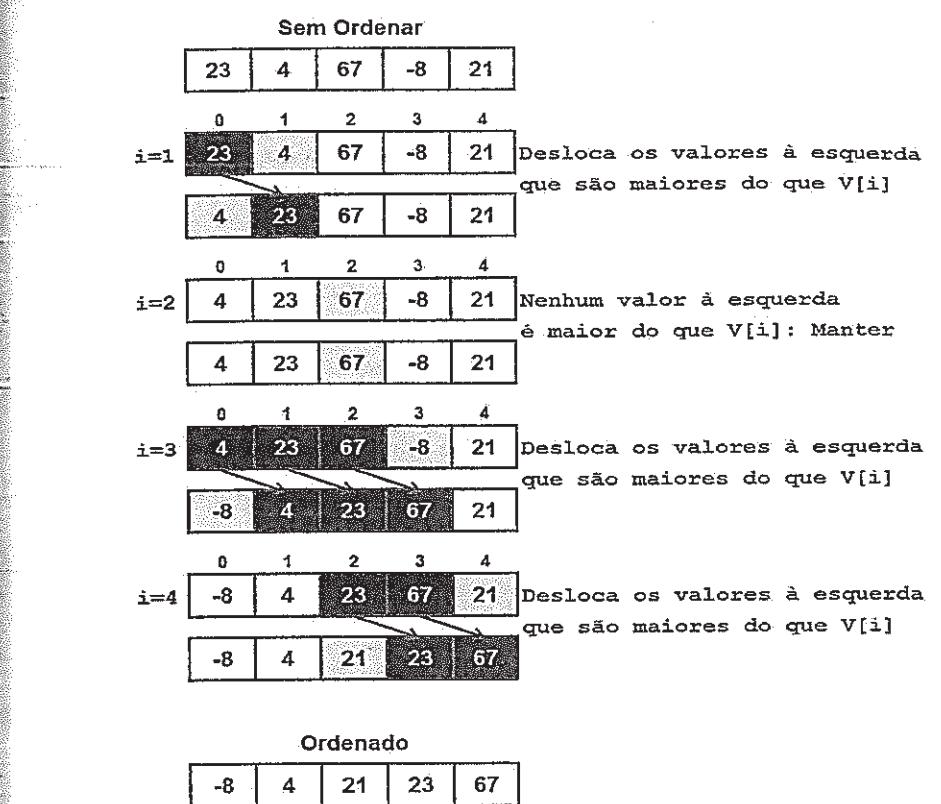


FIGURA 3.9

Considerando um array com  $N$  elementos, o tempo de execução do insertion sort é:

- $O(N)$ , melhor caso: os elementos já estão ordenados.
- $O(N^2)$ , pior caso: os elementos estão ordenados na ordem inversa.
- $O(N^2)$ , caso médio.



Na prática, o insertion sort é mais eficiente que a maioria dos algoritmos de ordem quadrática, como o selection sort e o bubble sort. De fato, trata-se de um dos mais rápidos algoritmos de ordenação para conjuntos pequenos de dados, superando inclusive o quick sort.

Além de ser um algoritmo de fácil implementação, o insertion sort tem a vantagem de ser:

- **Estável:** a ordem dos elementos iguais não muda durante a ordenação.
- **On-line:** pode ordenar elementos na medida em que os recebe, ou seja, não precisa ter todo o conjunto de dados para colocá-los em ordem.

### 3.3 ALGORITMOS SOFISTICADOS DE ORDENAÇÃO

#### 3.3.1 Algoritmo merge sort

O algoritmo merge sort, também conhecido como ordenação por “intercalação”, é um algoritmo recursivo que usa a ideia de *dividir para conquistar* para ordenar os dados de um array. Este algoritmo parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos. Sendo assim, o algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combiná-los por meio de intercalação (merge).



O algoritmo merge sort divide, recursivamente, o array em duas partes, até que cada posição dele seja considerada um array de um único elemento. Em seguida, o algoritmo combina dois arrays de forma a obter um array maior e ordenado. Essa combinação dos arrays é feita intercalando seus elementos de acordo com o sentido da ordenação (crescente ou decrescente). O processo se repete até que exista apenas um array.

A Figura 3.10 mostra a implementação do algoritmo merge sort. Este algoritmo trabalha com o conceito de recursividade *dividir para conquistar*: ele usa uma função para dividir os dados em arrays cada vez menores, **mergeSort** (linhas 31-39), e outra para intercalar os dados dos arrays de forma ordenada em um array maior, **merge** (linhas 1-29). A função **mergeSort** se encarrega de dividir o array ao meio se este tiver mais do que um elemento (linha 33). Se a afirmação for verdadeira, ela calcula o valor que corresponde ao meio do array (linha 34) e chama a si própria para as duas metades (linhas 35-36). Uma vez que todos os elementos do array tenham sido separados em arrays com um único elemento, a recursão termina e tem início a etapa de intercalação dos array pela função **mergeSort** (linha 37). Apesar de parecer grande, a função de intercalação é bastante simples, como mostra a Figura 3.11. A função recebe as duas metades do array (uma começando em **inicio** e outra em **meio+1**) e as combina em um array auxiliar **temp**. Para fazer isso, a função percorre o array **temp** e, a cada passo, verifica qual array tem o menor elemento. O menor elemento é inserido em **temp** e o algoritmo incrementa o contador do array escolhido (linhas 11-14). Caso um dos array tenha chegado ao final (linhas 16-17), na próxima iteração o algoritmo irá simplesmente copiar o restante do array que sobrou para o final de **temp** (linhas 19-22). Por fim, o algoritmo copia os dados do array **temp** para o array original (linhas 25-26).

```

Metodo merge sort

01 void merge(int *V, int inicio, int meio, int fim){
02     int *temp, p1, p2, tamanho, i, j, k;
03     int fim1 = 0, fim2 = 0;
04     tamanho = fim-inicio+1;
05     p1 = inicio;
06     p2 = meio+1;
07     temp = (int *) malloc(tamanho*sizeof(int));
08     if(temp != NULL){
09         for(i=0; i<tamanho; i++){
10             if(!fim1 && !fim2){
11                 if(V[p1] < V[p2])
12                     temp[i]=V[p1++];
13                 else
14                     temp[i]=V[p2++];
15             }
16             if(p1>meio) fim1=1;
17             if(p2>fim) fim2=1;
18         }
19         if(!fim1)
20             temp[i]=V[p1++];
21         else
22             temp[i]=V[p2++];
23     }
24     for(j=0, k=inicio; j<tamanho; j++, k++)
25         V[k]=temp[j];
26 }
27 free(temp);
28 }

30 void mergeSort(int *V, int inicio, int fim){
31     int meio;
32     if(inicio < fim){
33         meio = floor((inicio+fim)/2);
34         mergeSort(V,inicio,meio);
35         mergeSort(V,meio+1,fim);
36         merge(V,inicio,meio,fim);
37     }
38 }
39 }
```

FIGURA 3.10

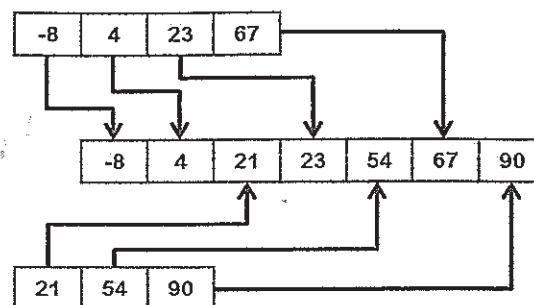


FIGURA 3.11

A Figura 3.12 mostra um exemplo de ordenação completa de um array em ordem crescente. Os itens coloridos são os valores calculados como sendo o meio do array e utilizados para dividi-lo em duas metades na função **mergeSort**.

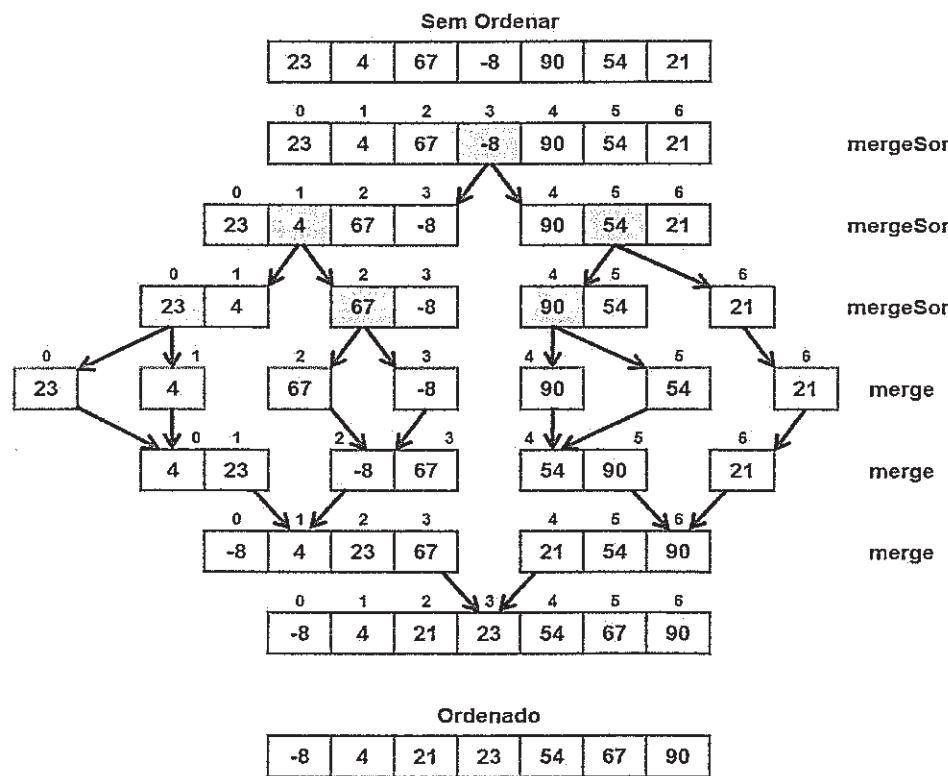


FIGURA 3.12

Considerando um array com  $N$  elementos, o tempo de execução do merge sort é sempre de ordem  $O(N \log N)$ . Como se pode notar, a eficiência do merge sort não depende da ordem inicial dos elementos.



Embora a eficiência do merge sort seja a mesma independente da ordem dos elementos, ele possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação.

Isso ocorre porque ele cria uma cópia do array para cada chamada recursiva. Em outra abordagem, é possível utilizar um único array auxiliar ao longo de toda a execução do merge sort.



No pior caso, o merge sort realiza cerca de 39 % menos comparações do que o quick sort faz no seu caso médio. Já no melhor caso, o merge sort realiza cerca de metade do número de iterações do seu pior caso.

### 3.3.2 Algoritmo quick sort

O algoritmo quick sort, também conhecido como ordenação por “partição”, é outro algoritmo recursivo que usa a ideia de *dividir para conquistar* para ordenar os dados de um array. Este algoritmo se baseia no problema da separação (partition subproblem). Tal problema consiste em rearranjar um array de modo que os valores menores que certo valor, chamado **pivô**, fiquem na parte esquerda do array, enquanto os valores maiores do que o **pivô** ficam na parte direita. Trata-se, em geral, de um algoritmo muito rápido, pois parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos. Porém, é um algoritmo lento em alguns casos especiais.



O algoritmo quick sort, primeiramente, rearranja os valores de array de modo que os valores menores do que o valor do **pivô** fiquem na parte esquerda do array, enquanto os valores maiores do que o **pivô** ficam na parte direita. Supondo um array de tamanho  $N$  e que o pivô esteja na posição  $X$ , esse processo cria duas partições:  $[0, \dots, X - 1]$  e  $[X + 1, \dots, N - 1]$ . Em seguida, o algoritmo é aplicado recursivamente a cada partição. O processo se repete até que cada partição contenha um único elemento.

A Figura 3.13 mostra uma implementação do algoritmo quick sort. Este algoritmo trabalha com o conceito de recursividade *dividir para conquistar*: ele usa uma função para rearranjar os dados do array em partições cada vez menores, **particiona** (linhas 1-20), e outra para gerenciar a ordenação de cada partição **quickSort** (linhas 22-29). A função **quickSort** se encarrega de verificar se o tamanho do array (ou partição) tem mais do que um elemento (linha 24). Se a afirmação for verdadeira, ela calcula o valor que corresponde ao **pivô** do array (linha 25) e chama a si própria para as duas partições criadas (linhas 26-27).

```

    MercadoQuickSort
01 int particiona(int *V, int inicio, int final ){
02     int esq, dir, pivo, aux;
03     esq = inicio;
04     dir = final;
05     pivo = V[inicio];
06     while(esq < dir){
07         while(V[esq] <= pivo)
08             esq++;
09         while(V[dir] > pivo)
10             dir--;
11         if(esq < dir){
12             aux = V[esq];
13             V[esq] = V[dir];
14             V[dir] = aux;
15         }
16     }
17     V[inicio] = V[dir];
18     V[dir] = pivo;
19     return dir;
20 }
21
22 void quickSort(int *V, int inicio, int fim) {
23     int pivo;
24     if(fim > inicio){
25         pivo = particiona(V, inicio, fim);
26         quickSort(V, inicio, pivo-1);
27         quickSort(V, pivo+1, fim);
28     }
29 }

```

FIGURA 3.13



Como se pode notar, o ponto principal para a ordenação dos dados usando o algoritmo quick sort é a escolha do **pivô**.

A escolha do pivô e a partição dos dados são a parte mais delicada deste algoritmo. Isso porque precisamos de um algoritmo que rearranje os dados de forma rápida e que não use um array auxiliar para realizar essa tarefa. A eficiência do quick sort está ligada à eficiência da sua função **particiona** (linhas 1-20) para separar os dados e calcular o **pivô**. Apesar de parecer grande, a função de particionamento dos dados é bastante simples, como mostra a Figura 3.14 (os itens coloridos são os **pívôs**, enquanto os valores hachurados são os que foram comparados e/ou mudados de lugar). A função recebe um array e as posições de **início** e **final** da partição sendo processada (a qual pode ser todo o array). Em seguida, ela define **esq** e **dir** como as posições mais a esquerda (**início**) e mais a direita (**final**) da partição, sendo o primeiro valor da partição escolhido como o **pivô** (linhas 3-5). Enquanto a posição da direita for maior do que a da esquerda (linha 6), a variável **esq** será incrementada até que se

encontre uma posição cujo valor seja maior do que o **pivô** (linhas 7-8). Já a variável **dir** será decrementada até que se encontre uma posição cujo valor seja menor ou igual ao **pivô** (linhas 9-10). Ao final desse processo, **esq** e **dir** são comparados e se **esq** for menor do que **dir**, seus valores são trocados de lugar (linhas 11-15). Esse processo se repete enquanto **esq** for menor do que **dir**. Terminado o comando **while**, a posição do **início** da partição recebe o valor na posição **dir** e esta recebe o valor do **pivô** (linhas 17-18). Por fim, o valor de **dir** é retornado como sendo o **pivô** escolhido (linha 19).

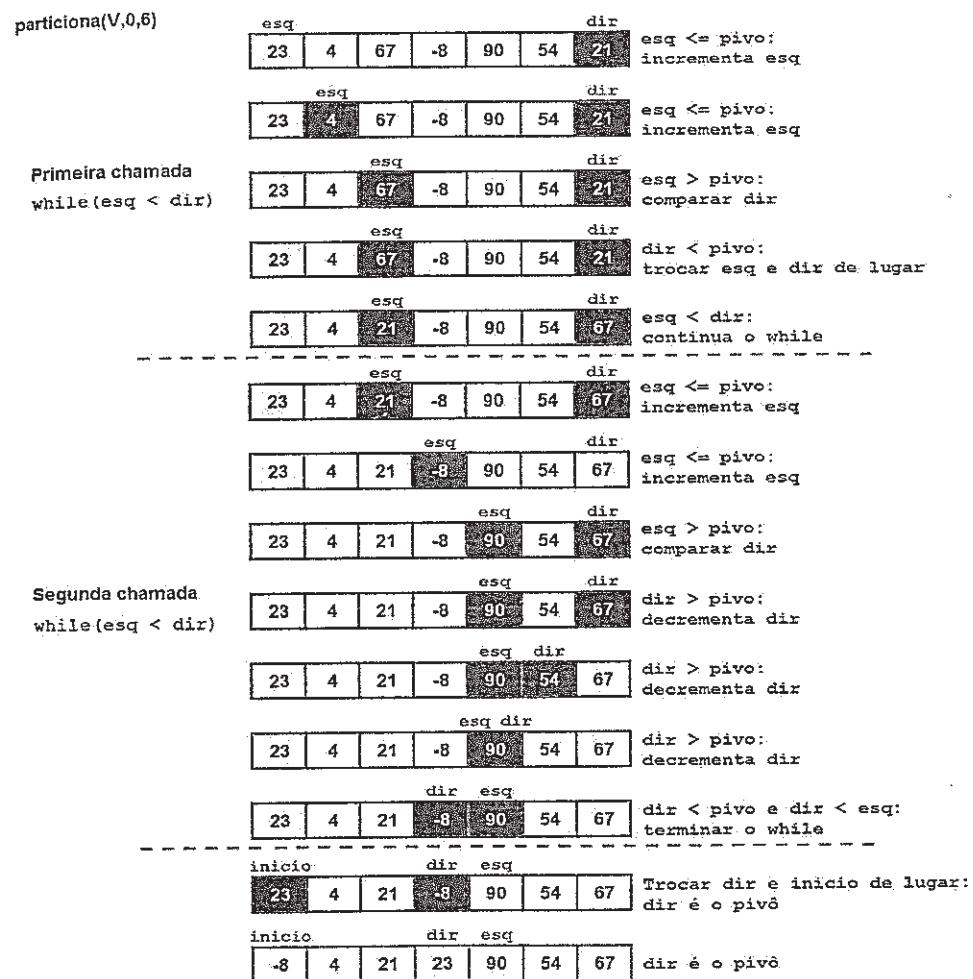


FIGURA 3.14

A Figura 3.15 mostra um exemplo de ordenação completa de um array em ordem crescente. Os itens coloridos são os valores calculados como sendo o pivô do array e os valores hachurados são os que foram mudados de lugar pela função `particiona`.

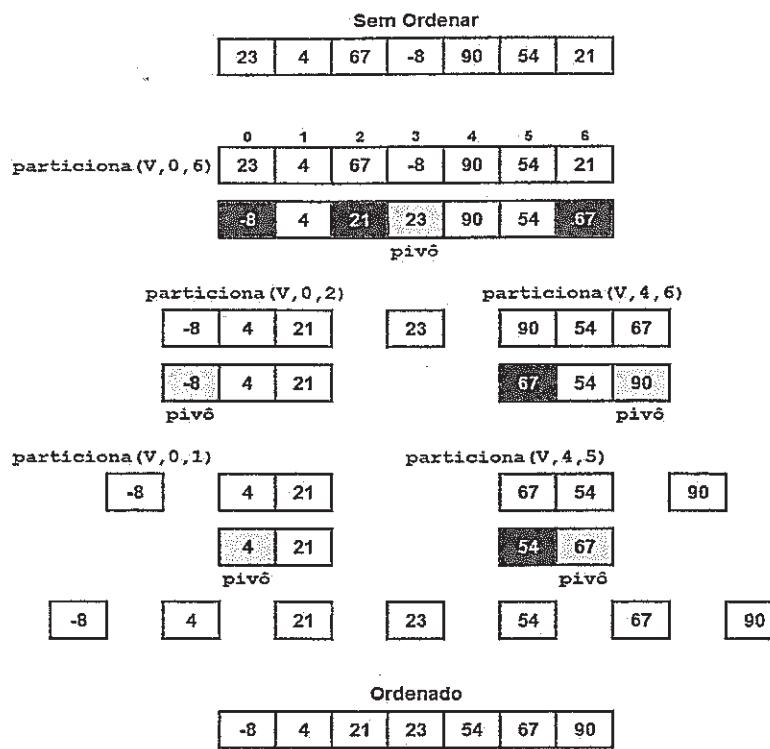


FIGURA 3.15

Considerando um array com  $N$  elementos, o tempo de execução do quick sort é:

- $O(N \log N)$ , melhor caso e caso médio.
- $O(N^2)$ , pior caso.



O pior caso do quick sort ocorre quando a função de partição calcula um pivô que divide o array de  $N$  em dois: uma partição com  $N-1$  elementos e outra com 0 elementos.

Neste caso, temos um particionamento que não é balanceado. Quando isso acontece a cada nível da recursão, temos o tempo de execução de  $O(N^2)$ . O insertion sort acaba sendo mais eficiente que o quick sort, pois o pior caso do quick sort ocorre quando o array já está ordenado, uma situação em que a complexidade é  $O(N)$  no insertion sort.



Apesar de seu pior caso ser quadrático, o quick sort costuma ser a melhor opção prática para ordenação de grandes conjuntos de dados. Sua maior desvantagem talvez seja o fato de não ser um algoritmo de ordenação estável.

### 3.3.3 Algoritmo heap sort

O algoritmo heap sort, também conhecido como ordenação por “heap” (ou monte), é um algoritmo de ordenação bastante sofisticado e que compete em desempenho com o quick sort. A ideia básica deste algoritmo é transformar o array de dados em uma estrutura do tipo heap, isto é, uma árvore binária completa (com exceção do seu último nível). Essa estrutura permite a recuperação e remoção eficiente do elemento de maior valor do array. Desse modo, podemos repetidamente “remover” o maior elemento da heap, construindo assim o array ordenado de trás para frente.



O algoritmo heap sort simula uma árvore binária completa (exceção do último nível) a partir de uma array. Cada posição ( $i$ ) do array passa a ser considerada o pai de duas outras posições, chamadas filhos:  $(2i + 1)$  e  $(2i + 2)$ . Em seguida, o algoritmo reorganiza o array para que o pai seja sempre maior que os dois filhos. Ao final desse processo, o elemento que é pai de todos é também o maior elemento do array. Este elemento poderá ser removido da heap e colocado na última posição do array, e o processo continua para o restante da heap/array.

A Figura 3.16 mostra a simulação de um heap a partir de uma array. Note que a posição ( $i$ ) do array passa a ser considerada o pai de duas outras posições, chamadas filhos:  $(2i + 1)$  e  $(2i + 2)$ .

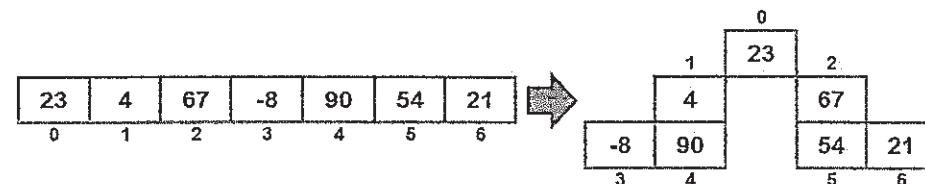


FIGURA 3.16

A Figura 3.17 mostra a implementação do algoritmo heap sort. Note que o algoritmo utiliza duas funções:

- `criaHeap` (linhas 1-19), responsável pela criação da heap a partir de certo elemento do array.
- `heapSort` (linhas 21-32), responsável por criar a heap e ordenar os dados.

A função `criaHeap` recebe como parâmetros o array a ser ordenado, determinada posição `pai` a ser verificada e a última posição do array, `fim`. A função, então, armazena o valor da

```

    Método de passagem por referência
01 void criaHeap(int *vet, int pai, int fim) {
02     int aux = vet[pai];
03     int filho = 2 * pai + 1;
04     while (filho <= fim) {
05         if(filho < fim){
06             if(vet[filho] < vet[filho + 1]){
07                 filho++;
08             }
09         }
10         if(aux < vet[filho]){
11             vet[pai] = vet[filho];
12             pai = filho;
13             filho = 2 * pai + 1;
14         }else{
15             filho = fim + 1;
16         }
17     }
18     vet[pai] = aux;
19 }
20
21 void heapSort(int *vet, int N){
22     int i, aux;
23     for(i=(N - 1)/2; i >= 0; i--){
24         criaHeap(vet, i, N-1);
25     }
26     for (i = N-1; i >= 1; i--){
27         aux = vet[0];
28         vet[0] = vet[i];
29         vet[i] = aux;
30         criaHeap(vet, 0, i - 1);
31     }
32 }

```

FIGURA 3.17

posição **pai** na variável **aux** (linha 2), calcula o seu primeiro **filho** (linha 3) e verifica se a posição do **filho** está no array (linha 4). Em caso afirmativo, a função verifica se existe o segundo **filho** (linha 5). Se o segundo **filho** existir, é necessário selecionar o maior deles (linhas 6-8). Do contrário, o primeiro será considerado o maior. Tendo sido selecionado o maior **filho**, é necessário verificar se o valor de **aux** é menor do que ele (linha 10). Se a afirmação for verdadeira, a posição **pai** receberá o valor do maior **filho** (linha 11), o **filho** será considerado o novo **pai** e também terá o seu **filho** calculado para que o processo e a ordenação continuem (linhas 11-13). Caso a afirmação seja falsa (linha 10), o **filho** recebe o valor de uma posição além do final do array (**fim**) para que o comando **while** termine (linha 15). Terminado o laço, o valor de **aux** é copiado para a posição do **pai** atual. Esse processo pode ser melhor entendido com a Figura 3.18.

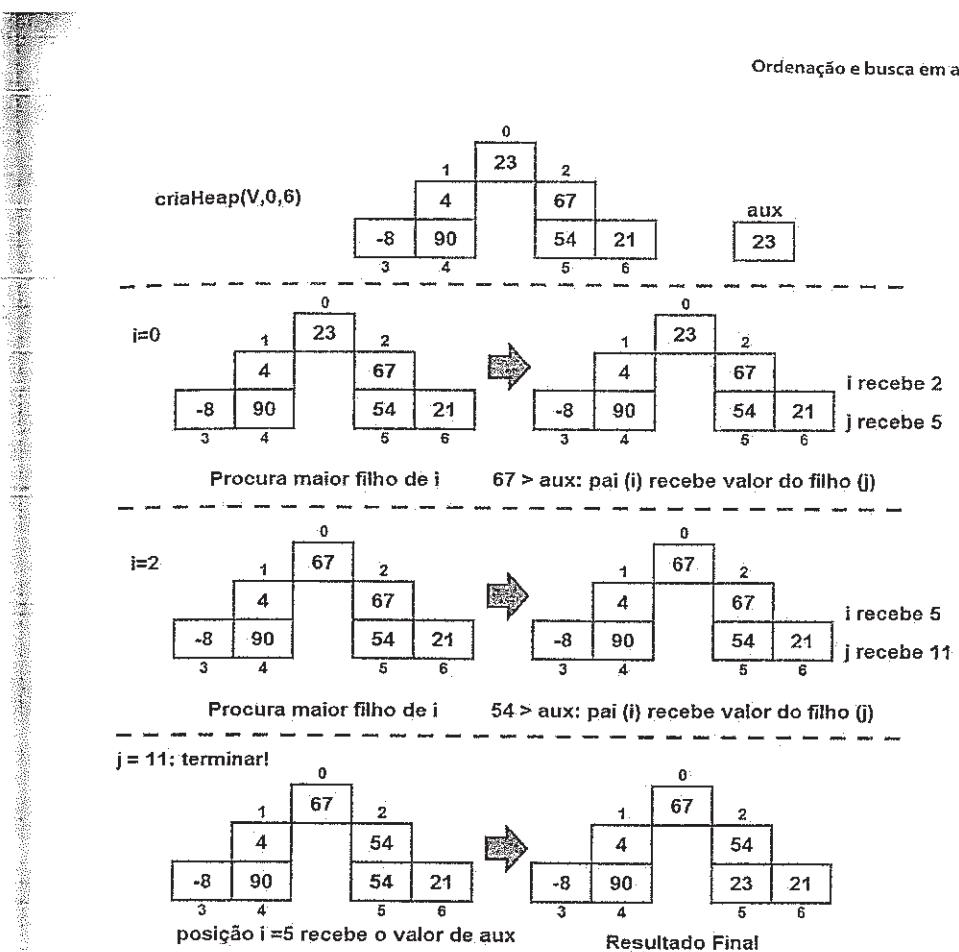


FIGURA 3.18



Como se pode notar, o objetivo da função **criaHeap** é fazer com que toda posição **pai** analisada seja sempre maior que os seus filhos. Porém, isso não significa que determinada posição **pai** será maior que os filhos de seus filhos.

É por esse motivo que a função **criaHeap** é chamada várias vezes dentro da função **heapSort** (linhas 22-25). É preciso analisar metade das posições do array para poder garantir que todo **pai** seja maior que seus filhos, e os filhos deles. A essa etapa damos o nome de criação da heap, como exemplificado na Figura 3.19.

A segunda etapa da função **heapSort** realiza a remoção do maior elemento e a reconstrução da heap (linhas 26-31). Sabemos agora que a primeira posição do array possui o maior elemento. Então, percorremos o array do seu final até o seu início (linha 26). A cada passo da repetição, trocamos de lugar os valores da posição **i** e do início do array (linhas 27-29). Em seguida,

recriamos a heap considerando que o array tenha um elemento a menos no seu final (linha 30). O elemento desconsiderado é o elemento copiado para o final, ou seja, o maior elemento, portanto, já está na sua posição correta, sendo necessário ordenar apenas os demais valores do array. A Figura 3.20 exemplifica esse processo.

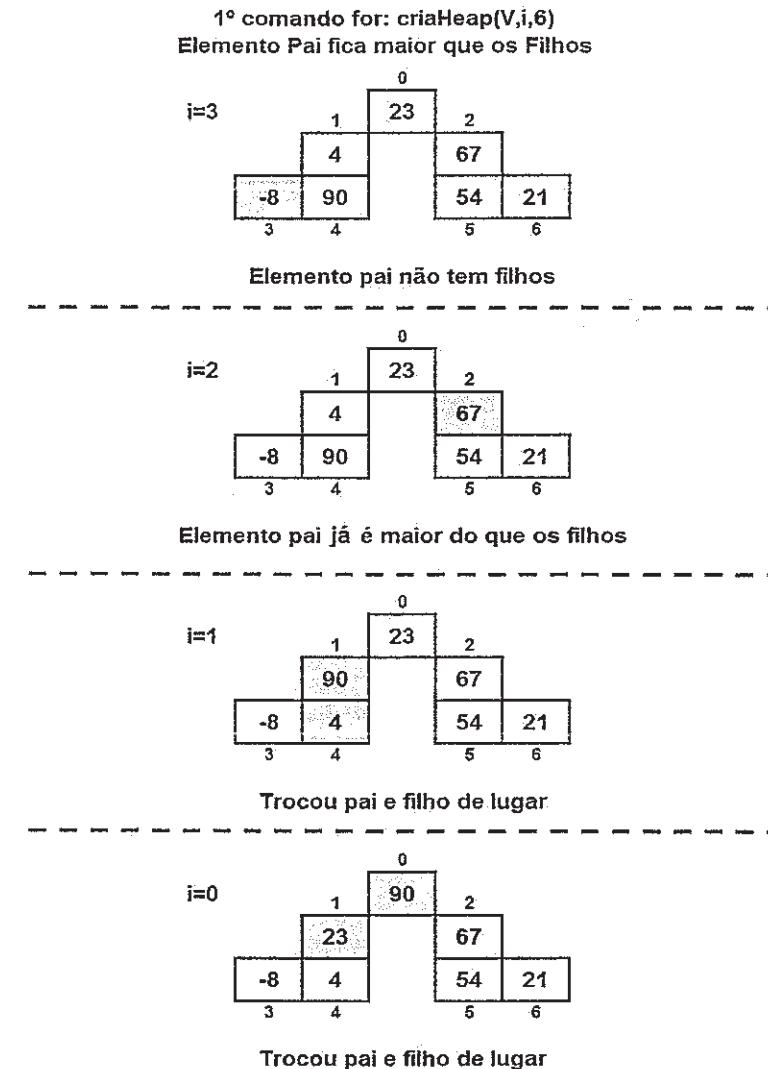


FIGURA 3.19

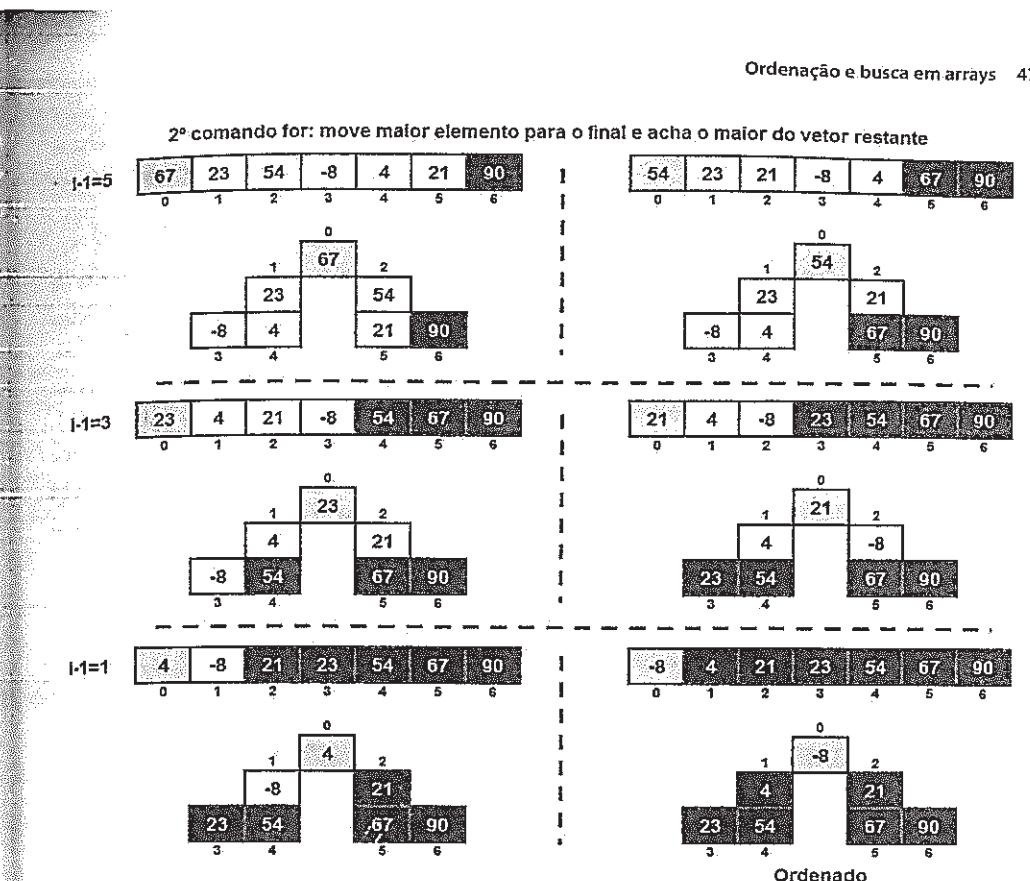


FIGURA 3.20

Considerando um array com  $N$  elementos, o tempo de execução do heap sort é sempre de ordem  $O(N \log N)$ . Como se pode notar, a eficiência do heap sort não depende da ordem inicial dos elementos.



Na prática, o heap sort é mais lento do que o quick sort. Exceto no pior caso, quando o quick sort tem complexidade  $O(N^2)$ , algo inaceitável para grandes conjuntos de elementos.

### 3.4 ORDENAÇÃO DE UM ARRAY DE STRUCT

Nas seções anteriores, vimos como realizar a ordenação de um conjunto de dados, mais especificamente um array de inteiros.



Como posso fazer para ordenar o meu array se nele estiver armazenado outro tipo de informação, como uma estrutura (struct)?

É importante lembrarmos que toda ordenação é feita utilizando como base uma chave específica. Esta chave é o “campo” utilizado para a comparação durante o processo de ordenação. No caso de uma estrutura, a chave é o campo da **struct** usado para a comparação.

A Figura 3.21 mostra dois exemplos de ordenação utilizando o algoritmo insertion sort (Seção 3.2.3) em uma **struct**. Nestes exemplos, optamos por usar uma estrutura que representa os dados associados a um aluno: número de matrícula, nome e três notas (linhas 1-5). O primeiro exemplo mostra a ordenação feita com base no número de matrícula do aluno (linhas 6-15). Perceba que a única mudança com relação à ordenação de um array de inteiros, apresentada na Figura 3.7, é a especificação do campo matrícula (**mat**) em cada posição do array (linha 11). Esta é nossa chave de comparação. Já o segundo exemplo realiza a ordenação com base no nome do aluno (linhas 17-26). Neste caso, além de especificarmos o campo **nome** em cada posição do array, temos também que utilizar a função **strcmp()** para realizar a comparação de duas strings (linha 22). Neste exemplo, é importante lembrar as saídas da função **strcmp()**:

- **strcmp(str1,str2) == 0:** str1 é igual a str2.
- **strcmp(str1,str2) > 0:** str1 vem depois de str2 no dicionário.
- **strcmp(str1,str2) < 0:** str1 vem antes de str2 no dicionário.

### Ordenação de um array de struct

```

01 struct aluno{
02     int mat;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 void insertionSortMatricula(struct aluno *V, int N){
07     int i, j;
08     struct aluno aux;
09     for(i = 1; i < N; i++){
10         aux = V[i];
11         for(j=i; (j>0) && (aux.mat<V[j-1].mat); j--)
12             V[j] = V[j - 1];
13         V[j] = aux;
14     }
15 }
16
17 void insertionSortNome(struct aluno *V, int N){
18     int i, j;
19     struct aluno aux;
20     for(i = 1; i < N; i++){
21         aux = V[i];
22         for(j=i; (j>0) && (strcmp(aux.nome,V[j-1].nome)<0); j--)
23             V[j] = V[j-1];
24         V[j] = aux;
25     }
26 }
```

FIGURA 3.21

A Figura 3.22 mostra um exemplo de ordenação usando como chave o nome do aluno.

```

Ordenação de um array de struct pelo nome do aluno
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main(){
05     int i;
06     struct aluno V[4] = {{2,"Andre",9.5,7.8,8.5},
07                           {4,"Ricardo",7.5,8.7,6.8},
08                           {1,"Bianca",9.7,6.7,8.4},
09                           {3,"Ana",5.7,6.1,7.4}};
10    insertionSortNome(V, 4);
11    for(i = 0; i < 4; i++)
12        printf("%d %s\n", V[i].mat, V[i].nome);
13
14    system("pause");
15    return 0;
16 }
```

FIGURA 3.22

### 3.5 ORDENAÇÃO EXTERNA

Os métodos de ordenação vistos até agora permitem ordenar apenas dados que estejam dentro da memória principal (RAM) do computador, sendo por isso chamados de métodos de ordenação **interna** ou **in-place**. Porém, esses métodos não são úteis quando a quantidade de dados a ser ordenada é maior do que a memória do computador. Neste tipo de situação, precisamos de um método de ordenação **externa**.



A ordenação externa busca ordenar conjuntos de dados maiores do que a memória principal (RAM) disponível no computador. Neste caso, a ordenação deve ser feita em arquivos, como mostra a Figura 3.23.

Na **ordenação externa** o conjunto de dados a ser ordenado não cabe na memória principal (está armazenado em memória secundária, por exemplo, um arquivo no disco rígido). Neste tipo de ordenação, os dados são acessados sequencialmente ou em grandes blocos, de modo que não temos acesso imediato a qualquer elemento. Além disso, os algoritmos devem minimizar o número de acesso às unidades de memória secundária (quando ocorre transferência de dados entre as memórias interna e externa) a fim de ter um bom desempenho, já que o custo de acesso à memória secundária é muito maior.



**FIGURA 3.2**

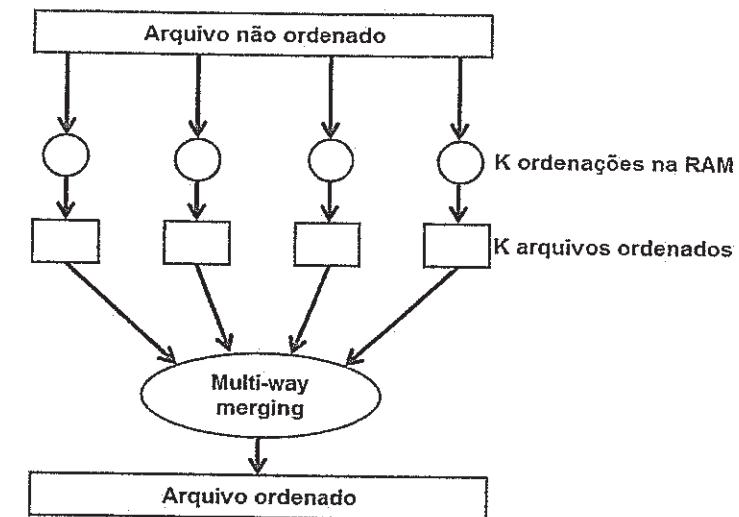
### 3.5.1 Merge sort externo

Um dos métodos mais importantes de ordenação externa é o **merge sort externo**. Seu funcionamento é similar ao merge tradicional: o algoritmo parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos. Sendo assim, o algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combiná-los por meio de intercalação (merge). A diferença está em que, agora, temos que considerar que os dados estão em um arquivo e que a etapa de intercalação será realizada entre arquivos.

A Figura 3.24 exemplifica o passo a passo de como funciona a ordenação em arquivos. Suponha que seja possível carregar na memória principal (RAM) do computador N registros de dados do arquivo a ser ordenado:

1. Carregar parte do arquivo na RAM (N registros de dados).
  2. Ordenar os dados na RAM com um algoritmo **in-place** (ex: quick sort).
  3. Salvar os dados ordenados em um arquivo separado.
  4. Repetir os passos de 1 a 3 até terminar o arquivo original. Ao final, teremos K arquivos ordenados.
  5. *Multi-way merging:* intercalar K blocos ordenados:
    - a. Criar K+1 buffers de tamanho  $N/(K+1)$ , sendo 1 buffer de saída e K buffers de entrada.
    - b. Carregar parte dos arquivos ordenados nos **buffers de entrada** e intercalar no **buffer de saída**.
    - c. Se um buffer de entrada ficar vazio, carregar mais dados do respectivo arquivo.
    - d. Se o buffer de saída ficar cheio, salvar os dados no arquivo final.

Perceba que o *multi-way merging* nada mais é do que a etapa de merge do algoritmo merge sort adaptada para fazer a intercalação de K conjuntos ordenados de dados, e não apenas dois.



**FIGURA 3.24**

A Figura 3.25 mostra um exemplo de programa de teste para a ordenação externa. Nele, criamos uma função para gerar um arquivo com valores inteiros aleatórios, `criArquivoTeste` (linhas 7-15), como o mostrado na Figura 3.23. Esse arquivo é então passado para a função `mergeSortExterno`, que irá ordená-lo. Note que estamos definindo um valor para N (linha 5). O valor será o máximo de memória disponível para a criação dos buffers durante o processo de ordenação.

**Exemplo: mergeSortExterno**

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <time.h>
04
05 #define N 100
06
07 void criArquivoTeste(char *nome){
08     int i;
09     FILE *f = fopen(nome, "w");
10    srand(time(NULL));
11    for(i=1; i < 1000; i++)
12        fprintf(f,"%d\n",rand());
13    fprintf(f,"%d",rand());
14    fclose(f);
15 }
16
17 int main(){
18     criArquivoTeste("dados.txt");
19     mergeSortExterno("dados.txt");
20     system("pause");
21     return 0;
22 }
```

FIGURA 3.25

A implementação da função **mergeSortExterno** é mostrada na Figura 3.26. Esta função utiliza outra função para criar os arquivos ordenados, **criaArquivosOrdenados** (linha 3), e retorna o número de arquivos ordenados gerados, K. Em seguida, a função calcula o tamanho, T, dos K+1 buffers necessários para a etapa de merge, apaga o arquivo original e chama a função responsável pela etapa de intercalação (linhas 4-6). Por fim, a função apenas apaga os arquivos temporários gerados (linhas 7-10).

**Merge sort externo**

```

01 void mergeSortExterno(char *nome) {
02     char novo[20];
03     int K = criaArquivosOrdenados(nome);
04     int i, T = N / (K + 1);
05     remove(nome);
06     merge(nome, K, T);
07     for(i=0; i<K; i++){
08         sprintf(novo,"Temp%d.txt",i+1);
09         remove(novo);
10     }
11 }
```

FIGURA 3.26

A criação dos arquivos ordenados é descrita pela função **criaArquivosOrdenados**, mostrada na Figura 3.27. Basicamente, esta função abre o arquivo original (linha 16) e, enquanto houver dados no arquivo (linhas 17-27), lê um valor do arquivo e armazena no vetor V (linha 18). Sempre que o total de valores lidos for igual ao tamanho do vetor, os dados são ordenados usando a função **quickSort** (Figura 3.13) e um novo arquivo temporário é gerado (linhas 20-26). Uma vez terminado o arquivo, verifica-se se não existem ainda dados no buffer (linha 28). Em caso afirmativo, um novo arquivo de dados ordenados é gerado (linhas 29-32). Por fim, a função retorna o número de arquivos ordenados gerados (linha 35). Perceba que utilizamos uma função auxiliar **salvaArquivo** (linhas 1-11) apenas para facilitar a etapa de salvar os dados de um vetor em um arquivo.

**Merge sort externo criaArquivosOrdenados**

```

01 void salvaArquivo(char *nome, int *V, int tam,
                     int mudaLinhaFinal){
02     int i;
03     FILE *f = fopen(nome, "a");
04     for(i=0; i < tam-1; i++)
05         fprintf(f,"%d\n",V[i]);
06     if(mudaLinhaFinal == 0)
07         fprintf(f,"%d",V[tam-1]);
08     else
09         fprintf(f,"%d\n",V[tam-1]);
10    fclose(f);
11 }
12 int criaArquivosOrdenados(char *nome) {
13     int V[N];
14     char novo[20];
15     int K = 0, total = 0;
16     FILE *f = fopen(nome, "r");
17     while(!feof(f)){
18         fscanf(f,"%d",&V[total]);
19         total++;
20         if(total == N){
21             K++;
22             sprintf(novo,"Temp%d.txt",K);
23             quickSort(V,0,N-1);
24             salvaArquivo(novo, V, total,0);
25             total = 0;
26         }
27     }
28     if(total > 0){
29         K++;
30         sprintf(novo,"Temp%d.txt",K);
31         qsort(V,total,sizeof(int),compara);
32         salvaArquivo(novo, V, total,0);
33     }
34     fclose(f);
35     return K;
36 }
```

FIGURA 3.27

Uma vez gerados os arquivos ordenados, é preciso fazer a intercalação deles utilizando a função **merge**, mostrada na Figura 3.28. Para facilitar o gerenciamento dos buffers, criamos uma **struct arquivo** (linhas 1-4), a qual armazena o ponteiro do arquivo ordenado relativo àquele buffer, assim como o seu tamanho máximo e a posição onde estamos no buffer. A função **merge** recebe como parâmetros o nome do arquivo a ser salvo com os dados ordenados, o número de arquivos (K) e o tamanho máximo de cada buffer (T). Inicialmente, a função cria o buffer de saída e os buffers de entrada (linhas 8-10). Para cada buffer de entrada é aberto o seu arquivo, alocada memória para o seu buffer e T elementos do arquivo são copiados para dentro do buffer, utilizando a função **preencheBuffer** (linhas 11-16). Em seguida, enquanto for possível achar o menor elemento de um dos buffers de entrada (linha 18), copiamos esse elemento para o buffer de saída (linhas 19-20) e, sempre que a quantidade de elementos no buffer de saída

### Merge sort externo: multi-way merging

```

01 struct arquivo{
02     FILE *f;
03     int pos, MAX, *buffer;
04 };
05 void merge(char *nome, int K, int T){
06     char novo[20];
07     int i;
08     int *saida = (int*)malloc(T*sizeof(int));
09     struct arquivo* arq;
10     arq=(struct arquivo*)malloc(K*sizeof(struct arquivo));
11     for(i=0; i < K; i++){
12         sprintf(novo,"Temp%d.txt",i+1);
13         arq[i].f = fopen(novo,"r");
14         arq[i].buffer = (int*)malloc(T*sizeof(int));
15         preencheBuffer(&arq[i],T);
16     }
17     int menor, qtdSaida = 0;
18     while(procuraMenor(arq,K,T,&menor) == 1){
19         saida[qtdSaida] = menor;
20         qtdSaida++;
21         if(qtdSaida == T){
22             salvaArquivo(nome, saida, T,1);
23             qtdSaida = 0;
24         }
25     }
26     if(qtdSaida != 0)
27         salvaArquivo(nome, saida, qtdSaida,1);
28
29     for(i=0; i<K; i++)
30         free(arq[i].buffer);
31     free(arq);
32     free(saida);
33 }

```

FIGURA 3.28

for igual ao tamanho máximo do buffer (T), os dados são salvos em arquivo e a quantidade de dados no buffer é zerada (linhas 21-24). Uma vez que não existam mais dados nos buffers de entrada (linha 18), verificamos se ainda existem dados no buffer de saída e, em caso afirmativo, salvamo-los em arquivo (linhas 26-27). Por fim, liberamos a memória associada a cada buffer.

A Figura 3.29 apresenta as funções responsáveis por copiar os dados do arquivo para o buffer, **preencheBuffer** (linhas 1-17), e por achar o menor elemento em um dos buffers de entrada, **procuraMenor** (linhas 18-38). Basicamente, a função **preencheBuffer** apenas lê T

### Merge sort externo: manipulando o buffer

```

01 void preencheBuffer(struct arquivo* arq, int T){
02     int i;
03     if(arq->f == NULL)
04         return;
05     arq->pos = 0;
06     arq->MAX = 0;
07     for(i=0; i < T; i++){
08         if(!feof(arq->f)){
09             fscanf(arq->f,"%d",&arq->buffer[arq->MAX]);
10             arq->MAX++;
11         }else{
12             fclose(arq->f);
13             arq->f = NULL;
14             break;
15         }
16     }
17 }
18 int procuraMenor(struct arquivo* arq,int K,int T,
19                   int* menor){
20     int i, idx = -1;
21     for(i=0; i < K; i++){
22         if(arq[i].pos < arq[i].MAX){
23             if(idx == -1)
24                 idx = i;
25             else{
26                 if(arq[i].buffer[arq[i].pos] <
27                     arq[idx].buffer[arq[idx].pos])
28                     idx = i;
29             }
30         }
31         if(idx != -1){
32             *menor = arq[idx].buffer[arq[idx].pos];
33             arq[idx].pos++;
34             if(arq[idx].pos == arq[idx].MAX)
35                 preencheBuffer(&arq[idx],T);
36         }
37     }
38 }

```

FIGURA 3.29

elementos de um arquivo e armazena no respectivo buffer (linhas 7-16). Caso o arquivo termine antes que se possam ler os  $T$  elementos, ele é fechado (linhas 12-14). Perceba que a função não é executada se o arquivo já tiver sido fechado anteriormente (linhas 3-4).

Já a função **procuraMenor** percorre todos os buffers de entrada que ainda contêm dados (linhas 20-21) e os compara para descobrir o índice do buffer (**idx**) cuja posição atual (**pos**) é a menor dentre todos os buffers (linhas 22-27).

- Caso se tenha encontrado o menor elemento (linha 30), este é copiado para a variável passada por **referência**, a posição atual desse buffer de entrada é incrementada e, caso necessário, ele é novamente preenchido com dados do arquivo (linhas 31-34). Ao final, a função retorna **UM** para indicar sucesso na operação;
- Caso não se tenha encontrado o menor elemento, significa que não mais existem dados nos buffers de entrada e o processo de intercalação deve terminar. Assim, a função retorna **ZERO** para indicar uma falha na operação (linha 37).

### Complexidade

Diferente dos métodos de ordenação interna, uma boa medida de complexidade para a ordenação externa é o número de vezes que um elemento é lido/escrito na memória externa. No caso do algoritmo apresentado, temos:

- A leitura do arquivo não ordenado ( $M$  elementos).
- $K$  gravações sequenciais de  $N$  elementos ordenados ( $K = M/N$ ).
- Cada buffer de entrada comporta  $T$  elementos ( $T = N/(K+1)$ ).
- São necessárias  $K+1$  passadas por cada um dos  $K$  arquivos ordenados para fazer a intercalação, o que resulta em  $K^*(K+1)$  operações de seek no disco.

Por esses cálculos, temos que a complexidade do merge sort externo, em termos de seeks, é  $O(K^2)$ . Como  $K$  é diretamente proporcional ao tamanho do arquivo ( $M$ ), podemos dizer que a sua complexidade é  $O(M^2)$ .



A complexidade do merge sort externo é muita alta. Como melhorar o desempenho?

Bons métodos de ordenação externa envolvem, ao todo, menos de 10 passadas sobre o arquivo. Existem várias maneiras de reduzir esse tempo:

- Usar mais hardware durante a ordenação (memória principal, canais de I/O).
- Fazer a intercalação em mais de uma etapa, o que aumenta o tamanho do buffer e reduz o número de seeks (**Multistep merging**, em vez de **Multi-way merging**).
- Aumentar o tamanho dos blocos ordenados de dados (**replacement selection**).

## 3.6 BUSCA EM ARRAYS

A busca nada mais é do que o ato de procurar um elemento em um conjunto de dados.



A operação de busca visa responder se determinado valor está ou não presente em um conjunto de elementos (por exemplo, um array).



A chave de busca é o "campo" do item utilizado para comparação. É por meio dele que sabemos se dado elemento é o que buscamos.

Existem vários tipos de busca. Sua utilização depende de como são os dados:

- Os dados estão estruturados (array, lista, árvore etc.)? Existe também a busca em dados não estruturados.
- Os dados estão ordenados?
- Existem valores duplicados?

Nas próximas seções, iremos abordar a busca em dados armazenados em arrays, sejam eles ordenados ou não.

### 3.6.1 Busca sequencial ou linear

A busca sequencial é a estratégia de busca mais simples que existe. Basicamente, o algoritmo percorre o array que contém os dados desde a sua primeira posição até a última.



A busca sequencial assume que os dados não estão ordenados, por isso a necessidade de percorrer o array do seu início até o seu fim.

Para cada posição do array, o algoritmo compara se a posição atual do array é igual ao valor buscado, como mostra a sua implementação, na Figura 3.30. Se os valores forem iguais, o valor existe e o algoritmo retorna a sua posição no array (linha 5). Do contrário, a busca continua com a próxima posição do array. Ao término do array, o algoritmo retorna o valor -1, indicando que o valor buscado não existe nesse array (linha 7). Um exemplo de busca é mostrado na Figura 3.31.

**Busca sequencial ou linear**

```

01 int buscaLinear(int *V, int N, int elem){
02     int i;
03     for(i = 0; i < N; i++){
04         if(elem == V[i])
05             return i;//elemento encontrado
06     }
07     return -1;//elemento não encontrado
08 }
```

FIGURA 3.30

V	0	1	2	3	4	5	6
	23	4	67	-8	54	90	21

elem 54 Elemento procurado

i=0	0	1	2	3	4	5	6	Valor diferente: continua a busca
	23	4	67	-8	54	90	21	
i=1	0	1	2	3	4	5	6	Valor diferente: continua a busca
	23	4	67	-8	54	90	21	
i=2	0	1	2	3	4	5	6	Valor diferente: continua a busca
	23	4	67	-8	54	90	21	
i=3	0	1	2	3	4	5	6	Valor diferente: continua a busca
	23	4	67	-8	54	90	21	
i=4	0	1	2	3	4	5	6	Valor igual: termina a busca
	23	4	67	-8	54	90	21	

FIGURA 3.31



Quanto tempo demora para executar esse algoritmo de busca?

Considerando um array com N elementos:

- $O(1)$ , melhor caso: o elemento é o primeiro do array.
- $O(N)$ , pior caso: o elemento é o último do array ou não existe.
- $O(N/2)$ , caso médio.

Procurar por determinado valor em um array desordenado é uma tarefa bastante cara. Porem, quando organizamos o array segundo alguma ordem, isto é, quando ordenamos nosso array, a tarefa de busca se torna muito mais fácil.

 A busca sequencial ordenada assume que os dados estão ordenados. Assim, se o elemento procurado for menor do que o valor em determinada posição do array, temos a certeza de que ele não estará no restante do array. Isso evita a necessidade de percorrer o array do seu início até o seu fim.

A Figura 3.32 mostra a implementação da busca sequencial ordenada. Para cada posição do array, o algoritmo compara se a posição atual do array é igual ao valor buscado (linha 4). Se os valores forem iguais, o valor existe e o algoritmo retorna a sua posição no array (linha 5). Do contrário, verificamos se o elemento procurado é menor do que o valor da posição atual do array (linha 7). Em caso afirmativo, o algoritmo retorna o valor -1, indicando que o valor buscado não existe (linha 8). Do contrário, a busca continua com a próxima posição do array. Ao término do array, o algoritmo retorna o valor -1, indicando que o valor buscado é maior do que todos os valores nesse array (linha 10). Um exemplo de busca é mostrado na Figura 3.33.

**Busca sequencial ou linear ordenada**

```

01 int buscaOrdenada(int *V, int N, int elem){
02     int i;
03     for(i = 0; i < N; i++){
04         if(elem == V[i])
05             return i;//elemento encontrado
06     }
07     else
08         if(elem < V[i])
09             return -1;//parar a busca
10    }
11 }
```

FIGURA 3.32

V	0	1	2	3	4	5	6
	-8	4	21	23	54	67	90

elem 34 Elemento procurado

i=0	0	1	2	3	4	5	6	Valor diferente: continua a busca
	-8	4	21	23	54	67	90	
i=1	0	1	2	3	4	5	6	Valor diferente: continua a busca
	-8	4	21	23	54	67	90	
i=2	0	1	2	3	4	5	6	Valor diferente: continua a busca
	-8	4	21	23	54	67	90	
i=3	0	1	2	3	4	5	6	Valor diferente: continua a busca
	-8	4	21	23	54	67	90	
i=4	0	1	2	3	4	5	6	Valor é maior: elemento não existe
	-8	4	21	23	54	67	90	

FIGURA 3.33



Ordenar um array também tem um custo. Este custo é superior ao custo da busca sequencial no seu pior caso.

Isso significa que, se for para fazer a busca de um único elemento, não vale a pena ordenar o array. Porém, se mais de um elemento for recuperado do array, o esforço de ordenar o array irá valer a pena.

### 3.6.2 Busca binária

Vimos que fazer a busca em um array ordenado representa um ganho de tempo, pois podemos terminar a busca mais cedo se o elemento procurado for menor que o valor da posição atual do array. Porém, a busca sequencial é uma estratégia de busca extremamente simples, que percorre todo o array, linearmente. Uma estratégia mais sofisticada é a busca binária.



A busca binária é uma estratégia baseada na ideia de *dividir para conquistar*. A cada passo, esse algoritmo analisa o valor do meio do array. Caso o valor seja igual ao elemento procurado, a busca termina. Do contrário, a busca continua na metade do array que condiz com o valor procurado.

A Figura 3.34 mostra a implementação da busca binária. Dado o **início** e o **final** do array, o algoritmo calcula a posição do **meio** do array (linha 6). Em seguida, o valor desta posição é comparado com o elemento buscado (linha 7). Se o elemento buscado tiver valor menor, definimos a posição do **meio** como sendo o novo **final** do array (linha 8). Se o elemento buscado tiver valor maior, definimos a posição do **meio** como sendo o novo **início** do array (linha 11). Se os valores forem iguais, o algoritmo retorna o valor **meio** (linha 13). Esse processo de busca

```
Busca binária
01 int buscaBinaria(int *V, int N, int elem) {
02     int i, inicio, meio, final;
03     inicio = 0;
04     final = N-1;
05     while(inicio <= final){
06         meio = (inicio + final)/2;
07         if(elem < V[meio])
08             final = meio-1;//busca na metade da esquerda
09         else
10             if(elem > V[meio])
11                 inicio = meio+1;//busca na metade direita
12             else
13                 return meio;
14     }
15     return -1;//elemento não encontrado
16 }
```

FIGURA 3.34

continua, enquanto o valor do **início** for menor do que o **final**. Do contrário, a busca termina e o algoritmo retorna o valor -1, indicando que o elemento não existe no array. Um exemplo de busca binária é mostrado na Figura 3.35.

V	0	1	2	3	4	5	6	7	8	9
	-8	-5	1	4	14	21	23	54	67	90
<b>elem</b> 4 Elemento procurado										
meio=4	-8	-5	1	4	14	21	23	54	67	90
Valor é menor: buscar no início										
meio=1	-8	-5	1	4	14	21	23	54	67	90
Valor é maior: buscar no final										
meio=2	-8	-5	1	4	14	21	23	54	67	90
Valor é maior: buscar no final										
meio=3	-8	-5	1	4	14	21	23	54	67	90
Valor é igual: terminar a busca										

FIGURA 3.35



A busca binária é uma estratégia de busca muito mais eficiente do que a busca sequencial ordenada.

Considerando um array com N elementos, o tempo de execução da busca binária é:

- $O(1)$ , melhor caso: o elemento procurado está no meio do array.
- $O(\log_2 N)$ , pior caso: o elemento não existe.
- $O(\log_2 N)$ , caso médio.

Para se ter uma ideia da sua vantagem, se  $N = 1000$  o algoritmo de busca sequencial irá executar 1.000 comparações no pior caso, enquanto a busca binária irá executar apenas 10 comparações.

### 3.6.3 Busca em um array de struct

Nas seções anteriores, vimos como realizar a busca por um elemento em um conjunto de dados, mais especificamente um array de inteiros.



Como posso fazer uma busca se no meu array estiver armazenado outro tipo de informação, como uma estrutura (**struct**)?

É importante lembrarmos que toda busca é feita utilizando como base uma chave específica. Esta chave é o “campo” utilizado para comparação. No caso de uma estrutura, a chave é o campo da **struct** usado para a comparação.

A Figura 3.36 mostra dois exemplos de buscas lineares em uma **struct**. Nesses exemplos, optamos por usar uma estrutura que representa os dados associados a um aluno: número de matrícula, nome e três notas (linhas 1-5). O primeiro exemplo mostra a busca feita com base no número de matrícula do aluno (linhas 6-13). Perceba que a única mudança com relação à busca linear, apresentada na Figura 3.30, é a especificação do campo matrícula (**mat**) em cada posição do array (linha 9). Esta é nossa chave de comparação. Já o segundo exemplo realiza a busca com base no nome do aluno (linhas 15-22). Neste caso, além de especificarmos o campo **nome** em cada posição do array, temos também que utilizar a função **strcmp()** para realizar a comparação de duas strings (linha 18). A Figura 3.37 mostra um exemplo de busca linear usando como chave o nome do aluno.

### Busca em um array de struct

```

01 struct aluno{
02     int mat;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 int buscaLinearMatricula(struct aluno *V, int N, int elem){
07     int i;
08     for(i = 0; i<N; i++){
09         if(elem == V[i].mat)
10             return i;//elemento encontrado
11     }
12     return -1;//elemento não encontrado
13 }
14 int buscaLinearNome(struct aluno *V, int N, char* elem){
15     int i;
16     for(i = 0; i<N; i++){
17         if(strcmp(elem,V[i].nome)==0)
18             return i;//elemento encontrado
19     }
20     return -1;//elemento não encontrado
21 }
22 }
```

FIGURA 3.36

### Busca de um aluno pelo nome

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 int main(){
04     struct aluno V[4] = {{2,"Andre",9.5,7.8,8.5},
05                           {4,"Ricardo",7.5,8.7,6.8},
06                           {1,"Bianca",9.7,6.7,8.4},
07                           {3,"Ana",5.7,6.1,7.4}};
08     int pos = buscaLinearNome(V, 4, "Andre");
09     if(pos != -1)
10         printf("Nome encontrado\n");
11     else
12         printf("Nome NAO encontrado\n");
13
14     system("pause");
15     return 0;
16 }
```

FIGURA 3.37

## 3.7 EXERCÍCIOS

- 1) Defina, usando as suas palavras, o problema de ordenação.
- 2) Defina, usando as suas palavras, o problema de encontrar o menor valor de um array.
- 3) Cite exemplos de aplicação que envolvam o problema de ordenação no mundo real.
- 4) Modifique os algoritmos de ordenação vistos para que ordenem os números de forma decrescente (do maior para o menor).
- 5) Dado um número e um array ordenado, escreva um algoritmo para inserir esse valor na sua posição correta. Desloque os outros números, se necessário (considere que há espaço vago no array).
- 6) Escreva um algoritmo que retorne quantas vezes dado número aparece em um array. Reescreva a função para considerar um array ordenado.
- 7) Modifique o algoritmo de ordenação por inserção para que este retorne o número de trocas realizadas.
- 8) Para cada sequência de números faça um teste de mesa com cada método de ordenação visto. Mostre o número de comparações e trocas realizadas por cada método:
  - 1, 4, 7, 9, 14, 17.
  - 21, 19, 17, 9, 5, 1.
  - 15, 27, 2, 18, 11, 6.
  - 2, 4, 6, 8, 10, 12, 11, 9, 7, 5, 3, 1.
  - 2, 4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11.
  - 18, 29, 17, 29, 23, 21, 23, 8, 14, 6.

- 9) Qual o resultado de se aplicar o algoritmo de particionamento no seguinte array: 23 12 56  
48 20 98 75 77 45 15?
- 10) Modifique o algoritmo *bubble sort*. O algoritmo deverá receber uma *string* e colocar as suas letras em ordem crescente.
- 11) Modifique o algoritmo *bubble sort* para que possa ordenar um conjunto de alunos por seus nomes. Cada aluno é representado por uma estrutura contendo seu nome e número de matrícula.

## CAPÍTULO 4

# Tipo abstrato de dados – TAD

## 4.1 DEFINIÇÃO

Quando iniciamos nossos estudos em uma linguagem de programação, vários conceitos foram apresentados. Um desses conceitos foi o de **tipo de dados**.



Um **tipo de dado** define o conjunto de **valores** (domínio) e **operações** que uma variável pode assumir.

Por exemplo, o tipo **char**, da linguagem C, suporta valores inteiros que vão de -128 até +127. Além disso, esse tipo também suporta operações de soma, subtração etc. Vimos também que não possuem nenhum tipo de estrutura sobre seus valores. Porém, existem outros tipos de dados, chamados **dados estruturados** ou **estruturas de dados**. Neles, existe uma relação estrutural (que pode ser linear ou não linear) entre seus valores.



Uma **estrutura de dados** consiste em um conjunto de tipos de dados em que existe algum tipo de relacionamento lógico estrutural.

Ou seja, uma **estrutura de dados** é apenas uma forma de armazenar e organizar os dados de modo que eles possam ser usados de forma eficiente. Alguns exemplos das estruturas de dados presentes na linguagem C são os **array**, **struct**, **union** e **enum**, todas criadas a partir dos tipos de dados básicos.

Às vezes, os tipos de dados e as estruturas de dados presentes na linguagem podem não ser suficientes para nossa aplicação. Podemos necessitar de uma melhor estruturação dos dados, assim como precisamos especificar quais operações estarão disponíveis para manipular esses dados. Neste caso, convém criar um **tipo abstrato de dados**, também conhecido como **TAD**.



Um **tipo abstrato de dados**, ou **TAD**, é um conjunto de dados estruturados e as operações que podem ser executadas sobre esses dados.

Basicamente, o tipo abstrato de dados é um conjunto de valores com seu comportamento definido por operações implementadas na forma de funções. Ele é construído a partir dos tipos básicos (`int`, `char`, `float` e `double`) ou dos tipos estruturados (`array`, `struct`, `union` e `enum`) da linguagem C. Assim, tipos abstratos de dados são entidades puramente teóricas, usadas para simplificar a descrição de algoritmos abstratos, classificar e avaliar as estruturas de dados e descrever formalmente certos tipos de sistemas.



Tanto a representação quanto as operações do TAD são especificadas pelo programador. O usuário utiliza o TAD como uma caixa-preta por meio de sua interface.

Para a criação de um TAD é essencial ocultar os dados do usuário, ou seja, devemos tornar invisível a sua implementação para o usuário. Assim, o TAD é como uma caixa-preta para o usuário, que nunca tem acesso direto à informação lá armazenada. A implementação de um TAD está desvinculada da sua utilização, ou seja, quando definimos um TAD, estamos preocupados com o que ele faz e não em como ele faz.

Um TAD é, muitas vezes, implementado na forma de dois módulos: **implementação** e **interface**. O módulo de **interface** declara as funções que correspondem às operações do TAD e é visível pelo usuário. A estratégia de ocultação de informações permite a implementação e a manutenção de módulos sem afetar os programas do usuário, como mostra a Figura 4.1.

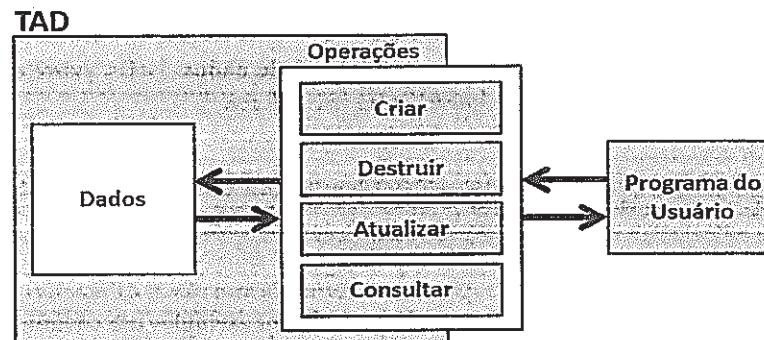


FIGURA 4.1

#### 4.1.1 Vantagens de usar um TAD

O uso de um TAD traz consigo uma série de vantagens:

- **Encapsulamento:** ao ocultarmos a implementação, fornecemos um conjunto de operações possíveis para o TAD. Isso é tudo o que o usuário precisa saber para fazer uso do TAD. O usuário não precisa de nenhum conhecimento técnico sobre como a implementação trabalha para usá-lo, tornando o seu uso muito mais fácil.

- **Segurança:** o usuário não tem acesso direto aos dados. Isso evita que ele manipule os dados de uma maneira imprópria.
- **Flexibilidade:** podemos alterar o TAD sem alterar as aplicações que o utilizam. De fato, podemos ter diferentes implementações de um TAD, desde que todos respeitem a mesma interface. Assim, podemos usar a implementação mais eficiente para determinada situação.
- **Reutilização:** a implementação do TAD é feita em um módulo diferente do programa do usuário.

#### 4.1.2 O tipo FILE

Se você já trabalhou com arquivos na linguagem C, então muito provavelmente já teve o seu primeiro contato com um tipo abstrato de dados: TAD. Trata-se do tipo FILE.

O tipo FILE é uma estrutura que contém as informações sobre um arquivo ou fluxo de texto necessário para realizar as operações de entrada ou saída sobre ele, tais como o descritor do arquivo, a posição atual dentro do arquivo, um indicador de fim de arquivo e um indicador de erro, como mostra a Figura 4.2. Tenha em mente que o conteúdo dessa estrutura parece mudar significativamente em outras implementações.

#### Exemplo: estrutura do tipo FILE

```

01 typedef struct{
02     int      level;      // nível do buffer
03     unsigned flags;    // flag de status do arquivo
04     char    *fd;        // descritor do arquivo
05     unsigned char hold; // retorna caractere se sem buffer
06     int      bsize;    // tamanho do Buffer
07     unsigned char *buffer; // buffer de transferência de dados
08     unsigned char *curp; // ponteiro atualmente ativo
09     unsigned   istemp; // indicador de arquivo temporário
10     short    token;    // usado para validação
11 }FILE;
12

```

FIGURA 4.2

Alguns acreditam que ninguém, em sã consciência, deve fazer uso direto dos campos dessa estrutura. Então, a única maneira de trabalhar com arquivos em linguagem C é declarando um ponteiro de arquivo da seguinte maneira:

```
FILE* f;
```

Desse modo, o usuário possui apenas um ponteiro para onde os dados estão armazenados, mas não pode acessá-los diretamente.



A única maneira de acessar o conteúdo do ponteiro FILE é por meio das operações definidas em sua interface.

Assim, os dados do ponteiro `f` somente podem ser acessados pelas funções de manipulação do tipo `FILE`:

- `fopen()`
- `fclose()`
- `fputc()`
- `fgetc()`
- `feof()`
- etc

#### 4.1.3 Tipo opaco

Sempre que trabalhamos com arquivos na linguagem C temos a necessidade de declarar um ponteiro do tipo `FILE` para poder manipular o arquivo.



Se o tipo `FILE` é, na verdade, uma estrutura, por que não podemos simplesmente declarar uma variável em vez de um ponteiro para ela?

Isso acontece porque o tipo `FILE` é um tipo **opaco** do ponto de vista dos usuários da biblioteca.

Apenas a própria biblioteca conhece o conteúdo do tipo e consegue manipulá-lo.



Diz-se que um tipo de dado é opaco quando ele é incompletamente definido em uma interface. Assim, os seus valores só podem ser acessados por funções específicas.

Basicamente, um tipo opaco representa uma forma de esconder os detalhes de sua implementação dos programadores que apenas farão uso do módulo ou biblioteca. Para criar um tipo opaco, utilizamos dois arquivos e os princípios de modularização:

- Arquivo “.C”: declara o tipo de dados que deverá ficar oculto do usuário.
- Arquivo “.H”: declara o tipo que irá representar os dados ocultos do arquivo “.C” e que somente poderá ser declarado pelo usuário na forma de um ponteiro.

Por meio dos tipos opacos, nossos programas utilizam uma biblioteca apenas através de ponteiros para os dados. E os dados somente podem ser acessados por funções. Note que é justamente isso que acontece quando trabalhamos com arquivos na linguagem C.

#### 4.1.4 Operações básicas de um TAD

Tipos abstratos de dados incluem as operações para a manipulação de seus dados. Essas operações variam de acordo com o TAD criado, porém as seguintes operações básicas são possíveis:

- Criação do TAD.
- Inserção de um novo elemento no TAD.
- Remoção de um elemento do TAD.
- Acesso a um elemento do TAD.
- Destrução do TAD.

#### 4.2 MODULARIZANDO O PROGRAMA

Quando trabalhamos com TAD, a convenção em linguagem C é preparamos dois arquivos para implementá-lo. Assim, podemos separar o “conceito” (definição do tipo) de sua “implementação”:

- Um arquivo “.H”: são declarados os protótipos das funções visíveis para o usuário, os tipos de ponteiro e os dados globalmente acessíveis. Aqui, é definida a **interface visível** pelo usuário.
- Um arquivo “.C”: declaração do tipo de dados que ficarão ocultos do usuário do TAD e implementação das suas funções. Aqui, é definido tudo que ficará **oculto** do usuário.



A esse processo de separação da definição do TAD em dois arquivos damos o nome de **modularização**.

Em programação, a **modularização** visa à criação de **módulos**. Um módulo é uma unidade com um propósito único e bem definido que pode ser compilado separadamente do restante do programa. Desse modo, um módulo pode ser facilmente reutilizado e modificado independentemente do programa do usuário.



Um módulo pode conter um ou mais tipos, variáveis, constantes e funções, além de uma interface apropriada com outros módulos existentes.

À medida que uma aplicação se torna maior, o uso de módulos se torna necessário. Isso ocorre porque o uso de um único arquivo causa uma série de problemas:

- Redação e modificação do programa se tornam mais difíceis, à medida que ele fica maior.
- A reutilização do código é um processo de copiar e colar.
- Qualquer modificação exige a recompilação de todo o código.
- O código é dividido em arquivos separados com entidades relacionadas.



A criação de módulos usa a estratégia de “dividir para conquistar” para a resolução de problemas.

A ideia básica da modularização é **dividir para conquistar**. Esta estratégia apresenta uma série de vantagens:

- Divide-se um problema maior em um conjunto de problemas menores.
- Aumenta as possibilidades de reutilização do código.
- Facilita o entendimento do programa.
- Encapsulam-se os dados. Assim, são agrupados os dados e processos logicamente relacionados.
- Código fica distribuído em vários arquivos. Isso permite trabalhar com equipes de programadores.
- Aumento da produtividade do programador.
- Apenas as partes alteradas do programa precisam ser recompiladas.
- Verificação independente dos módulos antes do uso no programa do usuário.
- Maior confiabilidade.

### 4.3 IMPLEMENTANDO UM TAD: PONTO

Nesta seção iremos mostrar passo a passo a criação de um TAD. Para o nosso exemplo, iremos criar um TAD que represente um ponto definido por suas coordenadas **x** e **y**. Como também estamos trabalhando com modularização, precisamos definir o tipo opaco que irá representar nosso ponto. Este tipo será um ponteiro para a estrutura que define o ponto. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **Ponto.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **Ponto.h**, ilustrado na Figura 4.3. Nele, temos que estabelecer:

- Um novo nome (**ponto**) para a **struct ponto** (linha 1). Esse é o tipo opaco que será usado sempre que se desejar trabalhar com nosso TAD.
- As funções disponíveis para trabalhar com nosso TAD (linhas 2-11) e que serão implementadas no arquivo **Ponto.c**.



No arquivo **Ponto.c** iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **Ponto.h**.

Basicamente, o arquivo **Ponto.c** (Figura 4.3) contém apenas:

- As chamadas às bibliotecas necessárias à implementação do TAD (linhas 1-3).
- A definição do tipo que descreve o nosso TAD, **struct ponto** (linhas 4-7).
- As implementações das funções definidas no arquivo **Ponto.h**, as quais serão vistas adiante.

Por estarem definidos dentro do arquivo .c, os campos da estrutura **struct ponto** não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **Ponto.h** (linha 1), que pode apenas declarar um ponteiro para ele, da seguinte forma:

**Ponto \*p;**

Arquivo **Ponto.h**

```
01 typedef struct ponto Ponto;
02 //Cria um novo ponto
03 Ponto* Ponto_cria(float x, float y);
04 //Libera um ponto
05 void Ponto_libera(Ponto* p);
06 //Acessa os valores "x" e "y" de um ponto
07 int Ponto_acessa(Ponto* p, float* x, float* y);
08 //Atribui os valores "x" e "y" a um ponto
09 int Ponto_atribui(Ponto* p, float x, float y);
10 //Calcula a distância entre dois pontos
11 float Ponto_distancia(Ponto* p1, Ponto* p2);
```

Arquivo **Ponto.c**

```
01 #include <stdlib.h>
02 #include <math.h>
03 #include "Ponto.h" //inclui os Protótipos
04 struct ponto{//Definição do tipo de dados
05     float x;
06     float y;
07 };
```

FIGURA 4.3

Para utilizar um TAD em seu programa, a primeira coisa a fazer é criar um novo ponto. Essa tarefa é executada pela função descrita na Figura 4.4. Basicamente, o que esta função faz é alocar uma área de memória para o TAD (linha 2). Esta área corresponde à memória necessária para armazenar a estrutura que define o ponto armazenado no TAD, **struct ponto**, a qual é devolvida para o nosso ponteiro para o TAD. Em seguida, essa função inicializa os campos da estrutura com os valores fornecidos pelo usuário (linhas 4-5) e retorna para o usuário o TAD (linha 7).

```
01 Ponto* Ponto_cria(float x, float y){
02     Ponto* p = (Ponto*) malloc(sizeof(Ponto));
03     if(p != NULL) {
04         p->x = x;
05         p->y = y;
06     }
07     return p;
08 }
```

FIGURA 4.4

Sempre que terminarmos de utilizar nosso **TAD**, é necessário que ele seja destruído, como mostra o código contido na Figura 4.5. Basicamente, o que temos que fazer é liberar a memória alocada para a estrutura que representa o ponto. Isso é feito utilizando apenas uma chamada da função `free()`.



Por que criar uma função para destruir o **TAD** sendo que tudo que precisamos fazer é chamar a função `free()`?

Por questões de modularização. Destruir nosso **TAD Ponto** é uma tarefa simples, porém para outros **TADs** essa pode ser uma tarefa mais complicada. Ao criar essa função, estamos escondendo a implementação dessa tarefa do usuário, que não precisa saber como um **TAD** é destruído para utilizá-lo.

```
01 void Ponto_liberar(Ponto* p){
02     free(p);
03 }
```

FIGURA 4.5

Outra tarefa importante é o acesso à informação armazenada dentro do **TAD**. É preciso criar uma função que recupere, por referência, o valor das coordenadas de um ponto, como mostra o código contido na Figura 4.6. Primeiramente, a função verifica se o **TAD** é válido, isto é, se o ponteiro `Ponto* p` é igual a `NULL`. Se essa condição for verdadeira, a função retorna o valor **ZERO** (linha 3), indicando erro na operação. Caso contrário, os dados são copiados para o conteúdo dos ponteiros passados por referência para a função (linhas 4 e 5) e a função retorna o valor **UM**, indicando sucesso nessa operação.

```
01 int Ponto_acessa(Ponto* p, float* x, float* y){
02     if(p == NULL)
03         return 0;
04     *x = p->x;
05     *y = p->y;
06     return 1;
07 }
```

FIGURA 4.6

Podemos também querer modificar o valor atribuído ao **TAD**. Assim, é preciso criar uma função que atribua um novo valor às coordenadas de um ponto, como mostra o código contido na Figura 4.7. Primeiramente, a função verifica se o **TAD** é válido, isto é, se o ponteiro `Ponto* p` é igual a `NULL`. Se essa condição for verdadeira, a função retorna o valor **ZERO** (linha 3), indicando erro na operação. Caso contrário, os dados passados por parâmetro são copiados para dentro da estrutura que representa o **TAD** (linhas 4-5) e a função retorna o valor **UM**, indicando sucesso nessa operação.

```
01 int Ponto_atribui(Ponto* p, float x, float y){
02     if(p == NULL)
03         return 0;
04     p->x = x;
05     p->y = y;
06     return 1;
07 }
```

FIGURA 4.7

Dependendo de nossa aplicação, pode ser interessante saber a distância entre dois pontos. Então, vamos criar uma função que calcule a distância entre as coordenadas de dois **TAD ponto**, como mostra o código contido na Figura 4.8. Primeiramente, a função verifica se os dois **TADs** são válidos. Se algum deles for igual a `NULL`, a função irá retornar o valor **-1** (linha 3), indicando erro na operação (não existem distâncias negativas). Caso contrário, será calculada e retornada para o usuário da função o valor da distância entre os pontos (linhas 4-6).

**Calculando a distância entre dois pontos**

```

01 float Ponto_distancia(Ponto* p1, Ponto* p2){
02     if(p1 == NULL || p2 == NULL)
03         return -1;
04     float dx = p1->x - p2->x;
05     float dy = p1->y - p2->y;
06     return sqrt(dx * dx + dy * dy);
07 }
```

FIGURA 4.8

Por fim, podemos ver na Figura 4.9 um exemplo de como podemos utilizar nosso TAD em uma aplicação.

**Exemplo: utilizando o TAD ponto**

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Ponto.h"
04 int main(){
05     float d;
06     Ponto *p,*q;
07     //Ponto r; //ERRO
08     p = Ponto_cria(10,21);
09     q = Ponto_cria(7,25);
10     //q->x = 2; //ERRO
11     d = Ponto_distancia(p,q);
12     printf("Distancia entre pontos: %f\n",d);
13     Ponto_liberar(q);
14     Ponto_liberar(p);
15     system("pause");
16     return 0;
17 }
```

FIGURA 4.9

**4.4 EXERCÍCIOS**

- 1) O que é um tipo abstrato de dados (TAD)? Qual a característica fundamental na sua utilização?
- 2) Quais as vantagens de se programar utilizando um TAD?
- 3) Desenvolva um TAD que represente um cubo. Inclua as funções de inicializações necessárias e as operações que retornem os tamanhos de cada lado, a sua área e o seu volume.

- 4) Desenvolva um TAD que represente um cilindro. Inclua as funções de inicializações necessárias e as operações que retornem a sua altura e o raio, a sua área e o seu volume.
- 5) Desenvolva um TAD que represente uma esfera. Inclua as funções de inicializações necessárias e as operações que retornem o seu raio, a sua área e o seu volume.
- 6) Desenvolva um TAD que represente um número complexo  $z = x + iy$ , em que  $i^2 = -1$ , sendo  $x$  a sua parte real e  $y$  a parte imaginária. O TAD deverá conter as seguintes funções:
  - Criar um número complexo.
  - Destruir um número complexo.
  - Soma de dois números complexos.
  - Subtração de dois números complexos.
  - Multiplicação de dois números complexos.
  - Divisão de dois números complexos.
- 7) Desenvolva um TAD que represente um conjunto de inteiros. Para isso, utilize um vetor de inteiros. O TAD deverá conter as seguintes funções:
  - Criar um conjunto vazio.
  - União de dois conjuntos.
  - Inserir um elemento no conjunto.
  - Remover um elemento do conjunto.
  - Intersecção entre dois conjuntos.
  - Diferença de dois conjuntos.
  - Testar se um número pertence ao conjunto.
  - Menor valor do conjunto.
  - Maior valor do conjunto.
  - Testar se dois conjuntos são iguais.
  - Tamanho do conjunto.
  - Testar se o conjunto é vazio.

## Listas

### 5.1 DEFINIÇÃO

O conceito de lista é algo muito comum para as pessoas. Trata-se de um relação finita de itens, todos eles contidos dentro de um mesmo tema. Vários são os exemplos possíveis de listas: itens em estoque em uma empresa, dias da semana, lista de compras do supermercado, convidados de uma festa etc.

Em ciência da computação, uma lista é uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador. Uma estrutura do tipo lista é uma sequência de elementos do mesmo tipo, como ilustrado na Figura 5.1. Seus elementos possuem estrutura interna abstruída, ou seja, sua complexidade é arbitrária e não afeta o seu funcionamento. Além disso, uma lista pode possuir elementos repetidos, assim como ser ordenada ou não, dependendo da aplicação. Como nas listas que conhecemos, a estrutura do tipo lista pode possuir  $N$  ( $N \geq 0$ ) elementos ou itens. Se  $N = 0$ , dizemos que a lista está vazia.

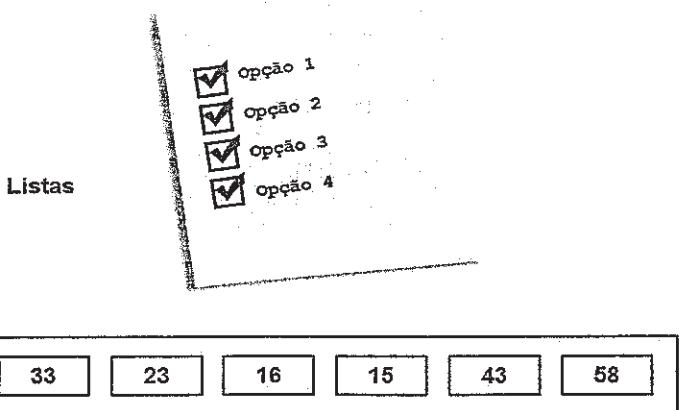


FIGURA 5.1

## 5.2 TIPOS DE LISTAS

Existem várias representações diferentes para uma lista.



Estas representações variam a depender de como os elementos são inseridos ou removidos da lista, tipo de alocação usada e tipo de acesso aos elementos.

Quanto à inserção/remoção de elementos da lista, temos:

- **Lista convencional:** pode ter elementos inseridos ou removidos de qualquer lugar dela.
- **Fila:** estrutura do tipo FIFO (First In First Out), os elementos só podem ser inseridos no final, e acessados ou removidos do início da lista. Mais informações no Capítulo 6.
- **Pilha:** estrutura do tipo LIFO (Last In First Out), os elementos só podem ser inseridos, acessados ou removidos do final da lista. Mais informações no Capítulo 8.

Quanto à alocação de memória, podemos utilizar **alocação estática** ou **dinâmica** para implementar uma lista:

- **Alociação estática:** o espaço de memória é alocado no momento da compilação do programa. É necessário definir o número máximo de elementos que a lista irá possuir.
- **Alociação dinâmica:** o espaço de memória é alocado em tempo de execução. A lista cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos.

E, independentemente de como a memória foi alocada, podemos acessar os seus elementos de duas formas:

- **Acesso sequencial:** os elementos são armazenados de forma consecutiva na memória (como em um array ou vetor). A posição de um elemento pode ser facilmente obtida a partir do início da lista.
- **Acesso encadeado:** cada elemento pode estar em uma área distinta da memória, não necessariamente consecutivas. É necessário que cada elemento da lista armazene, além da sua informação, o endereço de memória onde se encontra o próximo elemento. Para acessar um elemento, é preciso percorrer todos os seus antecessores na lista.

Nas próximas seções, serão apresentadas as diferentes implementações de listas com relação a alocação e acesso aos elementos.

## 5.3 OPERAÇÕES BÁSICAS DE UMA LISTA

Independentemente do tipo de alocação e acesso usado na implementação de uma lista, as seguintes operações básicas são possíveis:

- Criação da lista.
- Inserção de um elemento na lista.
- Remoção de um elemento da lista.
- Busca por um elemento da lista.
- Destrução da lista.
- Além de informações com tamanho, se a lista está cheia ou vazia.

### 5.3.1 A operação de inserção na lista

A operação de inserção é o ato de guardar elementos dentro da lista.



Existem três tipos de inserção: inserção no início, no final ou no meio (isto é, entre dois elementos) da lista.

A operação de inserção no meio da lista é comumente usada quando se deseja inserir um elemento de forma ordenada na lista.



A operação de inserção envolve o teste de estouro da lista, ou seja, precisamos verificar se é possível inserir um novo elemento na lista (ela ainda não está cheia).

### 5.3.2 A operação de remoção da lista

Existindo uma lista, e ela possuindo elementos, é possível excluí-los. É disso que se trata a operação de remoção.



Existem três tipos de remoção: remoção do início, do final ou do meio (isto é, entre dois elementos) da lista.

A operação de remoção do meio da lista é comumente usada quando se deseja remover um elemento específico da lista.



A operação de remoção envolve o teste de lista vazia, ou seja, precisamos verificar se existem elementos dentro da lista antes de tentar removê-los.

## 5.4 LISTA SEQUENCIAL ESTÁTICA

Uma **lista sequencial estática** ou **lista linear estática** é uma lista definida utilizando alocação estática e acesso sequencial dos elementos. Trata-se do tipo mais simples de lista possível. Ela é definida utilizando um array, de modo que o sucessor de um elemento ocupa a posição física seguinte deste.



Além do array, essa lista utiliza um campo adicional (**qtd**) que serve para indicar o quanto do array já está ocupado pelos elementos (dados) inseridos na lista.

Considerando uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo lista (ela é um **tipo opaco**), como ilustrado na Figura 5.2. Isso impede o usuário de saber como foi realmente implementada a lista, e limita o seu acesso apenas às funções que manipulam a lista.

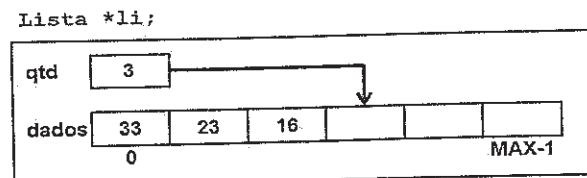


FIGURA 5.2



O uso de um array na definição de uma **lista sequencial estática** tem vantagens e desvantagens, que devem ser consideradas para um melhor desempenho da aplicação.

Várias são as vantagens em se definir uma lista utilizando um array

- Acesso rápido e direto aos elementos (índice do array).
- Tempo constante para acessar um elemento.
- Facilidade para modificar as suas informações.

Infelizmente, o uso de arrays também tem suas desvantagens:

- Definição prévia do tamanho do array e, consequentemente, da lista.
- Dificuldade para inserir e remover um elemento entre outros dois: é necessário deslocar os elementos para abrir espaço dentro do array.



Considerando suas vantagens e desvantagens, quando devo utilizar uma **lista sequencial estática**?

Em geral, usamos esse tipo de lista nas seguintes situações:

- Listas pequenas.
- Inserção e remoção apenas no final da lista.
- Tamanho máximo da lista bem definido.
- A busca é a operação mais frequente.

#### 5.4.1 Definindo o tipo lista sequencial estática

Antes de começar a implementar a nossa lista, é preciso definir o tipo de dado que será armazenado nela. Uma lista pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa lista. Este tipo será um ponteiro para a estrutura que define a lista. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando:



No arquivo **ListaSequencial.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **ListaSequencial.h**, ilustrado na Figura 5.3. Por se tratar de uma lista estática, temos que estabelecer:

- O tamanho máximo do array utilizado na lista, representada pela constante **MAX** (linha 1).
- O tipo de dado que será armazenado na lista, **struct aluno** (linhas 2-6).
- Para fins de padronização, um novo nome para o tipo lista (linha 7). Esse é o tipo que será usado sempre que se desejar trabalhar com a lista.
- As funções disponíveis para trabalhar com essa lista em especial (linhas 9-21) e que serão implementadas no arquivo **ListaSequencial.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da lista. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



No arquivo **ListaSequencial.c**, iremos definir tudo aquilo que deve ficar oculto da usuária da nossa biblioteca e implementar as funções definidas em **ListaSequencial.h**.

Basicamente, o arquivo **ListaSequencial.c** (Figura 5.3) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da lista (linhas 1-3).
- A definição do tipo que descreve o funcionamento da lista, **struct lista** (linhas 5-8).
- As implementações das funções definidas no arquivo **ListaSequencial.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que o nosso tipo lista nada mais é do que uma estrutura contendo dois campos: um inteiro **qtd**, que indica o quanto do array já está ocupado pelos elementos inseridos na lista, e o nosso array do tipo **struct aluno**, que é o tipo de dado a ser armazenado na lista.

Por estarem definidos dentro do arquivo .c, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo `main()`, apenas o seu outro nome, definido no arquivo `Listasequencial.h` (linha 7), que pode somente declarar um ponteiro para ele, da seguinte forma:

```
Lista *li;
```

#### Arquivo `Listasequencial.h`

```
01 #define MAX 100
02 struct aluno{
03     int matricula;
04     char nome[30];
05     float n1,n2,n3;
06 };
07 typedef struct lista Lista;
08
09 Lista* cria_lista();
10 void libera_lista(Lista* li);
11 int busca_lista_pos(Lista* li, int pos, struct aluno *al);
12 int busca_lista_mat(Lista* li, int mat, struct aluno *al);
13 int insere_lista_final(Lista* li, struct aluno al);
14 int insere_lista_inicio(Lista* li, struct aluno al);
15 int insere_lista_ordenada(Lista* li, struct aluno al);
16 int remove_lista(Lista* li, int mat);
17 int remove_lista_inicio(Lista* li);
18 int remove_lista_final(Lista* li);
19 int tamanho_lista(Lista* li);
20 int lista_chega(Lista* li);
21 int lista_vazia(Lista* li);
```

#### Arquivo `Listasequencial.c`

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Listasequencial.h" //inclui os protótipos
04 //Definição do tipo lista
05 struct lista{
06     int qtd;
07     struct aluno dados[MAX];
08 };
```

FIGURA 5.3

#### 5.4.2 Criando e destruindo uma lista

Para utilizar uma lista em seu programa, a primeira coisa a fazer é criar uma lista vazia. Essa tarefa é executada pela função descrita na Figura 5.4. Basicamente, o que esta função faz é a alocação de uma área de memória para a lista (linha 3). Esta área de memória corresponde à

memória necessária para armazenar a estrutura que define a lista, `struct lista`. Em seguida, a função inicializa o campo `qtd` com o valor `ZERO`. Este campo indica o quanto do array já está ocupado pelos elementos inseridos na lista, que, no caso, mostra que nenhum elemento foi inserido ainda. A Figura 5.5 indica o conteúdo do nosso ponteiro `Lista* li` após a chamada da função que cria a lista.

#### Criando uma lista

```
01 Lista* cria_lista(){
02     Lista *li;
03     li = (Lista*) malloc(sizeof(struct lista));
04     if(li != NULL)
05         li->qtd = 0;
06     return li;
07 }
```

FIGURA 5.4

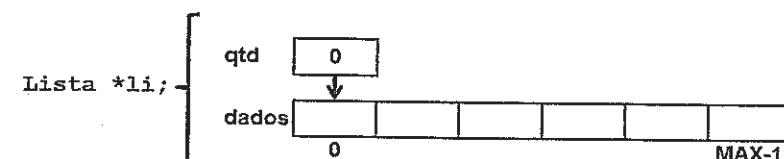


FIGURA 5.5

Destruir uma lista estática é bastante simples, como mostra o código contido na Figura 5.6. Basicamente, o que temos que fazer é liberar a memória alocada para a estrutura que representa a lista. Isso é feito utilizando apenas uma chamada da função `free()`.



Por que criar uma função para destruir a lista sendo que tudo que precisamos fazer é chamar a função `free()`?

Por questões de modularização. Destruir uma lista estática é bastante simples, porém destruir uma lista alocada dinamicamente é uma tarefa mais complicada. Ao criar essa função, estamos escondendo a implementação dessa tarefa do usuário, ao mesmo tempo que mantemos a notação utilizada por uma lista com alocação **estática** ou **dinâmica**. Desse modo, utilizar uma lista **estática** ou **dinâmica** será indiferente para o programador.

### Destruindo uma lista

```
01 void libera_lista(Lista* li) {
02     free(li);
03 }
```

FIGURA 5.6

### 5.4.3 Informações básicas sobre a lista

As operações de inserção, remoção e busca são consideradas as principais de uma lista. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a lista. Por exemplo, não podemos remover um elemento da lista se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o "status" atual da lista: seu tamanho, se ela está cheia ou se ela está vazia.

#### Tamanho da lista

Saber o tamanho de uma **lista sequencial estática** é uma tarefa relativamente simples. Isso ocorre porque essa lista possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na lista, como mostra a Figura 5.7.



Basicamente, retornar o tamanho de uma **lista sequencial estática** consiste em retornar o valor do seu campo **qtd**.

A implementação da função que retorna o tamanho da lista é mostrada na Figura 5.8. Note que essa função, em primeiro lugar, verifica se o ponteiro **Lista\*** **li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Porém, se a lista foi criada com sucesso, então é possível acessar o seu campo **qtd** e retornar o seu valor, que nada mais é do que o tamanho da lista.

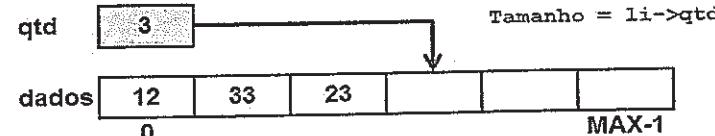


FIGURA 5.7

### Tamanho da lista

```
01 int tamanho_lista(Lista* li) {
02     if(li == NULL)
03         return -1;
04     else
05         return li->qtd;
06 }
```

FIGURA 5.8

#### Lista cheia

Saber se uma **lista sequencial estática** está cheia é outra tarefa relativamente simples. Novamente, isso ocorre porque essa lista possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na lista, como mostra a Figura 5.9.



Basicamente, retornar se uma **lista sequencial estática** está cheia consiste em verificar se o valor do seu campo **qtd** é igual a **MAX**.

A implementação da função que retorna se a lista está cheia é mostrada na Figura 5.10. Note que essa função, em primeiro lugar, verifica se o ponteiro **Lista\*** **li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma lista inválida. Porém, se a lista foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o tamanho máximo definido para o seu array (vetor) de elementos: **MAX**. Se os valores forem iguais (ou seja, lista cheia), a expressão da **linha 4** irá retornar o valor **1**. Caso contrário, irá retornar o valor **0**.

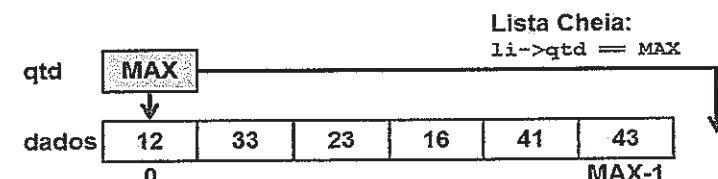


FIGURA 5.9

### Retornando se a lista está cheia

```

01 int lista_cheia(Lista* li){
02     if(li == NULL)
03         return -1;
04     return (li->qtd == MAX);
05 }
  
```

FIGURA 5.10

### Lista vazia

Saber se uma **lista sequencial estática** está vazia é outra tarefa bastante simples. Como no caso do tamanho da lista e da lista cheia, isso ocorre porque esse tipo de lista possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na lista, como mostra a Figura 5.11.



Basicamente, retornar se uma **lista sequencial estática** está vazia consiste em verificar se o valor do seu campo **qtd** é igual a **ZERO**.

A implementação da função que retorna se a lista está vazia é mostrada na Figura 5.12. Note que essa função, em primeiro lugar, verifica se o ponteiro **Lista\* li** é igual a **NONE**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma lista inválida. Porém, se a lista foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com **0**, que é o valor inicial do campo quando criamos uma lista. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da lista), a expressão da linha 4 irá retornar o valor **1**. Caso contrário, irá retornar o valor **0**.

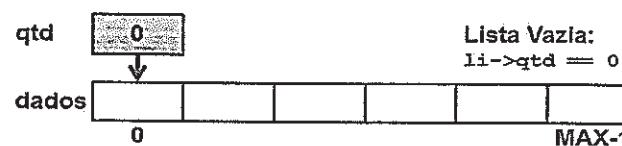


FIGURA 5.11

### Retornando se a lista está vazia

```

01 int lista_vazia(Lista* li){
02     if(li == NULL)
03         return -1;
04     return (li->qtd == 0);
05 }
  
```

FIGURA 5.12

### 5.4.4 Inserindo um elemento na lista

#### Preparando a inserção na lista

Antes de inserir um elemento em uma **lista sequencial estática**, algumas verificações são necessárias. Isso vale para os três tipos de inserção: no início, no final ou no meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.13, 5.15 e 5.17).

Primeiramente, verificamos se o ponteiro **Lista\* li** é igual a **NONE**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, precisamos verificar se ela não está cheia, isto é, se existe espaço para um novo elemento. Caso a lista esteja cheia, a função irá retornar o valor **ZERO** (linhas 4 e 5).



No caso de uma lista implementada usando um array, ela somente será considerada cheia quando a quantidade de elementos (campo **qtd**) for igual ao tamanho do array (**MAX**), indicando que o array está completamente ocupado por elementos.

#### Inserindo no início da lista

Inserir um elemento no início de uma **lista sequencial estática** é uma tarefa simples, mas um tanto trabalhosa.



Isso ocorre porque a inserção no início de uma **lista sequencial estática** necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é movimentar todos os elementos da lista uma posição para frente dentro do array. Isso deixa o início da lista (a posição **ZERO** do array) livre para inserir um novo elemento, como mostra a sua implementação na Figura 5.13. Note que as linhas 2 a 5 verificam se a inserção é possível.

Como se trata de uma inserção no início, temos que percorrer todos os elementos da lista e copiá-los uma posição para frente. Isto deve ser feito do último elemento até o primeiro, evitando assim que a cópia de um elemento sobrecreva o outro (linhas 7 e 8). Em seguida, podemos

copiar os dados que vamos armazenar para dentro da posição **ZERO** do array que representa a lista (linha 9). Por fim, devemos incrementar a quantidade (**li->qtd**) de elementos armazenados na lista e retornamos o valor **UM** (linhas 10 e 11), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 5.14.

#### Inserindo um elemento no início da lista

```

01 int insere_lista_inicio(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     if(li->qtd == MAX)//lista cheia
05         return 0;
06     int i;
07     for(i=li->qtd-1; i>=0; i--)
08         li->dados[i+1] = li->dados[i];
09     li->dados[0] = al;
10    li->qtd++;
11    return 1;
12 }
```

FIGURA 5.13

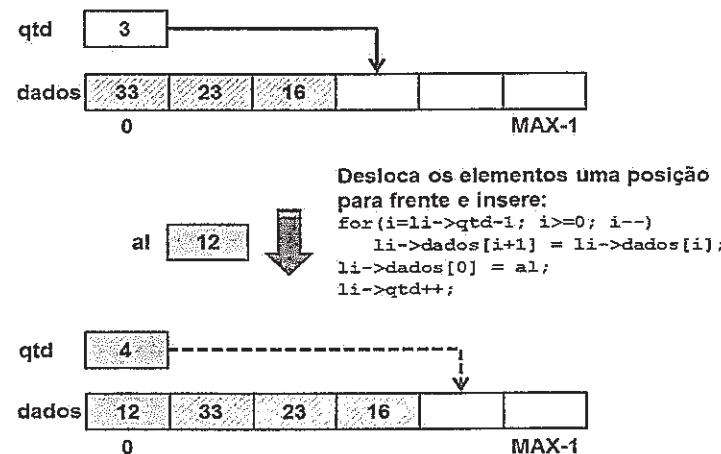


FIGURA 5.14

#### Inserindo no final da lista

Inserir um elemento no final de uma **lista sequencial estática** é uma tarefa extremamente simples.

Diferente da inserção no início, a inserção no final de uma **lista sequencial estática** não necessita que se mude o lugar dos demais elementos da lista.

Como a inserção é no final da lista, devemos inserir nosso elemento após a última posição ocupada do array que representa a lista, como mostra a sua implementação na Figura 5.15. Note que as linhas 2 a 5 verificam se a inserção é possível.

Como se trata de uma inserção no final, basta copiar os dados que vamos armazenar para dentro da posição **li->qtd** do array que representa a lista (linha 6). Por fim, devemos incrementar a quantidade (**li->qtd**) de elementos armazenados na lista e retornamos o valor **UM** (linhas 7 e 8), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 5.16.

#### Inserindo um elemento no final da lista

```

01 int insere_lista_final(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     if(li->qtd == MAX)//lista cheia
05         return 0;
06     li->dados[li->qtd] = al;
07     li->qtd++;
08     return 1;
09 }
```

FIGURA 5.15

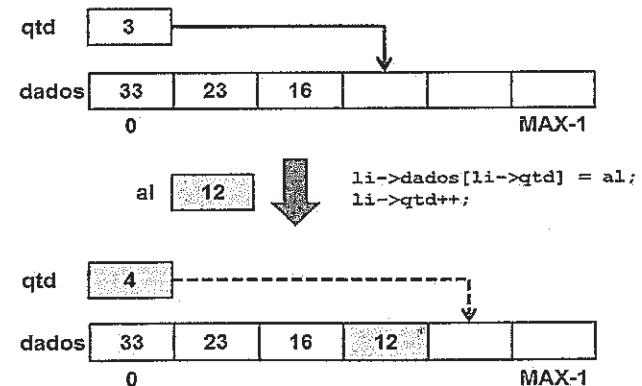


FIGURA 5.16

### Inserindo de forma ordenada na lista

Inserir um elemento de forma ordenada em uma **lista sequencial estática** é uma tarefa simples, mas trabalhosa.



Isso ocorre porque precisamos procurar o ponto de inserção do elemento na lista, o qual pode ser no início, no meio ou no final da lista. Nos dois primeiros casos (início ou meio), é preciso mudar o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é procurar em que lugar da lista será inserido o novo elemento (no caso, iremos ordenar pelo campo matrícula) e, dependendo do lugar, movimentar todos os elementos a partir daquele ponto da lista uma posição para frente dentro do array. Isso deixa aquela posição livre para inserir um novo elemento, como mostra a sua implementação na Figura 5.17. Note que as linhas 2 a 5 verificam se a inserção é possível.

Como se trata de uma inserção ordenada, temos que continuar percorrendo a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento atual for menor do que a matrícula do novo elemento a ser inserido (linhas 7 e 8). Uma vez que chegamos ao final da lista ou a um elemento com matrícula maior, iniciamos o processo de cópia dos elementos: a partir desse ponto da lista (i), todos os elementos são copiados uma posição para frente. Isto deve ser feito do último elemento até o atual, evitando assim que a cópia de um elemento sobrescreva o outro (linhas 10 e 11). Em seguida, podemos copiar os dados que vamos armazenar para dentro da posição atual (i) do array que representa a lista (linha 13). Por fim, devemos incrementar a quantidade (**li->qtd**) de elementos armazenados na lista e retornamos o valor UM (linhas 14 e 15), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 5.18.

#### Inserindo um elemento de forma ordenada na lista

```

01 int insere_lista_ordenada(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     if(li->qtd == MAX)//lista cheia
05         return 0;
06     int k, i = 0;
07     while(i<li->qtd && li->dados[i].matricula < al.matricula)
08         i++;
09
10    for(k=li->qtd-1; k >= i; k--)
11        li->dados[k+1] = li->dados [k];
12
13    li->dados[i] = al;
14    li->qtd++;
15    return 1;
16 }
```

FIGURA 5.17

**Lista Inicial**  
**Busca onde Inserir:**

dados	16	23	33		
	0				MAX-1

```
int k,i = 0;
while(i < li->qtd && li->dados[i].matricula < al.matricula)
    i++;
```

**Inserção no início ou no meio: desloca elementos**

```
for(k=li->qtd-1; k >= i; k--)
    li->dados[k+1] = li->dados [k];
```

dados	12	16	23	33		
	0					MAX-1

dados	16	19	23	33		
	0					MAX-1

**Inserir elemento**

```
li->dados[i] = al;
li->qtd++;
```

dados	16	23	33	40		
	0					MAX-1

FIGURA 5.18

### 5.4.5 Removendo um elemento da lista

#### Preparando a remoção da lista

Antes de remover um elemento de uma **lista sequencial estática**, algumas verificações são necessárias. Isso vale para os três tipos de remoção: do início, do final ou do meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.19, 5.21 e 5.23).

Primeiramente, verificamos se o ponteiro **Lista\* li** é igual a **NONE**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, precisamos verificar se ela não está vazia, isto é, se existem elementos dentro dela. Caso a lista esteja vazia, a função irá retornar o valor **ZERO** (linhas 4 e 5).



No caso de uma lista implementada usando um array, ela somente será considerada vazia quando a quantidade de elementos (campo **qtd**) for igual ao valor **ZERO**, indicando que nenhuma posição do array está ocupada por elementos.

#### Removendo do início da lista

Apesar de simples, remover um elemento do início de uma **lista sequencial estática** é uma tarefa trabalhosa.



Como na inserção no início, a remoção de um elemento do início de uma **lista sequencial estática** necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é movimentar todos os elementos da lista uma posição para trás dentro do array. Isso sobrescreve o início da lista (a posição **ZERO** do array), ao mesmo tempo que diminui o número de elementos, como mostra a sua implementação na Figura 5.19. Note que as linhas 2 a 5 verificam se a remoção é possível.

Como se trata de uma remoção do início, temos que percorrer do primeiro ao penúltimo elemento da lista e copiá-los uma posição para trás (linhas 7 e 8). Por fim, devemos diminuir em uma unidade a quantidade (**li->qtd**) de elementos armazenados na lista e retornamos o valor **UM** (linhas 9 e 10), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.20.



Note que o último elemento da lista fica duplicado. Isso não é um problema, já que aquela posição duplicada é considerada não ocupada por elementos da lista.

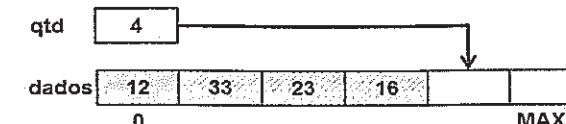
Lembre-se: o campo **qtd** indica a próxima posição vaga no final da lista.

#### Removendo um elemento do início da lista

```

01 int remove_lista_inicio(Lista* li){
02     if(li == NULL)
03         return 0;
04     if(li->qtd == 0)//lista vazia
05         return 0;
06     int k = 0;
07     for(k=0; k< li->qtd-1; k++)
08         li->dados[k] = li->dados[k+1];
09     li->qtd--;
10     return 1;
11 }
```

FIGURA 5.19



**Desloca os elementos uma posição para trás:**  
`for (k=0; k< li->qtd-1; k++)  
 li->dados[k] = li->dados[k+1];  
li->qtd--;`

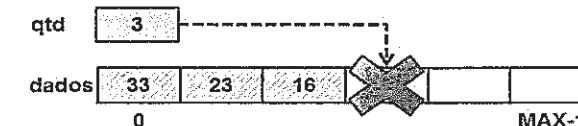


FIGURA 5.20

#### Removendo do final da lista

Remover um elemento do final de uma **lista sequencial estática** é uma tarefa extremamente simples.



Diferente da remoção do início, a remoção do final de uma **lista sequencial estática** não necessita que se mude o lugar dos demais elementos da lista. Temos apenas que alterar a quantidade de elementos na lista.

A Figura 5.21 mostra a implementação da função de remoção do final da lista. Note que as linhas 2 a 5 verificam se a remoção é possível. Como a remoção é no final da lista, basta diminuir em uma unidade a quantidade (**li->qtd**) de elementos armazenados na lista e retornar o valor **UM** (linhas 6 e 7), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.22.



Note que o elemento removido continua no final da lista. Isso não é um problema, já que aquela posição é considerada não ocupada por elementos da lista.

Lembre-se: o campo **qtd** indica a próxima posição vaga no final da lista.

### Removendo um elemento do final da lista

```

01 int remove_lista_final(Lista* li) {
02     if(li == NULL)
03         return 0;
04     if(li->qtd == 0)//lista vazia
05         return 0;
06     li->qtd--;
07     return 1;
08 }

```

FIGURA 5.21

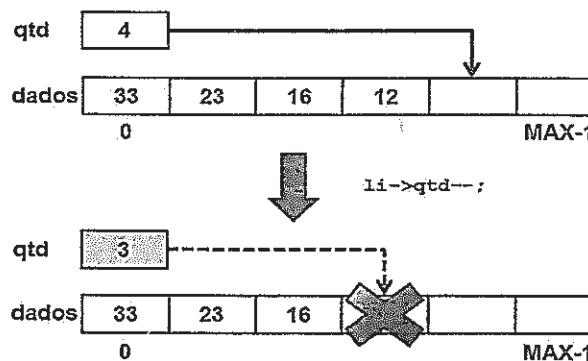


FIGURA 5.22

### Removendo um elemento específico da lista

Apesar de simples, remover um elemento específico de uma lista sequencial estática é uma tarefa trabalhosa.



Isso ocorre porque precisamos procurar o elemento a ser removido na lista, o qual pode estar no início, no meio ou no final da lista. Nos dois primeiros casos (início ou meio), é preciso mudar o lugar dos demais elementos da lista após a remoção.

Basicamente, o que temos que fazer é procurar esse elemento na lista e movimentar todos os elementos que estão à frente na lista uma posição para trás dentro do array. Isso sobrescreve o elemento a ser removido, ao mesmo tempo que diminui o número de elementos, como mostra a sua implementação na Figura 5.23. Note que as linhas 2 a 5 verificam se a remoção é possível.

No nosso exemplo, vamos remover um elemento de acordo com o campo matrícula. Assim, temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento atual for diferente do valor de matrícula procurado (linhas 7 e 8).

Terminado o processo de busca, verificamos se estamos no final da lista ou não (linha 9). Em caso afirmativo, o elemento não existe na lista e a remoção não é possível (linha 10). Caso contrário, percorremos da posição atual até o penúltimo elemento da lista e copiamos esse elemento uma posição para trás (linhas 12 e 13). Por fim, devemos diminuir em uma unidade a quantidade (*li->qtd*) de elementos armazenados na lista e retornamos o valor UM (linhas 14 e 15), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.24.

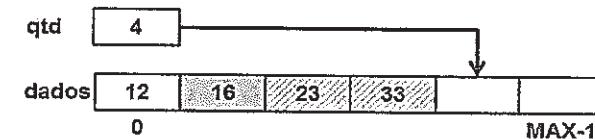
### Removendo um elemento específico da lista

```

01 int remove_lista(Lista* li, int mat) {
02     if(li == NULL)
03         return 0;
04     if(li->qtd == 0)//lista vazia
05         return 0;
06     int k, i = 0;
07     while(i<li->qtd && li->dados[i].matricula != mat)
08         i++;
09     if(i == li->qtd)//elemento não encontrado
10         return 0;
11
12     for(k=i; k< li->qtd-1; k++)
13         li->dados[k] = li->dados[k+1];
14     li->qtd--;
15     return 1;
16 }

```

FIGURA 5.23



Procura elemento a ser removido:

```
while(i<li->qtd && li->dados[i].matricula != mat)
    i++;
```

Desloca os elementos uma posição para trás:

```
for(k=i; k< li->qtd-1; k++)
    li->dados[k] = li->dados[k+1];
li->qtd--;
```

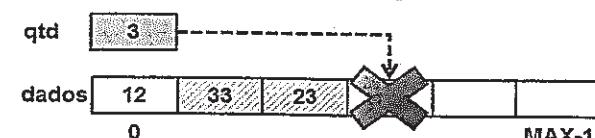


FIGURA 5.24

#### Otimizando a remoção da lista

Remover um elemento que esteja no início ou no meio de uma **lista sequencial** é uma tarefa trabalhosa, pois envolve o deslocamento de todos os elementos à frente. Porém, se a ordem dos elementos dentro da lista não for uma característica importante, então é possível otimizar essa tarefa.



A otimização da remoção é possível da seguinte forma: em vez de deslocar os elementos que estão à frente do elemento removido, apenas copie o último elemento para o lugar do elemento removido. Desse modo, apenas um único elemento será movimentado. Porém, a ordem dos elementos na lista será alterada.

Basicamente, o que temos que fazer é procurar esse elemento na lista e movimentar o último elemento para a posição do elemento removido. Isso sobrescreve o elemento a ser removido ao mesmo tempo que diminui o número de elementos, como mostra a sua implementação na Figura 5.25. Note que as linhas 2 a 5 verificam se a remoção é possível.

No nosso exemplo, vamos remover um elemento de acordo com o campo matrícula. Assim temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento atual for diferente do valor de matrícula procurado (linhas 7 e 8). Terminado o processo de busca, verificamos se estamos no final da lista ou não (linha 9). Em caso afirmativo, o elemento não existe na lista e a remoção não é possível (linha 10). Caso contrário, diminuímos em uma unidade a quantidade ( $li->qtd$ ) de elementos armazenados na lista e copiamos o último elemento para a posição do elemento a ser removido (linhas 12-13). Por fim, retornamos o valor UM (linhas 14), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.26.

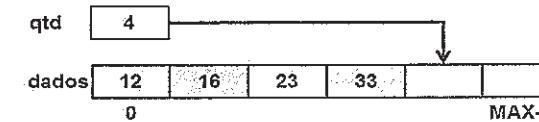
**Remover de uma lista elementos específicos**

```
01 int remove_lista_optimizado(Lista* li, int mat){
02     if(li == NULL)
03         return 0;
04     if(li->qtd == 0)
05         return 0;
06     int i = 0;
07     while(i<li->qtd && li->dados[i].matricula != mat)
08         i++;
09     if(i == li->qtd)//elemento não encontrado
10         return 0;
11
12     li->qtd--;
13     li->dados[i] = li->dados[li->qtd];
14
15 }
```

FIGURA 5.24



Não se esqueça: a otimização da operação de remoção somente deve ser utilizada quando alterar a ordem dos elementos da lista não comprometer o desempenho da aplicação. Se a ordem dos elementos é um atributo importante da lista, então essa otimização não deve ser utilizada.



```
Procura elemento a ser removido:  
while(i<li->qtd && li->dados[i].matricula != mat)  
    i++;
```

```
Copia o último elemento da lista para  
o lugar do elemento removido:  
li->qtd--;  
li->dados[i] = li->dados[li->qtd];
```

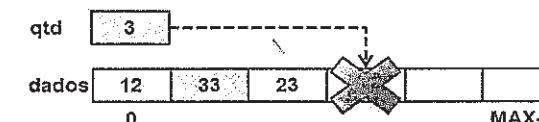


FIGURA 5.26

#### 5.4.6 Busca por um elemento da lista

A operação de busca consiste em recuperar as informações contidas em determinado elemento da lista.



De modo geral, a operação de busca pode ser feita de diversas maneiras, dependendo da necessidade da aplicação.

Por exemplo, podemos buscar as informações:

- Do segundo elemento da lista: busca por posição.
  - Do elemento que possui um pedaço de informação conhecida: busca por conteúdo.

A seguir, veremos como esses dois tipos de busca funcionam.

Busca por posição na lista

Buscar um elemento por sua posição em uma **lista sequencial estática** é uma tarefa quase imediata, como mostra a sua implementação na Figura 5.27. Em primeiro lugar, a função verifica se a busca é válida. Para tanto, três condições são verificadas: se o ponteiro **Lista\*** *l1* é igual a

**NULL**, se a posição buscada (**pos**) é um valor negativo e se essa posição é maior que o tamanho da lista. Se alguma dessas condições for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Caso contrário, a posição selecionada é copiada para o conteúdo do ponteiro passado por referência (**al**) para a função (linha 4). Como se nota, esse tipo de busca consiste em um simples acesso à determinada posição do array (Figura 5.28).

```

Busca um elemento por posição
01 int busca_lista_pos(Lista* li, int pos, struct aluno *al) {
02     if(li == NULL || pos <= 0 || pos > li->qtd)
03         return 0;
04     *al = li->dados[pos-1];
05     return 1;
06 }
```

FIGURA 5.27



FIGURA 5.28

#### Busca por conteúdo na lista

Buscar um elemento por seu conteúdo em uma **lista sequencial estática** é uma tarefa um pouco mais complicada.



Na busca por conteúdo, precisamos percorrer a lista à procura do elemento desejado.

A Figura 5.29 mostra a implementação da busca por conteúdo. Neste caso, estamos procurando um aluno pelo seu número de matrícula. Em primeiro lugar, a função verifica se a lista é válida, ou seja, se o ponteiro **Lista\* li** é igual a **NULL**. Se essa condição for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Caso contrário, temos que percorrer a lista. Essa tarefa é realizada enquanto não tivermos chegado ao final da lista e enquanto o número de matrícula não tiver sido encontrado (linhas 4-6).



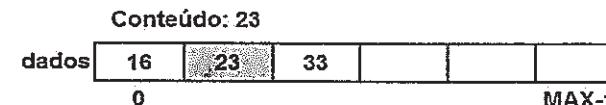
Perceba que a primeira condição do comando **while** será sempre falsa se a lista estiver vazia (campo **qtd** igual a **ZERO**). Assim, não é preciso tratar isoladamente esse caso.

Terminado o laço, verificamos se estamos no final da lista (linha 7). Se essa condição for verdadeira, o elemento não foi encontrado na lista. Do contrário, a posição atual (**i**) é copiada para o conteúdo do ponteiro passado por referência (**al**) para a função (linha 10). A Figura 5.30 ilustra os principais pontos dessa busca.

```

Busca um elemento por conteúdo
01 int busca_lista_mat(Lista* li, int mat, struct aluno *al) {
02     if(li == NULL)
03         return 0;
04     int i = 0;
05     while(i<li->qtd && li->dados[i].matricula != mat)
06         i++;
07     if(i == li->qtd) //elemento não encontrado
08         return 0;
09     *al = li->dados[i];
10     return 1;
11 }
```

FIGURA 5.29



Busca pelo elemento:

```
while(i<li->qtd && li->dados[i].matricula != mat)
    i++;
```

Achou o elemento:

```
*al = li->dados[i];
```

FIGURA 5.30

#### 5.4.7 Análise de complexidade

Um aspecto importante quando manipulamos listas tem relação com os custos das suas operações. Na sequência, são mostradas as complexidades computacionais das principais operações em uma **lista sequencial estática** contendo  $N$  elementos:

- **Inserção no início:** essa operação envolve o deslocamento de todos os elementos da lista, de modo que a sua complexidade é  $O(N)$ .
- **Inserção no final:** essa operação envolve apenas a manipulação de alguns índices. Desse modo, a sua complexidade é  $O(1)$ .
- **Inserção ordenada:** neste caso, é preciso procurar o ponto de inserção, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (inserção no início).

- **Remoção do início:** como na inserção, essa operação envolve o deslocamento de todos os elementos da lista, de modo que a sua complexidade é  $O(N)$ .
- **Remoção do final:** essa operação envolve apenas a manipulação de alguns índices. Desse modo, a sua complexidade é  $O(1)$ .
- **Remoção de um elemento específico:** essa operação envolve a busca pelo elemento a ser removido, que pode estar no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (remoção do início).
- **Consulta:** a operação de consulta envolve a busca de um elemento, que pode estar no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (último elemento).

## 5.5 LISTA DINÂMICA ENCADEADA

Uma **lista dinâmica encadeada** é uma lista definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da lista é alocado dinamicamente, à medida que os dados são inseridos dentro da lista, e tem sua memória liberada, à medida que é removido. Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação: um campo de **dado**, utilizado para armazenar a informação inserida na lista, e um campo **prox**, que nada mais é do que um ponteiro que indica o próximo elemento na lista.



Além da estrutura que define seus elementos, essa lista utiliza um **ponteiro para ponteiro** para guardar o primeiro elemento da lista.

Temos em uma **lista dinâmica encadeada** que todos os seus elementos são ponteiros alocados dinamicamente. Para inserir um elemento no início da lista, precisamos utilizar um campo que seja fixo, mas que, ao mesmo tempo, seja capaz de apontar para o novo elemento.

É necessário o uso de um **ponteiro para ponteiro** para guardar o endereço de um **ponteiro**. Utilizando um **ponteiro para ponteiro** para representar o início da lista, fica fácil mudar quem é o primeiro elemento da lista alterando apenas o **conteúdo do ponteiro para ponteiro**. Mais detalhes são apresentados na Seção 5.5.1.



Após o último elemento, não existe nenhum novo elemento alocado. Sendo assim, o último elemento da lista aponta para **NULL**.

Considerando uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo lista (ela é um **tipo opaco**), como ilustrado na Figura 5.31. Isso impede o usuário de saber como foi realmente implementada a lista e limita o seu acesso apenas às funções que manipulam a lista.

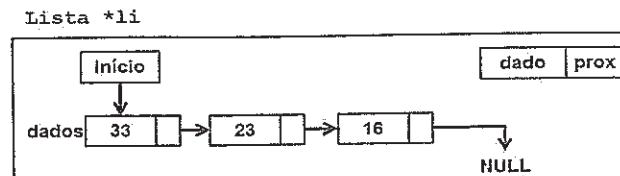


FIGURA 5.31



Note que a implementação em módulos impede o usuário de saber como a lista foi implementada. Tanto a **lista dinâmica encadeada** quanto a **lista sequencial estática** são declaradas como sendo do tipo **Lista \***.

Essa é a grande vantagem da modularização e da utilização de tipos opacos: mudar a maneira pela qual a lista foi implementada não altera nem interfere no funcionamento do programa que a utiliza.



O uso de alocação dinâmica e acesso encadeado na definição de uma **lista dinâmica encadeada** traz vantagens e desvantagens, que devem ser consideradas para um melhor desempenho da aplicação.

Várias são as vantagens de se definir uma lista utilizando uma abordagem dinâmica e encadeada:

- Melhor utilização dos recursos de memória.
- Não é preciso definir previamente o tamanho da lista.
- Não se precisa movimentar os elementos nas operações de inserção e remoção.

Infelizmente, esse tipo de implementação também tem suas desvantagens:

- Acesso indireto aos elementos.
- Necessidade de percorrer a lista para acessar determinado elemento.



Considerando suas vantagens e desvantagens, quando devo utilizar uma **lista dinâmica encadeada**?

Em geral, usamos esse tipo de lista nas seguintes situações:

- Não há necessidade de garantir um espaço mínimo para a execução da aplicação.
- Inserção e remoção em lista ordenada são as operações mais frequentes.
- Tamanho máximo da lista não é definido.

### 5.5.1 Trabalhando com ponteiro para ponteiro

Quando definimos uma **lista sequencial estática** precisamos declarar apenas um ponteiro para manipular a lista. Isso porque a lista em si era uma única estrutura contendo dois campos, um campo para o tamanho da lista e um array para armazenar os dados da lista.



Qualquer alteração (seja inserção ou remoção) em uma **lista sequencial estática** altera apenas o conteúdo da estrutura que define a lista, nunca o endereço onde ela se encontra na memória.

Já em uma **lista dinâmica encadeada** todos os seus elementos são ponteiros alocados dinamicamente e de forma independente. É dentro desse ponteiro que fica armazenada a informação daquele elemento da lista.



Em uma **lista dinâmica encadeada** não temos mais uma estrutura que define a lista, apenas a estrutura que define os seus elementos. Assim, quando inserirmos ou removemos um elemento do início da lista, estamos mudando o endereço onde essa lista se inicia.

Essa pequena diferença de implementação faz com que a passagem de uma **lista dinâmica encadeada** para uma função tenha que ser feita utilizando um **ponteiro para ponteiro** em vez de um simples **ponteiro**. Para entender melhor, veja a Figura 5.32. Nela, temos um pequeno trecho de código em que tentamos modificar o endereço associado a um ponteiro **x** (linha 9) dentro de uma função (linha 4). Perceba que mesmo que o valor seja modificado dentro da função (linha 5), o valor ao sair da função continua sendo o originalmente atribuído ao ponteiro (linhas 12 e 14).



Isso ocorre porque, quando passamos um ponteiro para uma função (passagem por referência), somente podemos alterar o conteúdo apontado por aquele ponteiro, nunca o endereço guardado dentro dele.

Para alterar o endereço do ponteiro é necessário o uso de um ponteiro para ponteiro para guardar o endereço do ponteiro que será modificado, como mostra a Figura 5.33. Nela, temos um pequeno trecho de código em que tentamos modificar o endereço associado a um ponteiro para ponteiro **x** (linha 10) dentro de uma função (linha 4). Perceba agora que o que está sendo modificado dentro da função não é mais o endereço de **x**, mas sim o conteúdo de **x** (linha 4). Como **x** é um ponteiro para ponteiro, o seu conteúdo nada mais é do que outro ponteiro: **w** (linha 9). Usando essa estratégia, o endereço guardado dentro de **x** nunca é modificado. O que é modificado é apenas o endereço guardado dentro de **w** (linha 4). Assim, ainda que o valor seja modificado dentro da função (linha 5), o valor ao sair da função será o mesmo recebido dentro da função e diferente do valor originalmente atribuído (linhas 13 e 15).



Desse modo, criamos um indicador que nunca muda sua posição na memória e que aponta para o **índice da lista dinâmica encadeada**.

### Trabalhando com ponteiro

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 void troca_ende(int *a, int *b) {
04     a = b;
05     printf("x (Dentro): %#p\n", a);
06 }
07 int main() {
08     int *x, y = 5, z = 6;
09     x = &y;
10     printf("Endereco y: %#p\n", &y);
11     printf("Endereco z: %#p\n", &z);
12     printf("x (Antes): %#p\n", x);
13     troca_ende(x, &z);
14     printf("x (Depois): %#p\n", x);
15     system("pause");
16 }
17 }
```

### Saída

```

01 Endereco y: 0x03E
02 Endereco z: 0x042
03 x (Antes) : 0x03E
04 x (Dentro): 0x042
05 x (Depois): 0x03E
```

### Memória

Memória		
Endereço	Variável	Conteúdo
00034		
00038		
00042		
00046	int *x;	00054
00050		
00054	int y;	5
00058		
00062	int z;	6
00066		

Antes da função

Memória		
Endereço	Variável	Conteúdo
00034		
00038		
00042		
00046	int *n;	00054
00050		
00054	int y;	5
00058		
00062	int z;	6
00066		

Depois da função

FIGURA 5.32

### Trabalhando com ponteiro para ponteiro

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 void troca_enderec(int **a, int *b) {
04     *a = b;
05     printf("x (Dentro): %#p\n", *a);
06 }
07 int main() {
08     int **x, *w, y = 5, z = 6;
09     x = &w;
10     *x = &y;
11     printf("Endereco y: %#p\n", &y);
12     printf("Endereco z: %#p\n", &z);
13     printf("x (Antes): %#p\n", *x);
14     troca_enderec(x, &z);
15     printf("x (Depois): %#p\n", *x);
16     system("pause");
17     return 0;
18 }
```

### Saída

```

01 Endereco y: 0x03E
02 Endereco z: 0x042
03 x (Antes) : 0x03E
04 x (Dentro) : 0x042
05 x (Depois): 0x042
```

### Memória

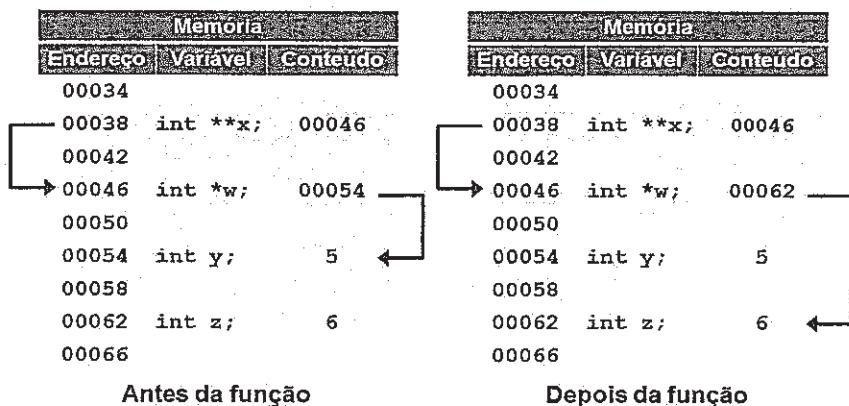


FIGURA 5.33

### 5.5.2 Definindo o tipo lista dinâmica encadeada

Antes de começar a implementar a nossa lista, é preciso definir o tipo de dado que será armazenado nela. Uma lista pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o tipo opaco que representa nossa lista. E, trabalhando com alocação dinâmica da lista, este tipo será um **ponteiro para ponteiro** da estrutura que define a lista. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **ListaDinEncad.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **ListaDinEncad.h**, ilustrado na Figura 5.34. Por se tratar de uma lista dinâmica encadeada, temos que estabelecer:

- O tipo de dado que será armazenado na lista, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para o **ponteiro** do tipo lista (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma lista.
- As funções disponíveis para trabalhar com essa lista em especial (linhas 8-20) e que serão implementadas no arquivo **ListaDinEncad.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da lista. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



Por que colocamos um ponteiro no comando **typedef** quando criamos um novo nome para o tipo (linha 6)?

Por estarmos trabalhando com uma lista dinâmica e encadeada, temos que fazê-lo com um ponteiro para ponteiro à fim de poder realizar modificações no início da lista. Por questões de modularização, e para manter a mesma notação utilizada pela **lista sequencial estática**, podemos esconder um dos ponteiros do usuário. Assim, utilizar uma **lista sequencial estática** ou uma **lista dinâmica encadeada** será indiferente para o programador, pois a sua implementação está escondida dele:

- Lista \*li; //Declaração de uma **lista sequencial estática** (ponteiro).
- Lista \*li; //Declaração de uma **lista dinâmica encadeada** (ponteiro para ponteiro).



No arquivo **ListaDinEncad.c**, iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em **ListaDinEncad.h**.

Basicamente, o arquivo **ListaDinEncad.c** (Figura 5.34) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da lista (linhas 1-3).
- A definição do tipo que descreve cada elemento da lista, **struct elemento** (linhas 5-8).
- A definição de um novo nome para a **struct elemento** (linhas 9). Isso é feito apenas para facilitar certas etapas de codificação.
- As implementações das funções definidas no arquivo **ListaDinEncad.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que a nossa estrutura **elemento** nada mais é do que uma estrutura contendo dois campos: um ponteiro **prox**, que indica o próximo elemento (também do tipo **struct elemento**) dentro da lista, e um campo **dado** do tipo **struct aluno**, que é o tipo de dado a ser armazenado na lista. Por estarem definidos dentro do arquivo **.c**, os campos dessa estrutura

**Arquivo ListaDinEncad.h**

```

01 struct aluno{
02     int matricula;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 typedef struct elemento* Lista;
07
08 Lista* cria_lista();
09 void libera_lista(Lista* li);
10 int insere_lista_final(Lista* li, struct aluno al);
11 int insere_lista_inicio(Lista* li, struct aluno al);
12 int insere_lista_ordenada(Lista* li, struct aluno al);
13 int remove_lista(Lista* li, int mat);
14 int remove_lista_inicio(Lista* li);
15 int remove_lista_final(Lista* li);
16 int tamanho_lista(Lista* li);
17 int lista_vazia(Lista* li);
18 int lista_cheia(Lista* li);
19 int busca_lista_mat(Lista* li, int mat, struct aluno *al);
20 int busca_lista_pos(Lista* li, int pos, struct aluno *al);

```

**Arquivo ListaDinEncad.c**

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ListaDinEncad.h" //inclui os protótipos
04 //Definição do tipo lista
05 struct elemento{
06     struct aluno dados;
07     struct elemento *prox;
08 };
09 typedef struct elemento Elemt;

```

FIGURA 5.34

não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **ListaDinEncad.h** (linha 6), que pode somente declarar um ponteiro para ele, da seguinte forma:

List\* li;

### 5.5.3 Criando e destruindo uma lista

Para utilizar uma lista em seu programa, a primeira coisa a fazer é criar uma lista vazia. Essa tarefa é executada pela função descrita na Figura 5.35. Basicamente, o que esta função faz é a alocação de uma área de memória para armazenar o endereço do início da lista (linha 2), que é um **ponteiro para ponteiro**. Esta área de memória corresponde à memória necessária para armazenar o endereço de um elemento da lista, **sizeof(Lista)** ou **sizeof(struct elemento\*)**. Em seguida, a função inicializa o conteúdo desse **ponteiro para ponteiro** com a constante **NULL**. Esta constante é utilizada em uma **lista dinâmica encadeada** para indicar que não existe nenhum elemento alocado após o atual. Como o início da lista aponta para tal constante, isso significa que a lista está vazia. A Figura 5.36 indica o conteúdo do nosso ponteiro **Lista\* li** após a chamada da função que cria a lista.

**Criando uma lista**

```

01 Lista* cria_lista(){
02     Lista* li = (Lista*) malloc(sizeof(Lista));
03     if(li != NULL)
04         *li = NULL;
05     return li;
06 }

```

FIGURA 5.35

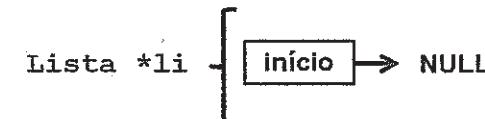


FIGURA 5.36

Destruir uma lista que utilize alocação dinâmica, e seja encadeada, não é uma tarefa tão simples quanto destruir uma **lista sequencial estática**.



Para liberar uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido nela.

O código que realiza a destruição da lista é mostrado na Figura 5.37. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso (linha 2). Em seguida, percorremos a lista até que o conteúdo do seu início (\*li) seja diferente de NULL, o final da lista. Enquanto não chegarmos ao final da lista, iremos liberar a memória do elemento que se encontra atualmente no início da lista e avançar para o próximo (linhas 5-7). Terminado o processo, liberaremos a memória alocada para o início da lista (linha 9). Esse processo é mais bem ilustrado pela Figura 5.38, que mostra a liberação de uma lista contendo dois elementos.

```
Destruindo uma lista
01 void libera_lista(Lista* li){
02     if(li != NULL){
03         Elem* no;
04         while((*li) != NULL){
05             no = *li;
06             *li = (*li)->prox;
07             free(no);
08         }
09         free(li);
10     }
11 }
```

FIGURA 5.37

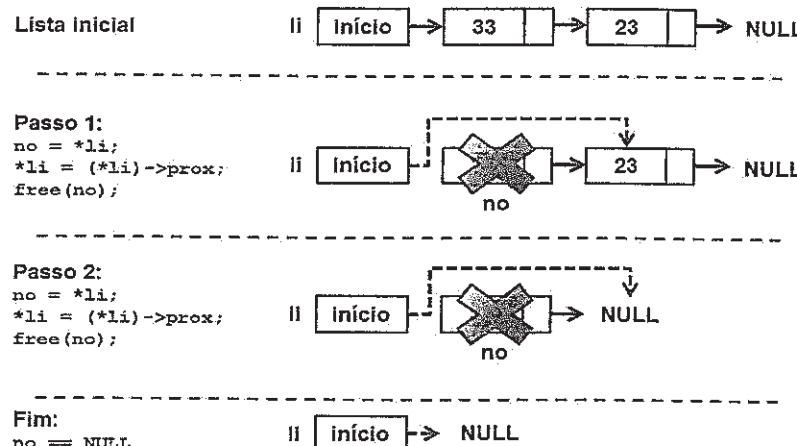


FIGURA 5.38

#### 5.5.4 Informações básicas sobre a lista

As operações de inserção, remoção e busca são consideradas as principais de uma lista. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a lista. Por exemplo, não podemos remover um elemento da lista se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da lista: seu tamanho, se ela está cheia ou se ela está vazia.

##### Tamanho da lista

Saber o tamanho de uma **lista dinâmica encadeada** não é uma tarefa tão simples como na **lista sequencial estática**. Isso ocorre porque agora não possuímos mais um campo armazenando a quantidade de elementos inseridos dentro da lista.



Para saber o tamanho de uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista contando os elementos inseridos nela, até encontrar o seu final.

O código que realiza a contagem dos elementos da lista é mostrado na Figura 5.39. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso e a lista é ou não igual a NULL (linha 2). Caso ela seja nula, não temos o que fazer na função e a terminamos (linha 3). Em seguida, criamos um contador iniciado em ZERO (linha 4) e um elemento auxiliar (no) apontado para o primeiro elemento da lista (linha 5). Então, percorremos a lista até que o valor de no seja diferente de NULL, o final da lista. Enquanto não chegarmos ao final da lista, iremos somar “+1” ao contador cont e avançar para o próximo elemento da lista (linhas 6-9). Terminado o processo, retornamos o valor da variável cont (linha 10). Esse processo é melhor ilustrado pela Figura 5.40, que mostra o cálculo do tamanho de uma lista contendo dois elementos.

#### Tamanho da lista

```
Tamanho da lista
01 int tamanho_lista(Lista* li){
02     if(li == NULL)
03         return 0;
04     int cont = 0;
05     Elem* no = *li;
06     while(no != NULL){
07         cont++;
08         no = no->prox;
09     }
10     return cont;
11 }
```

FIGURA 5.39

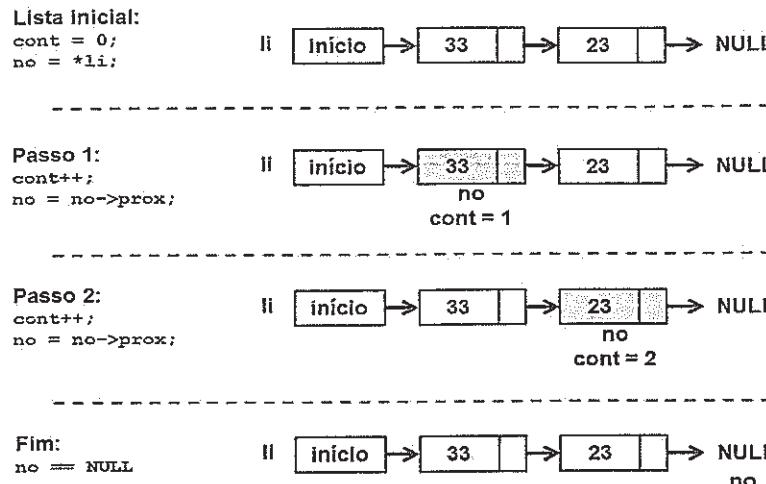


FIGURA 5.40

### Listas cheias

Implementar uma função que retorne se uma **lista dinâmica encadeada** está cheia é uma tarefa relativamente simples.



Uma lista dinâmica encadeada somente será considerada cheia quando não tivermos mais memória disponível para alocar novos elementos.

A implementação da função que retorna se a lista está cheia é mostrada na Figura 5.41. Como se pode notar, a função sempre irá retornar o valor **ZERO**, indicando que a lista não está cheia.

#### -Exemplo-

```
01 int lista_cheia(Lista* li){  
02     return 0;  
03 }
```

FIGURA 5.41

### Listas vazias

Implementar uma função que retorne se uma **lista dinâmica encadeada** está vazia é outra tarefa relativamente simples.



Uma lista dinâmica encadeada será considerada vazia sempre que o conteúdo do seu "íncio" apontar para a constante **NULL**.

A implementação da função, que retorna se a lista está vazia é mostrada na Figura 5.42. Note que essa função, em primeiro lugar, verifica se o ponteiro **Lista\*** **li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **UM** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, então é possível acessar o conteúdo do seu "íncio" (\*li) e comparar o seu valor com a constante **NULL**, que é o valor inicial do conteúdo do "íncio" quando criamos a lista. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da lista), a função irá retornar o valor **UM** (linha 5). Caso contrário, irá retornar o valor **ZERO** (linha 6).

#### Retornando se a lista está vazia

```
01 int lista_vazia(Lista* li){  
02     if(li == NULL)  
03         return 1;  
04     if(*li == NULL)  
05         return 1;  
06     return 0;  
07 }
```

FIGURA 5.42

### 5.5.5 Inserindo um elemento na lista

#### Preparando a inserção na lista

Antes de inserir um elemento em uma **lista dinâmica encadeada**, algumas verificações são necessárias. Isso vale para os três tipos de inserção: no início, no final ou no meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.44, 5.46 e 5.48).

Primeiramente, a função verifica se o ponteiro **Lista\*** **li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linhas 4 e 5). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 6 e 7). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 8).



No caso de uma lista com alocação dinâmica, ela somente será considerada cheia quando não tivermos mais memória disponível no computador para alocar novos elementos. Isso ocorrerá apenas quando a chamada da função **malloc()** retornar **NULL**.

Também existe a situação em que a inserção é feita em uma lista que está vazia, como mostrado na Figura 5.43. Neste caso, a lista, que inicialmente apontava para **NULL**, passa a apontar para o único elemento inserido até então.

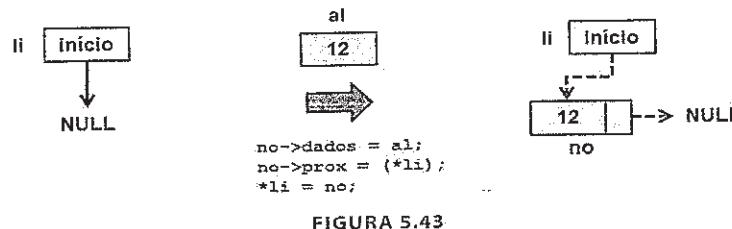


FIGURA 5.43

### Inserindo no início da lista

Inserir um elemento no **início** de uma **lista dinâmica encadeada** é uma tarefa bastante simples.



Diferente da **lista sequencial estática**, a inserção no **início** de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.44. Note que as linhas 2 a 8 verificam se a inserção é possível.

Como se trata de uma inserção no **início**, temos que fazer nosso elemento apontar para o **início** da lista, **\*li** (linha 9). Assim, o elemento **no** passa a ser o **início** da lista, enquanto o antigo **início** passa a ser o **próximo** elemento da lista. Por fim, mudamos o conteúdo do “**início**” da lista (**\*li**) para que ele passe a ser o nosso elemento **no** e retornamos o valor **UM** (linhas 10 e 11), indicando sucesso na operação de inserção. Esse processo é melhor ilustrado pela Figura 5.45.

#### Inserindo um elemento no **início** da lista

```

01 int insere_lista_inicio(Lista* li, struct aluno al) {
02     if(li == NULL)
03         return 0;
04     Elemt* no;
05     no = (Elemt*) malloc(sizeof(Elemt));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = (*li);
10     *li = no;
11     return 1;
12 }
```

FIGURA 5.44

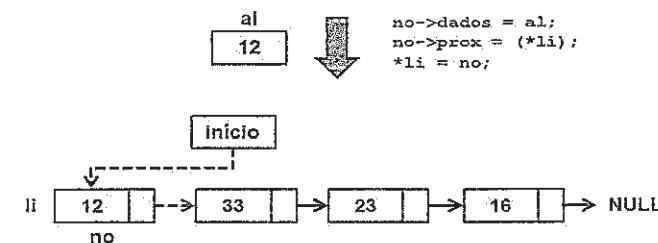
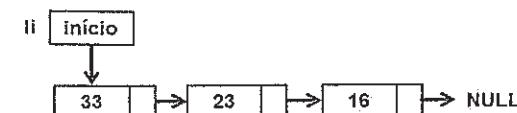


FIGURA 5.45

### Inserindo no final da lista

Inserir um elemento no **final** de uma **lista dinâmica encadeada** é uma tarefa um tanto trabalhosa.



Como na inserção no **início**, a inserção no **final** de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista. Porém, é preciso percorrer a lista toda para descobrir o último elemento e assim fazer a inserção após ele.

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista, encontrar o último da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.46. Note que as linhas 2 a 8 verificam se a inserção é possível.

Como se trata de uma inserção no **final**, o elemento a ser inserido obrigatoriamente irá apontar para a constante **NULL** (linha 9). Também temos que considerar que a lista pode ou não estar vazia (linha 10):

- No caso de ser uma lista vazia, mudamos o conteúdo do “**início**” da lista (**\*li**) para que ele passe a ser o nosso elemento **no** (linha 11). Esse processo é melhor ilustrado pela Figura 5.43.
- No caso de NÃO ser uma lista vazia, temos que achar o último elemento, pois ele aponta sempre para a constante **NULL**. Assim, devemos guardar em um ponteiro auxiliar (**aux**) o endereço do primeiro elemento da lista (**\*li**) e percorrer a lista até que o elemento seguinte ao elemento atual (**aux->prox**) seja a constante **NULL** (linhas 13-17). Ao final do processo, o elemento **no** passa a ser o elemento seguinte a **aux** (linha 18). Esse processo é melhor ilustrado pela Figura 5.47.

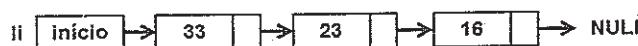
Terminado um dos dois processos de inserção, retornamos o valor **UM** (linha 20), indicando sucesso na operação de inserção.

**Inserindo um elemento no final da lista**

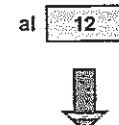
```

01 int insere_lista_final(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     ELEM *no;
05     no = (ELEM*) malloc(sizeof(ELEM));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = NULL;
10    if((*li) == NULL){//lista vazia: insere inicio
11        *li = no;
12    }else{
13        ELEM *aux;
14        aux = *li;
15        while(aux->prox != NULL){
16            aux = aux->prox;
17        }
18        aux->prox = no;
19    }
20    return 1;
21 }
```

FIGURA 5.46



**Busca onde inserir:**  
aux = \*li;  
while(aux->prox != NULL){  
 aux = aux->prox;  
}



**Insere depois de "aux":**  
no->dados = al;  
no->prox = NULL;  
aux->prox = no;

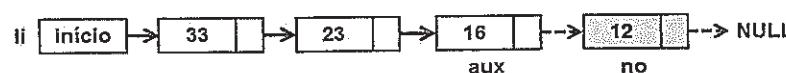


FIGURA 5.47

**Inserindo de forma ordenada na lista**

Inserir um elemento de forma ordenada em uma **lista dinâmica encadeada** é uma tarefa trabalhosa.



Isso ocorre porque precisamos procurar o ponto de inserção do elemento na lista, o qual pode ser no início, no meio ou no final da lista. Porém, diferente da **lista sequencial estática**, a inserção ordenada em uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é procurar em que lugar da lista será inserido o novo elemento (no caso, iremos ordenar pelo campo matrícula), alocar espaço para ele na lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.48. Note que as linhas 2 a 8 verificam se a inserção é possível.

Antes de fazer a inserção, temos que considerar que a lista pode ou não estar vazia (linha 9):

- No caso de ser uma lista vazia, mudamos o conteúdo do “início” da lista (\*li) para que ele passe a ser o nosso elemento no, que irá apontar para a constante NULL, sendo o valor UM retornado para indicar sucesso na operação de inserção (linhas 10-12). Esse processo é mais bem ilustrado pela Figura 5.43.
- No caso de NÃO ser uma lista vazia, temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento atual for menor do que a matrícula do novo elemento a ser inserido (linhas 15-19). Note que, além do elemento atual, também armazenamos o elemento anterior a ele (ant), que é necessário em uma inserção

**Inserindo um elemento de forma ordenada na lista**

```

01 int insere_lista_ordenada(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     ELEM *no;
05     no = (ELEM*) malloc(sizeof(ELEM));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     if((*li) == NULL){//lista vazia: insere inicio
10        no->prox = NULL;
11        *li = no;
12        return 1;
13    }
14    else{
15        ELEM *ant, *atual = *li;
16        while(atual != NULL &&
17              atual->dados.matricula < al.matricula){
18            ant = atual;
19            atual = atual->prox;
20        }
21        if(atual == *li){//insere inicio
22            no->prox = (*li);
23            *li = no;
24        }else{
25            no->prox = atual;
26            ant->prox = no;
27        }
28    }
29 }
```

FIGURA 5.48

no meio da lista. Ao final do processo, temos duas possibilidades de acordo com o valor de `atual`: inserção no início da lista (linhas 21 e 22) ou inserção entre os elementos `anterior` e `atual` (linhas 24 e 25). Perceba que se `atual` for igual a `NULL`, a inserção é no final da lista. No fim do processo, retornamos o valor `UM` para indicar sucesso na operação de inserção (linha 27). Esse processo é mais bem ilustrado pela Figura 5.49.

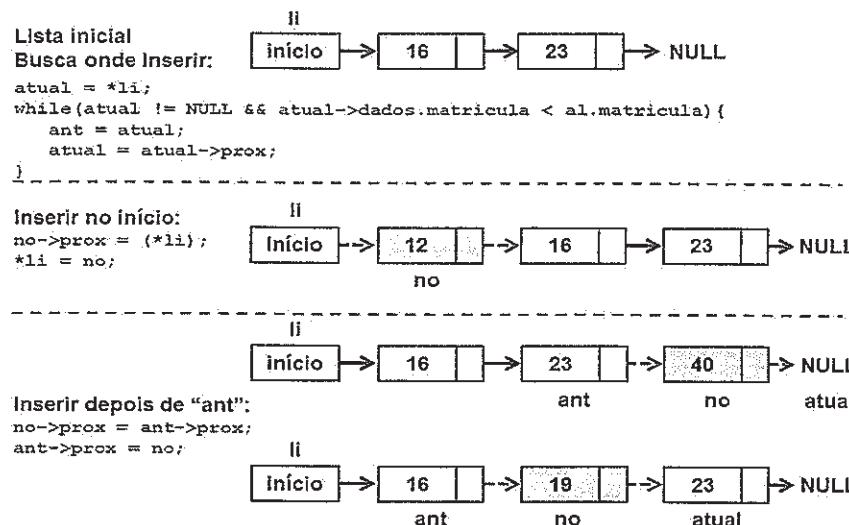


FIGURA 5.49

### 5.5.6 Removendo um elemento da lista

## Preparando a remoção da lista

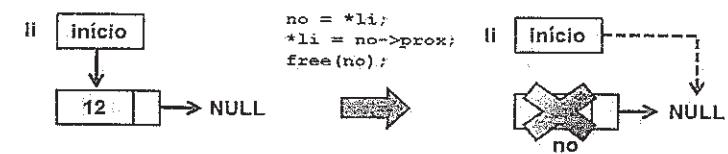
Antes de remover um elemento de uma **lista dinâmica encadeada**, algumas verificações são necessárias. Isso vale para os três tipos de remoção: do início, do final ou do meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.51, 5.53 e 5.55).

Primeiramente, verificamos se o ponteiro **Lista\* li** é igual a **NULL**. A condição seria verdadeira caso houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, precisamos verificar se ela não estiver vazia, isto é, se existem elementos dentro dela. Caso a lista esteja vazia, a função irá retornar o valor **ZERO** (linhas 4 e 5).



No caso de uma lista com alocação dinâmica, ela sómente será considerada vazia quando o seu início apontar para a constante **NULL**.

Também existe a situação em que a remoção é feita em uma lista que possui um único elemento, como mostrado na Figura 5.50. Neste caso, a lista fica vazia após a remoção.



**FIGURA 5.50**

### Removendo do início da lista

Remover um elemento do início de uma **lista dinâmica encadeada** é uma tarefa bastante simples.



Diferente da **lista sequencial estática**, a remoção do início de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é verificar se a lista não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.51. Note que as linhas 2 a 5 verificam se a remoção é possível.

Como se trata de uma remoção do **índice**, temos que fazer o **índice** da lista (**\*li**) apontar para o elemento seguinte a ele (linhas 7 e 8). Por fim, temos que liberar a memória associada ao antigo “**índice**” da lista (**no**) e retornarmos o valor **UM** (linhas 9 e 10), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.52.

## Removendo um elemento de inicio da lista

```

01 int remove_lista_inicio(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     Elemt *no = *li;
08     *li = no->prox;
09     free(no);
10
11 }
```

FIGURA 5.51

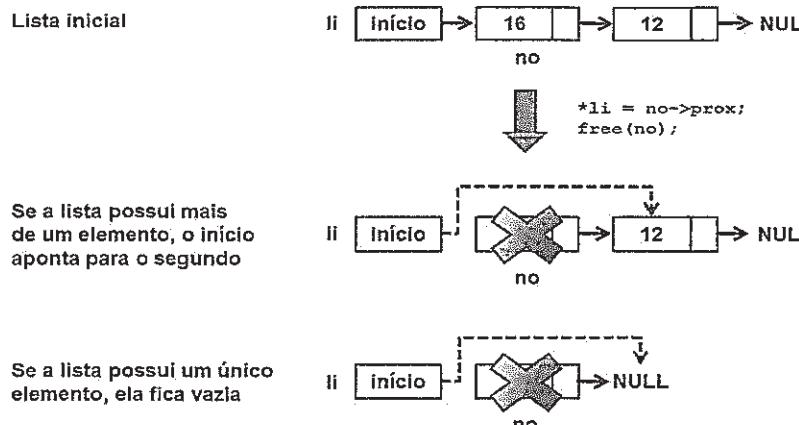


FIGURA 5.52

### Removendo do final da lista

Remover um elemento do final de uma **lista dinâmica encadeada** é uma tarefa um tanto trabalhosa.



Como na remoção do início, a remoção no final de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista. Porém, é preciso percorrer a lista toda para descobrir o último elemento e assim removê-lo.

Basicamente, o que temos que fazer é verificar se a lista não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.53. Note que as linhas 2 a 5 verificam se a remoção é possível.

Como se trata de uma remoção do final, temos que achar o último elemento da lista, ou seja, aquele que aponta para a constante **NULL**. Assim, devemos guardar em um ponteiro auxiliar (**no**) o endereço do primeiro elemento da lista (**\*li**) e percorrer a lista até que o elemento seguinte ao elemento atual (**no->prox**) seja a constante **NULL** (linhas 7-11). Note que usamos um segundo ponteiro auxiliar **ant** para guardar o elemento anterior ao último da lista. Ao final do processo, temos que considerar que o último elemento da lista talvez seja o primeiro e único (linha 13):

- Se **no** também é o início da lista, então o início da lista deverá apontar para a posição seguinte a ele, que, neste caso, é a constante **NULL**, ficando assim a lista vazia (linha 14).
- Caso contrário, o penúltimo elemento da lista (**ant**) irá apontar para o elemento seguinte ao último (**no**), que neste caso será a constante **NULL** (linha 16).

Terminado um dos dois processos de remoção, temos que liberar a memória associada ao antigo “final” da lista (**no**) e retornamos o valor **UM** (linhas 17 e 18), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.54.

### ... Removendo um elemento do final da lista

```

01 int remove_lista_final(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     ELEM *ant, *no = *li;
08     while(no->prox != NULL){
09         ant = no;
10         no = no->prox;
11     }
12
13     if(no == (*li))//remover o primeiro?
14         *li = no->prox;
15     else
16         ant->prox = no->prox;
17     free(no);
18     return 1;
19 }
```

FIGURA 5.53

**Lista inicial**  
**Busca o último elemento:**

```

no = *li;
while(no->prox != NULL) {
    ant = no;
    no = no->prox;
}
```



ant->prox = no->prox;  
free(no);



**Se a lista possui mais de um elemento, ant aponta para no**

**Se “no” é o único elemento da lista, a lista fica vazia.**

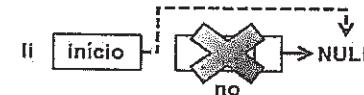


FIGURA 5.54

## Removendo um elemento específico da lista

Remover um elemento específico de uma lista dinâmica encadeada é uma tarefa trabalhosa.



Isso ocorre porque precisamos procurar o elemento a ser removido na lista, o qual pode estar no início, no meio ou no final da lista. Porém, diferente da lista sequencial estática, essa remoção não necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é procurar esse elemento na lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.55. Note que as linhas 2 a 5 verificam se a remoção é possível.

No nosso exemplo, vamos remover um elemento de acordo com o campo matrícula. Assim, temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento **no** for diferente do valor de matrícula procurado (linhas 6-10). Note que, além do elemento **no**, também armazenamos o elemento anterior à ele (**ant**), que é necessário em uma remoção no meio da lista. Terminado o processo de busca, verificamos se estamos no final da lista ou não (linha 11). Em caso afirmativo, o elemento não existe na lista e a remoção não é possível (linha 12). Caso contrário, a remoção pode ser no início da lista e, portanto, temos que mudar o valor de **li** (linhas 14-15), ou a remoção é no meio ou no final da lista (linha 17). Neste caso, basta apenas fazer o elemento **ant** apontar para o elemento seguinte a **no**.

Ao final do processo de remoção, temos que liberar a memória associada ao elemento **no** e retornamos o valor UM (linhas 18 e 19), indicando sucesso na operação de remoção. Esse processo é melhor ilustrado pela Figura 5.56.

### Removendo um elemento específico da lista

```

01 int remove_lista(Lista* li, int mat) {
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06     Elemt *ant, *no = *li;
07     while(no != NULL && no->dados.matricula != mat) {
08         ant = no;
09         no = no->prox;
10     }
11     if(no == NULL)//não encontrado
12         return 0;
13
14     if(no == *li)//remover o primeiro?
15         *li = no->prox;
16     else
17         ant->prox = no->prox;
18     free(no);
19     return 1;
20 }
```

FIGURA 5.55

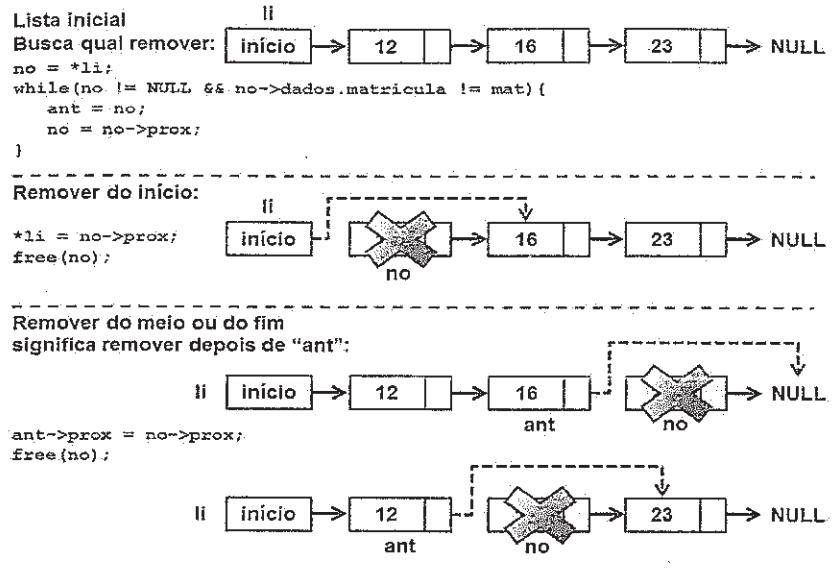


FIGURA 5.56

## 5.5.7 Busca por um elemento da lista

A operação de busca consiste em recuperar as informações contidas em determinado elemento da lista.



De modo geral, a operação de busca pode ser feita de diversas maneiras, dependendo da necessidade da aplicação.

Por exemplo, podemos buscar as informações:

- Do segundo elemento da lista: busca por posição.
- Do elemento que possui um pedaço de informação conhecida: busca por conteúdo.



Em uma lista que utilize alocação dinâmica, e seja encadeada, a busca sempre envolve a necessidade de percorrer a lista.

A seguir, veremos como esses dois tipos de busca funcionam.

### Busca por posição na lista

O código que realiza a busca de um elemento por sua posição em uma lista dinâmica encadeada é mostrado na Figura 5.57. Em primeiro lugar, a função verifica se a busca é válida.

Para tanto, duas condições são verificadas: se o ponteiro `Lista* li` é igual a `NULL` e se a posição buscada (`pos`) é um valor negativo. Se alguma dessas condições for verdadeira, a busca termina e a função retorna o valor `ZERO` (linha 3). Caso contrário, criamos um elemento auxiliar (`no`) apontado para o primeiro elemento da lista (linha 4) e um contador (`i`) iniciado em `UM` (linha 5). Então, percorremos a lista enquanto o valor de `no` for diferente de `NULL` e o valor do contador for menor do que a posição desejada (linhas 6-9). Terminado o laço, verificamos se estamos no final da lista (linha 10). Se essa condição for verdadeira, o elemento não foi encontrado na lista. Do contrário, a posição atual (`no`) é copiada para o conteúdo do ponteiro passado por referência (`al`) para a função (linha 13). A Figura 5.58 ilustra os principais pontos dessa busca.

```
BUSCA UM ELEMENTO POR POSIÇÃO
01 int busca_lista_pos(Lista* li, int pos, struct aluno *al){
02     if(li == NULL || pos <= 0)
03         return 0;
04     ELEM *no = *li;
05     int i = 1;
06     while(no != NULL && i < pos){
07         no = no->prox;
08         i++;
09     }
10     if(no == NULL)
11         return 0;
12     else{
13         *al = no->dados;
14         return 1;
15     }
16 }
```

FIGURA 5.57

```
Busca pela posição do elemento
no = *li;
int i = 1;
while(no != NULL && i < pos){
    no = no->prox;
    i++;
}

Verifica se a posição foi
encontrada e a retorna
if(no == NULL) return 0;
else{
    *al = no->dados;
    return 1;
}
```

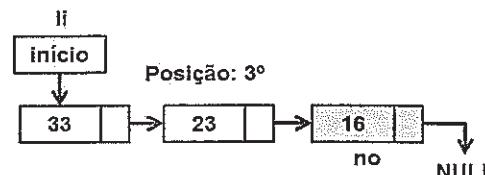


FIGURA 5.58

### Busca por conteúdo na lista

O código que realiza a busca de um elemento por sua posição em uma lista dinâmica encadeada é mostrado na Figura 5.59. Neste caso, estamos procurando um aluno pelo seu número de matrícula. Como se pode notar, o código é praticamente igual ao da busca por posição mostrado na Figura 5.57. A única diferença, é que agora não percorremos mais a lista comparando o valor da posição, mas sim o valor da matrícula (linha 5). Além disso, não precisamos mais de um contador ao percorrer a lista. A Figura 5.60 ilustra os principais pontos dessa busca.

```
Busca um elemento por conteúdo
01 int busca_lista_mat(Lista* li, int mat, struct aluno *al){
02     if(li == NULL)
03         return 0;
04     ELEM *no = *li;
05     while(no != NULL && no->dados.matricula != mat){
06         no = no->prox;
07     }
08     if(no == NULL)
09         return 0;
10     else{
11         *al = no->dados;
12         return 1;
13     }
14 }
```

FIGURA 5.59

### Busca pelo conteúdo do elemento

```
no = *li;
while(no != NULL && no->dados.matricula != mat)
    no = no->prox;
```

```
Verifica se o elemento foi
encontrado e o retorna
if(no == NULL) return 0;
else{
    *al = no->dados;
    return 1;
}
```

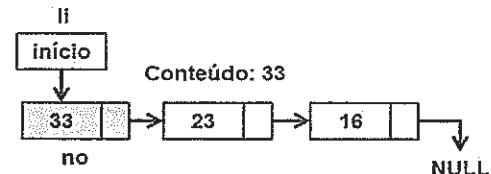


FIGURA 5.60

### 5.5.8 Análise de complexidade

Um aspecto importante quando manipulamos listas tem relação com os custos das suas operações. Na sequência, são mostradas as complexidades computacionais das principais operações em uma lista dinâmica encadeada contendo  $N$  elementos:

- **Inserção no início:** essa operação envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- **Inserção no final:** é preciso percorrer toda a lista até alcançar o seu final. Desse modo, a sua complexidade é  $O(N)$ .
- **Inserção ordenada:** neste caso, é preciso procurar o ponto de inserção, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (inserção no final).
- **Remoção do início:** é uma operação que envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- **Remoção do final:** é preciso percorrer toda a lista até alcançar o seu final. Desse modo, a sua complexidade é  $O(N)$ .
- **Remoção de um elemento específico:** essa operação envolve a busca pelo elemento a ser removido, que pode estar no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (remoção do final).
- **Consulta:** a operação de consulta envolve a busca de um elemento, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (último elemento).

## 5.6 LISTA DINÂMICA ENCADEADA CIRCULAR

Uma **lista dinâmica encadeada circular** é uma lista definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da lista é alocado dinamicamente, à medida que os dados são inseridos dentro da lista, e têm sua memória liberada, à medida que é removido. Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação: um campo de **dado**, utilizado para armazenar a informação inserida na lista, e um campo **prox**, que nada mais é do que um ponteiro que indica o próximo elemento na lista.



Além da estrutura que define seus elementos, essa lista utiliza um **ponteiro para ponteiro** para guardar o primeiro elemento da lista.

Temos em uma **lista dinâmica encadeada** que todos os seus elementos são ponteiros alocados dinamicamente. Para inserir um elemento no início da lista, precisamos utilizar um campo que seja fixo, mas que, ao mesmo tempo, seja capaz de apontar para o novo elemento.

É necessário o uso de um **ponteiro para ponteiro** para guardar o endereço de um **ponteiro**. Utilizando um **ponteiro para ponteiro** para representar o início da lista, fica fácil mudar quem é o primeiro elemento da lista mudando apenas o **conteúdo do ponteiro para ponteiro**. Mais detalhes são apresentados na Seção 5.5.1.



Até aqui, uma **lista dinâmica encadeada circular** se parece muito com a **lista dinâmica encadeada**. Qual a diferença entre elas?

Em uma **lista dinâmica encadeada**, após o último elemento não existe nenhum novo elemento alocado, de modo que o último elemento da lista aponta para **NULL**. Já em uma **lista dinâmica encadeada circular**, o último elemento tem como sucessor o primeiro elemento da lista, parecendo que esse tipo de lista não tem fim: nunca chegaremos a uma posição final a partir da qual não poderemos mais andar dentro da lista, porque depois do último elemento voltamos para o primeiro, como em um círculo.

Considerando uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo lista (ela é um **tipo opaco**), como ilustrado na Figura 5.61. Isso impede o usuário de saber como foi realmente implementada a lista, e limita o seu acesso apenas às funções que manipulam a lista.

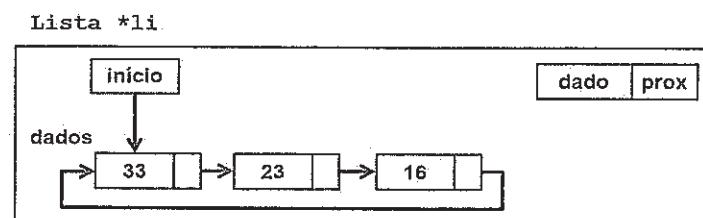


FIGURA 5.61



Note que a implementação em módulos impede o usuário de saber como a lista foi implementada. Tanto a **lista dinâmica encadeada circular** quanto as **listas dinâmica encadeada e sequencial estática** são declaradas como sendo do tipo **Lista \***.

Essa é a grande vantagem da modularização e da utilização de tipos opacos: mudar a maneira pela qual a lista foi implementada não altera nem interfere no funcionamento do programa que a utiliza.



O uso de alocação dinâmica e acesso encadeado na definição de uma **lista dinâmica encadeada circular** traz vantagens e desvantagens, que devem ser consideradas para um melhor desempenho da aplicação.

Várias são as vantagens de se definir uma lista utilizando uma abordagem dinâmica e encadeada:

- Melhor utilização dos recursos de memória.
- Não é preciso definir previamente o tamanho da lista.
- Possibilidade de percorrer a lista diversas vezes.
- Não se precisa movimentar os elementos nas operações de inserção e remoção.
- Não precisamos considerar casos especiais de inclusão e remoção de elementos (primeiro e último).

Infelizmente, esse tipo de implementação também tem suas desvantagens:

- Acesso indireto aos elementos.
- Necessidade de percorrer a lista para acessar determinado elemento.
- A lista não possui um final definido. Logo, não há como saber se já percorremos a lista inteira.



Considerando suas vantagens e desvantagens, quando devo utilizar uma **lista dinâmica encadeada circular**?

Em geral, usamos esse tipo de lista na seguinte situações:

- Não há necessidade de garantir um espaço mínimo para a execução da aplicação.
- Inserção e remoção em lista ordenada são as operações mais frequentes.
- Tamanho máximo da lista não é definido.
- Há a necessidade de voltar ao primeiro elemento da lista depois de percorrê-la.

### 5.6.1 Definindo o tipo lista dinâmica encadeada circular

Antes de começar a implementar a nossa lista, é preciso definir o tipo de dado que será armazenado. Uma lista pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o tipo opaco que representa nossa lista. E, trabalhando com alocação dinâmica da lista, este tipo será um **ponteiro para ponteiro** da estrutura que define a lista. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **ListaDinEncadCirc.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **ListaDinEncadCirc.h**, ilustrado na Figura 5.62. Pois se tratar de uma lista dinâmica encadeada, temos que estabelecer:

- O tipo de dado que será armazenado na lista, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para o **ponteiro** do tipo lista (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma lista.
- As funções disponíveis para trabalhar com essa lista em especial (linhas 8-20) e que serão implementadas no arquivo **ListaDinEncadCirc.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da lista. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



Por que colocamos um ponteiro no comando **typedef** quando criamos um novo nome para o tipo (linha 6)?

Por estarmos trabalhando com uma lista dinâmica e encadeada, temos que fazê-lo com um ponteiro para ponteiro a fim de poder realizar modificações no início da lista. Por questões de modularização, e para manter a mesma notação utilizada pela **lista sequencial estática**, podemos esconder um dos ponteiros do usuário. Assim, utilizar uma **lista sequencial estática**, uma **lista dinâmica encadeada** ou uma **lista dinâmica encadeada circular** será indiferente para o programador, pois a sua implementação está escondida dele:

- Lista \*li; //Declaração de uma **lista sequencial estática** (ponteiro).
- Lista \*li; //Declaração de uma **lista dinâmica encadeada** (ponteiro para ponteiro).
- Lista \*li; //Declaração de uma **lista dinâmica encadeada circular** (ponteiro para ponteiro).



No arquivo **ListaDinEncadCirc.c**, iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **ListaDinEncadCirc.h**.

Basicamente, o arquivo **ListaDinEncadCirc.c** (Figura 5.62) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da lista (linhas 1-3).
- A definição do tipo que descreve cada elemento da lista, **struct elemento** (linhas 5-8).
- A definição de um novo nome para a **struct elemento** (linhas 9). Isso é feito apenas para facilitar certas etapas de codificação.
- As implementações das funções definidas no arquivo **ListaDinEncadCirc.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que a nossa estrutura **elemento** nada mais é do que uma estrutura contendo dois campos: um **ponteiro prox**, que indica o próximo elemento (também do tipo **struct elemento**) dentro da lista, e um campo **dado** do tipo **struct aluno**, que é o tipo de dado a ser armazenado na lista. Por estarem definidos dentro do arquivo **.c**, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **ListaDinEncadCirc.h** (linha 6), que pode somente declarar um ponteiro para ele, da seguinte forma:

Lista \*li;



Note que não existe diferença na definição de uma **lista dinâmica encadeada** e de uma **lista dinâmica encadeada circular**. A diferença entre elas se dá na implementação de suas funções e não na definição dos seus tipos básicos.

Arquivo ListaDinEncadCirc.h

```

01 struct aluno{
02     int matricula;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 typedef struct elemento* Lista;
07
08 Lista* cria_lista();
09 void libera_lista(Lista* li);
10 int busca_lista_pos(Lista* li, int pos, struct aluno *al);
11 int busca_lista_mat(Lista* li, int mat, struct aluno *al);
12 int insere_lista_final(Lista* li, struct aluno al);
13 int insere_lista_inicio(Lista* li, struct aluno al);
14 int insere_lista_ordenada(Lista* li, struct aluno al);
15 int remove_lista(Lista* li, int mat);
16 int remove_lista_inicio(Lista* li);
17 int remove_lista_final(Lista* li);
18 int tamanho_lista(Lista* li);
19 int lista_vazia(Lista* li);
20 int lista_cheia(Lista* li);

```

Arquivo ListaDinEncadCirc.c

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ListaDinEncadCirc.h" //inclui os Protótipos
04 //Definição do tipo lista
05 struct elemento{
06     struct aluno dados;
07     struct elemento *prox;
08 };
09 typedef struct elemento Elemt;

```

FIGURA 5.62

## 5.6.2 Criando e destruindo uma lista

Para utilizar uma lista em seu programa, a primeira coisa a fazer é criar uma lista vazia. Essa tarefa é executada pela função descrita na Figura 5.63. Basicamente, o que esta função faz é a alocação de uma área de memória para armazenar o endereço do início da lista (linha 2), que é um **ponteiro para ponteiro**. Esta área de memória corresponde à memória necessária para armazenar o endereço de um elemento da lista, `sizeof(Lista)` ou `sizeof(struct elemento*)`. Em seguida, a função inicializa o conteúdo desse **ponteiro para ponteiro** com a constante `NULL`. Esta constante é utilizada em uma **lista dinâmica encadeada** para indicar que não existe nenhum elemento alocado até o momento. Isso significa que a lista está vazia. A Figura 5.64 indica o conteúdo do nosso ponteiro `Lista* li` após a chamada da função que cria a lista.



Note que não existe diferença entre criar uma **Lista dinâmica encadeada** e uma **lista dinâmica encadeada circular**.

```

01 Lista* cria_lista(){
02     Lista* li = (Lista*) malloc(sizeof(Lista));
03     if(li != NULL)
04         *li = NULL;
05     return li;
06 }

```

FIGURA 5.63

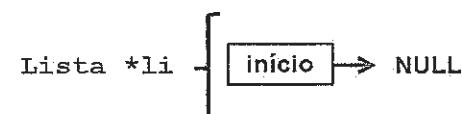


FIGURA 5.64

Destruir uma lista que utilize alocação dinâmica, e seja encadeada, não é uma tarefa tão simples quanto destruir uma **lista sequencial estática**.



Para liberar uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido nela.

O código que realiza a destruição da lista é mostrado na Figura 5.65. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso e se ela não está vazia (linha 2).



Para liberar uma **lista dinâmica encadeada circular**, não podemos mais percorrer a lista até que o conteúdo do seu início (`*li`) seja diferente de `NULL`.

Isso ocorre porque, em uma **lista dinâmica encadeada circular**, o último elemento tem como sucessor o primeiro elemento da lista, e não mais a constante `NULL`. O que devemos fazer é guardar em um ponteiro auxiliar (`no`) o endereço do primeiro elemento da lista (`*li`). Em seguida, devemos percorrer a lista até que o elemento seguinte ao elemento atual (`no->prox`) seja o início (`*li`), o que caracteriza uma volta completa na lista (linha 4). Enquanto não completarmos uma volta na lista, iremos liberar a memória do elemento que se encontra na posição atual e avançar para o próximo (linhas 5-7). Terminado o processo, liberamos a memória

alocada para o início da lista (linha 9). Esse processo é mais bem ilustrado pela Figura 5.66, que mostra a liberação de uma lista contendo dois elementos.

### Destruindo uma lista

```

01 void libera_lista(Lista* li){
02     if(li != NULL & & (*li) != NULL){
03         Elemt* aux, *no = *li;
04         while((*li) != no->prox) {
05             aux = no;
06             no = no->prox;
07             free(aux);
08         }
09         free(no);
10         free(li);
11     }
12 }
```

FIGURA 5.65

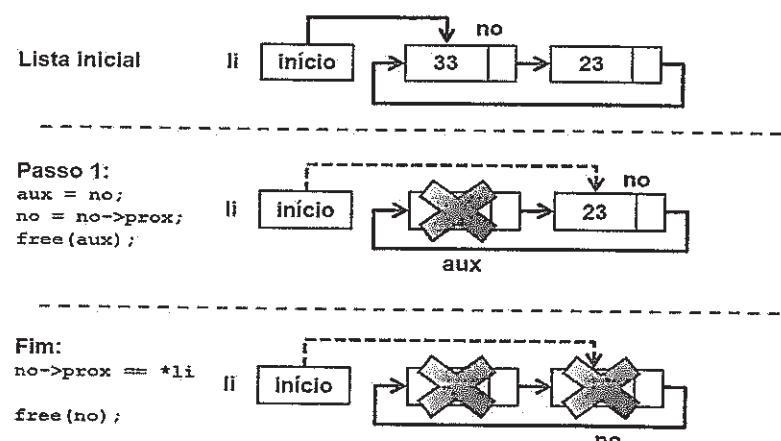


FIGURA 5.66

### 5.6.3 Informações básicas sobre a lista

As operações de inserção, remoção e busca são consideradas as principais de uma lista. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a lista. Por exemplo, não podemos remover um elemento da lista se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da lista: seu tamanho, se ela está cheia ou se ela está vazia.

### Tamanho da lista

Saber o tamanho de uma **lista dinâmica encadeada circular** é uma tarefa um pouco mais complexa do que saber o tamanho de uma **lista dinâmica encadeada**.



Para saber o tamanho de uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista contando os elementos inseridos nela, até encontrar o seu final. O problema é que uma **lista dinâmica encadeada circular** não possui final.

Para calcular o tamanho de uma **lista dinâmica encadeada circular**, não podemos mais percorrer a lista até que o conteúdo do seu início (\*li) seja diferente de **NONE**, como era feito com a **lista dinâmica encadeada**. Isso ocorre porque, em uma **lista dinâmica encadeada circular**, o último elemento tem como sucessor o primeiro elemento da lista, e não mais a constante **NONE**. Agora, devemos parar quando completarmos uma volta na lista.

O código que realiza a destruição da lista é mostrado na Figura 5.67. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso e se ela não está vazia (linha 2). Em seguida, criamos um contador iniciado em **ZERO** (linha 4) e um elemento auxiliar (no) apontado para o primeiro elemento da lista (linha 5). Então, percorremos a lista até que o valor de no seja novamente igual ao primeiro elemento da lista, \*li (linha 9). Enquanto não chegarmos novamente ao início da lista, iremos somar “+1” ao contador cont e avançar para o próximo elemento da lista (linhas 7-8). Terminado o processo, retornamos o valor da variável cont (linha 10). Esse processo é melhor ilustrado pela Figura 5.68, que mostra o cálculo do tamanho de uma lista contendo dois elementos.

### Tamanho da lista

```

01 int tamanho_lista(Lista* li){
02     if(li == NULL || (*li) == NULL)
03         return 0;
04     int cont = 0;
05     Elemt* no = *li;
06     do{
07         cont++;
08         no = no->prox;
09     }while(no != (*li));
10     return cont;
11 }
```

FIGURA 5.67

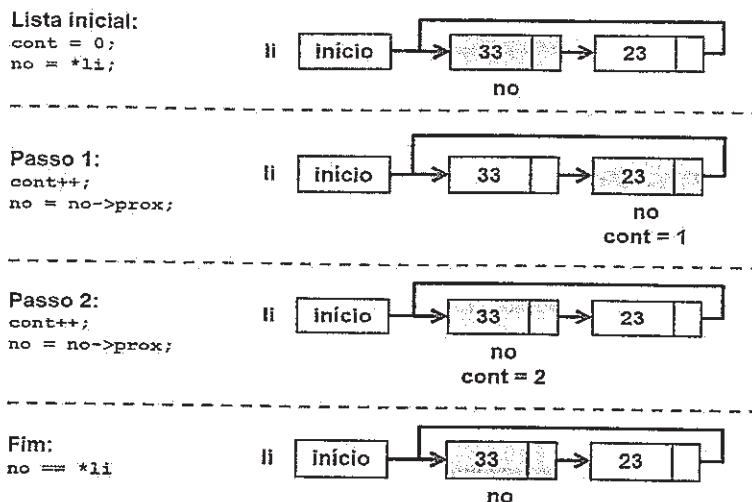


FIGURA 5.68

### Lista cheia

Implementar uma função que retorne se uma **lista dinâmica encadeada circular** está cheia é uma tarefa relativamente simples.



**Uma lista dinâmica encadeada circular** somente será considerada cheia quando não tivermos mais memória disponível para alocar novos elementos.

A implementação da função que retorna se a lista está cheia é mostrada na Figura 5.69. Como se pode notar, a função sempre irá retornar o valor **ZERO**, indicando que a lista não está cheia.

#### Retornando se a lista está cheia

```
01 int lista_cheia(Lista* li){  
02     return 0;  
03 }
```

FIGURA 5.69

### Lista vazia

Implementar uma função que retorne se uma **lista dinâmica encadeada circular** está vazia é outra tarefa relativamente simples.



Uma **lista dinâmica encadeada circular** será considerada vazia sempre que o conteúdo do seu “**íncio**” apontar para a constante **NULL**.

A implementação da função que retorna se a lista está vazia é mostrada na Figura 5.70. Note que essa função, em primeiro lugar, verifica se o ponteiro **Lista\* li** não é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **UM** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, então é possível acessar o conteúdo do seu “íncio” (\*li) e comparar o seu valor com a constante **NULL**, que é o valor inicial do conteúdo do “íncio” quando criamos a lista. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da lista), a função irá retornar o valor **UM** (linha 5). Caso contrário, irá retornar o valor **ZERO** (linha 6).

**Retornando se a lista está vazia**

```
01 int lista_vazia(Lista* li){  
02     if(li == NULL)  
03         return 1;  
04     if(*li == NULL)  
05         return 1;  
06     return 0;  
07 }
```

FIGURA 5.70

### 5.6.4 Inserindo um elemento na lista

#### Preparando a inserção na lista

Antes de inserir um elemento em uma **lista dinâmica encadeada circular**, algumas verificações são necessárias. Isso vale para os três tipos de inserção: no início, no final ou no meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.72, 5.74 e 5.76).

Primeiramente, a função verifica se o ponteiro **Lista\* li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linha 4). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 5 e 6). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 7).



No caso de uma lista com alocação dinâmica, ela somente será considerada cheia quando não tivermos mais memória disponível no computador para alocar novos elementos. Isso ocorrerá apenas quando a chamada da função **malloc()** retornar **NULL**.

Também existe a situação em que a inserção é feita em uma lista que está vazia, como mostrado na Figura 5.71. Neste caso, a lista, que inicialmente apontava para NULL, passa a apontar para o único elemento inserido até então.

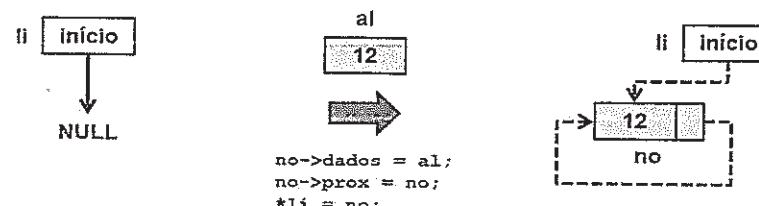


FIGURA 5.71

### Inserindo no início da lista

Inserir um elemento no início de uma **lista dinâmica encadeada circular** é uma tarefa um tanto trabalhosa.



Como na **lista dinâmica encadeada**, a inserção no início de uma **lista dinâmica encadeada circular** não necessita que se mude o lugar dos demais elementos da lista. Porém, como a lista é circular, é preciso percorrer a lista toda para descobrir o último elemento, ou seja, aquele que aponta para o primeiro elemento.

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista, encontrar o último da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.72. Note que as linhas 2 a 7 verificam se a inserção é possível.

Como se trata de uma inserção no início, temos que considerar que a lista pode ou não estar vazia (linha 8):

- No caso de ser uma lista vazia, mudamos o conteúdo do “início” da lista (\*li) para que ele passe a ser o nosso elemento no, que irá apontar para si mesmo (linhas 9 e 10), mantendo assim a circularidade da lista. Esse processo é mais bem ilustrado pela Figura 5.71.
- No caso de NÃO ser uma lista vazia, temos que achar o último elemento, pois ele aponta sempre para o primeiro da lista. Assim, devemos guardar em um ponteiro auxiliar (aux) o endereço do primeiro elemento da lista (\*li) e percorrer a lista até que o elemento seguinte ao elemento atual (aux->prox) seja o início (\*li), o que caracteriza uma volta completa na lista (linhas 12-15). Ao final do processo, o elemento no passa a ser o elemento seguinte a aux, sendo o elemento seguinte a no o antigo início da lista, \*li (linhas 16 e 17). Por fim, mudamos o conteúdo do “início” da lista (\*li) para que ele passe a ser o nosso elemento no (linha 18). Esse processo é mais bem ilustrado pela Figura 5.73.

Terminado um dos dois processos de inserção, retornamos o valor UM (linha 20), indicando sucesso na operação de inserção.

### Inserindo um elemento no início da lista

```

01 int insere_lista_inicio(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elemt *no = (Elemt*) malloc(sizeof(Elemt));
05     if(no == NULL)
06         return 0;
07     no->dados = al;
08     if((*li) == NULL){ //lista vazia: insere inicio
09         *li = no;
10         no->prox = no;
11     }else{
12         Elemt *aux = *li;
13         while(aux->prox != (*li)){
14             aux = aux->prox;
15         }
16         aux->prox = no;
17         no->prox = *li;
18         *li = no;
19     }
20     return 1;
21 }

```

FIGURA 5.72

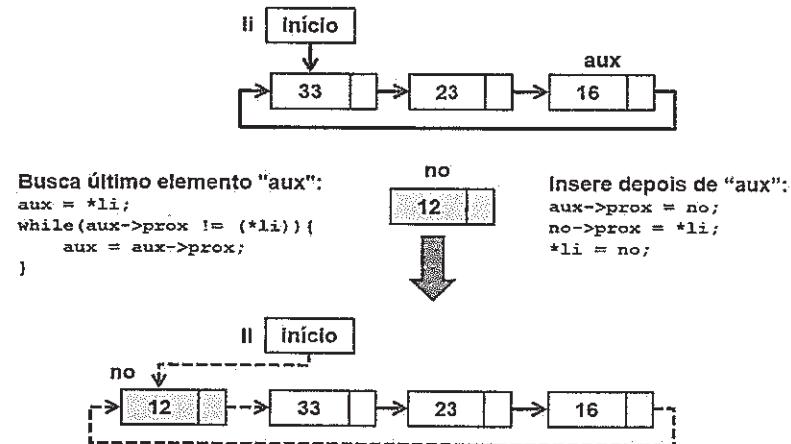


FIGURA 5.73

### Inserindo no final da lista

Inserir um elemento no final de uma **lista dinâmica encadeada circular** é uma tarefa praticamente igual à inserção no seu início, e igualmente trabalhosa. Isso ocorre porque é preciso

percorrer a lista toda para descobrir o último elemento, ou seja, aquele que aponta para o primeiro elemento. É entre o primeiro e o último que o novo elemento deverá ficar.

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista, encontrar o último da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.74. Note que as linhas 2 a 7 verificam se a inserção é possível.

Como se trata de uma inserção no final, temos que considerar que a lista pode ou não estar vazia (linha 8):

- No caso de ser uma lista vazia, mudamos o conteúdo do “íncio” da lista (\*li) para que ele passe a ser o nosso elemento **no**, que irá apontar para si mesmo (linhas 9 e 10), mantendo assim a circularidade da lista. Esse processo é mais bem ilustrado pela Figura 5.71;
- No caso de NÃO ser uma lista vazia, temos que achar o último elemento, pois ele aponta sempre para o primeiro da lista. Assim, devemos guardar em um ponteiro auxiliar (aux) o endereço do primeiro elemento da lista (\*li) e percorrer a lista até que o elemento seguinte ao elemento atual (**aux->prox**) seja o íncio (\*li), o que caracteriza uma volta completa na lista (linhas 12-15). Ao final do processo, o elemento **no** passa a ser o elemento seguinte a **aux**, sendo o elemento seguinte a **no** o íncio da lista, \*li (linhas 16 e 17). Esse processo é mais bem ilustrado pela Figura 5.75.

Terminado um dos dois processos de inserção, retornamos o valor **UM** (linha 19), indicando sucesso na operação de inserção.

#### Inserindo um elemento no final da lista

```

01 int insere_lista_final(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elem *no = (Elem*) malloc(sizeof(Elem));
05     if(no == NULL)
06         return 0;
07     no->dados = al;
08     if((*li) == NULL){ //lista vazia: insere inicio
09         *li = no;
10         no->prox = no;
11     }else{
12         Elem *aux = *li;
13         while(aux->prox != (*li)){
14             aux = aux->prox;
15         }
16         aux->prox = no;
17         no->prox = *li;
18     }
19     return 1;
20 }
```

FIGURA 5.74

Perceba que o código para inserir um elemento no início (Figura 5.72) e no final (Figura 5.74) difere em apenas uma única linha. Em uma lista circular, inserir no início ou no final equivale a colocar um novo elemento entre o último e o primeiro. A diferença é que temos que mudar o ponteiro que indica o íncio da lista (\*li) quando inserimos no início, mas não precisamos fazer isso quando inserimos no final.

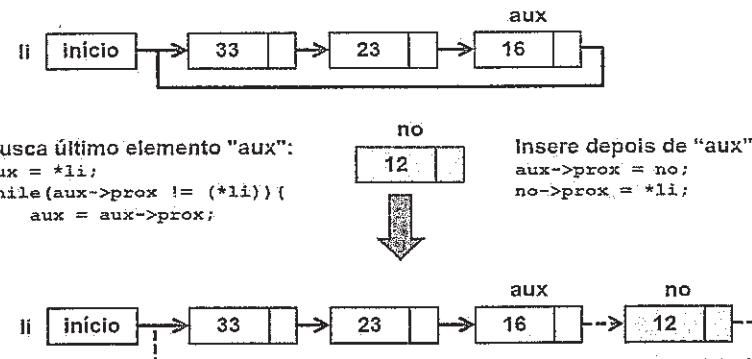


FIGURA 5.75

#### Inserindo de forma ordenada na lista

Inserir um elemento de forma ordenada em uma lista **dinâmica encadeada circular** é uma tarefa trabalhosa.

Isso ocorre porque precisamos procurar o ponto de inserção do elemento na lista, o qual pode ser no início, no meio ou no final da lista. Porém, diferente da **lista sequencial estática**, a inserção ordenada em uma **lista dinâmica encadeada circular** não necessita que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é procurar em que lugar da lista será inserido o novo elemento (no caso, iremos ordenar pelo campo matrícula), alocar espaço para ele na lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.76. Note que as linhas 2 a 7 verificam se a inserção é possível.

Antes de fazer a inserção, temos que considerar que a lista pode ou não estar vazia (linha 8):

- No caso de ser uma lista vazia, mudamos o conteúdo do “íncio” da lista (\*li) para que ele passe a ser o nosso elemento **no**, que irá apontar para si mesmo (linhas 9 e 10), mantendo assim a circularidade da lista. Esse processo é mais bem ilustrado pela Figura 5.71.

**Insertindo um elemento de forma ordenada na lista**

```

01 int insere_lista_ordenada(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     ELEM *no = (ELEM*) malloc(sizeof(ELEM));
05     if(no == NULL)
06         return 0;
07     no->dados = al;
08     if((*li) == NULL) //insere inicio
09         *li = no;
10         no->prox = no;
11         return 1;
12     }
13     else{
14         if((*li)->dados.matricula > al.matricula) //inicio
15             ELEM *atual = *li;
16             while(atual->prox != (*li)) //procura o ultimo
17                 atual = atual->prox;
18             no->prox = *li;
19             atual->prox = no;
20             *li = no;
21             return 1;
22         }
23         ELEM *ant = *li, *atual = (*li)->prox;
24         while(atual != (*li) &&
25             atual->dados.matricula < al.matricula){
26             ant = atual;
27             atual = atual->prox;
28         }
29         ant->prox = no;
30         no->prox = atual;
31         return 1;
32     }

```

FIGURA 5.76

- No caso de NÃO ser uma lista vazia, a inserção pode ser no início ou não (linha 14):
  - Se a inserção for no início, temos que achar o último elemento, pois ele aponta sempre para o primeiro da lista. Assim, devemos guardar em um ponteiro auxiliar (*atual*) o endereço do primeiro elemento da lista (*\*li*) e percorrer a lista até que o elemento seguinte ao elemento atual (*atual->prox*) seja o início (*\*li*), o que caracteriza uma volta completa na lista (linhas 15-17). Ao final do processo, o elemento *no* passa a ser o elemento seguinte a *atual*, sendo o elemento seguinte a *no* o início da lista, *\*li* (linhas 18 e 19). Por fim, mudamos o conteúdo do “início” da lista (*\*li*) para que ele passe a ser o nosso elemento *no* (linha 20). Esse processo é mais bem ilustrado pela Figura 5.77.
  - Se a inserção for no meio ou no final da lista, temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento *atual* for

menor do que a matrícula do novo elemento a ser inserido (linhas 23-27). Note que, além do elemento *atual*, também armazenamos o elemento anterior a ele (*ant*), que é necessário em uma inserção no meio da lista. Ao final do processo, o elemento *ant* irá apontar para o elemento *no*, que apontará para o elemento *atual* (linhas 28-29). Esse processo é mais bem ilustrado pela Figura 5.77.

Terminado algum dos processos de inserção, retornamos o valor **UM**, indicando sucesso na operação de inserção.

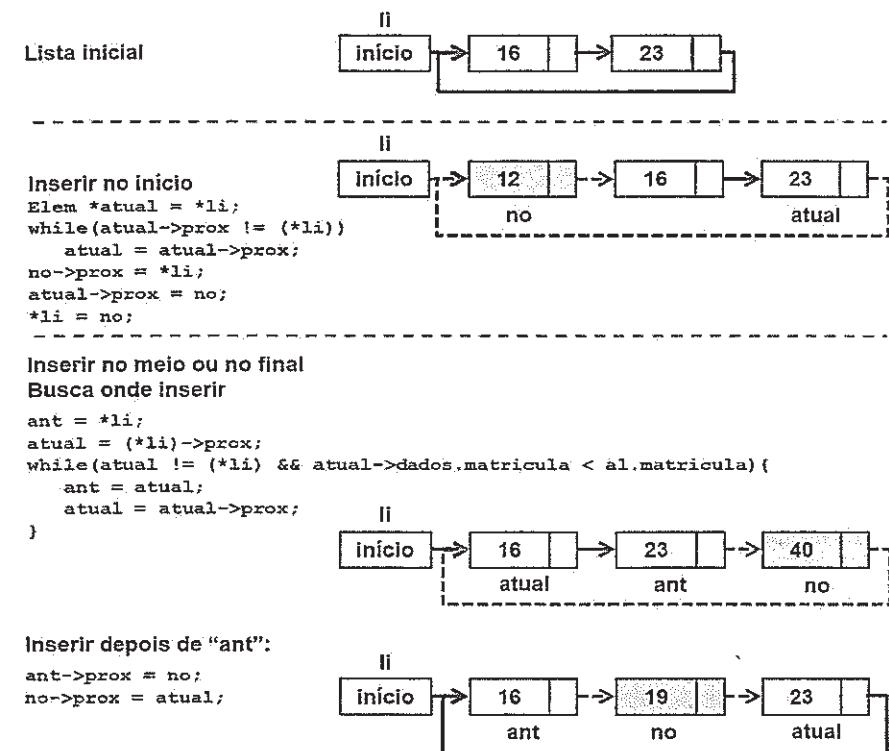


FIGURA 5.77

### 5.6.5 Removendo um elemento da lista

#### Preparando a remoção da lista

Antes de remover um elemento de uma lista **dinâmica encadeada circular**, algumas verificações são necessárias. Isso vale para os três tipos de remoção: do início, do final ou do meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.79, 5.81 e 5.83).

Primeiramente, verificamos se o ponteiro **Lista\* li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, precisamos verificar se ela não está vazia, isto é, se existem elementos dentro dela. Caso a lista esteja vazia, a função irá retornar o valor **ZERO** (linhas 4 e 5).



No caso de uma lista com alocação dinâmica, ela somente será considerada vazia quando o seu início apontar para a constante **NULL**.

Também existe a situação em que a remoção é feita em uma lista que possui um único elemento, como mostrado na Figura 5.78. Neste caso, a lista fica vazia após a remoção.

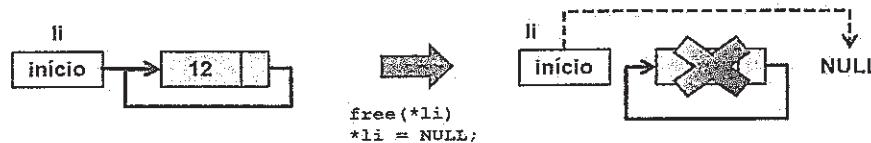


FIGURA 5.78

### Removendo do início da lista

Remover um elemento do início de uma **lista dinâmica encadeada circular** é uma tarefa um tanto trabalhosa.



Como na **lista dinâmica encadeada**, a remoção no início de uma **lista dinâmica encadeada circular** não necessita que se mude o lugar dos demais elementos da lista. Porém, como a lista é circular, é preciso percorrer a lista toda para descobrir o último elemento, ou seja, aquele que aponta para o primeiro elemento.

Basicamente, o que temos que fazer é verificar se a lista não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.79. Note que as linhas 2 a 5 verificam se a remoção é possível.

```

Removendo o elemento do início da lista
01 int remove_lista_inicio(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     if((*li) == (*li)->prox){//lista fica vazia
08         free(*li);
09         *li = NULL;
10         return 1;
11     }
12     Elemt *atual = *li;
13     while(atual->prox != (*li))//procura o último
14         atual = atual->prox;
15
16     Elemt *no = *li;
17     atual->prox = no->prox;
18     *li = no->prox;
19     free(no);
20 }

```

FIGURA 5.79

Por ser uma **lista dinâmica encadeada circular**, temos que considerar o caso de o elemento removido ser o único da lista, ou seja, dele apontar para si mesmo (linha 7):

- No caso de ser o único elemento da lista, liberamos a memória alocada para o primeiro item da lista e mudamos o conteúdo do “**início**” da lista (**\*li**) para que ele passe a apontar para a constante **NULL**, indicando assim uma lista vazia (linhas 8-9). Por fim, retornamos o valor **UM** (linha 10), indicando sucesso na operação de remoção.
- No caso de existirem mais elementos na lista, temos que achar o último elemento, pois ele aponta sempre para o primeiro da lista. Assim, devemos guardar em um ponteiro auxiliar (**atual**) o endereço do primeiro elemento da lista (**\*li**) e percorrer a lista até que o elemento seguinte ao elemento atual (**atual->prox**) seja o início (**\*li**), o que caracteriza uma volta completa na lista (linhas 12-14). Ao final do processo, o elemento **no** recebe o início da lista (linha 16). Em seguida, o último elemento passa a apontar para o segundo elemento da lista e o **início** passa a ser este segundo elemento (linhas 17 e 18). Por fim, temos que liberar a memória associada ao antigo “**início**” da lista (**no**) e retornamos o valor **UM** (linhas 19 e 20), indicando sucesso na operação de remoção.

O processo de remoção é melhor ilustrado pela Figura 5.80.

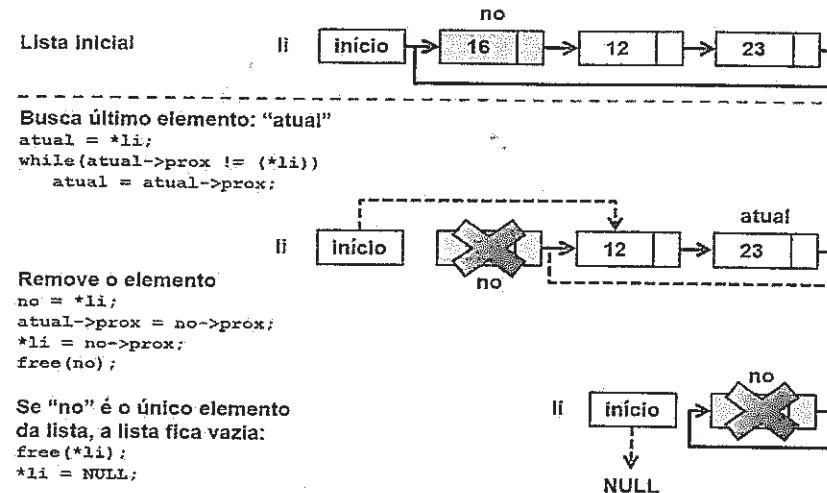


FIGURA 5.80

### Removendo do final da lista

Remover um elemento do final de uma **lista dinâmica encadeada circular** é uma tarefa praticamente igual à remoção no seu início, e igualmente trabalhosa. Isso ocorre porque é preciso percorrer a lista toda para descobrir o último elemento, ou seja, aquele que aponta para o primeiro elemento.

Basicamente, o que temos que fazer é verificar se a lista não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.81. Note que as linhas 2 a 5 verificam se a remoção é possível.

Por ser uma **lista dinâmica encadeada circular**, temos que considerar o caso de o elemento removido ser o único da lista, ou seja, dele apontar para si mesmo (linha 7):

- No caso de ser o único elemento da lista, liberamos a memória alocada para o primeiro item da lista e mudamos o conteúdo do “*início*” da lista (*\*li*) para que ele passe a apontar para a constante **NONE**, indicando assim uma lista vazia (linhas 8-9). Por fim, retornamos o valor **UM** (linha 10), indicando sucesso na operação de remoção.
- No caso de existirem mais elementos na lista, temos que achar o último elemento, pois ele aponta sempre para o primeiro da lista. Assim, devemos guardar em um ponteiro auxiliar (*no*) o endereço do primeiro elemento da lista (*\*li*) e percorrer a lista até que o elemento seguinte a ele (*no->prox*) seja o *início* (*\*li*), o que caracteriza uma volta completa na lista (linhas 12-16). Note que juntamente com o elemento *no*, guardamos também o elemento anterior a ele na lista, *ant*. Ao final do processo, o elemento *ant* passa a apontar para o elemento seguinte ao último (*no*), isto é, o *início* da lista (linha 17). Por

fim, temos que liberar a memória associada ao antigo “final” da lista (*no*) e retornamos o valor **UM** (linhas 18 e 19), indicando sucesso na operação de remoção.

O processo de remoção é melhor ilustrado pela Figura 5.82.

**Removendo um elemento do final da lista**

```

01 int remove_lista_final(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     if((*li) == (*li)->prox){//lista fica vazia
08         free(*li);
09         *li = NULL;
10         return 1;
11     }
12     ELEM *ant, *no = *li;
13     while(no->prox != (*li)){//procura o último
14         ant = no;
15         no = no->prox;
16     }
17     ant->prox = no->prox;
18     free(no);
19     return 1;
20 }
  
```

FIGURA 5.81

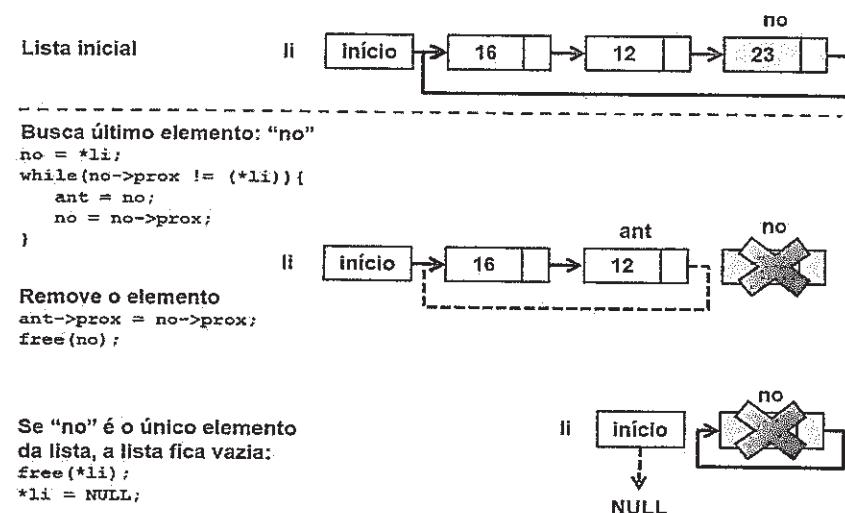


FIGURA 5.82

### Removendo um elemento específico da lista

Remover um elemento específico de uma **lista dinâmica encadeada circular** é uma tarefa bastante trabalhosa.



Isso ocorre porque, além de precisarmos procurar o elemento a ser removido, esse elemento pode estar no início ou no final da lista, o que nos leva ao problema da remoção do início ou final da lista visto anteriormente.

Basicamente, o que temos que fazer é procurar esse elemento na lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.83. Note que as linhas 2 a 5 verificam se a remoção é possível.

#### Removendo um elemento específico da lista

```

01 int remove_lista(Lista* li, int mat){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06     Elemt *no = *li;
07     if(no->dados.matricula == mat){//remover do inicio
08         if(no == no->prox){//lista fica vazia
09             free(no);
10             *li = NULL;
11             return 1;
12         }else{
13             Elemt *ult = *li;
14             while(ult->prox != (*li))//procura o ultimo
15                 ult = ult->prox;
16             ult->prox = (*li)->prox;
17             *li = (*li)->prox;
18             free(no);
19             return 1;
20         }
21     }
22     Elemt *ant = no;
23     no = no->prox;
24     while(no != (*li) && no->dados.matricula != mat){
25         ant = no;
26         no = no->prox;
27     }
28     if(no == *li)//não encontrado
29         return 0;
30
31     ant->prox = no->prox;
32     free(no);
33     return 1;
34 }
```

FIGURA 5.83

No nosso exemplo, vamos remover um elemento de acordo com o campo matrícula. Primeiro, verificamos se este é o primeiro elemento da lista (linha 7). Em caso afirmativo, tratarmos a remoção como se ela fosse uma remoção no início (linhas 7-21).

Caso o elemento procurado não seja o primeiro da lista, temos que percorrer a lista enquanto não chegarmos ao seu início (li) novamente, e enquanto o campo matrícula do elemento no for diferente do valor de matrícula procurado (linhas 22-27). Note que, além do elemento no, também armazenamos o elemento anterior a ele (ant), que é necessário em uma remoção no meio da lista. Terminado o processo de busca, verificamos se estamos no início da lista ou não (linha 28). Em caso afirmativo, o elemento não existe na lista e a remoção não é possível (linha 29). Caso contrário, basta apenas fazer o elemento ant apontar para o elemento seguinte a no (linha 31).

Ao final do processo de remoção, temos que liberar a memória associada ao elemento no e retornarmos o valor UM (linhas 32 e 33), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.84.

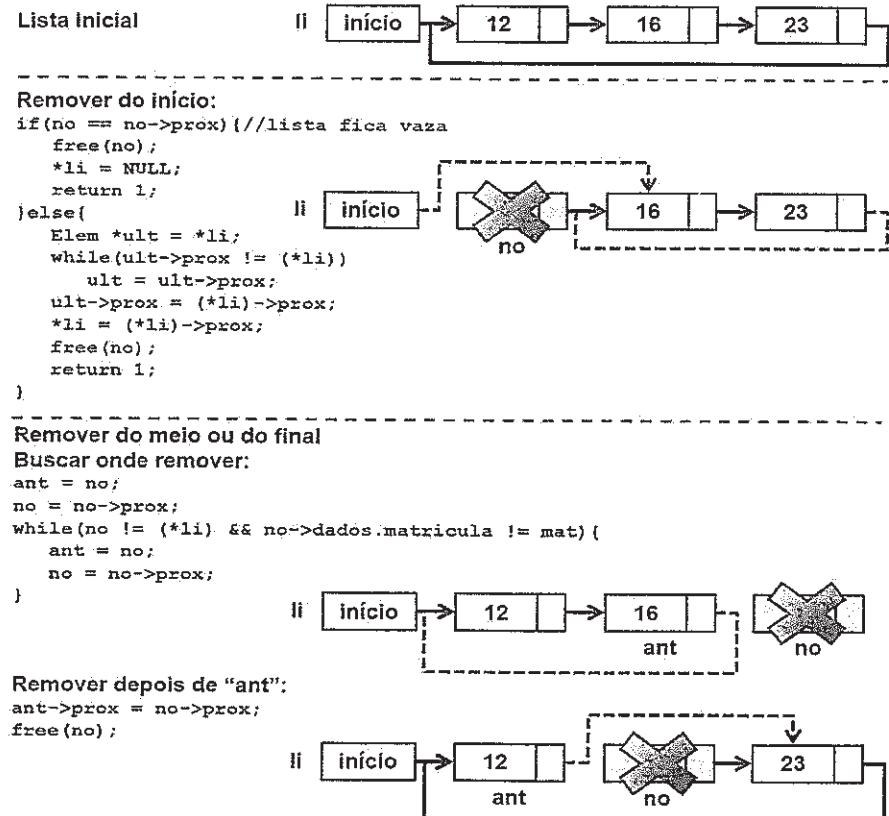


FIGURA 5.84

### 5.6.6 Busca por um elemento da lista

A operação de busca consiste em recuperar as informações contidas em determinado elemento da lista.



De modo geral, a operação de busca pode ser feita de diversas maneiras, dependendo da necessidade da aplicação.

Por exemplo, podemos buscar as informações:

- Do segundo elemento da lista: busca por posição.
- Do elemento que possui um pedaço de informação conhecida: busca por conteúdo.



Em uma lista que utilize alocação dinâmica, e seja encadeada, a busca sempre envolve a necessidade de percorrer a lista.

A seguir, veremos como esses dois tipos de busca funcionam.

#### Busca por posição na lista

O código que realiza a busca de um elemento por sua posição em uma **lista dinâmica encadeada circular** é mostrado na Figura 5.85. Em primeiro lugar, a função verifica se a busca é válida. Para tanto, três condições são verificadas: se o ponteiro `Lista* li` é igual a `NULL`, se o conteúdo do primeiro elemento da lista é `NULL` e se a posição buscada (`pos`) é um valor negativo. Se alguma dessas condições for verdadeira, a busca termina e a função retorna o valor

#### Busca um elemento por posição

```

01 int busca_lista_pos(Lista* li, int pos, struct aluno *al){
02     if(li == NULL || (*li) == NULL || pos <= 0)
03         return 0;
04     Elemt *no = *li;
05     int i = 1;
06     while(no->prox != (*li) && i < pos)
07         no = no->prox;
08         i++;
09     }
10     if(i != pos)
11         return 0;
12     else{
13         *al = no->dados;
14         return 1;
15     }
16 }
```

FIGURA 5.85

**ZERO** (linha 3). Caso contrário, criamos um elemento auxiliar (`no`) apontado para o primeiro elemento da lista (linha 4) e um contador (`i`) iniciado em UM (linha 5). Então, percorremos a lista enquanto o próximo elemento de `no` for igual ao primeiro elemento da lista, e enquanto o valor do contador for menor do que a posição desejada (linhas 6-9). Terminado o laço, verificamos se o valor do contador é diferente da posição desejada (linha 10). Se essa condição for verdadeira, o elemento não foi encontrado na lista. Do contrário, a posição atual (`no`) é copiada para o conteúdo do ponteiro passado por referência (`al`) para a função (linha 13). A Figura 5.86 ilustra os principais pontos dessa busca.

#### Busca pela posição do elemento

```

no = *li;
int i = 1;
while(no->prox != (*li) && i < pos){
    no = no->prox;
    i++;
}
Verifica se a posição foi
encontrada e a retorna
if(i != pos) return 0;
else{
    *al = no->dados;
    return 1;
}
```

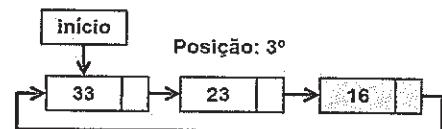


FIGURA 5.86

#### Busca por conteúdo na lista

O código que realiza a busca de um elemento por sua posição em uma **lista dinâmica encadeada circular** é mostrado na Figura 5.87. Neste caso, estamos procurando um aluno pelo seu número de matrícula. Como se pode notar, o código em muito se parece ao da busca por posição mostrado na Figura 5.85. As diferenças são poucas:

- Não precisamos verificar se a matrícula é válida como no caso da posição buscada (linha 2).
- Não percorremos mais a lista comparando o valor da posição, mas sim o valor da matrícula (linha 5).
- Terminado o laço, verificamos se o valor da matrícula do elemento atual é o que procurávamos (linha 7).

**Busca um elemento pelo conteúdo**

```

01 int busca_lista_mat(Lista* li, int mat, struct aluno *al){
02     if(li == NULL || (*li) == NULL)
03         return 0;
04     Elemt *no = *li;
05     while(no->prox != (*li) && no->dados.matricula != mat)
06         no = no->prox;
07     if(no->dados.matricula != mat)
08         return 0;
09     else{
10         *al = no->dados;
11         return 1;
12     }
13 }
```

FIGURA 5.87

A Figura 5.88 ilustra os principais pontos dessa busca.

**Busca pelo conteúdo do elemento**

```

no = *li;
while(no->prox != (*li) && no->dados.matricula != mat)
    no = no->prox;
```

**Verifica se o elemento foi encontrado e o retorna**

```

if(no->dados.matrícula != mat)
    return 0;
else{
    *al = no->dados;
    return 1;
}
```

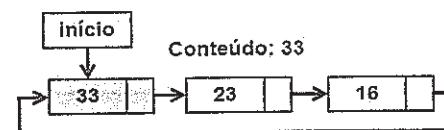


FIGURA 5.88

**5.6.7 Análise de complexidade**

Um aspecto importante quando manipulamos listas tem relação com os custos das suas operações. Na sequência, são mostradas as complexidades computacionais das principais operações em uma **lista dinâmica encadeada circular** contendo  $N$  elementos:

- Inserção no início:** é preciso percorrer toda a lista até alcançar o seu final, que aponta para o início. Desse modo, a sua complexidade é  $O(N)$ .
- Inserção no final:** é preciso percorrer toda a lista até alcançar o elemento anterior ao seu final. Desse modo, a sua complexidade é  $O(N)$ .
- Inserção ordenada:** neste caso, é preciso procurar o ponto de inserção, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (inserção no início ou no final).

- Remoção do início:** como na inserção, é preciso percorrer toda a lista até alcançar o seu final, que aponta para o início. Assim, a sua complexidade é  $O(N)$ .
- Remoção do final:** como na inserção, é preciso percorrer toda a lista até alcançar o elemento anterior ao seu final. Desse modo, a sua complexidade é  $O(N)$ .
- Remoção de um elemento específico:** essa operação envolve a busca pelo elemento a ser removido, que pode estar no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (remoção do final).
- Consulta:** a operação de consulta envolve a busca de um elemento, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (último elemento).

**5.6.8 Aumentando o desempenho de uma lista circular**

Como visto até agora, as operações de inserção e remoção na **lista dinâmica encadeada circular** são bastante trabalhosas, principalmente quando realizadas no início ou no final da lista.



Pelo fato de a lista ser circular, é preciso percorrer a lista toda para descobrir o último elemento, ou seja, aquele que aponta para o primeiro elemento.

Isso ocorre porque usamos uma representação de lista circular que guarda sempre a posição de início da lista. Assim, temos que andar a lista toda para encontrar o seu final e assim manter a circularidade da lista nas operações que alteram o seu início ou final.

Uma solução bastante simples seria mudar nossa representação da lista circular: em vez de guardar a posição de início da lista em  $(*li)$ , passamos a guardar a posição de final da lista. Essa alteração em nada modifica o armazenamento de elementos na lista. Trata-se apenas de uma mudança na lógica de operação da lista, como mostrado na Figura 5.89.

Lista \*li

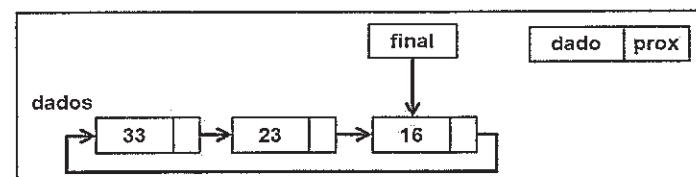


FIGURA 5.89



Em uma lista circular, inserir no início ou no final equivale a colocar um novo elemento entre o último e o primeiro.

Imagine que queremos inserir um novo elemento **X** em uma das extremidades da lista. Devido ao fato de a lista ser circular, inserir um novo elemento em qualquer uma de suas extremidades (início ou final) equivale a colocar esse novo elemento entre o seu final e o seu **íncio**, como mostrado na Figura 5.90. Assim, fazer com que a lista armazene o final dela no ponteiro (**\*li**) não muda o funcionamento da lista, mas evita que se percorra a lista em alguns tipos de inserção e remoção.



Essa modificação simplifica as operações de inserção, no **íncio** e no **final**, e de remoção, no **íncio** da lista. As demais operações não têm seu desempenho modificado, mas podem apresentar algumas pequenas modificações para lidar com a nova posição de final da lista armazenada.

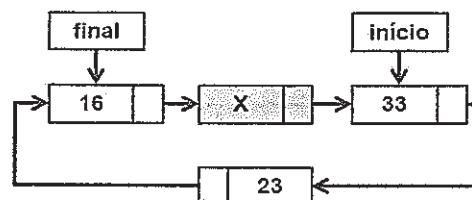
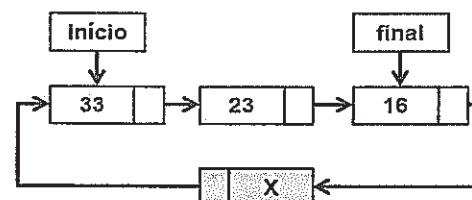


FIGURA 5.90

### Inserindo no **íncio** da lista

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.91. Primeiro, a função verifica se o ponteiro **Lista\* li** é igual a **NUL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linha 4). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 5 e 6). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 7).

Até aqui, o que fizemos foi verificar se podíamos inserir na lista e criar um novo elemento (**no**) com os dados passados por parâmetro. Falta, obviamente, inserir este elemento na lista. Como se trata de uma inserção no **íncio**, temos que considerar que a lista pode ou não estar vazia (linha 8):

- No caso de ser uma lista vazia, mudamos o conteúdo do “**final**” da lista (**\*li**) para que ele passe a ser o nosso elemento **no**, que irá apontar para si mesmo (linhas 9 e 10), mantendo assim a circularidade da lista.
- No caso de NÃO ser uma lista vazia, o elemento **no** irá apontar para o elemento seguinte ao final da lista, (**\*li**)**->prox**, que nada mais é do que o primeiro elemento da lista (linha 12). Em seguida, o elemento **no** passa a ser o elemento seguinte ao final da lista (linha 13). Esse processo é mais bem ilustrado pela Figura 5.92.

Terminado um dos dois processos de inserção, retornamos o valor **UM** (linha 15), indicando sucesso na operação de inserção.

### Inserindo um elemento no **íncio** da lista

```

01 int insere_lista_inicio(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elemt *no = (Elemt*) malloc(sizeof(Elemt));
05     if(no == NULL)
06         return 0;
07     no->dados = al;
08     if((*li) == NULL){//lista vazia: insere inicio
09         *li = no;
10         no->prox = no;
11     }else{
12         no->prox = (*li)->prox;
13         (*li)->prox = no;
14     }
15     return 1;
16 }
```

FIGURA 5.91

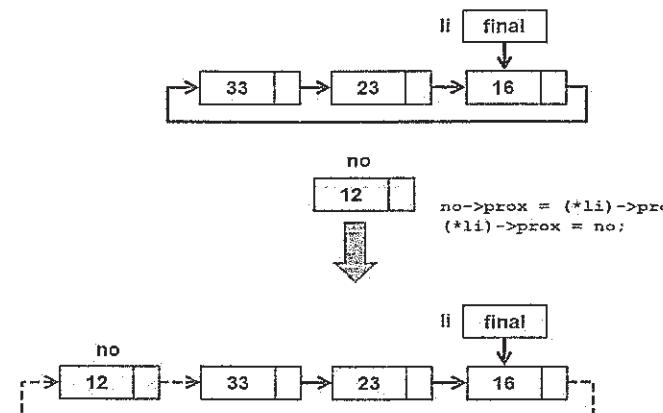


FIGURA 5.92

### Inserindo no final da lista

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.93. Primeiro, a função verifica se o ponteiro **Lista\*** **li** é igual a **NUL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linha 4). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 5 e 6). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 7).

Até aqui, o que fizemos foi verificar se podíamos inserir na lista e criar um novo elemento (**no**) com os dados passados por parâmetro. Falta, obviamente, inserir este elemento na lista. Como se trata de uma inserção no final, temos que considerar que a lista pode ou não estar vazia (linha 8):

- No caso de ser uma lista vazia, mudamos o conteúdo do “final” da lista (**\*li**) para que ele passe a ser o nosso elemento **no**, que irá apontar para si mesmo (linhas 9 e 10), mantendo assim a circularidade da lista.
- No caso de NÃO ser uma lista vazia, o elemento **no** irá apontar para o elemento seguinte ao final da lista, **(\*li)->prox**, que nada mais é do que o primeiro elemento da lista (linha 12). Em seguida, o elemento **no** passa a ser o elemento seguinte ao final da lista (linha 13). Como estamos mudando o final da lista, o elemento **no** passa a ser o novo final da lista (linha 14). Esse processo é mais bem ilustrado pela Figura 5.94.

Terminado um dos dois processos de inserção, retornamos o valor **UM** (linha 16), indicando sucesso na operação de inserção.

Perceba que os códigos para inserir um elemento no início (Figura 5.91) e no final (Figura 5.93) diferem apenas por uma única linha. Em uma lista circular, inserir no início ou no final equivale a colocar um novo elemento entre o último e o primeiro. A diferença é que temos que mudar o ponteiro que indica o final da lista (**\*li**) quando inserimos no final, mas não precisamos fazer isso quando inserimos no início.

FIGURA 5.93

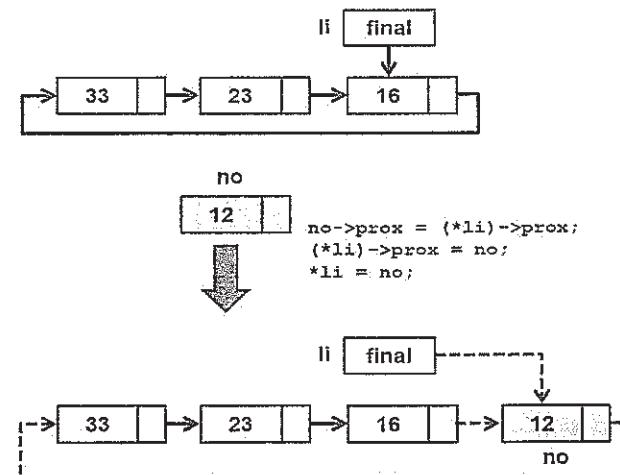


FIGURA 5.94

### Removendo do início da lista

Basicamente, o que temos que fazer é verificar se existem elementos na lista para serem removidos e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.95. Primeiro, verificamos se o ponteiro `Lista* li` é igual a `NULL`. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor `ZERO` para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, precisamos verificar se ela não está vazia, isto é, se existem elementos dentro dela. Caso a lista esteja vazia, a função irá retornar o valor `ZERO` (linhas 4 e 5).

Até aqui, o que fizemos foi verificar se podíamos remover um elemento da lista. Falta, obviamente, remover este elemento. Por ser uma **lista dinâmica encadeada circular**, temos que considerar o caso de o elemento removido ser o único da lista, ou seja, dele apontar para si mesmo (linha 7):

- No caso de ser o único elemento da lista, liberamos a memória alocada para o primeiro item da lista e mudamos o conteúdo do “início” da lista (`*li`) para que ele passe a apontar para a constante `NULL`, indicando assim uma lista vazia (linhas 8-9). Por fim, retornamos o valor `UM` (linha 10), indicando sucesso na operação de remoção.
- No caso de existirem mais elementos na lista, criamos um elemento `no` para armazenar o elemento seguinte ao final da lista, `(*li)->prox`, que nada mais é do que o primeiro elemento da lista (linha 12). Em seguida, o elemento seguinte a `no` passa a ser o elemento seguinte ao final da lista (linha 13). Por fim, temos que liberar a memória associada ao antigo “início” da lista (`no`) e retornamos o valor `UM` (linhas 14 e 15), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.96.

#### Removendo um elemento do início da lista

```

01 int remove_lista_inicio(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     if((*li) == (*li)->prox){//lista fica vazia
08         free(*li);
09         *li = NULL;
10         return 1;
11     }
12     ELEM *no = (*li)->prox;
13     (*li)->prox = no->prox;
14     free(no);
15     return 1;
16 }
```

FIGURA 5.95

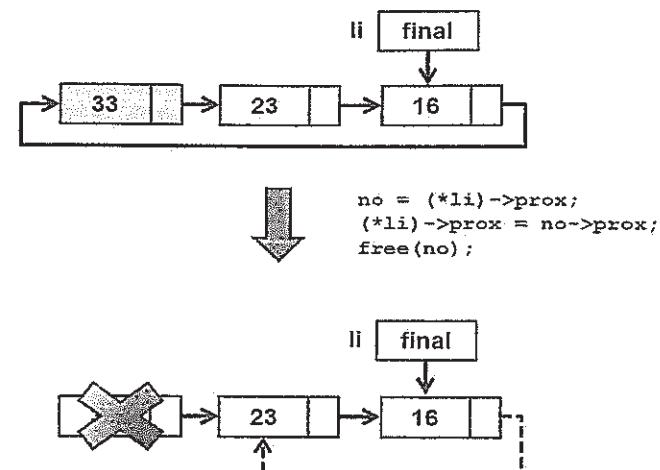


FIGURA 5.96

### 5.7 LISTA DINÂMICA DUPLAMENTE ENCADEADA

Uma **lista dinâmica duplamente encadeada** é uma lista definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da lista é alocado dinamicamente, à medida que os dados são inseridos dentro da lista, e tem sua memória liberada, à medida que é removido.



Diferente da **lista dinâmica encadeada**, esse tipo de lista não possui dois, mas sim três campos de informação dentro de cada elemento: os campos `dado`, `prox` e `ant`.

Como na **lista dinâmica encadeada**, cada elemento da lista nada mais é do que um ponteiro para uma estrutura. No entanto, esta estrutura contém agora três campos de informação: um campo de `dado`, utilizado para armazenar a informação inserida na lista, um campo `prox`, utilizado para indicar o próximo elemento na lista, e um campo `ant`, idêntico ao campo `prox` mas utilizado para indicar o elemento anterior na lista.



É a existência dos campos `prox` e `ant` que garante que a lista é **duplamente encadeada**.

A presença dos ponteiros `prox` e `ant` garantem que a lista seja encadeada em dois sentidos: no seu sentido normal, aquele usado para percorrer um lista do seu início até o seu final, e no sentido inverso, quando percorremos a lista de volta ao seu início. Por isso, ela é chamada **duplamente encadeada**: cada elemento aponta para o sucessor (`prox`) e o antecessor (`ant`) dentro da lista.



Além da estrutura que define seus elementos, essa lista utiliza um **ponteiro para ponteiro** para guardar o primeiro elemento da lista.

Como na **lista dinâmica encadeada**, temos aqui que todos os elementos da lista são ponteiros alocados dinamicamente. Para inserir um elemento no início da lista é necessário utilizar um campo que seja fixo, mas que, ao mesmo tempo, seja capaz de apontar para o novo elemento.

É necessário o uso de um **ponteiro para ponteiro** para guardar o endereço de um **ponteiro**. Utilizando um **ponteiro para ponteiro** para representar o início da lista, fica fácil mudar quem é o primeiro elemento da lista mudando o apenas o **conteúdo do ponteiro para ponteiro**. Mais detalhes são apresentados na Seção 5.5.1.



Após o último elemento, não existe nenhum novo elemento alocado. Sendo assim, o último elemento da lista aponta o campo **prox** para **NULL**. O mesmo vale para o primeiro elemento da lista: não existe ninguém antes dele. Sendo assim, ele aponta o seu campo **ant** para **NULL**.

Considerando uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo lista (ela é um **tipo opaco**), como ilustrado na Figura 5.97. Isso impede o usuário de saber como foi realmente implementada a lista, e limita o seu acesso apenas às funções que manipulam a lista.

Lista \*li

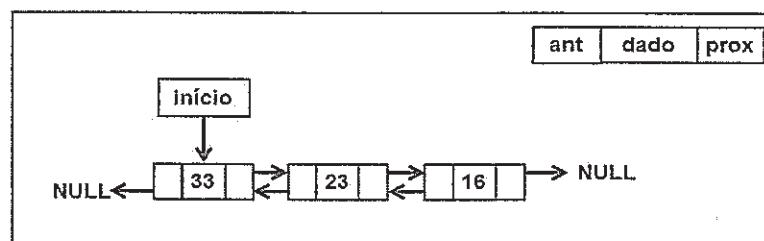


FIGURA 5.97



Note que a implementação em módulos impede o usuário de saber como a lista foi implementada. Tanto a **Lista dinâmica duplamente encadeada** quanto as outras listas vistas até agora são declaradas como sendo do tipo **Lista \***.

Essa é a grande vantagem da modularização e da utilização de tipos opacos: mudar a maneira pela qual a lista foi implementada não altera nem interfere no funcionamento do programa que a utiliza.



O uso de alocação dinâmica e acesso encadeado na definição de uma **lista dinâmica duplamente encadeada** traz vantagens e desvantagens, que devem ser consideradas para um melhor desempenho da aplicação.

Várias são as vantagens de se definir uma lista utilizando uma abordagem dinâmica e encadeada:

- Melhor utilização dos recursos de memória.
- Não é preciso definir previamente o tamanho da lista.
- Não se precisa movimentar os elementos nas operações de inserção e remoção.

Infelizmente, esse tipo de implementação também tem suas desvantagens:

- Acesso indireto aos elementos.
- Necessidade de percorrer a lista para acessar determinado elemento.



Considerando suas vantagens e desvantagens, quando devo utilizar uma **lista dinâmica duplamente encadeada**?

Em geral, usamos esse tipo de lista na seguinte situações:

- Não há necessidade de garantir um espaço mínimo para a execução da aplicação.
- Inserção e remoção em lista ordenada são as operações mais frequentes.
- Tamanho máximo da lista não é definido.
- Necessidade de acessar a informação de um elemento antecessor.

### 5.7.1 Definindo o tipo lista dinâmica duplamente encadeada

Antes de começar a implementar a nossa lista, é preciso definir o tipo de dado que será armazenado nela. Uma lista pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa lista. E, trabalhando com alocação dinâmica da lista, este tipo será um **ponteiro para ponteiro** da estrutura que define a lista. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **ListaDinEncadDupla.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **ListaDinEncadDupla.h**, ilustrado na Figura 5.98. Por se tratar de uma lista dinâmica duplamente encadeada, temos que estabelecer:

- O tipo de dado que será armazenado na lista, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para o **ponteiro** do tipo lista (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma lista.
- As funções disponíveis para trabalhar com essa lista em especial (linhas 8-20) e que serão implementadas no arquivo **ListaDinEncadDupla.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da lista. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



Por que colocamos um ponteiro no comando **typedef** quando criamos um novo nome para o tipo (linha 6)?

Por estarmos trabalhando com uma lista dinâmica e encadeada, temos que fazê-lo com um ponteiro para ponteiro a fim de poder realizar modificações no início da lista. Por questões de modularização, e para manter a mesma notação utilizada pela **lista sequencial estática**, podemos esconder um dos ponteiros do usuário. Assim, utilizar uma **lista sequencial estática**, uma **lista dinâmica encadeada**, uma **lista dinâmica encadeada circular** ou uma **lista dinâmica duplamente encadeada** será indiferente para o programador, pois a sua implementação está escondida dele:

- **Lista \*li;** //Declaração de uma **lista sequencial estática** (ponteiro).
- **Lista \*li;** //Declaração de uma **lista dinâmica encadeada** (ponteiro para ponteiro).
- **Lista \*li;** //Declaração de uma **lista dinâmica encadeada circular** (ponteiro para ponteiro).
- **Lista \*li;** //Declaração de uma **lista dinâmica duplamente encadeada** (ponteiro para ponteiro).



No arquivo **ListaDinEncadDupla.c**, iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em **ListaDinEncadDupla.h**.

Basicamente, o arquivo **ListaDinEncadDupla.c** (Figura 5.98) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da lista (linhas 1-3).
- A definição do tipo que descreve cada elemento da lista, **struct elemento** (linhas 5-9).
- A definição de um novo nome para a **struct elemento** (linha 10). Isso é feito apenas para facilitar certas etapas de codificação.
- As implementações das funções definidas no arquivo **listadinencaddupla.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que a nossa estrutura **elemento** nada mais é do que uma estrutura contendo três campos:

- Um ponteiro **ant**, que indica o elemento (também do tipo **struct elemento**) anterior ao elemento atual dentro da lista.
- Um campo **dado** do tipo **struct aluno**, que é o tipo de dado a ser armazenado na lista.
- Um ponteiro **prox**, que indica o elemento (também do tipo **struct elemento**) seguinte ao elemento atual dentro da lista.

Por estarem definidos dentro do arquivo .c, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **ListaDinEncadDupla.h** (linha 6), que pode somente declarar um ponteiro para ele, da seguinte forma:

**Lista \*li;**

#### Arquivo **ListaDinEncadDupla.h**

```

01 struct aluno{
02     int matricula;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 typedef struct elemento* Lista;
07
08 Lista* cria_lista();
09 void libera_lista(Lista* li);
10 int busca_lista_pos(Lista* li, int pos, struct aluno *al);
11 int busca_lista_mat(Lista* li, int mat, struct aluno *al);
12 int insere_lista_final(Lista* li, struct aluno al);
13 int insere_lista_inicio(Lista* li, struct aluno al);
14 int insere_lista_ordenada(Lista* li, struct aluno al);
15 int remove_lista(Lista* li, int mat);
16 int remove_lista_inicio(Lista* li);
17 int remove_lista_final(Lista* li);
18 int tamanho_lista(Lista* li);
19 int lista_vazia(Lista* li);
20 int lista_cheia(Lista* li);

```

#### Arquivo **ListaDinEncadDupla.c**

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ListaDinEncadDupla.h" //inclui os protótipos
04 //Definição do tipo lista
05 struct elemento{
06     struct elemento *ant;
07     struct aluno dados;
08     struct elemento *prox;
09 };
10 typedef struct elemento Elemt;

```

FIGURA 5.98

### 5.7.2 Criando e destruindo uma lista

Para utilizar uma lista em seu programa, a primeira coisa a fazer é criar uma lista vazia. Essa tarefa é executada pela função descrita na Figura 5.99. Basicamente, o que esta função faz é a alocação de uma área de memória para armazenar o endereço do início da lista (linha 2), que é um **ponteiro para ponteiro**. Esta área de memória corresponde à memória necessária para armazenar o endereço de um elemento da lista, `sizeof(Lista)` ou `sizeof(struct elemento*)`. Em seguida, a função inicializa o conteúdo desse **ponteiro para ponteiro** com a constante `NULL`. Esta constante é utilizada em uma **lista dinâmica encadeada** para indicar que não existe nenhum elemento alocado após o atual. Como o início da lista aponta para tal constante, isso significa que a lista está vazia. A Figura 5.100 indica o conteúdo do nosso ponteiro `List* li` após a chamada da função que cria a lista.



Note que não existe diferença entre criar uma **lista dinâmica encadeada** e uma **lista dinâmica duplamente encadeada**.

FIGURA 5.99

```
01 Lista* cria_lista(){
02     Lista* li = (Lista*) malloc(sizeof(Lista));
03     if(li != NULL)
04         *li = NULL;
05     return li;
06 }
```

FIGURA 5.99

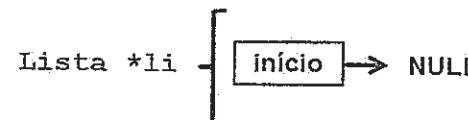


FIGURA 5.100

Destruir uma lista que utilize alocação dinâmica, e seja encadeada, não é uma tarefa tão simples quanto destruir uma **lista sequencial estática**.



Para liberar uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido nela.

O código que realiza a destruição da lista é mostrado na Figura 5.101. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso (linha 2).

Em seguida, percorremos a lista até que o conteúdo do seu início (`*li`) seja diferente de `NULL`, o final da lista. Enquanto não chegarmos ao final da lista, iremos liberar a memória do elemento que se encontra atualmente no início da lista e avançar para o próximo (linhas 5-7). Terminado o processo, liberaremos a memória alocada para o início da lista (linha 9). Esse processo é mais bem ilustrado pela Figura 5.102, que mostra a liberação de uma lista contendo dois elementos.



Note que não existe diferença entre liberar uma **lista dinâmica encadeada** e uma **lista dinâmica duplamente encadeada**. Isso ocorre porque precisamos percorrer a lista apenas em um sentido para liberá-la, sendo o outro sentido ignorado.

```
01 void libera_lista(Lista* li){
02     if(li != NULL){
03         Elemt* no;
04         while((*li) != NULL){
05             no = *li;
06             *li = (*li)->prox;
07             free(no);
08         }
09     }
10 }
```

FIGURA 5.101

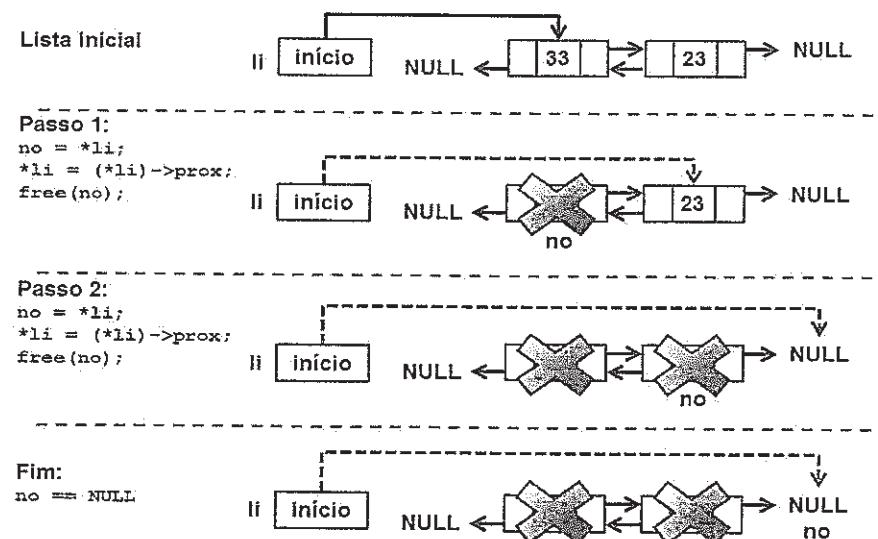


FIGURA 5.102

### 5.7.3 Informações básicas sobre a lista

As operações de inserção, remoção e busca são consideradas as principais de uma lista. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a lista. Por exemplo, não podemos remover um elemento da lista se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da lista: seu tamanho, se ela está cheia ou se ela está vazia.

#### Tamanho da lista

Saber o tamanho de uma lista dinâmica duplamente encadeada ou de uma lista dinâmica encadeada é uma tarefa idêntica.



Para saber o tamanho de uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista contando os elementos inseridos nela, até encontrar o seu final.

Não existe diferença entre saber o tamanho de uma **lista dinâmica encadeada** ou de uma **lista dinâmica duplamente encadeada**, porque precisamos percorrer a lista em apenas um sentido para contar os seus elementos, sendo o outro sentido ignorado.

O código que realiza a contagem dos elementos da lista é mostrado na Figura 5.103. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso e a lista é ou não igual a **NULL** (linha 2). Caso ela seja nula, não temos o que fazer na função e a terminamos (linha 3). Em seguida, criamos um contador iniciado em **ZERO** (linha 4) e um elemento auxiliar (**no**) apontado para o primeiro elemento da lista (linha 5). Então, percorremos a lista até que o valor de **no** seja diferente de **NULL**, o final da lista. Enquanto não chegarmos ao final da lista, iremos somar “+1” ao contador **cont** e avançar para o próximo elemento da lista (linhas 6-9). Terminado o processo, retornamos o valor da variável **cont** (linha 10). Esse processo é melhor ilustrado pela Figura 5.104, que mostra o cálculo do tamanho de uma lista contendo dois elementos.

#### Tamanho da lista:

```

01 int tamanho_lista(Lista* li){
02     if(li == NULL)
03         return 0;
04     int cont = 0;
05     Elém* no = *li;
06     while(no != NULL){
07         cont++;
08         no = no->prox;
09     }
10     return cont;
11 }
```

FIGURA 5.103

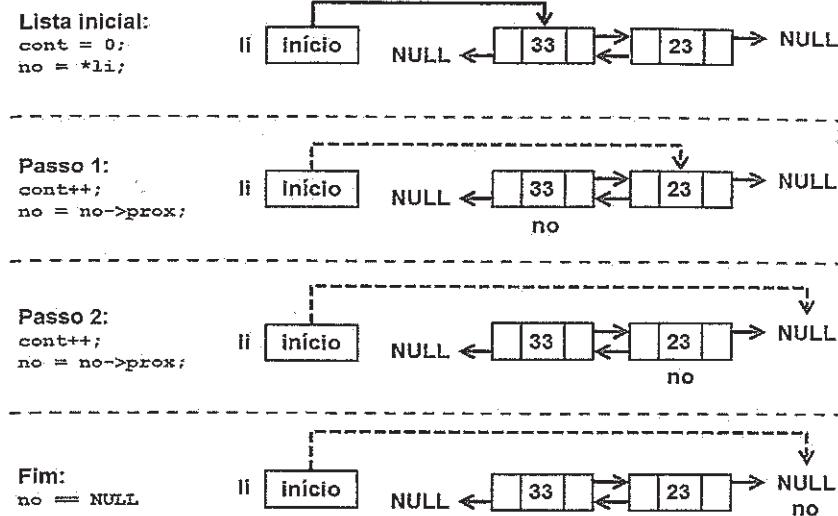


FIGURA 5.104

#### Lista cheia

Implementar uma função que retorne se uma **lista dinâmica duplamente encadeada** está cheia é uma tarefa relativamente simples.



Uma **lista dinâmica duplamente encadeada** somente será considerada cheia quando não tivermos mais memória disponível para alocar novos elementos.

A implementação da função que retorna se a lista está cheia é mostrada na Figura 5.105. Como se pode notar, a função sempre irá retornar o valor **ZERO**, indicando que a lista não está cheia.

#### Retornando se a lista está cheia

```

01 int lista_cheia(Lista* li){
02     return 0;
03 }
```

FIGURA 5.105

### Listas vazias

Implementar uma função que retorne se uma lista dinâmica duplamente encadeada está vazia é outra tarefa relativamente simples.



Uma lista dinâmica duplamente encadeada será considerada vazia sempre que o conteúdo do seu "íncio" apontar para a constante NULL.

A implementação da função que retorna se a lista está vazia é mostrada na Figura 5.106. Note que essa função, em primeiro lugar, verifica se o ponteiro `Lista* li` não é igual a `NULL`. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor `UM` para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, então é possível acessar o conteúdo do seu "íncio" (`*li`) e comparar o seu valor com a constante `NULL`, que é o valor inicial do conteúdo do "íncio" quando criamos a lista. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da lista), a função irá retornar o valor `UM` (linha 5). Caso contrário, irá retornar o valor `ZERO` (linha 6).

#### Retornando se a lista está vazia

```

01 int lista_vazia(Lista* li){
02     if(li == NULL)
03         return 1;
04     if(*li == NULL)
05         return 1;
06     return 0;
07 }
```

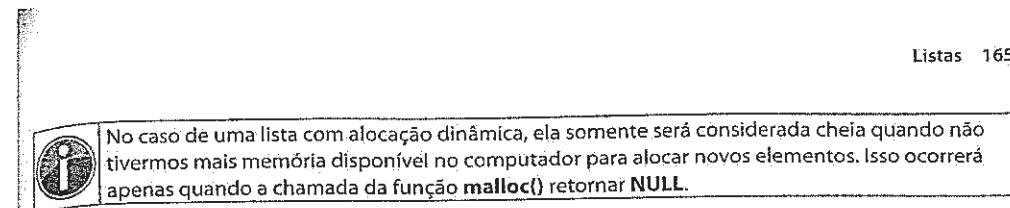
FIGURA 5.106

### 5.7.4 Inserindo um elemento na lista

#### Preparando a inserção na lista

Antes de inserir um elemento em uma lista dinâmica duplamente encadeada, algumas verificações são necessárias. Isso vale para os três tipos de inserção: no início, no final ou no meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.108, 5.110 e 5.112).

Primeiramente, a função verifica se o ponteiro `Lista* li` é igual a `NULL`. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor `ZERO` para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linhas 4 e 5). Caso a alocação de memória não seja possível, a função irá retornar o valor `ZERO` (linhas 6 e 7). Tendo a função `malloc()` retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 8).



Também existe a situação em que a inserção é feita em uma lista que está vazia, como mostrado na Figura 5.107. Neste caso, a lista, que inicialmente apontava para `NULL`, passa a apontar para o único elemento inserido até então.

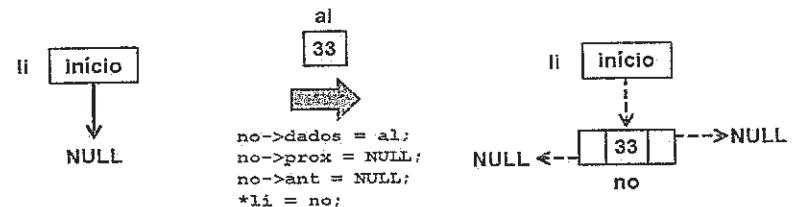


FIGURA 5.107

#### Inserindo no início da lista

Inserir um elemento no início de uma lista dinâmica duplamente encadeada é uma tarefa bastante simples.



De fato, essa inserção é semelhante à inserção no início de uma lista dinâmica encadeada. Porém, temos agora que lidar com o ponteiro que aponta para o elemento anterior da lista.

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.108. Note que as linhas 2 a 8 verificam se a inserção é possível.

Como se trata de uma inserção no início, temos que fazer nosso elemento apontar para o início da lista, `*li` (linha 9). Assim, o elemento `no` passa a ser o início da lista, enquanto o antigo início passa a ser o próximo elemento da lista. Sendo o elemento `no` o início, não existe nenhum elemento antes dele (linha 10).

Como agora existe o ponteiro para o elemento anterior e se trata de uma inserção no início, temos que considerar que a lista pode não estar vazia (linha 12). Neste caso, o elemento anterior do antigo início da lista, `(*li)->ant`, passa a ser o elemento `no` (linha 13). Note que se a lista for vazia, o campo `no->prox` já irá valer `NULL` (linha 9). Por fim, mudamos o conteúdo do "íncio" da lista (`*li`) para que ele passe a ser o nosso elemento `no` (linha 14) e retornamos o valor `UM` (linha 15), indicando sucesso na operação de inserção. Esse processo é melhor ilustrado pela Figura 5.109.

**Inserindo um elemento no início da lista**

```

01 int insere_lista_inicio(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elem* no;
05     no = (ELEM*) malloc(sizeof(ELEM));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = (*li);
10    no->ant = NULL;
11    //lista não vazia: apontar para o anterior!
12    if(*li != NULL)
13        (*li)->ant = no;
14    *li = no;
15    return 1;
16 }

```

FIGURA 5.108

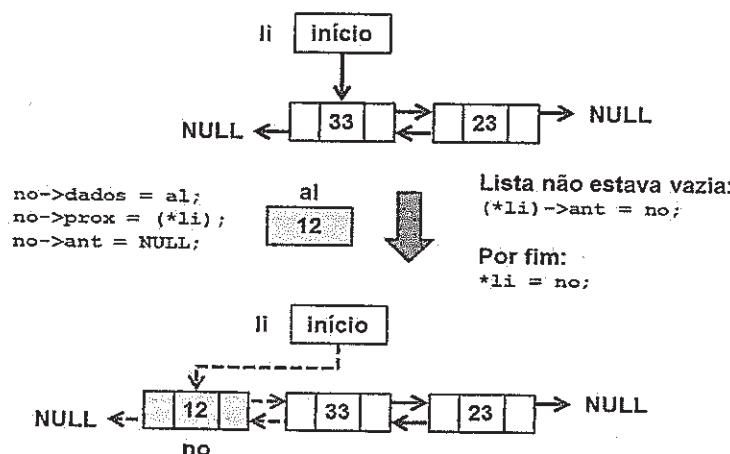


FIGURA 5.109

**Inserindo no final da lista**

Inserir um elemento no final de uma lista dinâmica duplamente encadeada é uma tarefa um tanto trabalhosa.



Como na inserção do início, a inserção no final de uma lista dinâmica duplamente encadeada não necessita que se mude o lugar dos demais elementos da lista. Porém, é preciso percorrer a lista toda para descobrir o último elemento e assim fazer a inserção após ele.

Basicamente, o que temos que fazer é alocar espaço para o novo elemento da lista, encontrar o último da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.110. Note que as linhas 2 a 8 verificam se a inserção é possível.

Como se trata de uma inserção no final, temos que considerar que a lista pode ou não estar vazia (linha 10):

- No caso de ser uma lista vazia, o elemento anterior ao elemento **no** passa a ser a constante **NULL** (linha 11), pois ele é agora o primeiro da lista, e mudamos o conteúdo do “início” da lista (**\*li**) para que ele passe a ser o nosso elemento **no** (linha 12). Esse processo é mais bem ilustrado pela Figura 5.107.
- No caso de NÃO ser uma lista vazia, temos que achar o último elemento, pois ele aponta sempre para a constante **NULL**. Assim, devemos guardar em um ponteiro auxiliar (**aux**) o endereço do primeiro elemento da lista (**\*li**) e percorrer a lista até que o elemento seguinte ao elemento atual (**aux->prox**) seja a constante **NULL** (linhas 14-17). Ao final do processo, o elemento **no** passa a ser o elemento seguinte a **aux**, enquanto **aux** passa a ser o elemento anterior a **no** (linhas 18 e 19). Esse processo é mais bem ilustrado pela Figura 5.111.

Terminado um dos dois processos de inserção, retornamos o valor **UM** (linha 21), indicando sucesso na operação de inserção.

**Inserindo um elemento no final da lista**

```

01 int insere_lista_final(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elem *no;
05     no = (ELEM*) malloc(sizeof(ELEM));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = NULL;
10    if((*li) == NULL){//lista vazia: insere inicio
11        no->ant = NULL;
12        *li = no;
13    }else{
14        Elem *aux = *li;
15        while(aux->prox != NULL){
16            aux = aux->prox;
17        }
18        aux->prox = no;
19        no->ant = aux;
20    }
21    return 1;
22 }

```

FIGURA 5.110

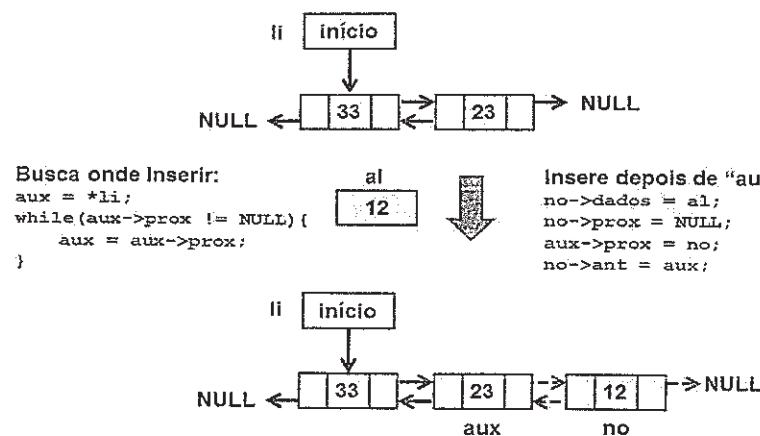


FIGURA 5.111

### Inserindo de forma ordenada na lista

Inserir um elemento de forma ordenada em uma **lista dinâmica duplamente encadeada** é uma tarefa trabalhosa.



Isso ocorre porque precisamos procurar o ponto de inserção do elemento na lista, o qual pode ser no início, no meio ou no final da lista. Felizmente, como na **lista dinâmica encadeada**, não é preciso que se mude o lugar dos demais elementos da lista.

Basicamente, o que temos que fazer é procurar em que lugar da lista será inserido o novo elemento (no caso, iremos ordenar pelo campo matrícula), alocar espaço para ele na lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.112. Note que as linhas 2 a 8 verificam se a inserção é possível.

Antes de fazer a inserção, temos que considerar que a lista pode ou não estar vazia (linha 9):

- No caso de ser uma lista vazia, mudamos o conteúdo do “**início**” da lista (**\*li**) para que ele passe a ser o nosso elemento **no**, que terá a constante **NUL** como elemento anterior e seguinte a ele, sendo o valor **UM** retornado para indicar sucesso na operação de inserção (linhas 10-13). Esse processo é mais bem ilustrado pela Figura 5.107.
- No caso de NÃO ser uma lista vazia, temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento **atual** for menor do que a matrícula do novo elemento a ser inserido (linhas 16-20). Note que, além do elemento **atual**, também armazenamos o elemento anterior a ele (**ante**), que é necessário em uma inserção no meio da lista. Ao final do processo, temos duas possibilidades de acordo com o valor de **atual**: inserção no **início** da lista (linhas 22-25) ou inserção entre os elementos **ante** e **atual** (linhas 27-31). Note que se **atual** for diferente de **NUL**, a

inserção é no meio da lista e, portanto, devemos indicar que o elemento anterior ao **atual** é o novo elemento (linhas 30 e 31). Ao final do processo, retornamos o valor **UM** para indicar sucesso na operação de inserção (linha 33). Esse processo é mais bem ilustrado pela Figura 5.113.

**Inserindo um elemento de forma ordenada na lista**

```

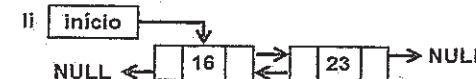
01 int insere_lista_ordenada(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elemt *no;
05     no = (Elemt*) malloc(sizeof(Elemt));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     if((*li) == NULL){//lista vazia: insere inicio
10         no->prox = NULL;
11         no->ant = NULL;
12         *li = no;
13         return 1;
14     }
15     else{
16         Elemt *ante, *atual = *li;
17         while(atual != NULL &&
18               atual->dados.matricula < al.matricula){
19             ante = atual;
20             atual = atual->prox;
21         }
22         if(atual == *li){//insere inicio
23             no->ant = NULL;
24             (*li)->ant = no;
25             no->prox = (*li);
26             *li = no;
27         }
28         else{
29             no->prox = ante->prox;
30             no->ant = ante;
31             ante->prox = no;
32             if(atual != NULL)
33                 atual->ant = no;
34         }
35     }
36     return 1;
37 }
  
```

FIGURA 5.112

```

Lista inicial atual = *li;
Busca onde while(atual!=NULL && atual->dados.matricula < al.matricula) {
Inserir:     ante = atual;
            atual = atual->prox;
}

```

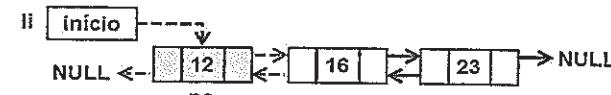


Inserir no início:

```

no->ant = NULL;
(*li)->ant = no;
no->prox = (*li);
*li = no;

```

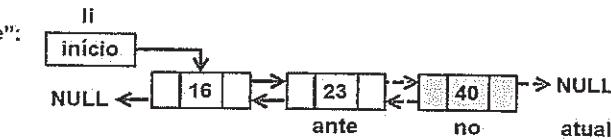


Inserir depois de "ante":

```

no->prox=ante->prox;
no->ant = ante;
ante->prox = no;

```



Não é final da lista:  
atual->ant = no;

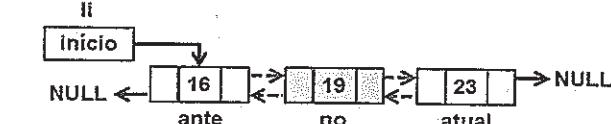


FIGURA 5.113

## 5.7.5 Removendo um elemento da lista

### Preparando a remoção da lista

Antes de remover um elemento de uma **lista dinâmica duplamente encadeada**, algumas verificações são necessárias. Isso vale para os três tipos de remoção: do início, do final ou do meio da lista, como mostrados nas suas respectivas implementações (Figuras 5.115, 5.117 e 5.119).

Primeiramente, verificamos se o ponteiro **Lista\* li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma lista inválida (linha 3). Porém, se a lista foi criada com sucesso, precisamos verificar se ela não está vazia, isto é, se existem elementos dentro dela. Caso a lista esteja vazia, a função irá retornar o valor **ZERO** (linhas 4 e 5).



No caso de uma lista com alocação dinâmica, ela somente será considerada vazia quando o seu início apontar para a constante **NULL**.

Também existe a situação em que a remoção é feita em uma lista que possui um único elemento, como mostrado na Figura 5.114. Neste caso, a lista fica vazia após a remoção.

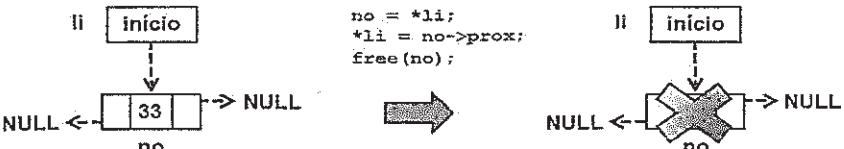


FIGURA 5.114

### Removendo do início da lista

Remover um elemento do início de uma **lista dinâmica duplamente encadeada** é uma tarefa bastante simples.



De fato, essa remoção é semelhante à remoção do início de uma **lista dinâmica encadeada**. Porém, temos agora que lidar com o ponteiro que aponta para o elemento anterior da lista.

Basicamente, o que temos que fazer é verificar se a lista não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.115. Note que as linhas 2 a 5 verificam se a remoção é possível.

Primeiro, criamos um elemento auxiliar (**no**) para armazenar o início da lista e fazemos o início da lista (**\*li**) apontar para o elemento seguinte a ele (linhas 7 e 8). Como se trata de uma remoção do início, verificamos se existe um elemento seguinte ao início da lista (linha 9). Caso esse elemento exista, ele será o novo início da lista e, portanto, seu anterior deverá ser a constante **NULL** (linha 10). Do contrário, a lista ficará vazia.

### Removendo um elemento do início da lista

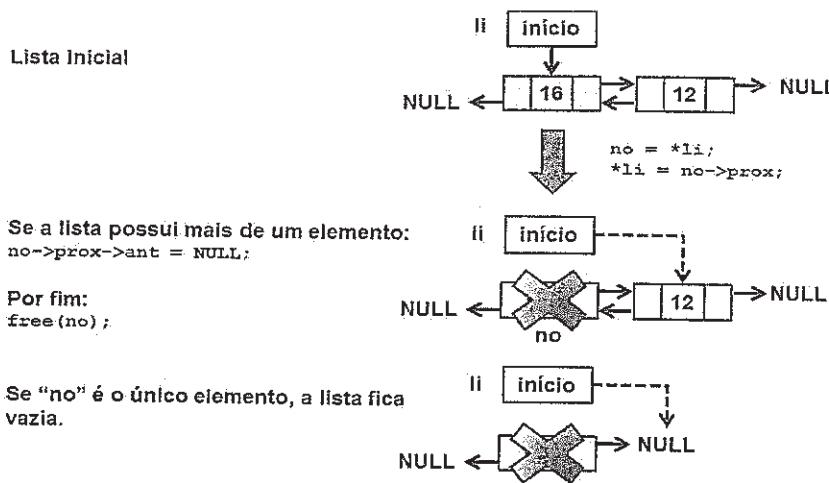
```

01 int remove_lista_inicio(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     Elemt *no = *li;
08     *li = no->prox;
09     if(no->prox != NULL)
10         no->prox->ant = NULL;
11
12     free(no);
13     return 1;
14 }

```

FIGURA 5.115

Por fim, temos que liberar a memória associada ao antigo “íncio” da lista (`no`) e retornamos o valor UM (linhas 12 e 13), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.116.



### Removendo do final da lista

Remover um elemento do final de uma **lista dinâmica encadeada** é uma tarefa um tanto trabalhosa.



Como na remoção do início, a remoção no final de uma **lista dinâmica duplamente encadeada** não necessita que se mude o lugar dos demais elementos da lista. Porém, é preciso percorrer a lista toda para descobrir o último elemento e assim removê-lo.

Basicamente, o que temos que fazer é verificar se a lista não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.117. Note que as linhas 2 a 5 verificam se a remoção é possível.

Como se trata de uma remoção do final, temos que achar o último elemento da lista, ou seja, aquele que aponta para a constante `NULL`. Assim, devemos guardar em um ponteiro auxiliar (`no`) o endereço do primeiro elemento da lista (`*li`) e percorrer a lista até que o elemento seguinte ao elemento atual (`no->prox`) seja a constante `NULL` (linhas 7-9). Ao final do processo, temos que considerar que o último elemento da lista talvez seja o primeiro e único (linha 11):

- Se `no` também é o íncio da lista, então o íncio da lista deverá apontar para a posição seguinte a ele, que, neste caso, é a constante `NULL`, ficando assim a lista vazia (linha 12);

- Caso contrário, o elemento anterior ao final da lista (`no->ant`) irá ter como próximo elemento (`no->ant->prox`) a constante `NULL` (linha 14).

Terminado um dos dois processos de remoção, temos que liberar a memória associada ao antigo “final” da lista (`no`) e retornamos o valor UM (linhas 16 e 17), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.118.

### Removendo um elemento do final da lista

```

01 int remove_lista_final(Lista* li){
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06
07     ELEM *no = *li;
08     while(no->prox != NULL)
09         no = no->prox;
10
11    if(no->ant == NULL)//remover o primeiro e único
12        *li = no->prox;
13    else
14        no->ant->prox = NULL;
15
16    free(no);
17    return 1;
18 }

```

FIGURA 5.117

### Procura último elemento da lista:

```

no = *li;
while(no->prox != NULL)
    no = no->prox;

```

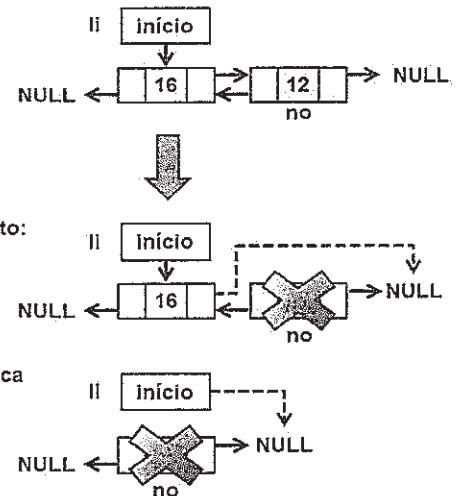


FIGURA 5.118

### Removendo um elemento específico da lista

Remover um elemento específico de uma **Lista Dinâmica Duplamente Encadeada** é similar à remoção de uma **lista dinâmica encadeada**. A diferença é que temos agora que lidar com o ponteiro que aponta para o elemento anterior da lista. Assim, o que temos que fazer nessa remoção é procurar o elemento a ser removido na lista e mudar os valores de alguns ponteiros, como mostramos a sua implementação na Figura 5.119. Note que as linhas 2 a 5 verificam se a remoção é possível.

No nosso exemplo, vamos remover um elemento de acordo com o campo matrícula. Assim, temos que percorrer a lista enquanto não chegarmos ao seu final, e enquanto o campo matrícula do elemento **no** for diferente do valor de matrícula procurado (linhas 6-9). Note que não precisamos mais armazenar o elemento anterior, como era feito com a **lista dinâmica encadeada**. Terminado o processo de busca, verificamos se estamos no final da lista ou não (linha 10). Em caso afirmativo, o elemento não existe na lista e a remoção não é possível (linha 11). Caso contrário, a remoção pode ser no início da lista e, portanto, temos que mudar o valor de **li** (linhas 13-14), ou a remoção é no meio ou no final da lista (linha 16). Neste caso, basta apenas fazer o elemento anterior a **no** apontar para o elemento seguinte a **no**. Temos também que verificar se o elemento removido não é o último. Assim, fazemos com que o elemento seguinte a **no** tenha como elemento anterior o elemento anterior a **no** (linhas 18-19).

Ao final do processo de remoção, temos que liberar a memória associada ao elemento **no** e retornarmos o valor UM (linhas 21 e 22), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.120.

#### Removendo um elemento específico da lista

```

01 int remove_lista(Lista* li, int mat) {
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06     Elemt *no = *li;
07     while(no != NULL && no->dados.matricula != mat) {
08         no = no->prox;
09     }
10     if(no == NULL)//não encontrado
11         return 0;
12
13     if(no->ant == NULL)//remover o primeiro
14         *li = no->prox;
15     else
16         no->ant->prox = no->prox;
17
18     if(no->prox != NULL)//não é o último
19         no->prox->ant = no->ant;
20
21     free(no);
22     return 1;
23 }
```

FIGURA 5.119

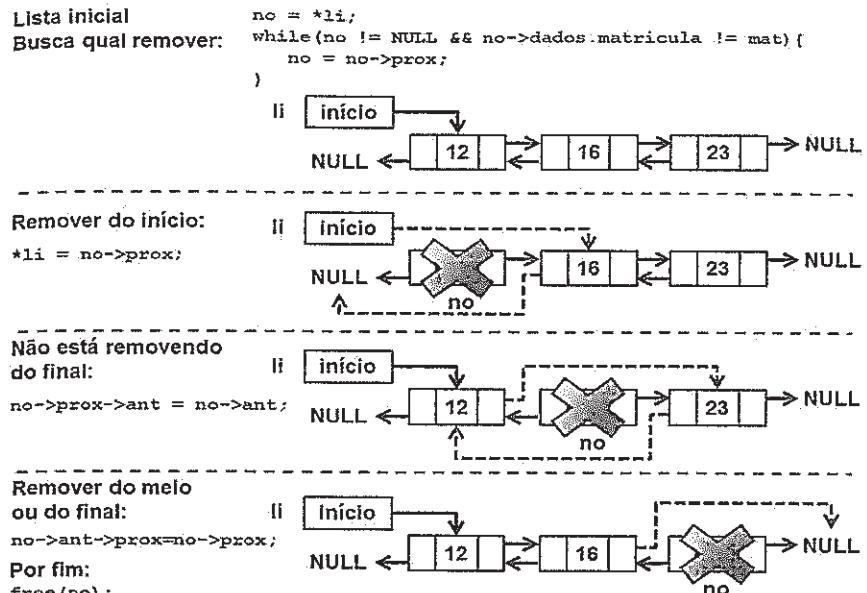


FIGURA 5.120

### 5.7.6 Busca por um elemento da lista

A operação de busca consiste em recuperar as informações contidas em determinado elemento da lista.



De modo geral, a operação de busca pode ser feita de diversas maneiras, dependendo da necessidade da aplicação.

Por exemplo, podemos buscar as informações:

- Do segundo elemento da lista: busca por posição.
- Do elemento que possui um pedaço de informação conhecida: busca por conteúdo.



Em uma lista que utilize alocação dinâmica, e seja encadeada, a busca sempre envolve a necessidade de percorrer a lista.

A seguir, veremos como esses dois tipos de busca funcionam.

### Busca por posição na lista

O código que realiza a busca de um elemento por sua posição em uma **lista dinâmica duplamente encadeada** é mostrado na Figura 5.121. Em primeiro lugar, a função verifica se a busca é válida. Para tanto, duas condições são verificadas: se o ponteiro **Lista\*** *li* é igual a **NULL** e se a posição buscada (*pos*) é um valor negativo. Se alguma dessas condições for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Caso contrário, criamos um elemento auxiliar (*no*) apontando para o primeiro elemento da lista (linha 4) e um contador (*i*) iniciado em **UM** (linha 5). Então, percorremos a lista enquanto o valor de *no* for diferente de **NULL** e o valor do contador for menor do que a posição desejada (linhas 6-9). Terminado o laço, verificamos se estamos no final da lista (linha 10). Se essa condição for verdadeira, o elemento não foi encontrado na lista. Do contrário, a posição atual (*no*) é copiada para o conteúdo do ponteiro passado por referência (*al*) para a função (linha 13). A Figura 5.122 ilustra os principais pontos dessa busca.

#### Busca um elemento por posição

```

01 int busca_lista_pos(Lista* li, int pos, struct aluno *al) {
02     if(li == NULL || pos <= 0)
03         return 0;
04     Elem *no = *li;
05     int i = 1;
06     while(no != NULL && i < pos) {
07         no = no->prox;
08         i++;
09     }
10     if(no == NULL)
11         return 0;
12     else{
13         *al = no->dados;
14         return 1;
15     }
16 }
```

FIGURA 5.121

#### Busca pela posição do elemento

```

no = *li;
int i = 1;
while(no != NULL && i < pos) {
    no = no->prox;
    i++;
}
Verifica se o elemento foi
encontrado e o retorna
if(no == NULL) return 0;
else{
    *al = no->dados;
    return 1;
}
```

FIGURA 5.122

### Busca por conteúdo na lista

O código que realiza a busca de um elemento por sua posição em uma **lista dinâmica duplamente encadeada** é mostrado na Figura 5.123. Neste caso, estamos procurando um aluno pelo seu número de matrícula. Como se pode notar, o código é praticamente igual ao da busca por posição mostrado na Figura 5.121. A única diferença é que agora não percorremos mais a lista comparando o valor da posição, mas sim o valor da matrícula (linha 5). Além disso, não precisamos mais de um contador ao percorrer a lista. A Figura 5.124 ilustra os principais pontos dessa busca.

#### Busca um elemento por conteúdo

```

01 int busca_lista_mat(Lista* li, int mat, struct aluno *al){
02     if(li == NULL)
03         return 0;
04     Elem *no = *li;
05     while(no != NULL && no->dados.matricula != mat) {
06         no = no->prox;
07     }
08     if(no == NULL)
09         return 0;
10     else{
11         *al = no->dados;
12         return 1;
13     }
14 }
```

FIGURA 5.123

#### Busca pelo conteúdo do elemento

```

no = *li;
while(no != NULL && no->dados.matricula != mat)
    no = no->prox;
```

```

Verifica se o elemento foi
encontrado e o retorna
if(no == NULL)
    return 0;
else{
    *al = no->dados;
    return 1;
}
```

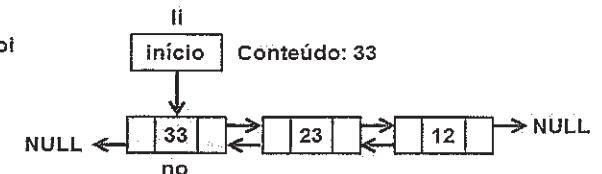


FIGURA 5.124

### 5.7.7 Análise de complexidade

Um aspecto importante quando manipulamos listas tem relação com os custos das suas operações. Na sequência, são mostradas as complexidades computacionais das principais operações em uma **lista dinâmica duplamente encadeada** contendo *N* elementos:

- Inserção no início:** essa operação envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- Inserção no final:** é preciso percorrer toda a lista até alcançar o seu final. Desse modo, a sua complexidade é  $O(N)$ .
- Inserção ordenada:** neste caso, é preciso procurar o ponto de inserção, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (inserção no final).
- Remoção do início:** é uma operação que envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- Remoção do final:** é preciso percorrer toda a lista até alcançar o seu final. Desse modo, a sua complexidade é  $O(N)$ .
- Remoção de um elemento específico:** essa operação envolve a busca pelo elemento a ser removido, que pode estar no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (remoção do final).
- Consulta:** a operação de consulta envolve a busca de um elemento, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (último elemento).

## 5.8 LISTA DINÂMICA ENCADEADA COM NÓ DESCRIPTOR

Sempre que trabalhamos com uma lista dinâmica e encadeada (seja ela simples, circular ou duplamente encadeada), utilizamos um **ponteiro para ponteiro** para guardar o início da lista. Optamos por essa abordagem porque, utilizando um **ponteiro para ponteiro** para representar o início da lista, fica fácil mudar quem é o primeiro elemento da lista alterando apenas o **conteúdo do ponteiro para ponteiro**. Outra abordagem que substitui o **ponteiro para ponteiro** é utilizar um **nó descriptor**.



Um **nó descriptor** é uma estrutura especial que possui um campo que aponta para o primeiro elemento da lista (seja ela simples, circular ou duplamente encadeada), além de armazenar outras informações sobre a lista que o programador julgue necessário.

Podemos entender que o **nó descriptor** é um elemento especial da lista. Dentro dele, armazenamos qualquer informação que julgarmos necessária. De modo geral, optamos por armazenar dentro dessa estrutura informações que facilitem a manipulação da lista, como o seu início, o seu final e a quantidade de elementos, como mostrado na Figura 5.125.

Considerando uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo lista (ela é um **tipo opaco**), como ilustrado na Figura 5.125. Isso impede o usuário de saber como foi realmente implementada a lista, e limita o seu acesso apenas às funções que manipulam a lista. Além disso, essa implementação em módulos impede o usuário de saber como a lista foi implementada. Note que qualquer uma das listas definidas até o momento é sempre declarada como sendo do tipo **Lista \***.

Lista \*li

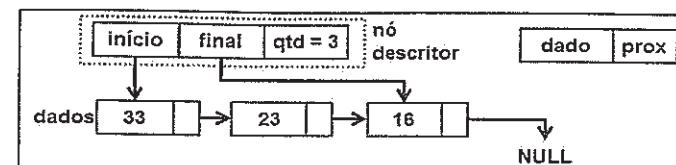


FIGURA 5.125

### 5.8.1 Definindo o tipo lista com nó descriptor

Antes de começar a implementar a nossa lista com **nó descriptor**, é preciso definir o tipo de dado que será armazenado nela. Uma lista pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa lista, neste caso, nosso **nó descriptor**. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **ListaDinEncadDesc.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **ListaDinEncadDesc.h**, ilustrado na Figura 5.126. Por se tratar de uma lista dinâmica encadeada, temos que estabelecer:

- O tipo de dado que será armazenado na lista, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para a **struct descriptor** (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma lista.
- As funções disponíveis para trabalhar com essa lista em especial (linhas 8-18) e que serão implementadas no arquivo **ListaDinEncadDesc.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da lista. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



No arquivo **ListaDinEncadDesc.c**, iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **ListaDinEncadDesc.h**.

Basicamente, o arquivo **ListaDinEncadDesc.c** (Figura 5.126) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da lista (linhas 1-3).
- A definição do tipo que descreve cada elemento da lista, **struct elemento** (linhas 5-8).
- A definição de um novo nome para a **struct elemento** (linha 9). Isso é feito apenas para facilitar certas etapas de codificação.

```
Arquivo ListaDinEncadDesc.h
01 struct aluno{
02     int matricula;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 typedef struct descriptor Lista;
07
08 Lista* cria_lista();
09 void libera_lista(Lista* li);
10 int insere_lista_final(Lista* li, struct aluno al);
11 int insere_lista_inicio(Lista* li, struct aluno al);
12 int remove_lista_inicio(Lista* li);
13 int remove_lista_final(Lista* li);
14 int tamanho_lista(Lista* li);
15 int lista_vazia(Lista* li);
16 int lista_cheia(Lista* li);
17 int busca_lista_mat(Lista* li, int mat, struct aluno *al);
18 int busca_lista_pos(Lista* li, int pos, struct aluno *al);
```

```
Arquivo ListaDinEncadDesc.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ListaDinEncadDesc.h" //inclui os protótipos
04 //Definição do tipo lista
05 struct elemento{
06     struct aluno dados;
07     struct elemento *prox;
08 };
09 typedef struct elemento Elem;
10
11 //Definição do Nô Descritor
12 struct descriptor{
13     struct elemento *inicio;
14     struct elemento *final;
15     int tamanho;
16 };
```

FIGURA 5.126

- A definição do tipo que descreve o **nô descritor**, **struct descriptor** (linhas 12-16).
- As implementações das funções definidas no arquivo **ListaDinEncadDesc.c**. As implementações dessas funções serão vistas nas seções seguintes.

Note que a **struct elemento** nada mais é do que uma estrutura contendo dois campos:

- Um ponteiro **prox**, que indica o próximo elemento (também do tipo **struct elemento**) dentro da lista. Isso porque estamos definindo uma **lista dinâmica encadeada** e não uma **lista dinâmica duplamente encadeada**.
- Um campo **dado** do tipo **struct aluno**, que é o tipo de dado a ser armazenado na lista.

Perceba também que a **struct descriptor** nada mais é do que uma estrutura contendo três campos:

- Um ponteiro **inicio**, que indica o primeiro elemento da lista (do tipo **struct elemento**).
- Um ponteiro **final**, que indica o último elemento da lista (também do tipo **struct elemento**).
- Um campo **tamanho** do tipo **int**, que armazena o número de elementos dentro da lista.

Por estarem definidos dentro do arquivo .c, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **ListaDinEncadDesc.h** (linha 6), que pode somente declarar um ponteiro para ele, da seguinte forma:

Lista \*li;

### 5.8.2 Criando e destruindo uma lista

Para utilizar uma lista com nó descritor em seu programa, a primeira coisa a fazer é criar uma lista vazia. Essa tarefa é executada pela função descrita na Figura 5.127. Basicamente, o que esta função faz é a alocação de uma área de memória para a lista (linha 2). Esta área de memória corresponde à memória necessária para armazenar a estrutura que define a lista, **struct descriptor**. Em seguida, a função inicializa os três campos da lista, como descrito a seguir:

- **inicio** (que aponta para o elemento que está no início da lista) recebe **NULL**.
- **final** (que aponta para o elemento que está no final da lista) recebe **NULL**.
- **qtd** (que indica a quantidade de elementos na lista) recebe **ZERO** (ou seja, nenhum elemento na lista).

A Figura 5.128 indica o conteúdo do nosso ponteiro **Lista\* li** após a chamada da função que cria a lista.

```
Criando uma lista
01 Lista* cria_lista(){
02     Lista* li = (Lista*) malloc(sizeof(Lista));
03     if(li != NULL){
04         li->inicio = NULL;
05         li->final = NULL;
06         li->tamanho = 0;
07     }
08     return li;
09 }
```

FIGURA 5.127

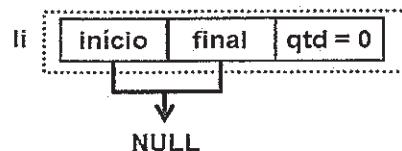


FIGURA 5.128

Destruir uma lista dinâmica que utilize nó descritor é uma tarefa semelhante a destruir uma lista dinâmica encadeada.



Para liberar uma lista dinâmica que utilize nó descritor, é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido nela.

O código que realiza a destruição da lista é mostrado na Figura 5.129. Inicialmente, verificamos se a lista é válida, ou seja, se a tarefa de criação da lista foi realizada com sucesso (linha 2). Em seguida, percorremos a lista até que o conteúdo do seu início seja diferente de NULL, o final da lista (linha 4). Enquanto não chegarmos ao final da lista, iremos liberar a memória do elemento que se encontra atualmente no início da lista e avançar para o próximo (linhas 5-7). Terminado o processo, liberaremos a memória alocada para o nó descritor que representa o início da lista (linha 9). Esse processo é mais bem ilustrado pela Figura 5.130, que mostra a liberação de uma lista contendo dois elementos.

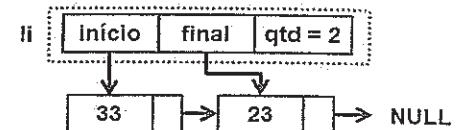
#### Destruindo uma lista

```

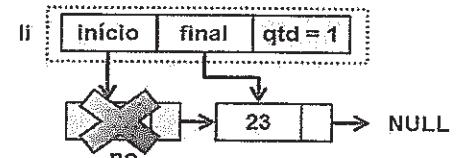
01 void libera_lista(Lista* li){
02     if(li != NULL){
03         ELEM* no;
04         while((li->inicio) != NULL){
05             no = li->inicio;
06             li->inicio = li->inicio->prox;
07             free(no);
08         }
09         free(li);
10     }
11 }
```

FIGURA 5.129

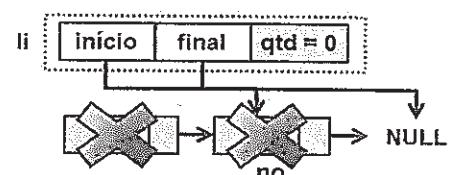
#### Lista inicial



**Passo 1:**  
no = li->inicio;  
li->inicio = li->inicio->prox;  
free(no);



**Passo 2:**  
no = li->inicio;  
li->inicio = li->inicio->prox;  
free(no);



**Fim:**  
(li->inicio) = NULL  
free(li)

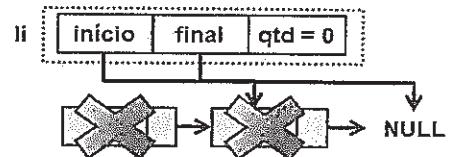


FIGURA 5.130

#### 5.8.3 Informações básicas sobre a lista

Saber o tamanho de uma lista dinâmica com nó descritor é uma tarefa relativamente simples. Isso ocorre porque seu **nó descritor** possui um campo inteiro **qtd** que indica a quantidade de elementos inseridos na lista, como mostra a Figura 5.131.



Basicamente, retornar o tamanho de uma **lista dinâmica com nó descritor** consiste em retornar o valor do seu campo **qtd**.

A implementação da função que retorna o tamanho da lista é mostrada na Figura 5.132. Note que essa função, em primeiro lugar, verifica se o ponteiro **Lista\*** **li** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da lista e, neste caso, não teríamos uma lista válida para trabalhar. Porém, se a lista foi criada com sucesso, então é possível acessar o seu campo **qtd** e retornar o seu valor, que nada mais é do que o tamanho da lista.

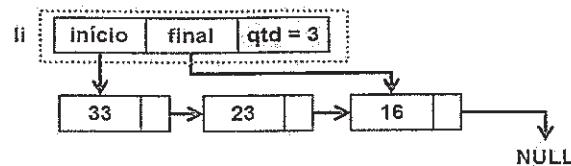


FIGURA 5.131

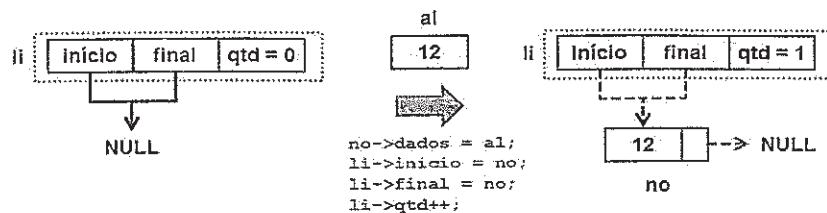


FIGURA 5.133

#### Inserindo no início da lista

Inserir um elemento no início de uma **lista dinâmica com nó descritor** é muito parecido com inserir um elemento em uma **lista dinâmica encadeada**. Basicamente, temos que alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.134. Note que as linhas 2 a 8 verificam se a inserção é possível.

Como se trata de uma inserção no início, temos que fazer nosso elemento apontar para o início da lista, **li->inicio** (linha 9). Caso o valor do início seja **NULL** (linha 10), trata-se da inserção do primeiro elemento da lista e devemos definir-o também como o último (linha 11), como mostrado na Figura 5.133. Em seguida, mudamos o conteúdo do “início” da lista (**li->inicio**) para que ele passe a ser o nosso elemento **no**, incrementamos o valor do tamanho da lista e retornamos o valor **UM** (linhas 12-14), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 5.135.

**Função de lista**

```

01 int tamanho_lista(Lista* li){
02     if(li == NULL)
03         return 0;
04     return li->tamanho;
05 }

```

FIGURA 5.132

**Informações**  
As operações de lista cheia e lista vazia em uma **lista dinâmica com nó descritor** são praticamente iguais às de uma **lista dinâmica encadeada**.

#### 5.8.4 Inserindo um elemento na lista

A inserção em uma **lista dinâmica com nó descritor** se assemelha muito à inserção em uma **lista dinâmica encadeada**.

**Informações**  
Como na **lista dinâmica encadeada**, é preciso verificar se a lista é válida, se está vazia ou cheia. Essas operações não apresentam grandes diferenças, pelo fato de estarmos agora em uma **lista dinâmica com nó descritor**.

A maior diferença é na inserção de um elemento em uma lista vazia, como mostrado na Figura 5.133. Como estamos usando um nó descritor para representar a lista, temos que considerar os campos que apontam para o primeiro e o último elemento da lista quando inserirmos o primeiro elemento. Neste caso, o primeiro elemento também é o último.

**Inserindo um elemento no início da lista**

```

01 int insere_lista_inicio(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elemt* no;
05     no = (Elemt*) malloc(sizeof(Elemt));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = li->inicio;
10    if(li->inicio == NULL)
11        li->final = no;
12    li->inicio = no;
13    li->tamanho++;
14    return 1;
15 }

```

FIGURA 5.134

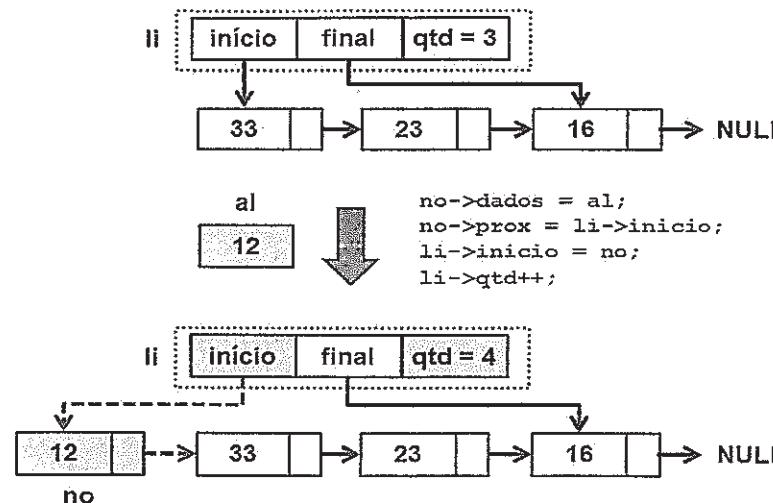


FIGURA 5.135

### Inserindo no final da lista

Inserir um elemento no final de uma lista dinâmica com nó descritor é muito parecido com inserir um elemento no início de uma lista dinâmica encadeada.



Diferente da inserção no final de uma lista dinâmica encadeada, não precisamos percorrer a lista até chegar ao seu final, pois temos um ponteiro nó descritor para isso.

Basicamente, temos que alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 5.136. Note que as linhas 2 a 8 verificam se a inserção é possível.

Como se trata de uma inserção no final, o elemento a ser inserido obrigatoriamente irá apontar para a constante `NULL` (linha 9). Também temos que considerar que a lista pode ou não estar vazia (linha 10):

- No caso de ser uma lista vazia, mudamos o conteúdo do “`íncio`” da lista (`li->íncio`) para que ele passe a ser o nosso elemento `no` (linha 11).
- No caso de NÃO ser uma lista vazia, o elemento do final da lista deverá apontar para o novo elemento (linha 13).

Em seguida, mudamos o conteúdo do “`final`” da lista (`li->final`) para que ele passe a ser o nosso elemento `no`, incrementamos o valor do tamanho da lista e retornamos o valor UM (linhas 15-17), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 5.137.

```

Inserindo um elemento no final da lista
01 int insere_lista_final(Lista* li, struct aluno al){
02     if(li == NULL)
03         return 0;
04     Elemt *no;
05     no = (Elemt*) malloc(sizeof(Elemt));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = NULL;
10    if(li->inicio == NULL)//lista vazia: insere inicio
11        li->inicio = no;
12    else
13        li->final->prox = no;
14
15    li->final = no;
16    li->tamanho++;
17    return 1;
18 }

```

FIGURA 5.136

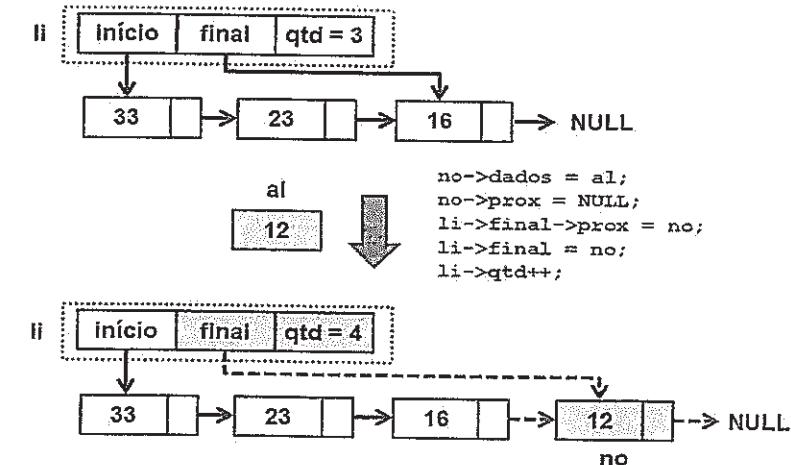


FIGURA 5.137

### 5.8.5 Removendo um elemento da lista

A remoção de um elemento de uma lista dinâmica com nó descritor não difere muito da remoção na lista dinâmica encadeada. O uso de um nó descritor não facilita muito as coisas e, portanto, não representa um ganho de performance. A maior diferença está na remoção

do último elemento da lista, como mostrado na Figura 5.138. Como estamos usando um nó descritor para representar a lista, temos que considerar os campos que apontam para o primeiro e o último elemento quando removemos o único elemento da lista. Neste caso, ambos os ponteiros terão que apontar para **NULL**.

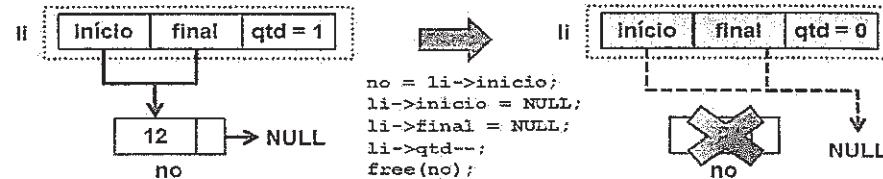


FIGURA 5.138

A Figura 5.139 mostra a implementação da remoção do início da lista. Como na remoção na **lista dinâmica encadeada**, aqui também temos que verificar se a lista não está vazia e mudar os valores de alguns ponteiros. Note que as linhas 2 a 5 verificam se a remoção é possível.

Como se trata de uma remoção do início, temos que fazer o **início** da lista (**li->inicio**) apontar para o elemento seguinte a ele (linhas 7 e 8). Em seguida, temos que liberar a memória associada ao antigo “**início**” da lista (**no**) (linha 9). Como estamos trabalhando com um nó descritor, temos que considerar que a lista pode ficar vazia após a remoção. Neste caso, é necessário mudar o valor do seu **final** (linhas 10-11). Por fim, diminuímos o valor do tamanho da lista e retornamos o valor **UM** (linhas 12-13), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 5.140.

```

Removendo um elemento do início da lista
01 int remove_lista_inicio(Lista* li) {
02     if(li == NULL)
03         return 0;
04     if(li->inicio == NULL)//lista vazia
05         return 0;
06
07     Elemt *no = li->inicio;
08     li->inicio = no->prox;
09     free(no);
10     if(li->inicio == NULL)
11         li->final = NULL;
12     li->tamanho--;
13     return 1;
14 }
  
```

FIGURA 5.139

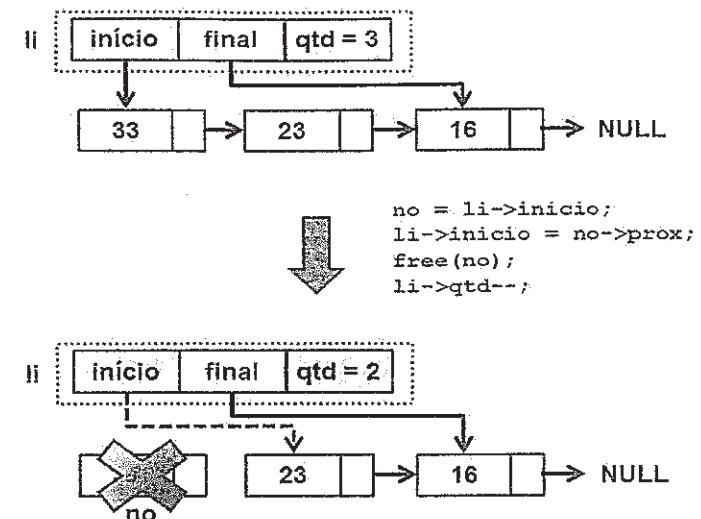


FIGURA 5.140

### 5.8.6 Análise de complexidade

Um aspecto importante quando manipulamos listas tem relação com os custos das suas operações. Na sequência, são mostradas as complexidades computacionais das principais operações em uma **lista dinâmica com nó descritor** contendo  $N$  elementos:

- Inserção no início:** essa operação envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- Inserção no final:** essa operação envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- Inserção ordenada:** neste caso, é preciso procurar o ponto de inserção, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (inserção antes do último elemento).
- Remoção do início:** é uma operação que envolve apenas a manipulação de alguns ponteiros, de modo que a sua complexidade é  $O(1)$ .
- Remoção do final:** é preciso percorrer toda a lista até alcançar o anterior ao seu final. Desse modo, a sua complexidade é  $O(N)$ .
- Remoção de um elemento específico:** essa operação envolve a busca pelo elemento a ser removido, que pode estar no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (remoção do final).
- Consulta:** a operação de consulta envolve a busca de um elemento, que pode ser no início, no meio ou no final. Assim, no pior caso, a sua complexidade é  $O(N)$  (último elemento).

## 5.9 EXERCÍCIOS

- 1) Explique o que é a alocação sequencial de memória para um conjunto de elementos.
- 2) Descreva a diferença entre alocação sequencial e alocação encadeada.
- 3) Enumere as vantagens e desvantagens de se utilizar alocação encadeada para um conjunto de elementos.
- 4) Descreva a diferença entre alocação estática e alocação dinâmica.
- 5) Escreva uma função que receba duas listas e retorne uma terceira contendo as duas primeiras concatenadas. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).
- 6) Faça uma função para remover os  $n$  primeiros elementos de uma lista. A função deve retornar se a operação foi possível ou não. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).
- 7) Faça uma função para remover os  $n$  últimos elementos de uma lista. A função deve retornar se a operação foi possível ou não. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).
- 8) Dada uma lista que armazena a *struct* *produto*, escreva a função que busca o produto de menor preço. Faça a função para a lista sequencial estática e dinâmica encadeada.

```
struct produto{
    int codido;
    char nome[30];
    float preco;
    int qtd;
};
```

- 9) Escreva uma função que receba a posição de dois elementos da lista e os troque de lugar. A função deve retornar se a operação foi possível ou não. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).
- 10) Dada uma lista contendo números inteiros positivos, escreva uma função que calcule:
  - Quantos números pares existem.
  - A média da lista.
  - O maior valor.
  - O menor valor.
  - A posição do maior valor.
  - A posição do menor valor.
  - O número de nós com valor maior do que  $x$ .
  - A soma da lista.
  - O número de nós da lista que possuem um número primo.
- 11) Escreva uma função que, dada uma lista L1, crie uma cópia dela em L2.

- 12) Escreva uma função que, dada uma lista L1, crie uma cópia dela em L2 eliminando os valores repetidos.
- 13) Escreva uma função que, dada uma lista L1, inverta a lista e a armazene em L2.
- 14) Dadas duas listas ordenadas, L1 e L2, escreva uma função que faça o *merge* de ambas em uma terceira lista.
- 15) Escreva uma função para verificar se uma lista de inteiros está ordenada ou não. A ordenação pode ser crescente ou decrescente.
- 16) Escreva uma função para verificar se duas listas de inteiros são iguais.
- 17) Dadas duas listas ordenadas, L1 e L2, escreva uma função que faça a UNIÃO de ambas em uma terceira lista.
- 18) Dadas duas listas ordenadas, L1 e L2, escreva uma função que faça a INTERSEÇÃO de ambas em uma terceira lista.
- 19) Dada uma lista contendo números inteiros positivos, implemente uma função que receba como parâmetro uma lista e um valor  $n$ . Em seguida, a função divide a lista em duas, de modo que a segunda lista contenha os elementos que vêm depois de  $n$  na lista. A função deverá retornar a segunda lista.
- 20) Considere uma lista dinâmica contendo elementos repetidos. Escreva uma função para eliminar estes elementos. Faça isso para todos os tipos de lista dinâmica (encadeada, duplamente encadeada e circular).
- 21) Considere uma lista dinâmica contendo elementos repetidos. Escreva uma função para eliminar todas as ocorrências de um valor  $x$ . Faça isso para todos os tipos de lista dinâmica (encadeada, duplamente encadeada e circular).
- 22) Escreva uma função recursiva para calcular o tamanho de uma lista dinâmica encadeada.
- 23) Escreva uma função recursiva para imprimir uma lista dinâmica encadeada.
- 24) Modifique o algoritmo de ordenação *selection sort* para que ele possa ser utilizado para armazenar uma lista dinâmica encadeada contendo números inteiros. Faça o mesmo para uma lista duplamente encadeada.
- 25) Implemente o TAD lista circular duplamente encadeada. Implemente as funções necessárias para o gerenciamento desse TAD.

# Filas

## 6.1 DEFINIÇÃO

O conceito de fila, ou *fila de espera*, é algo bastante comum para as pessoas. Afinal, somos obrigados a enfrentar uma fila sempre que vamos ao banco, ao cinema etc. Na computação, uma fila nada mais é do que um conjunto finito de itens (de um mesmo tipo) esperando por um serviço ou processamento. Trata-se de um controle de fluxo muito comum na computação.



Um exemplo bastante comum da aplicação de filas é o gerenciamento de documentos enviados para a impressora.



As filas são implementadas e se comportam de modo muito similar às listas, sendo, muitas vezes, consideradas um tipo especial de lista em que a inserção e a remoção são realizadas sempre em extremidades distintas.

Neste caso, a inserção de um item é feita de um lado da fila, enquanto a retirada é feita do outro lado. Desse modo, se quisermos acessar determinado elemento da fila, deveremos remover todos os que estiverem à frente dele. Por esse motivo, as filas são conhecidas como estruturas do tipo **primeiro a entrar, primeiro a sair** ou FIFO (First In First Out): os elementos são removidos da fila na mesma ordem em que foram inseridos.

Em ciência da computação, uma fila é uma estrutura de dados linear utilizada para armazenar e controlar o fluxo de dados em um computador. Uma estrutura do tipo fila é uma sequência de elementos do mesmo tipo, como ilustrado na Figura 6.1. Seus elementos possuem estrutura interna abstruída, ou seja, sua complexidade é arbitrária e não afeta o seu funcionamento. Além disso, uma fila pode possuir elementos repetidos, dependendo da aplicação. Uma estrutura do tipo fila pode possuir  $N$  ( $N \geq 0$ ) elementos ou itens. Se  $N = 0$ , dizemos que a fila está vazia.

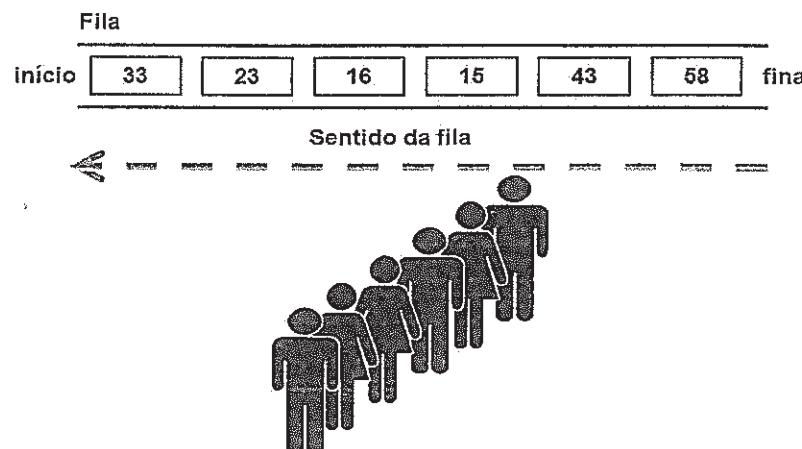


FIGURA 6.1

## 6.2 TIPOS DE FILAS

Basicamente, existem dois tipos de implementações principais para uma fila.



Estas implementações diferem entre si com relação ao tipo de alocação de memória usada e ao tipo de acesso aos elementos.

Uma fila pode ser implementada usando **alocação estática com acesso sequencial** ou **alocação dinâmica com acesso encadeado**, como descrito a seguir:

- **Alociação estática com acesso sequencial:** o espaço de memória é alocado no momento da compilação do programa, ou seja, é necessário definir o número máximo de elementos que a fila irá possuir. Desse modo, os elementos são armazenados de forma consecutiva na memória (como em um array ou vetor) e a posição de um elemento pode ser facilmente obtida a partir do início da fila.
- **Alociação dinâmica com acesso encadeado:** o espaço de memória é alocado em tempo de execução, ou seja, a fila cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos. Nessa implementação, cada elemento pode estar em uma área distinta da memória, ambas não necessariamente consecutivas. É necessário então que cada elemento da fila armazene, além da sua informação, o endereço de memória onde se encontra o próximo elemento. Para acessar um elemento, é preciso percorrer todos os seus antecessores na fila.

Nas próximas seções, serão apresentadas as diferentes implementações de fila com relação à alocação e ao acesso aos elementos.

## 6.3 OPERAÇÕES BÁSICAS DE UMA FILA

Independentemente do tipo de alocação e acesso usado na implementação de uma fila, as seguintes operações básicas são sempre possíveis:

- Criação da fila.
- Inserção de um elemento no final da fila.
- Remoção de um elemento do início da fila.
- Acesso ao elemento do início da fila.
- Destrução da fila.
- Além de informações com tamanho, se a fila está cheia ou vazia.

## 6.4 FILA SEQUENCIAL ESTÁTICA

Uma fila sequencial estática é uma fila definida utilizando alocação estática e acesso sequencial dos elementos. Trata-se do tipo mais simples de fila possível. Ela é definida utilizando um array, de modo que o sucessor de um elemento ocupa a posição física seguinte deste.



Além do array, essa fila utiliza três campos adicionais para guardar o **início**, o **final** e a **quantidade de elementos (dados)** inseridos na fila.

Em uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo fila (ela é um **tipo opaco**), como ilustrado na Figura 6.2. Isso impede o usuário de saber como foi realmente implementada a fila, e limita o seu acesso apenas às funções que manipulam o início e o final da fila.

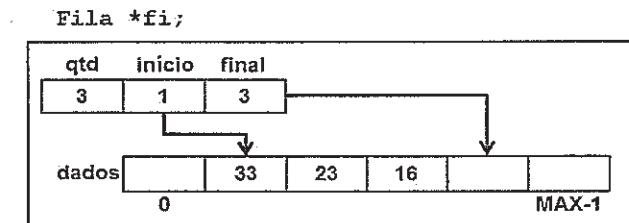


FIGURA 6.2



A principal vantagem de se utilizar um array na definição de uma fila sequencial estática é a facilidade de criar e destruir a fila. Já a sua principal desvantagem é a necessidade de definir previamente o tamanho do array e, consequentemente, da fila.

Considerando suas vantagens e desvantagens, o ideal é utilizar uma fila sequencial estática em filas pequenas ou quando o tamanho máximo da fila é bem definido.

#### 6.4.1 Definindo o tipo fila sequencial estática

Antes de começar a implementar a nossa fila, é preciso definir o tipo de dado que será armazenado nela. Uma fila pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o tipo opaco que representa nossa fila. Este tipo será um ponteiro para a estrutura que define a fila. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **FilaEstatica.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **FilaEstatica.h**, ilustrado na Figura 6.3. Por se tratar de uma fila estática, temos que estabelecer:

- O tamanho máximo do array utilizado na fila, representada pela constante **MAX** (linha 1).
- O tipo de dado que será armazenado na fila, **struct aluno** (linhas 2-6).
- Para fins de padronização, um novo nome para o tipo fila (linha 7). Esse é o tipo que será usado sempre que se desejar trabalhar com uma fila.
- As funções disponíveis para se trabalhar com essa fila em especial (linhas 9-16) e que serão implementadas no arquivo **FilaEstatica.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da fila. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



No arquivo **FilaEstatica.c**, iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **FilaEstatica.h**.

Basicamente, o arquivo **FilaEstatica.c** (Figura 6.3) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da fila (linhas 1-3).
- A definição do tipo que descreve o funcionamento da fila, **struct fila** (linhas 5-8).
- As implementações das funções definidas no arquivo **FilaEstatica.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que o nosso tipo fila nada mais é do que uma estrutura contendo quatro campos:

- Um inteiro **inicio**, que indica a posição no array do elemento que está no início da fila.
- Um inteiro **final**, que indica a posição no array que é o final da fila. Essa é a posição disponível para inserir um novo elemento.

- Um inteiro **qtd**, que indica o quanto do array já está ocupado pelos elementos inseridos na fila.
- Um array do tipo **struct aluno**, que é o tipo de dado a ser armazenado na fila.

Por estarem definidos dentro do arquivo **.c**, os campos dessa estrutura **struct aluno** não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome definido no arquivo **FilaEstatica.h** (linha 7), que pode somente declarar um ponteiro para ele, da seguinte forma:

**Fila \*fi;**



Note que a implementação de uma **fila sequencial estática** é praticamente igual à implementação de uma **lista sequencial estática**. A diferença é que uma fila possui uma regra para inserção e outra para remoção.

```
Arquivo FilaEstatica.h
01 #define MAX 100
02 struct aluno{
03     int matricula;
04     char nome[30];
05     float n1,n2,n3;
06 };
07 typedef struct fila Fila;
08
09 Fila* cria_Fila();
10 void libera_Fila(Fila* fi);
11 int consulta_Fila(Fila* fi, struct aluno *al);
12 int insere_Fila(Fila* fi, struct aluno al);
13 int remove_Fila(Fila* fi);
14 int tamanho_Fila(Fila* fi);
15 int Fila_vazia(Fila* fi);
16 int Fila_cheia(Fila* fi);
```

```
Arquivo FilaEstatica.c
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "FilaEstatica.h" //inclui os protótipos
04 //Definição do tipo Fila
05 struct fila{
06     int inicio, final, qtd;
07     struct aluno dados[MAX];
08 };
```

FIGURA 6.3.

#### 6.4.2 Criando e destruindo uma fila

Para utilizar uma fila em seu programa, a primeira coisa a fazer é criar uma fila vazia. Essa tarefa é executada pela função descrita na Figura 6.4. Basicamente, o que esta função faz é a alocação de uma área de memória para a fila (linha 3). Esta área de memória corresponde à memória necessária para armazenar a estrutura que define a fila, **struct fila**. Em seguida, essa função inicializa os três campos da fila com o valor **ZERO**: **inicio** (que indica a posição no array do elemento que está no início da fila), **final** (que mostra a posição no array que é o final da fila e que está disponível para inserir um novo elemento) e **qtd** (que indica o quanto do array já está ocupado, ou seja, nenhum). A Figura 6.5 mostra o conteúdo do nosso ponteiro **Fila\* fi** após a chamada da função que cria a fila.

**Criando uma fila**

```

01 Fila* cria_Fila(){
02     Fila *fi;
03     fi = (Fila*) malloc(sizeof(struct fila));
04     if(fi != NULL){
05         fi->inicio = 0;
06         fi->final = 0;
07         fi->qtd = 0;
08     }
09     return fi;
10 }
```

FIGURA 6.4

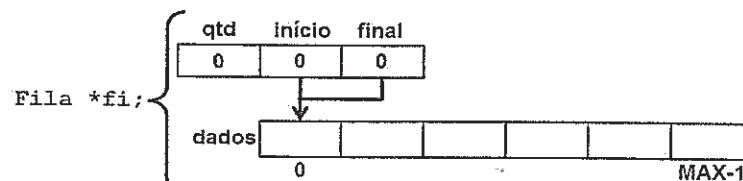


FIGURA 6.5

Destruir uma fila estática é bastante simples, como mostra o código contido na Figura 6.6. Basicamente, o que temos que fazer é liberar a memória alocada para a estrutura que representa a fila. Isso é feito utilizando apenas uma chamada da função **free()**.



Por que criar uma função para destruir a fila, sendo que tudo que precisamos fazer é chamar a função **free()**?

Por questões de modularização. Destruir uma fila estática é bastante simples, porém destruir uma fila alocada dinamicamente é uma tarefa mais complicada. Ao criar essa função, estamos

escondendo a implementação dessa tarefa do usuário, ao mesmo tempo que mantemos a notação utilizada por uma fila com alocação **estática** ou **dinâmica**. Desse modo, utilizar uma fila estática ou dinâmica será indiferente para o programador.

**Destruidor de fila**

```

01 void libera_Fila(Fila* fi){
02     free(fi);
03 }
```

FIGURA 6.6

#### 6.4.3 Informações básicas sobre a fila

As operações de inserção, remoção e consulta são consideradas as principais de uma fila. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a fila. Por exemplo, não podemos remover um elemento da fila se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da fila: seu tamanho, se ela está cheia ou se ela está vazia.

##### Tamanho da fila

Saber o tamanho de uma fila **sequencial estática** é uma tarefa relativamente simples. Isso ocorre porque essa fila possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na fila, como mostra a Figura 6.7.



Basicamente, retornar o tamanho de uma fila **sequencial estática** consiste em retornar o valor do seu campo **qtd**.

A implementação da função que retorna o tamanho da fila é mostrada na Figura 6.8. Note que essa função, em primeiro lugar, verifica se o ponteiro **Fila\* fi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e retornar o seu valor, que nada mais é do que o tamanho da fila.

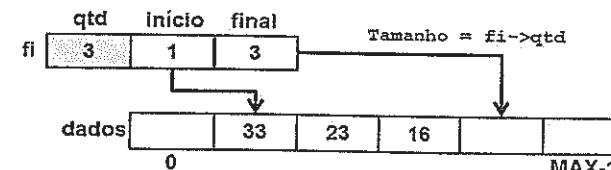


FIGURA 6.7

**Tamanho da fila**

```

01 int tamanho_Fila(Fila* fi){
02     if(fi == NULL)
03         return -1;
04     return fi->qtd;
05 }

```

FIGURA 6.8

**Fila cheia**

Saber se uma **fila sequencial estática** está cheia é outra tarefa relativamente simples. Novamente, isso ocorre porque essa fila possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na fila, como mostra a Figura 6.9.

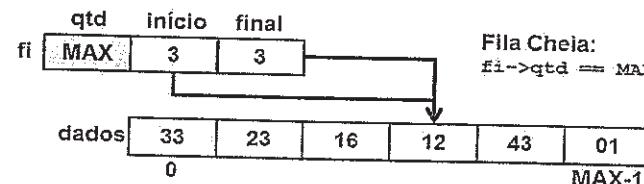


FIGURA 6.9



Basicamente, retornar se uma **fila sequencial estática** está cheia consiste em verificar se o valor do seu campo **qtd** é igual a **MAX**.

A implementação da função que retorna se a fila está cheia é mostrada na Figura 6.10. Note que essa função, em primeiro lugar, verifica se o ponteiro **Fila\*** **fi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma fila inválida. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o tamanho máximo definido para o seu array (vetor) de elementos: **MAX**. Se os valores forem iguais (ou seja, fila cheia), iremos retornar o valor **UM** (linha 5). Caso contrário, a função irá retornar o valor **ZERO**.



Note que apenas o valor do campo **qtd** é relevante para saber se a fila está cheia. Os campos **início** e **final** poderão ter qualquer valor, desde que seja o mesmo.

**Retornando se a fila está cheia**

```

01 int Fila_cheia(Fila* fi){
02     if(fi == NULL)
03         return -1;
04     if (fi->qtd == MAX)
05         return 1;
06     else
07         return 0;
08 }

```

FIGURA 6.10

**Fila vazia**

Saber se uma **fila sequencial estática** está vazia é outra tarefa bastante simples. Como no caso do tamanho da fila e da fila cheia, isso ocorre porque esse tipo de fila possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na fila, como mostra a Figura 6.11.

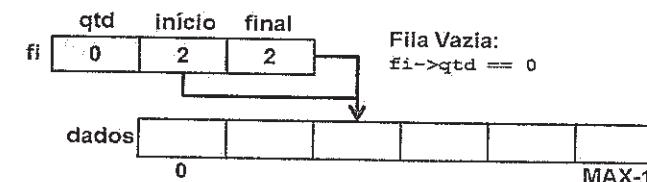


FIGURA 6.11



Basicamente, retornar se uma **fila sequencial estática** está vazia consiste em verificar se o valor do seu campo **qtd** é igual a **ZERO**.

A implementação da função que retorna se a fila está vazia é mostrada na Figura 6.12. Note que essa função, em primeiro lugar, verifica se o ponteiro **Fila\*** **fi** é igual a **NULL**. A condição seria verdadeira caso houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma fila inválida. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o valor **ZERO**, que é o valor inicial do campo quando criamos uma fila. Se os valores forem iguais (ou seja, fila vazia), iremos retornar o valor **UM** (linha 5). Caso contrário, a função irá retornar o valor **ZERO**.



Note que apenas o valor do campo **qtd** é relevante para saber se a fila está vazia. Os campos **início** e **final** poderão ter qualquer valor, desde que seja o mesmo.

### Retornando se a fila está vazia

```

01 int Fila_vazia(Fila* fi){
02     if(fi == NULL)
03         return -1;
04     if (fi->qtd == 0)
05         return 1;
06     else
07         return 0;
08 }

```

FIGURA 6.12

#### 6.4.4 Inserindo um elemento na fila

Inserir um elemento em uma fila sequencial estática é uma tarefa simples e bastante semelhante à inserção no final de uma lista sequencial estática, como mostra a sua implementação na Figura 6.13.

Primeiramente, verificamos se o ponteiro `Fila* fi` é igual a `NULL` (linha 2). A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor `ZERO` para indicar uma fila inválida (linha 3). Porém, se a fila foi criada com sucesso, precisamos verificar se ela não está cheia, isto é, se existe espaço para um novo elemento. Caso a fila esteja cheia, a função irá retornar o valor `ZERO` (linhas 4 e 5).

A inserção em uma fila ocorre sempre no seu final. Sendo assim, copiamos os dados a ser inseridos para a posição apontada pelo campo `final` da fila (linha 6). Em seguida, devemos incrementar o valor do campo `final`, indicando assim a próxima posição vaga na fila (linha 7).



Note que utilizamos a operação de resto da divisão no cálculo do novo final da fila. Fazemos isso para simular uma fila circular. Assim, ao chegar à posição `MAX` (que não existe no array) o final da fila será colocado na posição `ZERO`, de modo que as posições no começo do array que ficarem vagas, à medida que inserirmos e removermos elementos da fila, poderão ser usadas pelo final da fila.

### Inserindo um elemento na fila

```

01 int insere_Fila(Fila* fi, struct aluno al){
02     if(fi == NULL)
03         return 0;
04     if(fi->qtd == MAX)
05         return 0;
06     fi->dados[fi->final] = al;
07     fi->final = (fi->final+1)%MAX;
08     fi->qtd++;
09 }
10 }

```

FIGURA 6.13

Por fim, devemos incrementar a quantidade (`fi->qtd`) de elementos armazenados na fila e retornamos o valor `UM` (linhas 8 e 9), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 6.14.

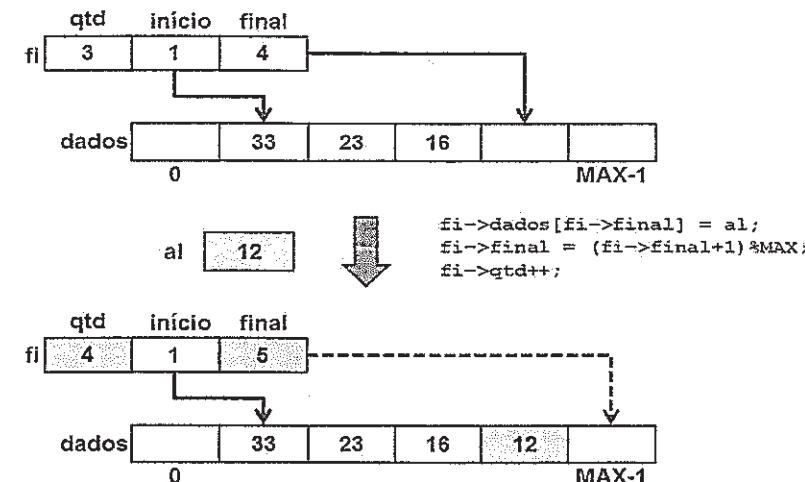


FIGURA 6.14

#### 6.4.5 Removendo um elemento da fila

Remover um elemento de uma fila sequencial estática é uma tarefa simples e bastante semelhante à remoção do final de uma lista sequencial estática, como mostra a sua implementação na Figura 6.15.

Primeiramente, verificamos se o ponteiro `Fila* fi` é igual a `NULL` ou se a fila está vazia (linha 2). As condições garantem que temos uma fila válida para trabalhar (ou seja, não houve problemas na criação da fila) e que existem elementos que podem ser removidos da fila. Assim, optamos por retornar o valor `ZERO` para indicar que uma das condições é falsa (linha 3). Como a remoção é feita no início da fila, basta incrementar em uma unidade o seu valor (linha 4).



Note que utilizamos a operação de resto da divisão no cálculo do novo início da fila. Fazemos isso para simular uma fila circular. Assim, ao chegar à posição `MAX` (que não existe no array) o início da fila será colocado na posição `ZERO`, de modo que as posições no começo do array que ficarem vagas, à medida que inserirmos e removermos elementos da fila, poderão ser usadas pelo início da fila.

Por fim, devemos diminuir a quantidade (`fi->qtd`) de elementos armazenados na fila e retornamos o valor `UM` (linhas 5 e 6), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 6.16.

### Removendo um elemento da fila

```

01 int remove_Fila(Fila* fi){
02     if(fi == NULL || fi->qtd == 0)
03         return 0;
04     fi->inicio = (fi->inicio+1)%MAX;
05     fi->qtd--;
06     return 1;
07 }

```

FIGURA 6.15

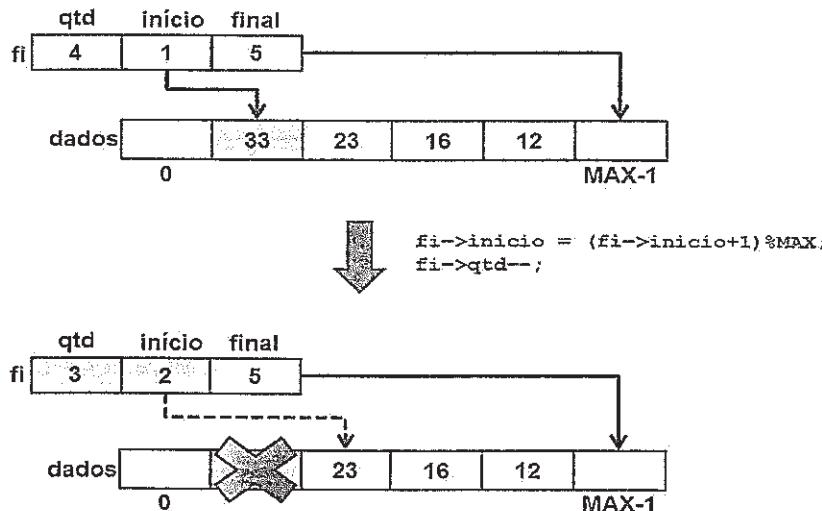


FIGURA 6.16

### 6.4.6 Consultando o elemento no início da fila

Apesar de possuir uma implementação quase idêntica à da **lista sequencial estática**, o acesso a um elemento de uma **fila sequencial estática** é um pouco diferente.



Em uma lista, pode-se acessar e recuperar as informações contidas em qualquer um dos seus elementos. Já em uma fila, podemos acessar as informações apenas do elemento no **início** da fila.

Acessar um elemento que se encontra no **início** de uma **fila sequencial estática** é uma tarefa quase imediata, como mostra a sua implementação na Figura 6.17. Em primeiro lugar, a função verifica se a consulta é válida. Para tanto, duas condições são verificadas: se o ponteiro **Fila\* fi** é igual a **NULL** e se a fila está vazia (quantidade de elementos na fila é diferente de zero). Se alguma destas condições for verdadeira, a busca termina e a função retorna o valor

**ZERO** (linha 3). Caso contrário, a posição equivalente ao “**início**” da fila é copiada para o conteúdo do ponteiro passado por referência (**al**) para a função (linha 4). Como se nota, esse tipo de consulta consiste em um simples acesso ao item há mais tempo inserido no array que representa a fila (Figura 6.18).

```

01 int consulta_Fila(Fila* fi, struct aluno *al){
02     if(fi == NULL || fi->qtd == 0)
03         return 0;
04     *al = fi->dados[fi->inicio];
05     return 1;
06 }

```

FIGURA 6.17

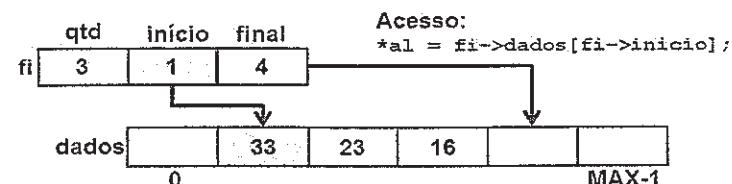


FIGURA 6.18

### 6.4.7 Análise de complexidade

Um aspecto importante quando manipulamos filas tem relação com os custos das suas operações. Em uma **fila sequencial estática** as operações de inserção, remoção e consulta envolvem apenas a manipulação de alguns índices, independentemente do número de elementos na fila. Desse modo, a complexidade das operações é  $O(1)$ .

### 6.5 FILA DINÂMICA ENCADEADA

Uma **fila dinâmica encadeada** é uma fila definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da fila é alocado dinamicamente, à medida que os dados são inseridos dentro da fila, e tem sua memória liberada, à medida que é removido. Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação: um campo de **dado**, utilizado para armazenar a informação inserida na fila, e um campo **prox**, que nada mais é do que um ponteiro que indica o próximo elemento na fila.



Além da estrutura que define seus elementos, essa fila utiliza um **nó descriptor** para guardar o **início**, o **final** e a **quantidade** de elementos (**dados**) inseridos na fila.

Como visto na Seção 5.8, um **nó descritor** é um elemento especial da fila. Dentro dele, podemos armazenar qualquer informação que julgarmos necessária. De modo geral, optamos por armazenar dentro dessa estrutura informações que facilitem a manipulação da fila, como o seu início, o seu final e a quantidade de elementos, como mostrado na Figura 6.19.

Em uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo fila (ela é um **tipo opaco**), como ilustrado na Figura 6.19. Isso impede o usuário de saber como foi realmente implementada a fila, e limita o seu acesso apenas às funções que manipulam o início e o final da fila.



Note que a implementação em módulos impede o usuário de saber como a fila foi implementada. Tanto a **fila dinâmica encadeada** quanto a **fila sequencial estática** são declaradas como sendo do tipo **Fila \***.

Essa é a grande vantagem da modularização e da utilização de tipos opacos: mudar a maneira pela qual a fila foi implementada não altera nem interfere no funcionamento do programa que a utiliza.

`Fila *fi;`

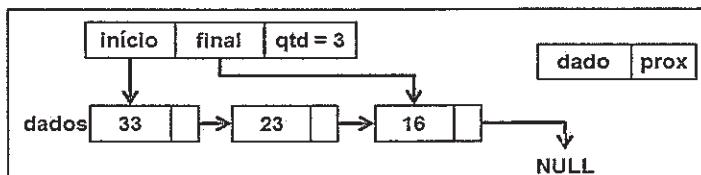


FIGURA 6.19



A principal vantagem de se utilizar uma abordagem dinâmica e encadeada na definição da fila é a melhor utilização dos recursos de memória, não sendo mais necessário definir previamente o tamanho da fila. Já a sua principal desvantagem é a necessidade de percorrer toda a fila para destruí-la.

Considerando suas vantagens e desvantagens, o ideal é utilizar uma **fila dinâmica encadeada** quando não há necessidade de garantir um espaço mínimo para a execução da aplicação, ou quando o tamanho máximo da fila não é bem definido.

### 6.5.1 Definindo o tipo fila dinâmica encadeada

Antes de começar a implementar a nossa fila, é preciso definir o tipo de dado que será armazenado nela. Uma fila pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa fila, neste caso, nosso **nó descritor**. Este tipo será um ponteiro para a estrutura que define a fila. Além disso, também

precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **FilaDin.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **FilaDin.h**, ilustrado na Figura 6.20. Por se tratar de uma fila dinâmica encadeada, temos que estabelecer:

- O tipo de dado que será armazenado na fila, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para a **struct fila** (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma fila.
- As funções disponíveis para se trabalhar com essa fila em especial (linhas 8-15) e que serão implementadas no arquivo **FilaDin.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da fila. Este aluno é representado pelo seu número de matrícula, nome e três notas.



No arquivo **FilaDin.c**, iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em **FilaDin.h**.

Basicamente, o arquivo **FilaDin.c** (Figura 6.20) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da fila (linhas 1-3).
- A definição do tipo que descreve cada elemento da fila, **struct elemento** (linhas 5-8).
- A definição de um novo nome para a **struct elemento** (linha 9). Isso é feito apenas para facilitar certas etapas de codificação.
- A definição do tipo que descreve o **nó descritor** da fila, **struct fila** (linhas 11-15).
- As implementações das funções definidas no arquivo **FilaDin.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que a **struct elemento** nada mais é do que uma estrutura contendo dois campos:

- Um ponteiro **prox**, que indica o próximo elemento (também do tipo **struct elemento**) dentro da fila.
- Um campo **dado** do tipo **struct aluno**, que é o tipo de dado a ser armazenado na fila.

Perceba também que a **struct fila** nada mais é do que uma estrutura contendo três campos:

- Um ponteiro **inicio**, que indica o primeiro elemento da fila.
- Um ponteiro **final**, que indica o último elemento da fila.
- Um campo **qtd** do tipo **int**, que armazena o número de elementos dentro da fila.

Por estarem definidos dentro do arquivo .c, os campos dessa estrutura **struct aluno** não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **FilaDin.h** (linha 6), que pode somente declarar um ponteiro para ele da seguinte forma:

```
Fila *fi;
```



Note que a implementação de uma **fila dinâmica encadeada** é praticamente igual à implementação de uma **lista dinâmica encadeada com nó descritor**. A diferença é que uma fila possui uma regra para inserção e outra para remoção.

```
01 struct aluno{
02     int matricula;
03     char nome[30];
04     float n1, n2, n3;
05 };
06 typedef struct fila Fila;
07
08 Fila* cria_Fila();
09 void libera_Fila(Fila* fi);
10 int consulta_Fila(Fila* fi, struct aluno *al);
11 int insere_Fila(Fila* fi, struct aluno al);
12 int remove_Fila(Fila* fi);
13 int tamanho_Fila(Fila* fi);
14 int Fila_vazia(Fila* fi);
15 int Fila_cheia(Fila* fi);
```

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "FilaDin.h" // inclui os Protótipos
04 // Definição do tipo Fila
05 struct elemento{
06     struct aluno dados;
07     struct elemento *prox;
08 };
09 typedef struct elemento Elem;
10 // Definição do Nô Descritor da Fila
11 struct fila{
12     struct elemento *inicio;
13     struct elemento *final;
14     int qtd;
15 };
```

FIGURA 6.20

## 6.5.2 Criando e destruindo uma fila

Para utilizar uma fila em seu programa, a primeira coisa a fazer é criar uma fila vazia. Essa tarefa é executada pela função descrita na Figura 6.21. Basicamente, o que esta função faz é a alocação de uma área de memória para a fila (linha 2). Esta área de memória corresponde à memória necessária para armazenar a estrutura que define a fila, **struct fila**. Em seguida, a função inicializa os três campos da fila, como descrito a seguir:

- **inicio** (que aponta para o elemento que está no início da fila) recebe **NULL**.
- **final** (que aponta para o elemento que está no final da fila) recebe **NULL**.
- **qtd** (que indica a quantidade de elementos na fila) recebe **ZERO** (ou seja, nenhum elemento na fila).

A Figura 6.22 indica o conteúdo do nosso ponteiro **Fila\* fi** após a chamada da função que cria a fila.

```
01 Fila* cria_Fila(){
02     Fila* fi = (Fila*) malloc(sizeof(struct fila));
03     if(fi != NULL){
04         fi->final = NULL;
05         fi->inicio = NULL;
06         fi->qtd = 0;
07     }
08     return fi;
09 }
```

FIGURA 6.21

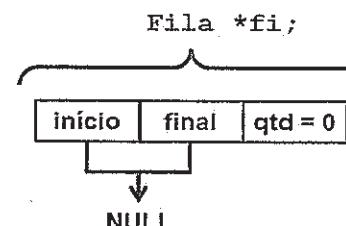


FIGURA 6.22

Destruir uma fila que utilize alocação dinâmica e seja encadeada não é uma tarefa tão simples quanto destruir uma **fila sequencial estática**.



Para liberar uma fila que utilize alocação dinâmica e seja encadeada é preciso percorrer toda a fila liberando a memória alocada para cada elemento inserido nela.

O código que realiza a destruição da fila é mostrado na Figura 6.23. Inicialmente, verificamos se a fila é válida, ou seja, se a tarefa de criação da fila foi realizada com sucesso (linha 2). Em seguida, percorremos a fila enquanto o conteúdo do seu início seja diferente de NULL, o final da fila (linha 4). Enquanto não chegarmos ao final da fila, iremos liberar a memória do elemento que se encontra atualmente no início da fila e avançar para o próximo (linhas 5-7). Terminado o processo, liberamos a memória alocada para o nó descritor que representa a fila (linha 9). Esse processo é mais bem ilustrado pela Figura 6.24, que mostra a liberação de uma fila contendo dois elementos.

### Destruindo uma fila

```

01 void libera_Fila(Fila* fi){
02     if(fi != NULL){
03         Elemt* no;
04         while(fi->inicio != NULL) {
05             no = fi->inicio;
06             fi->inicio = fi->inicio->prox;
07             free(no);
08         }
09         free(fi);
10     }
11 }
```

FIGURA 6.23

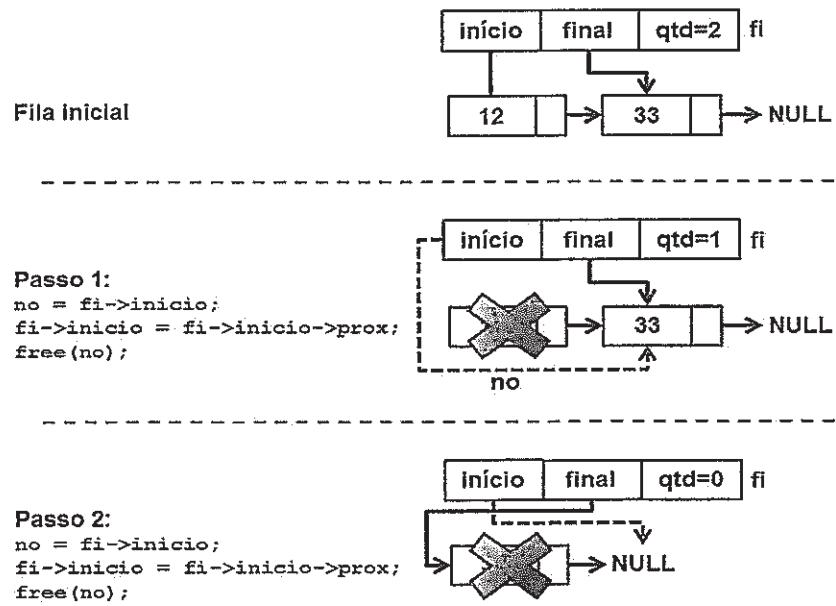


FIGURA 6.24

### 6.5.3 Informações básicas sobre a fila

As operações de inserção, remoção e consulta são consideradas as principais de uma fila. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a fila. Por exemplo, não podemos remover um elemento da fila se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da fila: seu tamanho, se ela está cheia ou se ela está vazia.

#### Tamanho da fila

Saber o tamanho de uma fila dinâmica encadeada é uma tarefa relativamente simples. Isso ocorre porque estamos trabalhando com um **nó descritor** e ele possui um campo inteiro **qtd** que indica a quantidade de elementos inseridos na fila, como mostra a Figura 6.25.

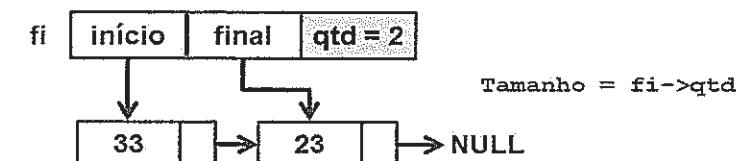


FIGURA 6.25



Basicamente, retornar o tamanho de uma fila dinâmica encadeada consiste em retornar o valor do seu campo **qtd**.

A implementação da função que retorna o tamanho da fila é mostrada na Figura 6.26. Note que essa função, em primeiro lugar, verifica se o ponteiro **Fila\* fi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e retornar o seu valor, que nada mais é do que o tamanho da fila.

### Tamanho da fila

```

01 int tamanho_Fila(Fila* fi){
02     if(fi == NULL)
03         return 0;
04     return fi->qtd;
05 }
```

FIGURA 6.26

## Fila cheia

Implementar uma função que retorne se uma **fila dinâmica encadeada** está cheia é uma tarefa relativamente simples.



Uma **fila dinâmica encadeada** somente será considerada cheia quando não tivermos mais memória disponível para alocar novos elementos.

A implementação da função que retorna se a fila está cheia é mostrada na Figura 6.27. Como se pode notar, a função sempre irá retornar o valor **ZERO**, indicando que a fila não está cheia.

### Retornando se a fila está cheia

```
01 int Fila_cheia(Fila* fi){
02     return 0;
03 }
```

FIGURA 6.27

## Fila vazia

Saber se uma **fila dinâmica encadeada** está vazia é outra tarefa bastante simples. Como no caso do tamanho da fila e da fila cheia, isso ocorre porque temos um nó descritor que possui um campo **inicio**. Este campo aponta para o primeiro elemento da fila, como mostra a Figura 6.28.



Basicamente, retornar se uma **fila dinâmica encadeada** está vazia consiste em verificar se o valor do seu campo **inicio** é igual a **NULL**.

A implementação da função que retorna se a fila está vazia é mostrada na Figura 6.29. Note que essa função, em primeiro lugar, verifica se o ponteiro **Fila\* fi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma fila inválida (linha 3). Porém, se a fila foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linha 4). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 5 e 6). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 7).



Outra maneira de saber se uma fila está vazia é verificando se o valor do campo **qtd** é igual a **ZERO**.

Neste caso, **ZERO** é o valor inicial do campo **qtd** quando criamos uma fila.

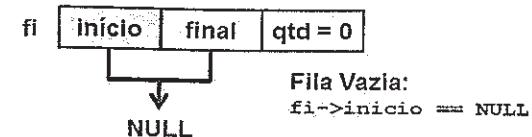


FIGURA 6.28

### Reformando se a fila está vazia

```
01 int Fila_vazia(Fila* fi){
02     if(fi == NULL)
03         return -1;
04     if(fi->início == NULL)
05         return 1;
06     return 0;
07 }
```

FIGURA 6.29

## 6.5.4 Inserindo um elemento na fila

Inserir um elemento em uma **fila dinâmica encadeada** é uma tarefa simples e bastante semelhante à inserção no final de uma **lista dinâmica com nó descritor**. Basicamente, temos que alocar espaço para o novo elemento da fila e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 6.30.

Primeiro, verificamos se o ponteiro **Fila\* fi** é igual a **NULL** (linha 2). A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma fila inválida (linha 3). Porém, se a fila foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linha 4). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 5 e 6). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 7).

Como se trata de uma inserção no final da fila, o elemento a ser inserido obrigatoriamente irá apontar para a constante **NULL** (linha 8). Também temos que considerar que a fila pode ou não estar vazia (linha 9):

- No caso de ser uma fila vazia, mudamos o conteúdo do “**início**” da fila (**fi->início**) para que ele passe a ser o nosso elemento **no** (linha 10).
- No caso de NÃO ser uma fila vazia, o elemento do final da fila deverá apontar para o novo elemento (linha 12).

Em seguida, mudamos o conteúdo do “**final**” da fila (**fi->final**) para que ele passe a ser o nosso elemento **no**, incrementamos o valor do tamanho da fila e retornamos o valor **UM**.

(linhas 13-15), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 6.31, na qual podemos ver um elemento sendo inserido em uma fila que está vazia ou que já contém um elemento.

```

FIGURA 6.30
Inserindo um elemento na fila
01 int insere_Fila(Fila* fi, struct aluno al) {
02     if(fi == NULL)
03         return 0;
04     Elem *no = (ELEM*) malloc(sizeof(ELEM));
05     if(no == NULL)
06         return 0;
07     no->dados = al;
08     no->prox = NULL;
09     if(fi->final == NULL)//fila vazia
10         fi->inicio = no;
11     else
12         fi->final->prox = no;
13     fi->final = no;
14     fi->qtd++;
15     return 1;
16 }
```

FIGURA 6.30

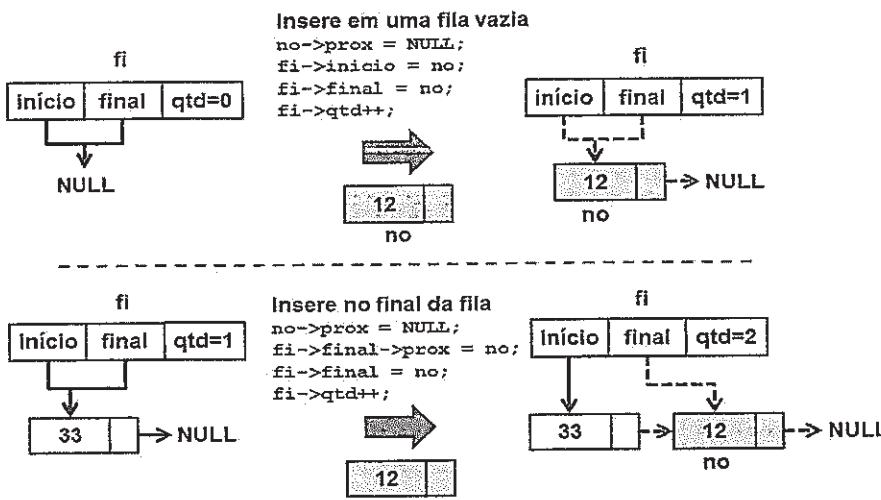


FIGURA 6.31

### 6.5.5 Removendo um elemento da fila

Remover um elemento de uma fila dinâmica encadeada é uma tarefa simples e bastante semelhante à remoção do início de uma lista dinâmica com nó descritor. Basicamente, o que temos que fazer é verificar se a fila não está vazia e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 6.32.

```

FIGURA 6.32
Removendo um elemento da fila
01 int remove_Fila(Fila* fi){
02     if(fi == NULL)
03         return 0;
04     if(fi->inicio == NULL)//fila vazia
05         return 0;
06     Elem *no = fi->inicio;
07     fi->inicio = fi->inicio->prox;
08     free(no);
09     if(fi->inicio == NULL)//fila ficou vazia
10         fi->final = NULL;
11     fi->qtd--;
12     return 1;
13 }
```

Primeiro, verificamos se o ponteiro `Fila* fi` é igual a `NULL` (linha 2). A condição seria verdadeira se houvesse ocorrido um problema na criação da fila e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor `ZERO` para indicar uma fila inválida (linha 3). Porém, se a fila foi criada com sucesso, precisamos verificar se ela não está vazia, isto é, se existem elementos dentro dela. Caso a fila esteja vazia, a função irá retornar o valor `ZERO` (linhas 4 e 5).

Como se trata de uma remoção do início da fila, criamos uma cópia do início em um elemento auxiliar (`no`) e fazemos com que o início da fila (`fi->inicio`) aponte para o elemento seguinte a ele (linhas 6 e 7). Em seguida, temos que liberar a memória associada ao antigo “início” da fila (`no`) (linha 8). Como estamos trabalhando com um nó descritor, temos que considerar que a fila pode ficar vazia após a remoção. Neste caso, é necessário mudar o valor do seu final (linhas 9-10). Por fim, diminuímos o valor do tamanho da fila e retornamos o valor `UM` (linhas 11-12), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 6.33.

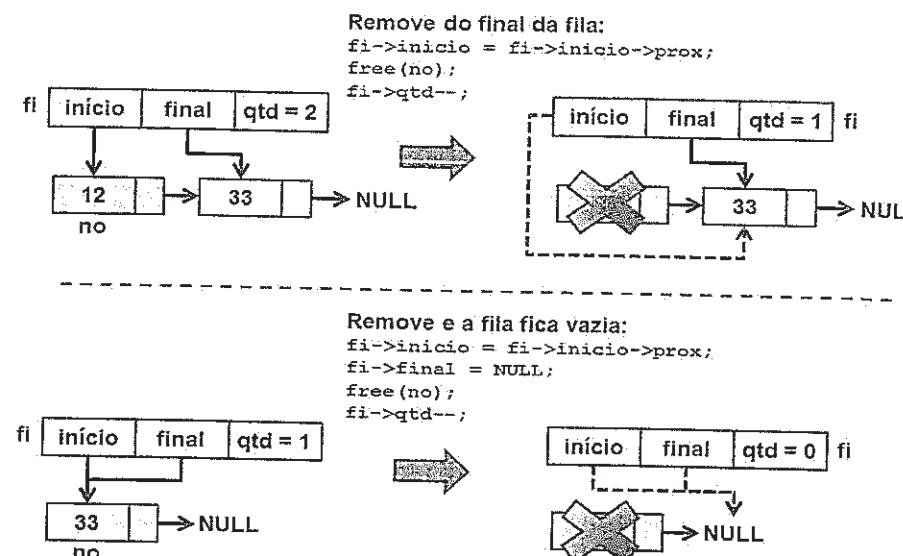


FIGURA 6.33

### 6.5.6 Consultando o elemento no início da fila

Apesar de possuir uma implementação quase idêntica à da lista dinâmica encadeada com nó descriptor, o acesso a um elemento de uma fila dinâmica encadeada é um pouco diferente.



Em uma lista, pode-se acessar e recuperar as informações contidas em qualquer um dos seus elementos. Já em uma fila, podemos acessar as informações apenas do elemento no **início** da fila.

Acessar um elemento que se encontra no início de uma fila dinâmica encadeada é uma tarefa quase imediata, como mostra a sua implementação na Figura 6.34. Em primeiro lugar, a função verifica se a consulta é válida. Para tanto, ela verifica se o ponteiro `Fila* fi` é igual a `NULL`, ou seja, se houve erro na criação da fila. Em caso afirmativo, a função retorna o valor `ZERO` e a busca termina (linhas 2-3). Em seguida, ela verifica se a fila está vazia. Isso é feito verificando se o seu campo `início` é igual a `NULL`. Novamente, a função retorna o valor `ZERO` se essa condição for verdadeira e a busca termina (linhas 4-5). Caso contrário, os dados contidos no campo “`início`” da fila são copiados para o conteúdo do ponteiro passado por referência (`al`) para a função (linha 6). Como se nota, esse tipo de consulta consiste em um simples acesso ao item há mais tempo alocado e inserido na fila (Figura 6.35).

**Consulta do elemento no início da fila:**

```

01 int consulta_Fila(Fila* fi, struct aluno *al){
02     if(fi == NULL)
03         return 0;
04     if(fi->início == NULL)//fila vazia
05         return 0;
06     *al = fi->início->dados;
07     return 1;
08 }
  
```

FIGURA 6.34

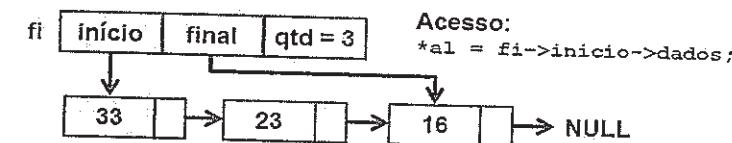


FIGURA 6.35

### 6.5.7 Análise de complexidade

Um aspecto importante quando manipulamos filas tem relação com os custos das suas operações. Em uma fila dinâmica encadeada as operações de inserção, remoção e consulta envolvem apenas a manipulação de alguns ponteiros, independentemente do número de elementos na fila. Desse modo, a complexidade das operações é  $O(1)$ .

### 6.6 CRIANDO UMA FILA USANDO UMA LISTA

Ao longo desta seção, vimos que as filas são implementadas e se comportam de modo muito similar às listas.



As filas são consideradas um tipo especial de lista em que a inserção e a remoção são realizadas sempre em extremidades distintas.

Podemos concluir, então, que uma fila nada mais é do que uma lista sujeita a uma ordem de entrada e saída. Sendo assim, podemos implementar uma fila utilizando uma lista, como mostram as Figuras 6.36 e 6.37. Neste caso, estamos implementando uma fila dinâmica usando uma lista dinâmica.

Vamos começar definindo o arquivo `FilaUsandoListaDinEncad.h`, ilustrado na Figura 6.36. Nele, temos que estabelecer:

- A biblioteca da lista usada para representar a fila (linha 1).
- Para fins de padronização, um novo nome para a **Lista** (linha 3). Esse é o tipo que será usado sempre que se desejar trabalhar com uma fila.
- As funções disponíveis para se trabalhar com essa fila (linhas 5-12) e que serão implementadas no arquivo **FilaUsandoListaDinEncad.c**.

No arquivo **FilaUsandoListaDinEncad.c**, ilustrado na Figura 6.37, iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **FilaUsandoListaDinEncad.h**. Neste caso, as funções definidas para a fila irão apenas encapsular as funções já definidas para a lista. O mais importante nessa implementação é a criação da regra de inserção e remoção da fila, assim como a consulta a um elemento:

- Consulta na fila: sempre o **primeiro** elemento da lista (linhas 9-11).
- Inserção na fila: sempre no **final** da lista (linhas 12-14).
- Remoção da fila: sempre no **íncio** da lista (linhas 15-17).

Arquivo **FilaUsandoListaDinEncad.h**

```

01 #include "ListaDinEncad.h"
02
03 typedef Lista Fila;
04
05 Fila* cria_Fila();
06 void libera_Fila(Fila* fi);
07 int consulta_Fila(Fila* fi, struct aluno *al);
08 int insere_Fila(Fila* fi, struct aluno al);
09 int remove_Fila(Fila* fi);
10 int tamanho_Fila(Fila* fi);
11 int Fila_vazia(Fila* fi);
12 int Fila_cheia(Fila* fi);

```

FIGURA 6.36

Arquivo **FilaUsandoListaDinEncad.c**

```

01 //inclui os Protótipos
02 #include "FilaUsandoListaDinEncad.h"
03 Fila* cria_Fila(){
04     return cria_lista();
05 }
06 void libera_Fila(Fila* fi){
07     libera_lista(fi);
08 }
09 int consulta_Fila(Fila* fi, struct aluno *al){
10     return consulta_lista_pos(fi, 1, al);
11 }
12 int insere_Fila(Fila* fi, struct aluno al){
13     return insere_lista_final(fi, al);
14 }
15 int remove_Fila(Fila* fi){
16     return remove_lista_inicio(fi);
17 }
18 int tamanho_Fila(Fila* fi){
19     return tamanho_lista(fi);
20 }
21 int Fila_vazia(Fila* fi){
22     return lista_vazia(fi);
23 }
24 int Fila_cheia(Fila* fi){
25     return lista_cheia(fi);
26 }

```

FIGURA 6.37

## 6.7 EXERCÍCIOS

- 1) Defina, usando as suas palavras, o que é uma estrutura do tipo fila.
- 2) Defina, usando as suas palavras, a diferença da fila sequencial estática para a fila dinâmica encadeada.
- 3) Liste as situações em que uma fila pode ser utilizada. Quando seria usada a implementação estática? E a dinâmica?
- 4) Implemente uma função que receba uma fila e a inverta. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 5) Considere uma fila que armazena números inteiros. Faça uma função que receba uma fila e exclua todos os números negativos. A ordem dos outros elementos não deve ser alterada. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 6) Crie uma função que imprima os elementos de uma fila.
- 7) Crie um TAD que utilize uma lista dinâmica duplamente encadeada para funcionar como uma fila. Implemente as operações desse TAD.

- 8) Implemente uma função que receba duas filas (F1 e F2) e as concatene. O resultado da concatenação deve ser colocado em F1. A fila F2 deve ficar vazia. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 9) Implemente uma função para intercalar filas. A função receberá duas filas (F1 e F2) e o resultado da intercalação deve ser colocado em F1. A fila F2 deve ficar vazia. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 10) Implemente uma função para unir filas. A função receberá duas filas (F1 e F2) e o resultado da união deve ser colocado em F1, sem repetições. A fila F2 deve ficar vazia. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 11) Implemente uma função que copie os elementos de uma fila F1 para uma fila F2.
- 12) Modifique a TAD fila para que ela funcione como um deque. Um deque é um tipo especial de fila que permite inserções e remoções tanto no início quanto no final. Implemente as funções do TAD.

## CAPÍTULO 7

# Filas de prioridade

## 7.1 DEFINIÇÃO

Uma **fila de prioridade** é um tipo especial de fila que generaliza a ideia de **ordenação**. Neste tipo de fila, os elementos nela inseridos possuem um dado extra, associado a eles: a sua **prioridade**. É o valor associado à **prioridade** que determina a posição de um elemento na fila, assim como quem deve ser o primeiro a ser removido desta, quando necessário.



Dois elementos podem ter a mesma prioridade na fila. O elemento inserido primeiro fica à frente de quem foi inserido posteriormente, se as prioridades forem iguais.

Várias são as aplicações existentes das filas de prioridades. Basicamente, qualquer problema em que seja preciso estabelecer uma prioridade de acesso aos elementos pode ser representado com uma fila de prioridade. Um exemplo é a fila de prioridade do processador. Nela, os processos com maior prioridade são executados antes dos outros. Outros exemplos:

- Uma fila de pacientes esperando transplante de fígado.
- Busca em grafos (algoritmo de Dijkstra).
- Compressão de dados (código de Huffman).
- Sistemas operacionais (manipulação de interrupções).
- Fila de pouso de aviões em um aeroporto (prioridade por combustível disponível).

## 7.2 OPERAÇÕES BÁSICAS DE UMA FILA

Independentemente do tipo de implementação utilizada para representar a fila de prioridades, as seguintes operações básicas são sempre possíveis:

- Criação da fila.
- Inserção de um elemento na fila com prioridade.
- Remoção de um elemento da fila com maior prioridade.
- Acesso a um elemento do início da fila (maior prioridade).

- Destrução da fila.
- Além de informações com tamanho, se a fila está cheia ou vazia.

### 7.3 IMPLEMENTAÇÃO DA FILA DE PRIORIDADES

Ao se modelar um problema utilizando uma fila de prioridades, surge a questão: como implementar essa fila no computador? Existem diversas maneiras de implementar uma fila no computador. São elas:

- Lista dinâmica encadeada.
- Array desordenado.
- Array ordenado.
- Heap binária.



Que implementação devemos usar para a fila de prioridades?

A implementação escolhida para uma fila de prioridades depende da sua aplicação. Não existe uma implementação que seja melhor que a outra em todas os casos. Algumas implementações são eficientes na operação de inserção, outras na de remoção. Há também implementações que são eficientes nas duas operações, como mostra o Quadro 7.1.

QUADRO 7.1

Implementação	Inserção	Remoção
lista dinâmica encadeada	$O(N)$	$O(1)$
array desordenado	$O(1)$	$O(N)$
array ordenado	$O(N)$	$O(1)$
heap binária	$O(\log N)$	$O(\log N)$

A seguir, veremos como implementar uma **fila de prioridades estática** usando a mesma estrutura da **lista sequencial estática**, como mostrado na Figura 7.1. Como visto anteriormente na Seção 5.4, esse tipo de implementação utiliza um array para armazenar os elementos e tem como desvantagem necessitar que se defina o tamanho do array previamente, o que limita o número de elementos que podemos armazenar. Porém, o fato de utilizarmos um array permite que adotemos a mesma estrutura da fila para duas implementações distintas: **array ordenado** e **heap binária**. Para tanto, basta modificar as funções de inserção, remoção e consulta, como veremos a seguir.

Fila \*fp;

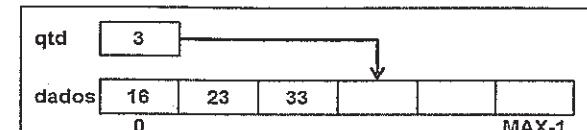


FIGURA 7.1

### 7.4 DEFININDO O TIPO FILA DE PRIORIDADES

Antes de começar a implementar a nossa fila de prioridades, é preciso definir o tipo de dado que será armazenado nela. Uma fila pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa fila. Este tipo será um ponteiro para a estrutura que define a fila de prioridades. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **FilaPrioridade.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **FilaPrioridade.h**, ilustrado na Figura 7.2. Por se tratar de uma fila de prioridades estática, temos que estabelecer:

- O tamanho máximo do array utilizado na fila de prioridades, representado pela constante **MÁX** (linha 1).
- Para fins de padronização, um novo nome para o tipo fila (linha 3). Esse é o tipo que será usado sempre que se desejar trabalhar com uma fila de prioridades.
- As funções disponíveis para se trabalhar com essa fila em especial (linhas 5-12) e que serão implementadas no arquivo **FilaPrioridade.c**.

Neste exemplo, a fila de prioridades irá armazenar o **nome** e a **prioridade** de pacientes esperando por atendimento em um pronto-socorro. Quanto maior a prioridade, mais no início da fila estará aquele paciente.



No arquivo **FilaPrioridade.c**, iremos definir tudo aquilo que deve ficar oculto da usuária da nossa biblioteca e implementar as funções definidas em **FilaPrioridade.h**.

Basicamente, o arquivo **FilaPrioridade.c** (Figura 7.2) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da fila (linhas 1-4).
- O tipo de dado que será armazenado na fila, **struct paciente** (linhas 6-9).

- A definição do tipo que descreve o funcionamento da fila de prioridades, `struct fila_prioridade` (linhas 11-14).
- As implementações das funções definidas no arquivo `FilaPrioridade.h`. As implementações dessas funções serão vistas nas seções seguintes.

Note que o nosso tipo `fila` nada mais é do que uma estrutura contendo dois campos: um inteiro `qtd`, que indica o quanto do array já está ocupado pelos elementos inseridos na fila, e o nosso array do tipo `struct paciente`, que é o tipo de dado a ser armazenado na fila de prioridades. Por estarem definidos dentro do arquivo `.c`, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo `main()`, apenas o seu outro nome, definido no arquivo `FilaPrioridade.h` (linha 3), que pode somente declarar um ponteiro para ele, da seguinte forma:

```
FilaPrio *fp;
```

```
01 #define MAX 100
02
03 typedef struct fila_prioridade FilaPrio;
04
05 FilaPrio* cria_FilaPrio();
06 void libera_FilaPrio(FilaPrio* fp);
07 int consulta_FilaPrio(FilaPrio* fp, char* nome);
08 int insere_FilaPrio(FilaPrio* fp, char *nome, int prio);
09 int remove_FilaPrio(FilaPrio* fp);
10 int tamanho_FilaPrio(FilaPrio* fp);
11 int estaCheia_FilaPrio(FilaPrio* fp);
12 int estaVazia_FilaPrio(FilaPrio* fp);
```

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04 #include "FilaPrioridade.h" //inclui os Protótipos
05
06 struct paciente{
07     char nome[30];
08     int prio;
09 };
10 //Definição do tipo fila de prioridade
11 struct fila_prioridade{
12     int qtd;
13     struct paciente dados[MAX];
14 };
```

FIGURA 7.2

## 7.5 CRIANDO E DESTRUINDO UMA FILA

Para utilizar uma fila de prioridades em seu programa, a primeira coisa a fazer é criar uma fila vazia. Essa tarefa é executada pela função descrita na Figura 7.3. Basicamente, o que esta função faz é a alocação de uma área de memória para a fila, `struct fila_prioridade`. Em seguida, a função inicializa o campo `qtd` com o valor `ZERO`. Este campo indica o quanto do array já está ocupado pelos elementos inseridos na fila de prioridades, que, no caso, mostra que nenhum elemento foi inserido ainda. A Figura 7.4 indica o conteúdo do nosso ponteiro `FilaPrio* fp` após a chamada da função que cria a fila de prioridades.

```
01 FilaPrio* cria_FilaPrio(){
02     FilaPrio *fp;
03     fp = (FilaPrio*) malloc(sizeof(struct fila_prioridade));
04     if(fp != NULL)
05         fp->qtd = 0;
06     return fp;
07 }
```

FIGURA 7.3

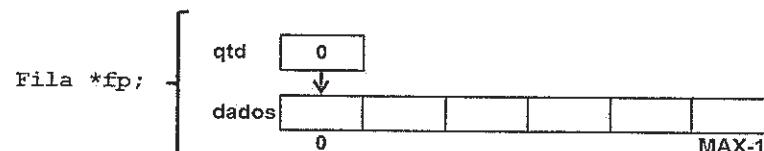


FIGURA 7.4

Destruir uma fila de prioridades estática é bastante simples, como mostra o código contido na Figura 7.5. Basicamente, o que temos que fazer é liberar a memória alocada para a estrutura que representa a fila de prioridades. Isso é feito utilizando apenas uma chamada da função `free()`.

Por que criar uma função para destruir a fila de prioridades sendo que tudo que precisamos fazer é chamar a função `free()`?

Por questões de modularização. Destruir uma fila de prioridades estática é bastante simples, porém destruir uma fila alocada dinamicamente é uma tarefa mais complicada. Ao criar essa função, estamos escondendo a implementação dessa tarefa do usuário, ao mesmo tempo que mantermos a notação utilizada por uma fila de prioridades que fosse implementada utilizando alocação **estática** ou **dinâmica**. Desse modo, utilizar uma fila de prioridades **estática** ou **dinâmica** será indiferente para o programador.

**Destruindo uma fila de prioridade**

```
01 void libera_FilaPrio(FilaPrio* fp){
02     free(fp);
03 }
```

FIGURA 7.5

**7.6 INFORMAÇÕES BÁSICAS SOBRE A FILA****7.6.1 Tamanho da fila**

Saber o tamanho de uma fila de prioridades estática é uma tarefa relativamente simples. Isso ocorre porque essa fila possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na fila de prioridades, como mostra a Figura 7.6.

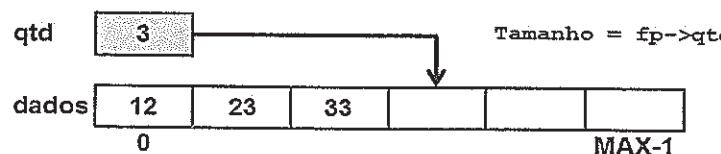


FIGURA 7.6



Basicamente, retornar o tamanho de uma fila de prioridades estática consiste em retornar o valor do seu campo **qtd**.

A implementação da função que retorna o tamanho da fila de prioridades é mostrada na Figura 7.7. Note que essa função, em primeiro lugar, verifica se o ponteiro **FilaPrio\*** **fp** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila de prioridades e, neste caso, não teríamos uma fila válida para trabalhar. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e retornar o seu valor, que nada mais é do que o tamanho da fila de prioridades.

**Tamanho da fila de prioridade**

```
01 int tamanho_FilaPrio(FilaPrio* fp) {
02     if(fp == NULL)
03         return -1;
04     else
05         return fp->qtd;
06 }
```

FIGURA 7.7

**7.6.2 Fila cheia**

Saber se uma fila de prioridades estática está cheia é outra tarefa relativamente simples. Novamente, isso ocorre porque essa fila possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na fila, como mostra a Figura 7.8.



Basicamente, retornar se uma fila de prioridades estática está cheia consiste em verificar se o valor do seu campo **qtd** é igual a **MAX**.

A implementação da função que retorna se a fila de prioridades está cheia é mostrada na Figura 7.9. Note que essa função, em primeiro lugar, verifica se o ponteiro **FilaPrio\*** **fp** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila de prioridades e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma fila inválida. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o tamanho máximo definido para o seu array (vetor) de elementos: **MAX**. Se os valores forem iguais (ou seja, fila cheia), a expressão da linha 4 irá retornar o valor **UM**. Caso contrário, irá retornar o valor **ZERO**.

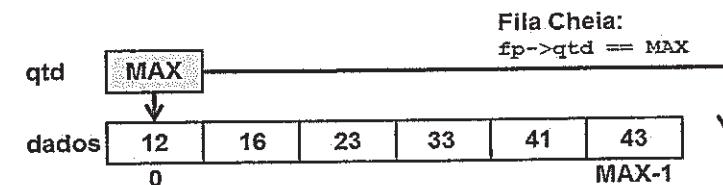


FIGURA 7.8

**Retornando se a fila de prioridade está cheia**

```
01 int estaCheia_FilaPrio(FilaPrio* fp) {
02     if(fp == NULL)
03         return -1;
04     return (fp->qtd == MAX);
05 }
```

FIGURA 7.9

**7.6.3 Fila vazia**

Saber se uma fila de prioridades estática está vazia é outra tarefa bastante simples. Como no caso do tamanho da fila e da fila cheia, isso ocorre porque esse tipo de fila possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na fila de prioridades, como mostra a Figura 7.10.



Basicamente, retornar se uma fila de prioridades estática está vazia consiste em verificar se o valor do seu campo **qtd** é igual a **ZERO**.

A implementação da função que retorna se a fila de prioridades está cheia é mostrada na Figura 7.11. Note que essa função, em primeiro lugar, verifica se o ponteiro **FilaPrio\*** **fp** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da fila de prioridades e, neste caso, não teríamos uma fila válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma fila inválida. Porém, se a fila foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o valor **ZERO**, que é o valor inicial do campo quando criamos uma fila de prioridades. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da fila), a expressão da **linha 4** irá retornar o valor **UM**. Caso contrário, irá retornar o valor **ZERO**.

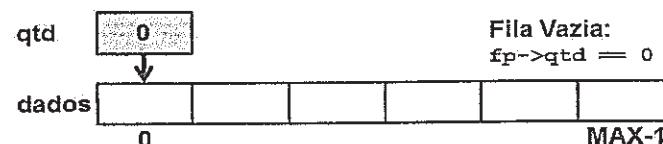


FIGURA 7.10

#### Retornando se a fila de prioridade está vazia

```
01 int estaVazia_FilaPrio(FilaPrio* fp) {
02     if(fp == NULL)
03         return -1;
04     return (fp->qtd == 0);
05 }
```

FIGURA 7.11

## 7.7 FILA DE PRIORIDADE USANDO UM ARRAY ORDENADO

Na implementação usando **array ordenado**, os elementos na fila de prioridade são ordenados de forma crescente dentro do array. Desse modo, o elemento de maior prioridade estará sempre no final do array (índice da fila), enquanto o de menor prioridade estará na primeira posição do array (final da fila). Isso resulta em um custo  $O(N)$  para inserção (precisamos procurar o ponto de inserção no array), sendo a remoção efetuada em tempo constante,  $O(1)$ .

### 7.7.1 Inserindo um elemento na fila de prioridade

Inserir um elemento em uma fila de prioridades implementada usando um **array ordenado** é uma tarefa simples, mas trabalhosa.



Isso ocorre porque precisamos procurar o ponto de inserção do elemento na fila de acordo com a sua prioridade, o qual pode ser no início, no meio ou no final do array. Nos dois primeiros casos (início ou meio), é preciso mudar o lugar dos demais elementos do array.

Basicamente, o que temos que fazer é procurar em que lugar da fila será inserido o novo elemento (lembre-se, estamos ordenando pelo campo **prio**) e, dependendo do lugar, movimentar todos os elementos a partir daquele ponto da fila uma posição para frente dentro do array. Isso deixa aquela posição livre para inserir um novo elemento, como mostra a sua implementação na Figura 7.12. Note que as linhas 2 a 5 verificam se a inserção é possível: se a fila foi criada com sucesso e se ela não está cheia (existe espaço para um novo elemento).

Começamos percorrendo a fila partindo do final do array em direção ao seu começo. Enquanto não tivermos chegado ao início do array ou a prioridade do elemento atual for maior ou igual a prioridade do elemento a ser inserido, realizamos a cópia desse elemento da fila uma posição para frente no array (linhas 7-11). Em seguida, podemos copiar os dados que vamos armazenar para dentro da posição atual (**i+1**) do array que representa a fila (linhas 13-14). Por fim, devemos incrementar a quantidade (**fp->qtd**) de elementos armazenados na fila de prioridades e retornarmos o valor **UM** (linhas 15-16), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 7.13. Nela, os itens hachurados são os elementos de maior prioridade deslocados uma posição para frente (se necessário) para que um novo elemento seja inserido.

#### Inserindo um elemento na fila de prioridade (array ordenado)

```
01 int insere_FilaPrio(FilaPrio* fp, char *nome, int prio) {
02     if(fp == NULL)
03         return 0;
04     if(fp->qtd == MAX)//fila cheia
05         return 0;
06
07     int i = fp->qtd-1;
08     while(i >= 0 && fp->dados[i].prio >= prio) {
09         fp->dados[i+1] = fp->dados[i];
10         i--;
11     }
12
13     strcpy(fp->dados[i+1].nome, nome);
14     fp->dados[i+1].prio = prio;
15     fp->qtd++;
16 }
17 }
```

FIGURA 7.12

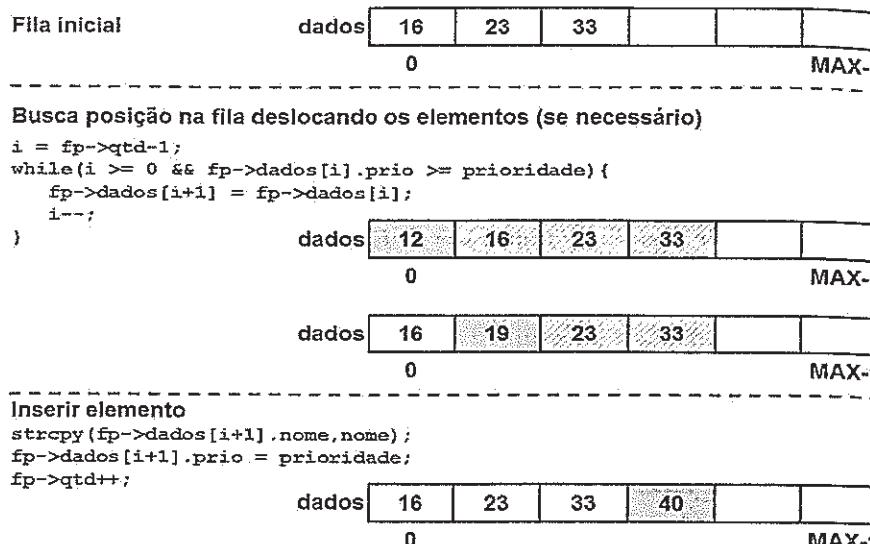


FIGURA 7.13

### 7.7.2 Removendo um elemento da fila de prioridade

Remover um elemento de uma fila de prioridades implementada usando um **array ordenado** é uma tarefa extremamente simples. Basicamente, temos apenas que alterar a quantidade de elementos na fila.

A Figura 7.14 mostra a implementação da função de remoção. Note que as linhas 2 a 5 verificam se a remoção é possível: se a fila foi criada com sucesso e se ela não está vazia (existe pelo menos um elemento para ser removido). Como a remoção ocorre sempre no final do array, basta diminuir em uma unidade a quantidade (**fp->qtd**) de elementos armazenados na fila de prioridades (linha 6). Em seguida, retornamos o valor **UM** (linha 7) para indicar sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 7.15.

#### Removendo um elemento da fila de prioridade (array ordenado)

```
01 int remove_FilaPrio(FilaPrio* fp) {
02     if(fp == NULL)
03         return 0;
04     if(fp->qtd == 0)
05         return 0;
06     fp->qtd--;
07     return 1;
08 }
```

FIGURA 7.14

**i** Note que o elemento removido continua no final da fila de prioridades. Isso não é um problema, já que aquela posição é considerada não ocupada por elementos da fila.

Lembre-se: o campo **qtd** indica a próxima posição vaga no final do array que representa a fila de prioridades.

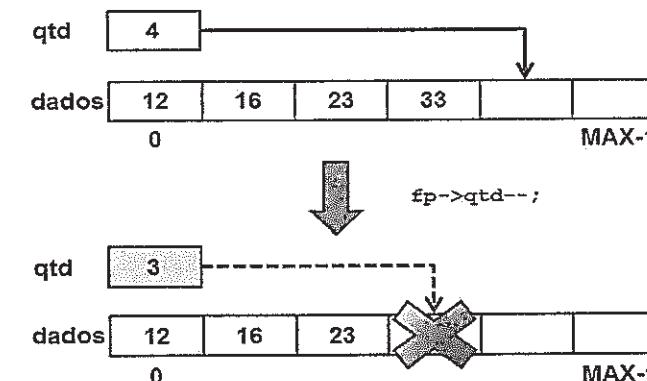


FIGURA 7.15

### 7.7.3 Consultando o elemento no início da fila de prioridade

Apesar de possuir uma implementação quase idêntica à da lista sequencial estática, o acesso a um elemento de uma fila de prioridades estática é um pouco diferente.

**i** Apesar da implementação se parecer com a de uma lista, uma fila de prioridades continua sendo uma fila. Isso significa que podemos acessar as informações apenas do elemento no **início** da fila.

Acessar um elemento que se encontra no início de uma fila de prioridades estática é uma tarefa quase imediata, como mostra a sua implementação na Figura 7.16. Em primeiro lugar, a função verifica se a consulta é válida. Para tanto, duas condições são verificadas: se o ponteiro **FilaPrio\* fp** é igual a **NONE** e se a fila de prioridades está vazia (quantidade de elementos na fila é diferente de zero). Se alguma destas condições for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Caso contrário, a posição equivalente ao “início” da fila é copiada para a string passada por referência (**nome**) para a função (linha 4). Como se nota, esse tipo de consulta consiste em um simples acesso à última posição do array que representa a fila de prioridades (Figura 7.17).



Em uma fila de prioridades implementada utilizando um array ordenado, o **índice** da fila é a última posição do array. Já na implementação usando uma **heap binária**, o **índice** da fila é a **primeira** posição do array.

~~Em uma fila de prioridades implementada utilizando um array ordenado, o índice da fila é a última posição do array. Já na implementação usando uma heap binária, o índice da fila é a primeira posição do array.~~

```
01 int consulta_FilaPrio(FilaPrio* fp, char* nome) {
02     if(fp == NULL || fp->qtd == 0)
03         return 0;
04     strcpy(nome, fp->dados[fp->qtd-1].nome);
05     return 1;
06 }
```

FIGURA 7.16

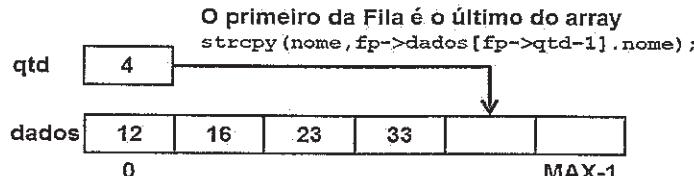


FIGURA 7.17

## 7.8 FILA DE PRIORIDADE USANDO UMA HEAP BINÁRIA

Neste tipo de implementação, utilizamos um array como uma estrutura de dados do tipo heap. Uma heap permite simular uma árvore binária **completa** ou **quase completa** (a exceção é o seu último nível).

O uso de um array como uma heap faz com que cada posição do array passe a ser considerada o pai de duas outras posições, chamadas filhos. Assim, a posição (i) do array passa a ser o pai das posições  $2i + 1$  (filho da esquerda) e  $2i + 2$  (filho da direita). Logo, os elementos na fila de prioridade são dispostos na heap de forma que um nó tem sempre uma prioridade maior ou igual à prioridade de seus filhos. Desse modo, o elemento de maior prioridade estará sempre no **índice** do array (índice da fila), enquanto o de menor prioridade estará na **última** posição do array (final da fila). A Figura 7.18 mostra uma fila de prioridade representada na forma de uma heap binária.

Além disso, a heap binária permite a recuperação e a remoção eficientes dos elementos do array, o que resulta em um custo  $O(\log N)$ , tanto para a inserção quanto para a remoção.



Tanto na inserção quanto na remoção de elementos da fila, precisamos verificar e corrigir violações de propriedades da heap para garantir que a prioridade do nó pai seja sempre maior ou igual a de seus filhos.

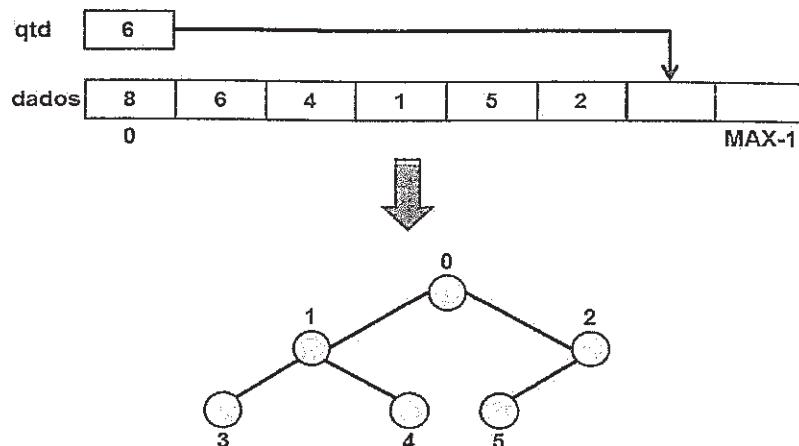


FIGURA 7.18

### 7.8.1 Inserindo um elemento na fila de prioridade

Inserir um elemento em uma fila de prioridades implementada usando uma **heap binária** é uma tarefa simples, mas trabalhosa.



De fato, a inserção é extremamente simples. Porém, terminada a inserção, precisamos verificar e corrigir violações de propriedades da heap para garantir que a prioridade do nó pai seja sempre maior ou igual a de seus filhos.

A Figura 7.19 mostra a implementação da função de inserção. Perceba que a inserção trabalha com duas funções: a função **insere\_FilaPrio**, responsável por gerenciar a inserção (linhas 16-26) e a função **promoverElemento**, que verifica e corrige as violações na heap (linhas 1-15).

Basicamente, a função **insere\_FilaPrio** insere o novo elemento na primeira posição vazia do array, ou seja, no final do array (linhas 21-22). Isso significa que o elemento é inserido como um nó folha na heap. Em seguida, chamamos a função **promoverElemento**, que se encarregará de levar o elemento inserido para a sua respectiva posição na heap, de acordo com a sua prioridade (linha 23). Por fim, devemos incrementar a quantidade (**fp->qtd**) de elementos armazenados na fila de prioridades e retornarmos o valor **UM** (linhas 24-25), indicando sucesso na operação de inserção. Note que as linhas 17 a 20 verificam se a inserção é possível: se a fila foi criada com sucesso e se ela não está cheia (existe espaço para um novo elemento).

A função **promoverElemento** recebe como parâmetros a fila de prioridade e a posição do último elemento inserido, **filho**. Então, a função calcula a posição do **pai** deste elemento (linha 4) e começa a subir a heap verificando se existe alguma violação. Perceba que esse processo de verificação é feito enquanto o valor do **filho** for maior do que **ZERO** (o que

significa que ele possui um **pai**) e a prioridade do **pai** for maior que a do **filho** (linhas 5-6). Se as duas condições forem verdadeiras, os elementos nas posições **pai** e **filho** são trocados de lugar (linhas 8-10). Em seguida, o **pai** será considerado um **filho** e o valor do seu **pai** será calculado para dar continuidade ao processo de subida e verificação da heap (linhas 12-13). Esse processo de inserção e promoção dos elementos dentro da heap pode ser mais bem ilustrado pela Figura 7.20.

#### Inserindo um elemento na fila de prioridade (heap binária)

```

01 void promoverElemento(FilaPrio* fp, int filho){
02     int pai;
03     struct paciente temp;
04     pai = (filho - 1) / 2;
05     while((filho > 0) &&
06           (fp->dados[pai].prio <= fp->dados[filho].prio)){
07
08         temp = fp->dados[filho];
09         fp->dados[filho] = fp->dados[pai];
10         fp->dados[pai] = temp;
11
12         filho = pai;
13         pai = (pai - 1) / 2;
14     }
15 }
16 int insere_FilaPrio(FilaPrio* fp, char *nome, int prio){
17     if(fp == NULL)
18         return 0;
19     if(fp->qtd == MAX)//fila cheia
20         return 0;
21     strcpy(fp->dados[fp->qtd].nome,nome);
22     fp->dados[fp->qtd].prio = prio;
23     promoverElemento(fp,fp->qtd);
24     fp->qtd++;
25     return 1;
26 }
```

FIGURA 7.19

#### Insere elemento com prioridade 9

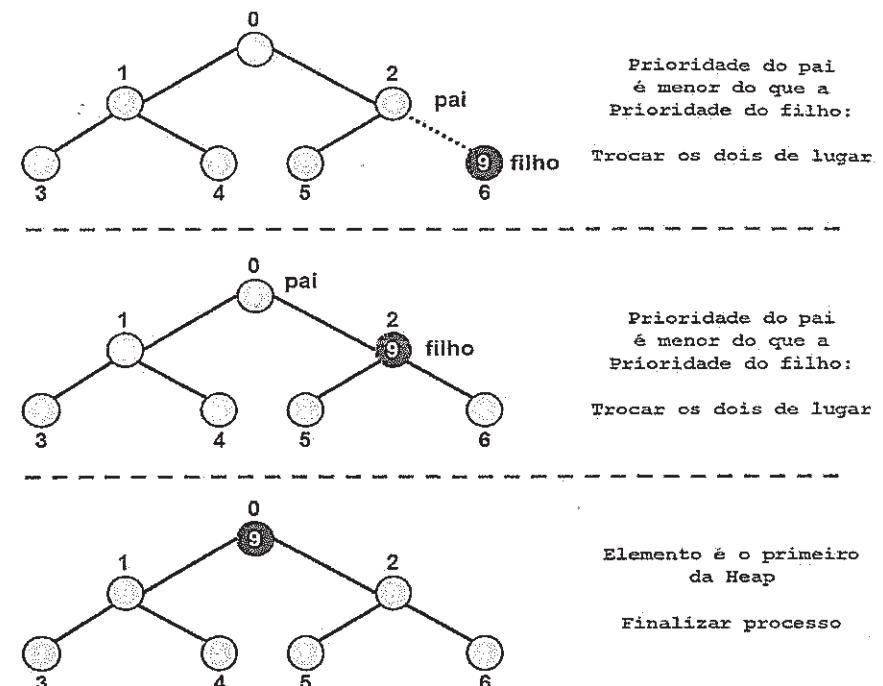


FIGURA 7.20

#### 7.8.2 Removendo um elemento da fila de prioridade

Como a inserção, a remoção de um elemento de uma fila de prioridades implementada usando uma **heap binária** é uma tarefa simples, mas trabalhosa.



Como na inserção, na remoção também precisamos verificar e corrigir violações de propriedades da heap para garantir que a prioridade do nó **pai** seja sempre maior ou igual a de seus filhos.

A Figura 7.21 mostra a implementação da função de remoção. Perceba que a remoção trabalha com duas funções: a função **remove\_FilaPrio**, responsável por gerenciar a remoção (linhas 21-31) e a função **rebaixarElemento**, que verifica e corrige as violações na heap (linhas 1-19).

Basicamente, a função **remove\_FilaPrio** remove o elemento que está no topo da heap, ou seja, no início do array. Isso é feito diminuindo a quantidade (**fp->qtd**) de elementos no array e copiando o elemento do final para o início do array (linhas 27-28). Em seguida, chamamos a função **rebaixarElemento**, que se encarregará de levar o elemento que foi colocado no topo da heap para a sua respectiva posição, de acordo com a sua prioridade (linha 29). Por fim, retornamos o valor **UM** (linha 30), indicando sucesso na operação de remoção. Note que as linhas

22 a 25 verificam se a remoção é possível: se a fila foi criada com sucesso e se ela não está vazia (existe pelo menos um elemento para ser removido).

A função **rebaixarElemento** recebe como parâmetros a fila de prioridade e a posição do elemento que está no topo da heap, **pai**. Note que, inicialmente, o valor de **pai** é igual a **ZERO**. Então, a função calcula o seu primeiro **filho** (linha 3) e verifica se a posição do filho está no array (linha 4). Em caso afirmativo, a função verifica se existe o segundo **filho** (linha 5). Se o segundo filho existir, é necessário selecionar o maior deles (linhas 6-7). Do contrário, o primeiro será considerado o maior. Tendo sido selecionado o maior **filho**, é necessário comparar as prioridades do **pai** e do **filho** (linha 9). Se a prioridade do **pai** é maior ou igual à do **filho**, isso significa que a heap está correta e o processo termina (linha 10). Do contrário, os elementos nas posições **pai** e **filho** são trocados de lugar (linhas 12-14). Em seguida, o **filho** será considerado um **pai** e o valor do seu **filho** será calculado para dar continuidade ao processo de descida e verificação da heap (linhas 16-17). Esse processo de remoção e rebaixamento dos elementos dentro da heap pode ser mais bem ilustrado pela Figura 7.22.

#### Removendo um elemento da fila de prioridade (heap dinâmica)

```

01 void rebaixarElemento(FilaPrio* fp, int pai){
02     struct paciente temp;
03     int filho = 2 * pai + 1;
04     while(filho < fp->qtd){
05         if(filho < fp->qtd-1)
06             if(fp->dados[filho].prio < fp->dados[filho+1].prio)
07                 filho++;
08
09         if(fp->dados[pai].prio >= fp->dados[filho].prio)
10             break;
11
12         temp = fp->dados[pai];
13         fp->dados[pai] = fp->dados[filho];
14         fp->dados[filho] = temp;
15
16         pai = filho;
17         filho = 2 * pai + 1;
18     }
19 }
20
21 int remove_FilaPrio(FilaPrio* fp){
22     if(fp == NULL)
23         return 0;
24     if(fp->qtd == 0)
25         return 0;
26
27     fp->qtd--;
28     fp->dados[0] = fp->dados[fp->qtd];
29     rebaixarElemento(fp,0);
30
31 }
```

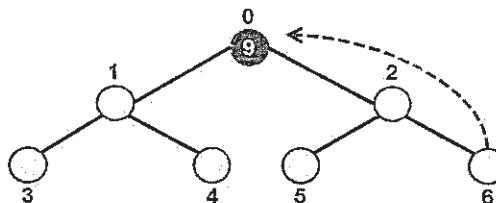
FIGURA 7.21



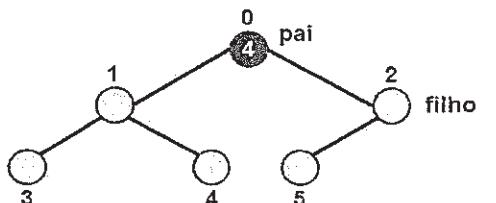
Note que o elemento removido continua no final da fila de prioridades. Isso não é um problema, já que aquela posição é considerada não ocupada por elementos da fila.

Lembre-se: o campo **qtd** indica a próxima posição vaga no final do array que representa a fila de prioridades.

#### Remove elemento com maior prioridade

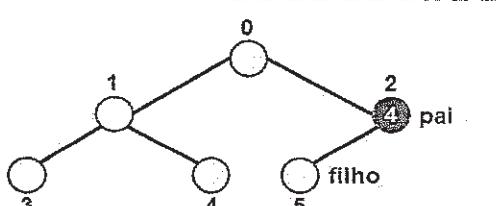


Copia o último elemento da Heap para o topo  
Iniciar ajuste da Heap



Prioridade do pai é menor do que a Prioridade do filho:

Trocar os dois de lugar



Prioridade do pai é maior do que a Prioridade do filho:

Finalizar processo

FIGURA 7.22

#### 7.8.3 Consultando o elemento no início da fila de prioridade

Apesar de possuir uma implementação quase idêntica à da lista sequencial estática, o acesso a um elemento de uma fila de prioridades estática é um pouco diferente.



Apesar da implementação se parecer com a de uma lista, uma fila de prioridades continua sendo uma fila. Isso significa que podemos acessar as informações apenas do elemento no **início** da fila.

Acessar um elemento que se encontra no início de uma fila de prioridades estática é uma tarefa quase imediata, como mostra a sua implementação na Figura 7.23. Em primeiro lugar, a função verifica se a consulta é válida. Para tanto, duas condições são verificadas: se o ponteiro **FilaPrio\* fp** é igual a **NULL** e se a fila de prioridades está vazia (quantidade de elementos na fila é igual a zero). Se alguma destas condições for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Caso contrário, a posição equivalente ao “início” da fila é copiada para a string passada por referência (**nome**) para a função (linha 4). Como se nota, esse tipo de consulta consiste em um simples acesso à primeira posição do array que representa a fila de prioridades (Figura 7.24).



Em uma fila de prioridades implementada utilizando um **array ordenado**, o **início** da fila é a **última posição** do array. Já na implementação usando uma **heap binária**, o **início** da fila é a **primeira posição** do array.

```
Retorna o elemento com maior prioridade da fila (heap binária)
01 int consulta_FilaPrio(FilaPrio* fp, char* nome) {
02     if(fp == NULL || fp->qtd == 0)
03         return 0;
04     strcpy(nome, fp->dados[0].nome);
05     return 1;
06 }
```

FIGURA 7.23

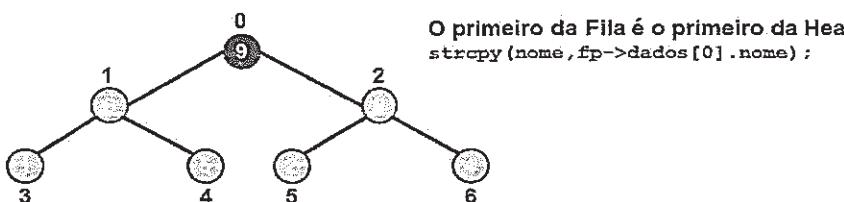


FIGURA 7.24

## 7.9 EXERCÍCIOS

- 1) Defina, usando as suas palavras, a diferença entre fila e fila de prioridade.
- 2) Defina, usando as suas palavras, o que é uma estrutura do tipo fila de prioridades.
- 3) Defina, usando as suas palavras, a diferença da fila de prioridades usando array ordenado e heap binária.
- 4) Liste as situações em que uma fila de prioridades pode ser utilizada.

- 5) Implemente uma função que receba uma fila de prioridades e a inverta (o elemento de maior prioridade passa a ter a menor). Faça isso para os dois tipos de implementação: array ordenado e heap binária.
- 6) Escreva uma função para achar o valor de maior prioridade da fila. Faça isso para os dois tipos de implementação: array ordenado e heap binária.
- 7) Implemente uma função que receba duas filas (F1 e F2) e as concatene. O resultado da concatenação deve ser colocado em F1. A fila F2 deve ficar vazia. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 8) Implemente uma função para unir filas de prioridade. A função receberá duas filas (F1 e F2) e o resultado da união deve ser colocado em F1, sem repetições. A fila F2 deve ficar vazia. Faça a função para ambos os tipos de fila: estática e dinâmica.
- 9) Implemente uma função que copie os elementos de uma fila de prioridades F1 para uma fila F2.

# Pilhas

## 8.1 DEFINIÇÃO

O conceito de pilha é algo bastante comum para as pessoas. Trata-se de um conjunto finito de itens sobre um mesmo tema. Mas, diferente das listas, os itens de uma pilha se encontram dispostos uns sobre os outros. Assim, somente podemos inserir um novo item na pilha se o colocarmos acima dos demais e apenas removeremos o item que está no topo da pilha.



As pilhas são implementadas e se comportam de modo muito similar às listas, sendo, muitas vezes, consideradas um tipo especial de lista em que a inserção e a remoção são realizadas sempre na mesma extremidade.

Desse modo, se quisermos acessar determinado elemento da pilha, deveremos remover todos os que estiverem sobre ele. Por esse motivo, as pilhas são conhecidas como estruturas do tipo último a entrar, primeiro a sair ou LIFO (Last In First Out): os elementos são removidos da pilha na ordem inversa daquela em que foram inseridos. Vários são os exemplos possíveis de pilhas no dia a dia: pilha de livros, pilhas de documentos sobre uma mesa, pilha de pratos etc.

Em ciência da computação, uma pilha é uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador. Uma estrutura do tipo pilha é uma sequência de elementos do mesmo tipo, como ilustrado na Figura 8.1. Seus elementos possuem estrutura

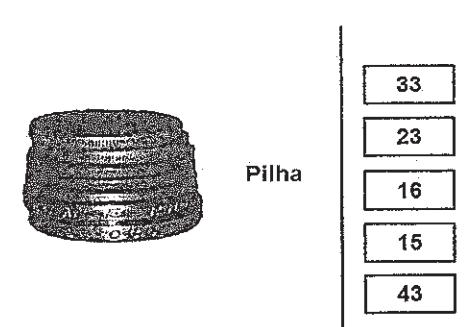


FIGURA 8.1

interna abstraída, ou seja, sua complexidade é arbitrária e não afeta o seu funcionamento. Além disso, uma pilha pode possuir elementos repetidos, dependendo da aplicação. Uma estrutura do tipo pilha pode possuir  $N$  ( $N \geq 0$ ) elementos ou itens. Se  $N = 0$ , dizemos que a pilha está vazia.

## 8.2 TIPOS DE PILHAS

Basicamente, existem dois tipos de implementações principais para uma pilha.



Estas implementações diferem entre si com relação ao tipo de alocação de memória usada e ao tipo de acesso aos elementos.

Uma pilha pode ser implementada usando **alocação estática com acesso sequencial** ou **alocação dinâmica com acesso encadeado**, como descrito a seguir:

- **Alociação estática com acesso sequencial:** o espaço de memória é alocado no momento da compilação do programa, ou seja, é necessário definir o número máximo de elementos que a pilha irá possuir. Desse modo, os elementos são armazenados de forma consecutiva na memória (como em um array ou vetor) e a posição de um elemento pode ser facilmente obtida a partir do início da pilha.
- **Alociação dinâmica com acesso encadeado:** o espaço de memória é alocado em tempo de execução, ou seja, a pilha cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos. Nessa implementação, cada elemento pode estar em uma área distinta da memória, não necessariamente consecutivas. É necessário então que cada elemento da pilha armazene, além da sua informação, o endereço de memória onde se encontra o próximo elemento. Para acessar um elemento, é preciso percorrer todos os seus antecessores na pilha.

Nas próximas seções, serão apresentadas as diferentes implementações de pilhas com relação à alocação e ao acesso aos elementos.

## 8.3 OPERAÇÕES BÁSICAS DE UMA PILHA

Independentemente do tipo de alocação e acesso usado na implementação de uma pilha, as seguintes operações básicas são sempre possíveis:

- Criação da pilha.
- Inserção de um elemento no topo da pilha.
- Remoção de um elemento do topo da pilha.
- Acesso ao elemento do topo da pilha.
- Destrução da pilha.
- Além de informações com tamanho, se a pilha está cheia ou vazia.

## 8.4 PILHA SEQUENCIAL ESTÁTICA

Uma **pilha sequencial estática** ou **pilha linear estática** é uma pilha definida utilizando alocação estática e acesso sequencial dos elementos. Trata-se do tipo mais simples de pilha possível. Ela é definida utilizando um array, de modo que o sucessor de um elemento ocupa a posição física seguinte deste.



Além do array, essa pilha utiliza um campo adicional (**qtd**) que serve para indicar o quanto do array já está ocupado pelos elementos (**dados**) inseridos na pilha.

Em uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo pilha (ela é um **tipo opaco**), como ilustrado na Figura 8.2. Isso impede o usuário de saber como foi realmente implementada a pilha, e limita o seu acesso apenas às funções que manipulam o topo da pilha.

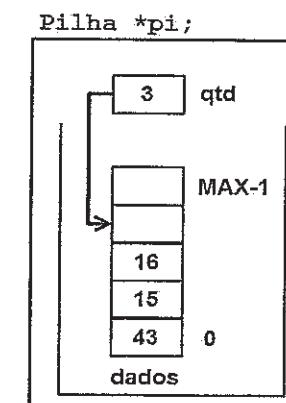


FIGURA 8.2



A principal vantagem de se utilizar um array na definição de uma **pilha sequencial estática** é a facilidade de criar e destruir a pilha. Já a sua principal desvantagem é a necessidade de definir previamente o tamanho do array e, consequentemente, da pilha.

Considerando suas vantagens e desvantagens, o ideal é utilizar uma **pilha sequencial estática** em pilha pequena ou quando o tamanho máximo da pilha é bem definido.

### 8.4.1 Definindo o tipo pilha sequencial estática

Antes de começar a implementar a nossa pilha, é preciso definir o tipo de dado que será armazenado nela. Uma pilha pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa pilha. Este tipo será um

ponteiro para a estrutura que define a pilha. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **PilhaSequencial.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **PilhaSequencial.h**, ilustrado na Figura 8.3. Por se tratar de uma pilha estática, temos que estabelecer:

- O tamanho máximo do array utilizado na pilha, representada pela constante **MAX** (linha 1).
- O tipo de dado que será armazenado na pilha, **struct aluno** (linhas 2-6).
- Para fins de padronização, um novo nome para o tipo pilha (linha 7). Esse é o tipo que será usado sempre que se desejar trabalhar com uma pilha.
- As funções disponíveis para se trabalhar com essa pilha em especial (linhas 9-16) e que serão implementadas no arquivo **PilhaSequencial.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da pilha. Este aluno é representado pelo seu número de matrícula, nome e três notas.



No arquivo **PilhaSequencial.c**, iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em **PilhaSequencial.h**.

Basicamente, o arquivo **PilhaSequencial.c** (Figura 8.3) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da pilha (linhas 1-3).
- A definição do tipo que descreve o funcionamento da pilha, **struct pilha** (linhas 5-8).
- As implementações das funções definidas no arquivo **PilhaSequencial.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que o nosso tipo pilha nada mais é do que uma estrutura contendo dois campos: um inteiro **qtd**, que indica o quanto do array já está ocupado pelos elementos inseridos na pilha, e o nosso array do tipo **struct aluno**, que é o tipo de dado a ser armazenado na pilha. Por estarem definidos dentro do arquivo **.c**, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **PilhaSequencial.h** (linha 7), que pode somente declarar um ponteiro para ele, da seguinte forma:

```
Pilha *pi;
```



Note que a implementação de uma **pilha sequencial estática** é exatamente igual à implementação de uma **lista sequencial estática**. A diferença é que uma pilha apenas permite um único tipo de inserção e remoção.

### Arquivo PilhaSequencial.h

```
01 #define MAX 100
02 struct aluno{
03     int matricula;
04     char nome[30];
05     float n1,n2,n3;
06 };
07 typedef struct pilha Pilha;
08
09 Pilha* cria_Pilha();
10 void libera_Pilha(Pilha* pi);
11 int acessa_topo_Pilha(Pilha* pi, struct aluno *al);
12 int insere_Pilha(Pilha* pi, struct aluno al);
13 int remove_Pilha(Pilha* pi);
14 int tamanho_Pilha(Pilha* pi);
15 int Pilha_vazia(Pilha* pi);
16 int Pilha_cheia(Pilha* pi);
```

### Arquivo PilhaSequencial.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "PilhaSequencial.h" //inclui os protótipos
04 //Definição do tipo Pilha
05 struct pilha{
06     int qtd;
07     struct aluno dados[MAX];
08 };
```

FIGURA 8.3

#### 8.4.2 Criando e destruindo uma pilha

Para utilizar uma pilha em seu programa, a primeira coisa a fazer é criar uma pilha vazia. Essa tarefa é executada pela função descrita na Figura 8.4. Basicamente, o que esta função faz é a alocação de uma área de memória para a pilha (linha 3). Esta área de memória corresponde à memória necessária para armazenar a estrutura que define a pilha, **struct pilha**. Em seguida, essa função inicializa o campo **qtd** com o valor **ZERO**. Este campo indica o quanto do array já

### Criando uma pilha

```
01 Pilha* cria_Pilha(){
02     Pilha *pi;
03     pi = (Pilha*) malloc(sizeof(struct pilha));
04     if(pi != NULL)
05         pi->qtd = 0;
06     return pi;
07 }
```

FIGURA 8.4

está ocupado pelos elementos inseridos na pilha, que, no caso, mostra que nenhum elemento foi inserido ainda. A Figura 8.5 indica o conteúdo do nosso ponteiro `Pilha* pi` após a chamada da função que cria a pilha.

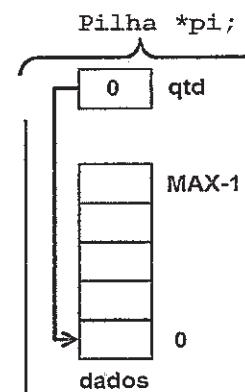


FIGURA 8.5

Destruir uma pilha estática é bastante simples, como mostra o código contido na Figura 8.6. Basicamente, o que temos que fazer é liberar a memória alocada para a estrutura que representa a pilha. Isso é feito utilizando apenas uma chamada da função `free()`.



Por que criar uma função para destruir a pilha sendo que tudo que precisamos fazer é chamar a função `free()`?

Por questões de modularização. Destruir uma pilha estática é bastante simples, porém destruir uma pilha alocada dinamicamente é uma tarefa mais complicada. Ao criar essa função, estamos escondendo a implementação dessa tarefa do usuário, ao mesmo tempo em que mantemos a mesma notação utilizada por uma pilha com alocação **estática** ou **dinâmica**. Desse modo, utilizar uma pilha **estática** ou **dinâmica** será indiferente para o programador.

#### Destruindo uma pilha

```
01 void libera_Pilha(Pilha* pi){
02     free(pi);
03 }
```

FIGURA 8.6

#### 8.4.3 Informações básicas sobre a pilha

As operações de inserção, remoção e acesso ao topo são consideradas as principais de uma pilha. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais

básicas sobre a pilha. Por exemplo, não podemos remover um elemento da pilha se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da pilha: seu tamanho, se ela está cheia ou se ela está vazia.

#### Tamanho da pilha

Saber o tamanho de uma **pilha sequencial estática** é uma tarefa relativamente simples. Isso ocorre porque essa pilha possui um campo inteiro `qtd` que indica o quanto do array já está ocupado pelos elementos inseridos na pilha, como mostra a Figura 8.7.



Basicamente, retornar o tamanho de uma **pilha sequencial estática** consiste em retornar o valor do seu campo `qtd`.

A implementação da função que retorna o tamanho da pilha é mostrada na Figura 8.8. Note que essa função, em primeiro lugar, verifica se o ponteiro `Pilha* pi` é igual a `NULL`. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Porém, se a pilha foi criada com sucesso, então é possível acessar o seu campo `qtd` e retornar o seu valor, que nada mais é do que o tamanho da pilha.

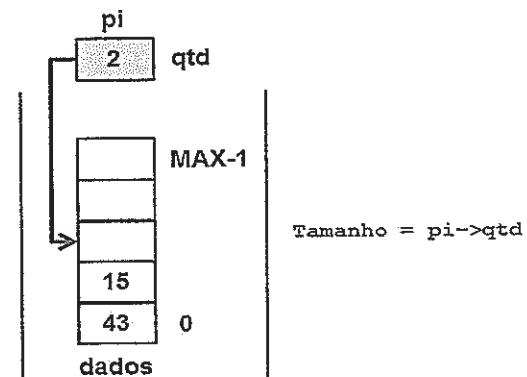


FIGURA 8.7

#### Tamanho da pilha

```
01 int tamanho_Pilha(Pilha* pi){
02     if(pi == NULL)
03         return -1;
04     else
05         return pi->qtd;
06 }
```

FIGURA 8.8

### Pilha cheia

Saber se uma pilha sequencial estática está cheia é outra tarefa relativamente simples. Note que isso ocorre porque essa pilha possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na pilha, como mostra a Figura 8.9.



Basicamente, retornar se uma pilha sequencial estática está cheia consiste em verificar se o valor do seu campo **qtd** é igual a **MAX**.

A implementação da função que retorna se a pilha está cheia é mostrada na Figura 8.10. Note que essa função, em primeiro lugar, verifica se o ponteiro **Pilha\*** **pi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma pilha inválida. Porém, se a pilha foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o tamanho máximo definido para o seu array (vetor) de elementos: **MAX**. Se os valores forem iguais (ou seja, pilha cheia), a expressão da **linha 4** irá retornar o valor **UM**. Caso contrário, irá retornar o valor **ZERO**.

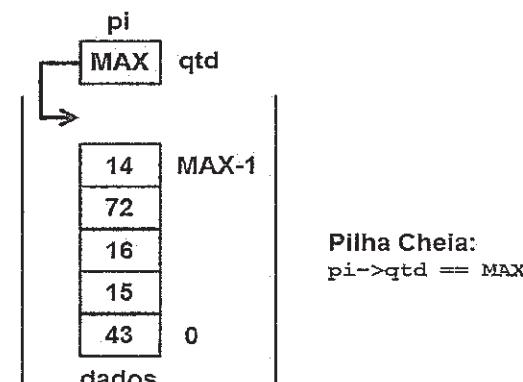


FIGURA 8.9

#### Retornando se a pilha está cheia:

```
01 int Pilha_cheria(Pilha* pi){
02     if(pi == NULL)
03         return -1;
04     return (pi->qtd == MAX);
05 }
```

FIGURA 8.10

### Pilha vazia

Saber se uma pilha sequencial estática está vazia é outra tarefa bastante simples. Como no caso do tamanho da pilha e da pilha cheia, isso ocorre porque esse tipo de pilha possui um campo inteiro **qtd** que indica o quanto do array já está ocupado pelos elementos inseridos na pilha, como mostra a Figura 8.11.



Basicamente, retornar se uma pilha sequencial estática está vazia consiste em verificar se o valor do seu campo **qtd** é igual a **ZERO**.

A implementação da função que retorna se a pilha está cheia é mostrada na Figura 8.12. Note que essa função, em primeiro lugar, verifica se o ponteiro **Pilha\*** **pi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Assim, optamos por retornar o valor **-1** para indicar uma pilha inválida. Porém, se a pilha foi criada com sucesso, então é possível acessar o seu campo **qtd** e comparar o seu valor com o valor **ZERO**, que é o valor inicial do campo quando criamos uma pilha. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da pilha), a expressão da **linha 4** irá retornar o valor **UM**. Caso contrário, irá retornar o valor **ZERO**.

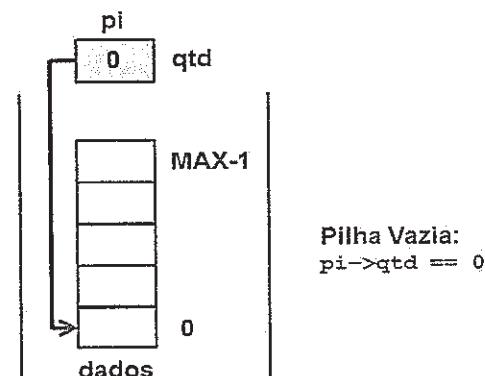


FIGURA 8.11

#### Retornando se a pilha está vazia:

```
01 int Pilha_vazia(Pilha* pi){
02     if(pi == NULL)
03         return -1;
04     return (pi->qtd == 0);
05 }
```

FIGURA 8.12

#### 8.4.4 Inserindo um elemento na pilha

Inserir um elemento no topo de uma **pilha sequencial estática** é uma tarefa extremamente simples, como mostra a sua implementação na Figura 8.13. Primeiro, a função verifica se o ponteiro **Pilha\* pi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma pilha inválida (linha 3). Porém, se a pilha foi criada com sucesso, precisamos verificar se ela não está cheia, isto é, se existe espaço para um novo elemento. Caso a pilha esteja cheia, a função irá retornar o valor **ZERO** (linhas 4 e 5).

Até aqui, o que fizemos foi verificar se podíamos inserir um elemento na pilha. Falta, obviamente, inserir este elemento. Para tanto, basta copiar os dados que vamos armazenar para dentro da posição **pi->qtd** do array que representa a pilha, **dados** (linha 6). Por fim, devemos incrementar a quantidade (**pi->qtd**) de elementos armazenados na pilha e retornarmos o valor **UM** (linhas 7 e 8), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 8.14.



Note que a inserção no topo de uma **pilha sequencial estática** é equivalente à inserção no final de uma **lista sequencial estática**.

#### Inserindo um elemento na pilha

```

01 int insere_Pilha(Pilha* pi, struct aluno al){
02     if(pi == NULL)
03         return 0;
04     if(pi->qtd == MAX)//pilha cheia
05         return 0;
06     pi->dados[pi->qtd] = al;
07     pi->qtd++;
08     return 1;
09 }
```

FIGURA 8.13

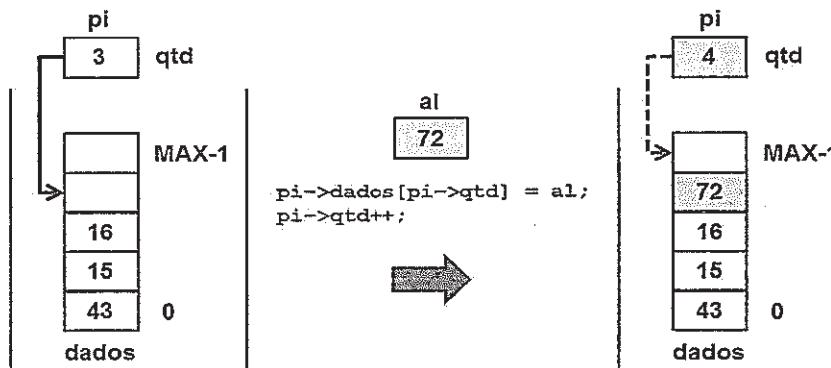


FIGURA 8.14

#### 8.4.5 Removendo um elemento da pilha

Remover um elemento do topo de uma **pilha sequencial estática** é uma tarefa extremamente simples, como mostra a sua implementação na Figura 8.15. Primeiro, a função verifica se o ponteiro **Pilha\* pi** é igual a **NULL** ou se a pilha está vazia (linha 2). Essas condições garantem que temos uma pilha válida para trabalhar (ou seja, não houve problemas na criação da pilha) e que existem elementos que podem ser removidos da pilha. Neste caso, optamos por retornar o valor **ZERO** para indicar que uma das condições é verdadeira (linha 3). Como a remoção é feita no topo da pilha, basta diminuir em uma unidade o valor do campo quantidade (linha 4). Por fim, retornarmos o valor **UM** (linha 5), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 8.16.



Note que a remoção do topo de uma **pilha sequencial estática** é equivalente à remoção do final de uma **lista sequencial estática**.

#### Removendo um elemento da pilha

```

01 int remove_Pilha(Pilha* pi){
02     if(pi == NULL || pi->qtd == 0)
03         return 0;
04     pi->qtd--;
05     return 1;
06 }
```

FIGURA 8.15

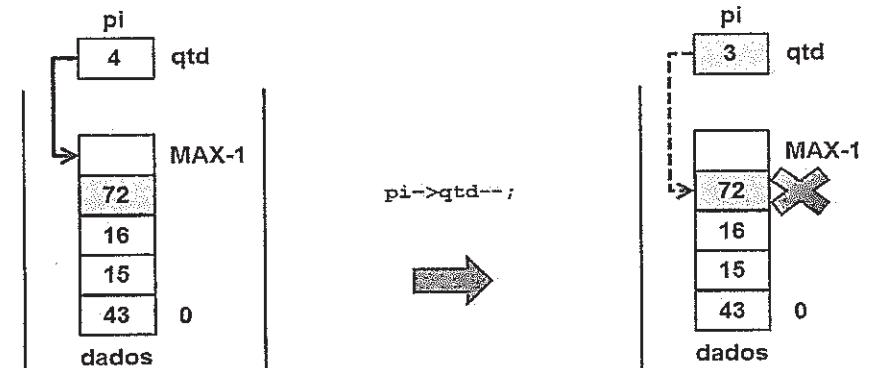


FIGURA 8.16

#### 8.4.6 Acessando o elemento no topo da pilha

Apesar de possuir uma implementação quase idêntica à da **lista sequencial estática**, o acesso a um elemento de uma **pilha sequencial estática** é um pouco diferente.



Em uma lista, pode-se acessar e recuperar as informações contidas em qualquer um dos seus elementos. Já em uma pilha, podemos acessar as informações apenas do elemento no **topo** da pilha.

Acessar um elemento que se encontra no topo de uma **pilha sequencial estática** é uma tarefa quase imediata, como mostra a sua implementação na Figura 8.17. Em primeiro lugar, a função verifica se a consulta é válida. Para tanto, duas condições são verificadas: se o ponteiro **Pilha\*** *pi* é igual a **NULL** e se a pilha está vazia (quantidade de elementos na pilha é diferente de zero). Se alguma destas condições for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Caso contrário, a posição equivalente ao “topo” da pilha é copiada para o conteúdo do ponteiro passado por referência (*al*) para a função (linha 4). Como se nota, esse tipo de consulta consiste em um simples acesso à última posição inserida no array que representa a pilha (Figura 8.18).

#### Acessando o topo da pilha

```
01 int acessa_topo_Pilha(Pilha* pi, struct aluno *al) {
02     if(pi == NULL || pi->qtd == 0)
03         return 0;
04     *al = pi->dados[pi->qtd-1];
05     return 1;
06 }
```

FIGURA 8.17

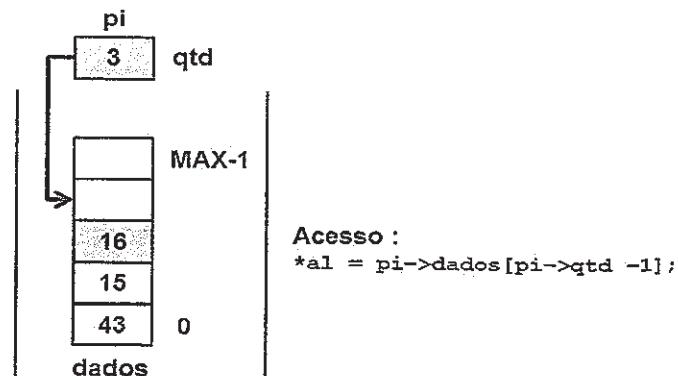


FIGURA 8.18

#### 8.4.7 Análise de complexidade

Um aspecto importante quando manipulamos pilhas tem relação com os custos das suas operações. Em uma **pilha sequencial estática** as operações de inserção, remoção e consulta envolvem apenas a manipulação de alguns índices, independentemente do número de elementos na pilha. Desse modo, a complexidade dessas operações é  $O(1)$ .

### 8.5 PILHA DINÂMICA ENCADEADA

Uma **pilha dinâmica encadeada** é uma pilha definida utilizando alocação dinâmica e acesso encadeado dos elementos. Cada elemento da pilha é alocado dinamicamente, à medida que os dados são inseridos no topo da pilha, e tem sua memória liberada, à medida que são removidos. Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação: um campo de **dado**, utilizado para armazenar a informação inserida na pilha, e um campo **prox**, que nada mais é do que um ponteiro que indica o próximo elemento na pilha.



Além da estrutura que define seus elementos, essa pilha utiliza um **ponteiro para ponteiro** para guardar o primeiro elemento ou “topo” da pilha.

Temos em uma **pilha dinâmica encadeada** que todos os seus elementos são ponteiros alocados dinamicamente. Para inserir um elemento no topo ou início da pilha é necessário utilizar um campo que seja fixo, mas que, ao mesmo tempo, seja capaz de apontar para o novo elemento.

É necessário o uso de um **ponteiro para ponteiro** para guardar o endereço de um **ponteiro**. Utilizando um **ponteiro para ponteiro** para representar o topo da pilha, fica fácil mudar quem é o primeiro elemento da pilha alterando apenas o **conteúdo do ponteiro para ponteiro**. Mais detalhes são apresentados na Seção 5.5.1.



Após o último elemento, não existe nenhum novo elemento alocado. Sendo assim, o último elemento da pilha aponta para **NULL**.

Em uma implementação em módulos, temos que o usuário tem acesso apenas a um ponteiro do tipo pilha (ela é um **tipo opaco**), como ilustrado na Figura 8.19. Isso impede o usuário de saber como foi realmente implementada a pilha, e limita o seu acesso apenas às funções que manipulam a pilha.

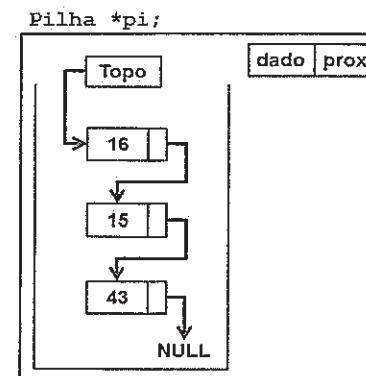


FIGURA 8.19



Note que a implementação em módulos impede o usuário de saber como a pilha foi implementada. Tanto a **pilha dinâmica encadeada** quanto a **pilha sequencial estática** são declaradas como sendo do tipo **Pilha \***.

Essa é a grande vantagem da modularização e da utilização de tipos opacos: mudar a maneira pela qual a pilha foi implementada não altera nem interfere no funcionamento do programa que a utiliza.



A principal vantagem de se utilizar uma abordagem dinâmica e encadeada na definição da pilha é a melhor utilização dos recursos de memória, não sendo mais necessário definir previamente o tamanho da pilha. Já a sua principal desvantagem é a necessidade de percorrer toda a pilha para destruí-la.

Considerando suas vantagens e desvantagens, o ideal é utilizar uma **pilha dinâmica encadeada** quando não há necessidade de garantir um espaço mínimo para a execução da aplicação, ou quando o tamanho máximo da pilha não é bem definido.

### 8.5.1 Definindo o tipo pilha dinâmica encadeada

Antes de começar a implementar a nossa pilha, é preciso definir o tipo de dado que será armazenado nela. Uma pilha pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa pilha. E, trabalhando com alocação dinâmica da pilha, este tipo será um **ponteiro para ponteiro** da estrutura que define a pilha. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **PilhaDin.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **PilhaDin.h**, ilustrado na Figura 8.20. Por se tratar de uma pilha dinâmica encadeada, temos que estabelecer:

- O tipo de dado que será armazenado na pilha, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para o **ponteiro** do tipo pilha (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma pilha.
- As funções disponíveis para se trabalhar com essa pilha em especial (linhas 8-15) e que serão implementadas no arquivo **PilhaDin.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da pilha. Este aluno é identificado pelo seu número de matrícula, nome e três notas.



Por que colocamos um ponteiro no comando **typedef** quando criamos um novo nome para o tipo (linha 6)?

Por estarmos trabalhando com uma pilha dinâmica e encadeada, temos que fazê-lo com um ponteiro para ponteiro a fim de poder realizar modificações no topo da pilha. Por questões de modularização e para manter a mesma notação utilizada pela **pilha sequencial estática**, podemos esconder um dos ponteiros do usuário. Assim, utilizar uma **pilha sequencial estática** ou uma **pilha dinâmica encadeada** será indiferente para o programador, pois a sua implementação está escondida dele:

- Pilha \*pi; //Declaração de uma Pilha Sequencial Estática (ponteiro).
- Pilha \*pi; //Declaração de uma Pilha Dinâmica Encadeada (ponteiro para ponteiro).



No arquivo **PilhaDin.c**, iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **PilhaDin.h**.

Basicamente, o arquivo **PilhaDin.c** (Figura 8.20) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da pilha (linhas 1-3).
- A definição do tipo que descreve cada elemento da pilha, **struct elemento** (linhas 5-8).
- A definição de um novo nome para a **struct elemento** (linhas 9). Isso é feito apenas para facilitar certas etapas de codificação.
- As implementações das funções definidas no arquivo **PilhaDin.h**. As implementações dessas funções serão vistas nas seções seguintes.

```

 01 struct aluno{
 02     int matricula;
 03     char nome[30];
 04     float n1,n2,n3;
 05 };
 06 typedef struct elemento* Pilha;
 07
 08 Pilha* cria_Pilha();
 09 void libera_Pilha(Pilha* pi);
10 int acessa_topo_Pilha(Pilha* pi, struct aluno *al);
11 int insere_Pilha(Pilha* pi, struct aluno al);
12 int remove_Pilha(Pilha* pi);
13 int tamanho_Pilha(Pilha* pi);
14 int Pilha_vazia(Pilha* pi);
15 int Pilha_cheia(Pilha* pi);

```

```

 01 #include <stdio.h>
 02 #include <stdlib.h>
 03 #include "PilhaDin.h" //inclui os protótipos
 04 //Definição do tipo Pilha
 05 struct elemento{
 06     struct aluno dados;
 07     struct elemento *prox;
 08 };
 09 typedef struct elemento Elemt;

```

FIGURA 8.20

Note que a nossa estrutura **elemento** nada mais é do que uma estrutura contendo dois campos: um ponteiro **prox**, que indica o próximo elemento (também do tipo **struct elemento**) dentro da pilha, e um campo **dados** do tipo **struct aluno**, que é o tipo de dado a ser armazenado na pilha. Por estarem definidos dentro do arquivo **.c**, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **PilhaDin.h** (linha 6), que pode somente declarar um ponteiro para ele, da seguinte forma:

```
Pilha *pi;
```



Note que a implementação de uma **pilha dinâmica encadeada** é exatamente igual à implementação de uma **lista dinâmica encadeada**. A diferença é que uma pilha apenas permite um único tipo de inserção e remoção.

## 8.5.2 Criando e destruindo uma pilha

Para utilizar uma pilha em seu programa, a primeira coisa a fazer é criar uma pilha vazia. Essa tarefa é executada pela função descrita na Figura 8.21. Basicamente, o que esta função faz é

alocar de uma área de memória para armazenar o endereço do início da pilha (linha 2), que é um **ponteiro para ponteiro**. Esta área de memória corresponde à memória necessária para armazenar o endereço de um elemento da pilha, **sizeof(Pilha)** ou **sizeof(struct elemento\*)**. Em seguida, a função inicializa o conteúdo desse **ponteiro para ponteiro** com a constante **NULL**. Esta constante é utilizada em uma **pilha dinâmica encadeada** para indicar que não existe nenhum elemento alocado após o atual. Como o início da pilha aponta para a constante, isso significa que a pilha está vazia. A Figura 8.22 indica o conteúdo do nosso **Pilha\*** **pi** após a chamada da função que cria a pilha.

```

 01 //Arquivo PilhaDin.c
 02
 03 Pilha* cria_Pilha();
 04 void libera_Pilha(Pilha* pi);
 05 int acessa_topo_Pilha(Pilha* pi, struct aluno *al);
 06 int insere_Pilha(Pilha* pi, struct aluno al);
 07 int remove_Pilha(Pilha* pi);
 08 int tamanho_Pilha(Pilha* pi);
 09 int Pilha_vazia(Pilha* pi);
 10 int Pilha_cheia(Pilha* pi);

```

```

 01 #include <stdio.h>
 02 #include <stdlib.h>
 03 #include "PilhaDin.h" //inclui os protótipos
 04 //Definição do tipo Pilha
 05 struct elemento{
 06     struct aluno dados;
 07     struct elemento *prox;
 08 };
 09 typedef struct elemento Elemt;

```

```

 01 //Arquivo PilhaDin.h
 02
 03 Pilha* cria_Pilha();
 04 void libera_Pilha(Pilha* pi);
 05 int acessa_topo_Pilha(Pilha* pi, struct aluno *al);
 06 int insere_Pilha(Pilha* pi, struct aluno al);
 07 int remove_Pilha(Pilha* pi);
 08 int tamanho_Pilha(Pilha* pi);
 09 int Pilha_vazia(Pilha* pi);
 10 int Pilha_cheia(Pilha* pi);

```

```

 01 Pilha* cria_Pilha(){
 02     Pilha* pi = (Pilha*) malloc(sizeof(Pilha));
 03     if(pi != NULL)
 04         *pi = NULL;
 05     return pi;
 06 }

```

FIGURA 8.21

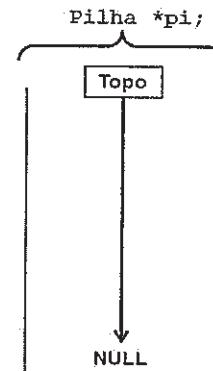


FIGURA 8.22

Destruir uma pilha que utilize alocação dinâmica e seja encadeada não é uma tarefa tão simples quanto destruir uma **pilha sequencial estática**.



Para liberar uma pilha que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a pilha liberando a memória alocada para cada elemento inserido nela.

O código que realiza a destruição da pilha é mostrado na Figura 8.23. Inicialmente, verificamos se a pilha é válida, ou seja, se a tarefa de criação da pilha foi realizada com sucesso (linha

2). Em seguida, percorremos a pilha até que o conteúdo do seu topo (\*pi) seja diferente de NULL, o final da pilha. Enquanto não chegarmos ao final da pilha, iremos liberar a memória do elemento que se encontra atualmente no topo da pilha e avançar para o próximo (linhas 5-7). Terminado o processo, liberamos a memória alocada para o topo da pilha (linha 9). Esse processo é mais bem ilustrado pela Figura 8.24, que mostra a liberação de uma pilha contendo dois elementos.

### Destruindo uma pilha

```

01 void libera_Pilha(Pilha* pi){
02     if(pi != NULL){
03         Elem* no;
04         while((*pi) != NULL){
05             no = *pi;
06             *pi = (*pi)->prox;
07             free(no);
08         }
09         free(pi);
10     }
11 }
```

FIGURA 8.23

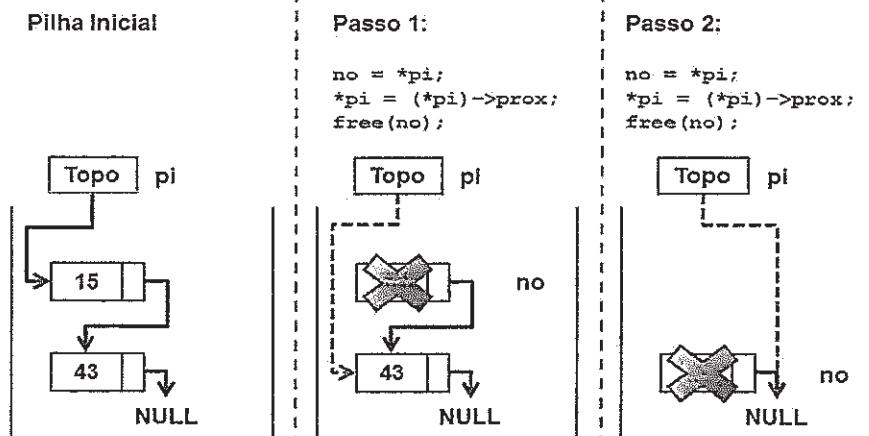


FIGURA 8.24

### 8.5.3 Informações básicas sobre a pilha

As operações de inserção, remoção e busca são consideradas as principais de uma pilha. Apesar disso, para realizar estas operações é necessário ter em mãos outras informações mais básicas sobre a pilha. Por exemplo, não podemos remover um elemento da pilha se ela estiver vazia. Sendo assim, é conveniente implementar uma função que retorne esse tipo de informação.

Na sequência, veremos como implementar as funções para retornar as três principais informações sobre o “status” atual da pilha: seu tamanho, se ela está cheia ou se ela está vazia.

#### Tamanho da pilha

Saber o tamanho de uma **pilha dinâmica encadeada** não é uma tarefa tão simples como na **pilha sequencial estática**. Isso ocorre porque agora não possuímos mais um campo armazenando a quantidade de elementos inseridos dentro da pilha.



Para saber o tamanho de uma pilha que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a pilha contando os elementos inseridos na pilha, até encontrar o seu final.

O código que realiza a contagem dos elementos da pilha é mostrado na Figura 8.25. Inicialmente, verificamos se a pilha é válida, ou seja, se a tarefa de criação da pilha foi realizada com sucesso e a pilha é ou não igual a NULL (linha 2). Caso ela seja nula, não temos o que fazer na função e a terminamos (linha 3). Em seguida, criamos um contador iniciado em ZERO (linha 4) e um elemento auxiliar (no) apontado para o topo da pilha (linha 5). Então, percorremos a pilha até que o valor de no seja diferente de NULL, o final da pilha. Enquanto não chegarmos ao final da pilha, iremos somar “+1” ao contador cont e avançar para o próximo elemento da pilha (linhas 6-9). Terminado o processo, retornamos o valor da variável cont (linha 10). Esse processo é mais bem ilustrado pela Figura 8.26, que mostra o cálculo do tamanho de uma pilha contendo dois elementos.

### Tamanho da pilha

```

01 int tamanho_Pilha(Pilha* pi){
02     if(pi == NULL)
03         return 0;
04     int cont = 0;
05     Elem* no = *pi;
06     while(no != NULL){
07         cont++;
08         no = no->prox;
09     }
10     return cont;
11 }
```

FIGURA 8.25

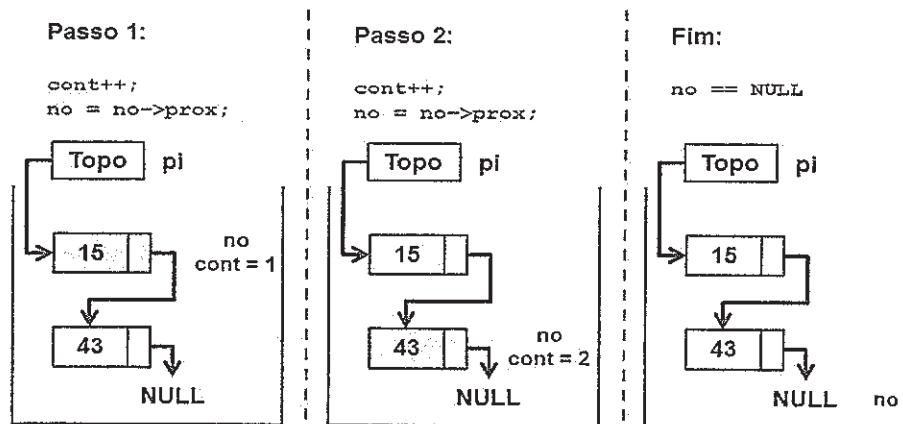


FIGURA 8.26

### Pilha cheia

Implementar uma função que retorne se uma **pilha dinâmica encadeada** está cheia é uma tarefa relativamente simples.



**Uma pilha dinâmica encadeada** somente será considerada cheia quando não tivermos mais memória disponível para alocar novos elementos.

A implementação da função que retorna se a pilha está cheia é mostrada na Figura 8.27. Como se pode notar, a função sempre irá retornar o valor **ZERO**, indicando que a pilha não está cheia.

#### Retornando se a pilha está cheia

```
01 int Pilha_cheria(Pilha* pi) {
02     return 0;
03 }
```

FIGURA 8.27

### Pilha vazia

Implementar uma função que retorne se uma **pilha dinâmica encadeada** está vazia é outra tarefa relativamente simples.



**Uma pilha dinâmica encadeada** será considerada vazia sempre que o conteúdo do seu “topo” apontar para a constante **NULL**.

A implementação da função que retorna se a pilha está vazia é mostrada na Figura 8.28. Note que essa função, em primeiro lugar, verifica se o ponteiro **Pilha \*pi** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Assim, optamos por retornar o valor **UM** para indicar uma pilha inválida (linha 3). Porém, se a pilha foi criada com sucesso, então é possível acessar o conteúdo do seu “topo” (“**pi**”) e comparar o seu valor com a constante **NULL**, que é o valor inicial do conteúdo do “topo” quando criamos a pilha. Se os valores forem iguais (ou seja, nenhum elemento contido dentro da pilha), a função irá retornar o valor **UM** (linha 5). Caso contrário, irá retornar o valor **ZERO** (linha 6).



#### Retornando se a pilha está vazia

```
01 int Pilha_vazia(Pilha* pi) {
02     if(pi == NULL)
03         return 1;
04     if(*pi == NULL)
05         return 1;
06     return 0;
07 }
```

FIGURA 8.28

### 8.5.4 Inserindo um elemento na pilha

Uma vez criada a pilha, podemos começar a guardar elementos em seu topo. É disso que se trata a operação de inserção.



A operação de inserção envolve o teste de estouro da pilha, ou seja, precisamos verificar se é possível inserir um novo elemento no topo da pilha (ela ainda não está cheia).

No caso de uma pilha com alocação dinâmica, ela somente será considerada cheia quando não tivermos mais memória disponível no computador para alocar novos elementos. Isso ocorrerá apenas quando a chamada da função **malloc()** retornar **NULL**.



Também existe o caso em que a inserção é feita em uma pilha que está vazia. Neste caso, a pilha, que inicialmente apontava para **NULL**, passa a apontar para o único elemento inserido até então.

Basicamente, o que temos que fazer para inserir um elemento no topo de uma pilha é alocar espaço para ele e mudar os valores de alguns ponteiros, como mostra a sua implementação na Figura 8.29. Primeiro, a função verifica se o ponteiro `Pilha* pi` é igual a `NULL`. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Assim, optamos por retornar o valor `ZERO` para indicar uma pilha inválida (linha 3). Porém, se a pilha foi criada com sucesso, podemos tentar alocar memória para um novo elemento (linhas 4 e 5). Caso a alocação de memória não seja possível, a função irá retornar o valor `ZERO` (linhas 6 e 7). Tendo a função `malloc()` retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 8).

Até aqui, o que fizemos foi verificar se podemos inserir na pilha e criar um novo elemento (`no`) com os dados passados por parâmetro. Falta, obviamente, inserir este elemento no topo da pilha. Então, temos que fazê-lo apontar para o topo da pilha, `*pi` (linha 9). Assim, o elemento `no` passa a ser o novo topo da pilha, enquanto o antigo topo passa a ser o próximo elemento da pilha. Por fim, mudamos o conteúdo do “topo” da pilha (`*pi`) para que ele passe a ser o nosso elemento `no` e retornamos o valor `UM` (linhas 10 e 11), indicando sucesso na operação de inserção. Esse processo é mais bem ilustrado pela Figura 8.30.



Note que a inserção no topo de uma **pilha dinâmica encadeada** é equivalente à inserção no início de uma **lista dinâmica encadeada**.

#### Inserindo um elemento na pilha

```

01 int insere_Pilha(Pilha* pi, struct aluno al){
02     if(pi == NULL)
03         return 0;
04     Elemt* no;
05     no = (Elemt*) malloc(sizeof(Elemt));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = (*pi);
10     *pi = no;
11     return 1;
12 }
```

FIGURA 8.29

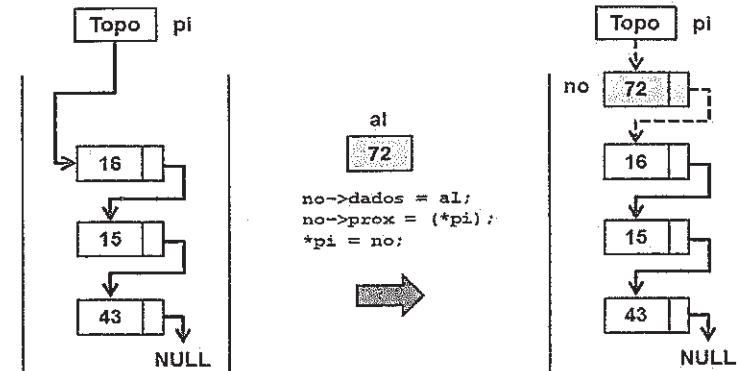


FIGURA 8.30

#### 8.5.5 Removendo um elemento da pilha

Remover um elemento do topo de uma **pilha dinâmica encadeada** é uma tarefa relativamente simples, como mostra a sua implementação na Figura 8.31. Primeiro, verificamos se o ponteiro `Pilha* pi` é igual a `NULL`. A condição seria verdadeira se houvesse ocorrido um problema na criação da pilha e, neste caso, não teríamos uma pilha válida para trabalhar. Assim, optamos por retornar o valor `ZERO` para indicar uma pilha inválida (linha 3). Porém, se a pilha foi criada com sucesso, precisamos verificar se ela não está vazia, isto é, se existem elementos dentro dela. Caso a pilha esteja vazia, a função irá retornar o valor `ZERO` (linhas 4 e 5).

Como se trata de uma remoção, temos que fazer o topo da pilha (`*pi`) apontar para o elemento seguinte a ele (linhas 6 e 7). Por fim, temos que liberar a memória associada ao antigo “topo” da pilha (`no`) e retornarmos o valor `UM` (linhas 8 e 9), indicando sucesso na operação de remoção. Esse processo é mais bem ilustrado pela Figura 8.32.



Note que a remoção do topo de uma **pilha dinâmica encadeada** é equivalente à remoção do início de uma **lista dinâmica encadeada**.

#### Removendo um elemento da pilha

```

01 int remove_Pilha(Pilha* pi){
02     if(pi == NULL)
03         return 0;
04     if((*pi) == NULL)
05         return 0;
06     Elemt *no = *pi;
07     *pi = no->prox;
08     free(no);
09     return 1;
10 }
```

FIGURA 8.31

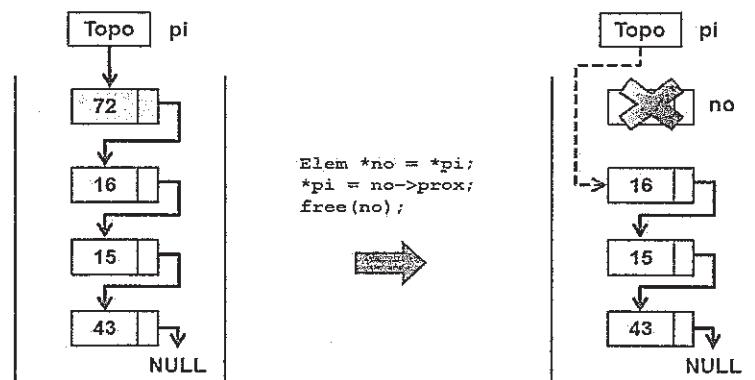


FIGURA 8.32

### 8.5.6 Acessando o elemento no topo da pilha

Apesar de possuir uma implementação quase idêntica à da lista dinâmica encadeada, o acesso a um elemento de uma pilha dinâmica encadeada é um pouco diferente.



Em uma lista, pode-se acessar e recuperar as informações contidas em qualquer um dos seus elementos. Já em uma pilha, podemos acessar as informações apenas do elemento no **topo** da pilha.

Acessar um elemento que se encontra no topo de uma **pilha dinâmica encadeada** é uma tarefa quase imediata, como mostra a sua implementação na Figura 8.33. Em primeiro lugar, a função verifica se a busca é válida. Para tanto, verificamos se o ponteiro **Pilha\*** **pi** é igual a **NULL**. Se esta condição for verdadeira, a busca termina e a função retorna o valor **ZERO** (linha 3). Em seguida, a função verifica se a pilha é vazia, ou seja, se o conteúdo do topo é igual a **NULL**. Novamente temos que, se esta condição for verdadeira a busca termina e a função retorna o valor **ZERO** (linha 5). Caso contrário, a posição equivalente ao “topo” da pilha é copiada para o conteúdo do ponteiro passado por referência (**al**) para a função (linha 6). Como se nota, esse tipo de busca consiste em um simples acesso ao conteúdo do primeiro elemento da pilha (Figura 8.34).

**Acessando o topo da pilha**

```

01. int acessa_topo_Pilha(Pilha* pi, struct aluno *al){
02.     if(pi == NULL)
03.         return 0;
04.     if((*pi) == NULL)
05.         return 0;
06.     *al = (*pi)->dados;
07. }

```

FIGURA 8.33

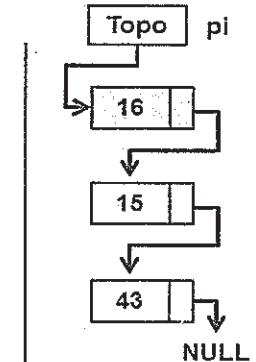


FIGURA 8.34

### 8.5.7 Análise de complexidade

Um aspecto importante quando manipulamos pilhas tem relação com os custos das suas operações. Em uma **pilha dinâmica encadeada** as operações de inserção, remoção e consulta envolvem apenas a manipulação de alguns ponteiros, independentemente do número de elementos na pilha. Desse modo, a complexidade dessas operações é  $O(1)$ .

## 8.6 CRIANDO UMA PILHA USANDO UMA LISTA

Ao longo desta seção, vimos que as pilhas são implementadas e se comportam de modo muito similar às listas.



As pilhas são consideradas um tipo especial de lista em que a inserção e remoção são realizadas sempre na mesma extremidade da lista.

Podemos concluir, então, que uma pilha nada mais é do que uma lista sujeita a uma ordem de entrada e saída. Sendo assim, podemos implementar uma pilha utilizando uma lista, como mostram as Figuras 8.35 e 8.36. Neste caso, estamos implementando uma pilha dinâmica usando uma lista dinâmica.

Vamos começar definindo o arquivo **PilhaUsandoListaDinEncad.h**, ilustrado na Figura 8.35. Nele, temos que estabelecer:

- A biblioteca da lista usada para representar a pilha (linha 1).
- Para fins de padronização, um novo nome para a **Lista** (linha 3). Esse é o tipo que será usado sempre que se desejar trabalhar com uma pilha.
- As funções disponíveis para se trabalhar com essa pilha (linhas 5-12) e que serão implementadas no arquivo **PilhaUsandoListaDinEncad.c**.

No arquivo **PilhaUsandoListaDinEncad.c**, ilustrado na Figura 8.36, iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em **PilhaUsandoListaDinEncad.h**. Neste caso, as funções definidas para a pilha irão apenas encapsular as funções já definidas para a lista. O mais importante nessa implementação é a criação da regra de inserção e remoção da pilha, assim como a consulta a um elemento:

- Consulta na pilha: sempre o **primeiro** elemento da lista (linhas 9-11).
- Inserção na pilha: sempre no **início** da lista (linhas 12-14).
- Remoção da pilha: sempre no **início** da lista (linhas 15-17).

Arquivo PilhaUsandoListaDinEncad.h

```

01 #include "ListaDinEncad.h"
02
03 typedef Lista Pilha;
04
05 Pilha* cria_Pilha();
06 void libera_Pilha(Pilha* pi);
07 int consulta_topo_Pilha(Pilha* pi, struct aluno *al);
08 int insere_Pilha(Pilha* pi, struct aluno al);
09 int remove_Pilha(Pilha* pi);
10 int tamanho_Pilha(Pilha* pi);
11 int Pilha_vazia(Pilha* pi);
12 int Pilha_cheia(Pilha* pi);

```

FIGURA 8.35

Arquivo PilhaUsandoListaDinEncad.c

```

01 //inclui os Protótipos
02 #include "PilhaUsandoListaDinEncad.h"
03 Pilha* cria_Pilha(){
04     return cria_lista();
05 }
06 void libera_Pilha(Pilha* pi){
07     libera_lista(pi);
08 }
09 int consulta_topo_Pilha(Pilha* pi, struct aluno *al){
10     return consulta_lista_pos(pi,1,al);
11 }
12 int insere_Pilha(Pilha* pi, struct aluno al){
13     return insere_lista_inicio(pi,al);
14 }
15 int remove_Pilha(Pilha* pi){
16     return remove_lista_inicio(pi);
17 }
18 int tamanho_Pilha(Pilha* pi){
19     return tamanho_lista(pi);
20 }
21 int Pilha_vazia(Pilha* pi){
22     return lista_vazia(pi);
23 }
24 int Pilha_cheia(Pilha* pi){
25     return lista_cheia(pi);
26 }

```

FIGURA 8.36

## 8.7 EXERCÍCIOS

- 1) Defina, usando as suas palavras, o que é uma estrutura do tipo pilha.
- 2) Defina, usando as suas palavras, a diferença da pilha sequencial estática para a pilha dinâmica encadeada.
- 3) Considere uma pilha com quatro valores inteiros, na seguinte ordem: 1, 2, 3 e 4. Que sequência de inserções (I) e remoções (R) devemos executar para obter os valores na ordem 2, 4, 3 e 1?
- 4) Uma sequência de inserções (I) e remoções (R) é considerada válida se tem igual número de Is e Rs. Além disso, todas as operações devem ser possíveis, isto é, a remoção só pode ocorrer se houver elementos na pilha. Crie um algoritmo que permita determinar se uma sequência (ex: IRIIRR) é ou não válida.
- 5) Dada uma pilha que armazena caracteres, crie uma função que verifique se uma palavra é um palíndromo.

- 6) Escreva uma função que receba uma string contendo uma expressão aritmética e retorne se ela está com a parentização correta. A função deverá verificar se cada “abre parênteses” tem um “fecha parênteses” correspondente. Exemplos:
- Correto:  $(()) - ((()) - ()()$
  - Incorreto:  $) - ((()) - ))(($
- 7) Implemente uma função para inverter a posição dos elementos de uma pilha. Faça a função para ambos os tipos de pilha: estática e dinâmica.
- 8) Implemente uma função para testar se duas pilhas contendo números inteiros são iguais, isto é, se possuem o mesmo conteúdo e na mesma ordem. Faça a função para ambos os tipos de pilha: estática e dinâmica.
- 9) Implemente uma função que copie os elementos de uma pilha P1 para uma P2.
- 10) Escreva um programa que armazene  $n$  valores da sequência de Fibonacci em uma pilha.
- 11) Considerando as operações de inserção (I) e remoção (R) em uma pilha, escreva a configuração da pilha após a seguinte sequência de operações: I(11), I(34), R, R, I(23), I(45), R, I(121), I(22), R, R.
- 12) Escreva um programa que faça a conversão numérica da base 10 para a base escolhida usando uma pilha:
- Decimal para binário.
  - Decimal para octal.
  - Decimal para hexadecimal.

## CAPÍTULO 9

# Tabela hash

## 9.1 DEFINIÇÃO

Uma grande variedade de métodos de busca funciona segundo o mesmo princípio: procurar a informação desejada com base na comparação de suas chaves, isto é com base em algum valor que a compõe. Um dos problemas disso é que, para obtermos algoritmos eficientes, os elementos devem ser armazenados de forma ordenada. Desconsiderando o custo da ordenação, que, no melhor caso, é  $O(N \log N)$ , temos que os algoritmos mais eficientes de busca possuem complexidade  $O(\log N)$ .



Uma operação de busca ideal seria aquela que permitisse o acesso direto ao elemento procurado, sem nenhuma etapa de comparação de chaves. Neste caso, teríamos um custo  $O(1)$ .

Arrays são estruturas que utilizam índices para armazenar informações. Através do índice, podemos acessar determinada posição do array com custo  $O(1)$ . Infelizmente, os arrays não possuem nenhum mecanismo que permita calcular a posição em que uma informação está armazenada, de modo que a operação de busca em um array não é  $O(1)$ .



O acesso direto a um elemento de um array com base em parte de sua informação (chave) é possível por meio do uso de **tabelas hash**.

Também conhecidas como tabelas de indexação ou de espalhamento, a tabela hash é uma generalização da ideia de array. Sua ideia central é utilizar uma função, chamada **função de hashing**, para espalhar os elementos que queremos armazenar na tabela. Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do array que define a tabela, como mostra a Figura 9.1.

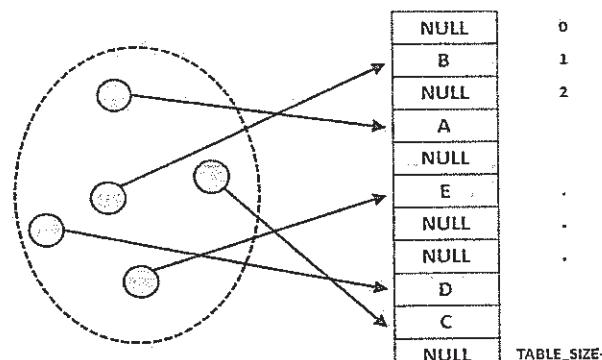


FIGURA 9.1



Por que espalhar os elementos de forma não ordenada pode melhorar a busca?

A tabela hash é uma estrutura de dados especial que permite a associação de valores a chaves. A chave é uma parte da informação que compõe o elemento a ser inserido ou buscado na tabela, sendo o valor retornado pela função a posição (índice) em que o elemento se encontra no array que define a tabela. Assim, a partir de uma chave, podemos acessar de forma rápida determinada posição do array. Na média, essa operação tem custo  $O(1)$ .

Várias são as vantagens de se utilizar uma tabela hash:

- Alta eficiência na operação de busca: caso médio é  $O(1)$  enquanto o da busca linear é  $O(N)$  e o da busca binária é  $O(\log_2 N)$ .
- Tempo de busca é praticamente independente do número de chaves armazenadas na tabela.
- Implementação simples.

Infelizmente, esse tipo de implementação também tem suas desvantagens:

- Alto custo para recuperar os elementos da tabela ordenados pela chave. Neste caso, é preciso ordenar a tabela.
- O pior caso é  $O(N)$ , sendo  $N$  o tamanho da tabela: alto número de colisões.



Uma **colisão** é a ocorrência de duas ou mais chaves na tabela hash com o mesmo valor de posição.

Ou seja, uma colisão ocorre quando duas (ou mais) chaves diferentes tentam ocupar a mesma posição na tabela hash. A colisão de chaves não é algo exatamente ruim, é apenas indesejável, pois diminui o desempenho do sistema.

## 9.2 APLICAÇÕES

Existem várias aplicações que fazem uso de tabelas hash. Ela pode ser utilizada para:

- Busca de elementos em base de dados: estruturas de dados em memória, bancos de dados e mecanismos de busca na internet.
- Verificação de integridade de dados e autenticação de mensagens: os dados são enviados juntamente com o resultado da função de hashing. Quem receber os dados recalculará a função de hashing usando os dados recebidos e compara o resultado obtido com o que recebeu. Se os resultados forem diferentes, houve erro de transmissão.
- Armazenamento de senhas com segurança: a senha não é armazenada no servidor, mas sim o resultado da função de hashing.
- Implementação da tabela de símbolos dos compiladores.
- Criptografia: MD5 e família SHA (Secure Hash Algorithm).

## 9.3 CRIANDO O TAD TABELA HASH

### 9.3.1 Definindo o tipo tabela hash

Antes de começar a implementar a nossa tabela hash, é preciso definir o tipo de dado que será armazenado nela. Uma tabela hash pode armazenar qualquer tipo de informação. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa tabela. Este tipo será um ponteiro para a estrutura que define a tabela. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **TabelaHash.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **TabelaHash.h**, ilustrado na Figura 9.2. Nele, iremos definir:

- O tipo de dado que será armazenado na tabela, **struct aluno** (linhas 1-5).
- Para fins de padronização, um novo nome para o tipo hash (linha 6). Esse é o tipo que será usado sempre que se desejar trabalhar com uma tabela hash.
- As funções disponíveis para trabalhar com essa tabela hash (linhas 8-14) e que serão implementadas no arquivo **TabelaHash.c**.

Neste exemplo, optamos por armazenar uma estrutura representando um aluno dentro da tabela. Este aluno é representado pelo seu número de matrícula, nome e três notas.



No arquivo **TabelaHash.c**, iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em **TabelaHash.h**.

Basicamente, o arquivo **TabelaHash.c** (Figura 9.2) contém:

- As chamadas às bibliotecas necessárias à implementação da tabela hash (linhas 1-3).
- A definição do tipo que descreve o funcionamento da tabela hash, **struct hash** (linhas 5-8).
- As implementações das funções definidas no arquivo **TabelaHash.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que o nosso tipo hash nada mais é do que uma estrutura contendo três campos: um inteiro **TABLE\_SIZE**, que indica o tamanho da tabela hash, um inteiro **qtd**, que mostra a quantidade de elementos armazenados, e um ponteiro para ponteiro **itens**. A ideia aqui é alocar um array de ponteiros de tamanho **TABLE\_SIZE** no campo **itens** para armazenar os elementos inseridos na tabela.



Por questão de desempenho, nossa tabela hash irá armazenar apenas o endereço para a estrutura que contém os dados do aluno e não os dados em si.

#### Arquivo TabelaHash.h

```

01 struct aluno{
02     int matricula;
03     char nome[30];
04     float n1,n2,n3;
05 };
06 typedef struct hash Hash;
07
08 Hash* criaHash(int tamanho);
09 void liberaHash(Hash* ha);
10 int valorString(char *str);
11 int insereHash_SemColisao(Hash* ha, struct aluno al);
12 int buscaHash_SemColisao(Hash* ha, int mat,
                           struct aluno* al);
13 int insereHash_EnderAberto(Hash* ha, struct aluno al);
14 int buscaHash_EnderAberto(Hash* ha, int mat,
                           struct aluno* al);

```

#### Arquivo TabelaHash.c

```

01 #include <stdlib.h>
02 #include <string.h>
03 #include "TabelaHash.h" //inclui os Protótipos
04 //Definição do tipo hash
05 struct hash{
06     int qtd, TABLE_SIZE;
07     struct aluno **itens;
08 };

```

FIGURA 9.2

Esse tipo de abordagem tem como objetivo evitar o gasto excessivo de memória. Em uma tabela hash, os elementos ficam dispersos, ou seja, várias posições do array podem não possuir nenhum dado. Assim, se o array armazenasse a **struct aluno**, teríamos uma grande quantidade de memória desperdiçada. Para evitar isso, utilizamos um array de ponteiros (que ocupa muito menos memória do que a **struct**) e, à medida que os elementos são inseridos na tabela, realizamos a alocação daquele único elemento.

Por fim, por estarem definidos dentro do arquivo **.c**, os campos da **struct hash** não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome definido no arquivo **TabelaHash.h** (linha 6), que pode somente declarar um ponteiro para ele, da seguinte forma:

Hash \*ha;

#### 9.3.2 Criando e destruindo uma tabela hash

Para utilizar uma tabela hash em seu programa, a primeira coisa a fazer é criar uma tabela vazia. Essa tarefa é executada pela função descrita na Figura 9.3. Basicamente, o que essa função faz é a alocação de uma área de memória para a tabela hash (linha 2). Esta área de memória corresponde à memória necessária para armazenar a estrutura que define a tabela, **struct hash**. Em seguida, essa função inicializa o campo **TABLE\_SIZE** com o tamanho de tabela informado pelo usuário (linha 5). Esse valor será também utilizado para alocar o array de ponteiros no campo **itens** (linha 6), que terá inicializado cada posição com o valor **NULL**, indicando que essa posição está vaga na tabela (linhas 12-13). Também inicializamos o campo **qtd** com o valor **ZERO** (linha 11), indicando que nenhum elemento foi inserido na tabela (tabela vazia).

#### Criando uma tabela hash

```

01 Hash* criaHash(int TABLE_SIZE){
02     Hash* ha = (Hash*) malloc(sizeof(Hash));
03     if(ha != NULL){
04         int i;
05         ha->TABLE_SIZE = TABLE_SIZE;
06         ha->itens = (struct aluno**) malloc(
07             TABLE_SIZE * sizeof(struct aluno*));
08         if(ha->itens == NULL){
09             free(ha);
10             return NULL;
11         }
12         ha->qtd = 0;
13         for(i=0; i < ha->TABLE_SIZE; i++)
14             ha->itens[i] = NULL;
15     }
16     return ha;
}

```

FIGURA 9.3

A Figura 9.4 indica o conteúdo do nosso ponteiro **Hash\*** **ha** após a chamada da função que cria a tabela hash.



Ao escolher o tamanho da tabela hash, o ideal é escolher um número primo e evitar valores que sejam uma potência de dois.

A escolha de um número primo para o tamanho da tabela hash reduz a probabilidade de colisões, mesmo que a função de hashing utilizada não seja muito eficaz. Usar uma potência de dois como o tamanho da tabela melhora a velocidade, mas pode aumentar os problemas de colisão se estivermos utilizando uma função de hashing mais simples.

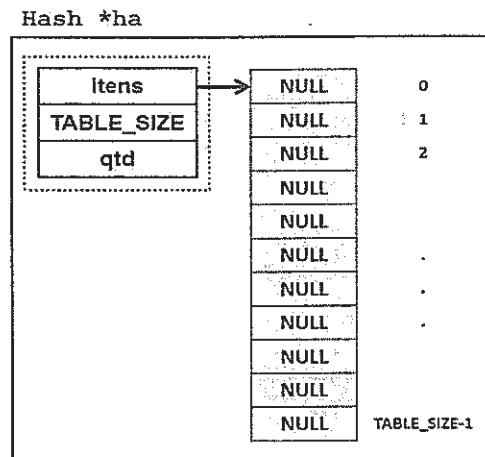


FIGURA 9.4

Destruir uma tabela hash é uma tarefa muito simples, como mostra o código contido na Figura 9.5. Basicamente, precisamos percorrer todo o array que define a tabela procurando por elementos que tenham sido armazenados nela (ou seja, que possuam um endereço de memória diferente de **NULL**) e os liberamos com a função **free()** (linhas 4-7). Ao final desse processo, temos que liberar a memória alocada para o array de ponteiros e para a estrutura que representa a tabela. Isso é feito utilizando apenas uma chamada da **free()** para cada (linhas 8-9).

### Destruindo uma tabela hash

```

01 void liberaHash(Hash* ha){
02     if(ha != NULL){
03         int i;
04         for(i=0; i < ha->MAX; i++){
05             if(ha->itens[i] != NULL)
06                 free(ha->itens[i]);
07         }
08         free(ha->itens);
09         free(ha);
10     }
11 }
  
```

FIGURA 9.5

## 9.4 CALCULANDO A POSIÇÃO DA CHAVE: FUNÇÃO DE HASHING

Tanto na operação de inserção quanto na de busca na tabela hash, é necessário calcular a posição dos dados dentro da tabela. Para tanto, utilizamos uma **função de hashing** para calcular esta posição a partir de uma **chave** escolhida a partir dos dados manipulados.



A função de hashing é extremamente importante para o bom desempenho da tabela. Ela é responsável por distribuir as informações de forma equilibrada pela tabela hash.

Ou seja, a função de hashing calcula, a partir do valor do dado, a posição dele na tabela, como mostra a Figura 9.6. Para tanto, esta função deve satisfazer as seguintes condições:

- Ser simples e barata de se calcular.
- Garantir que valores diferentes produzam posições diferentes.
- Gerar uma distribuição equilibrada dos dados na tabela, ou seja, cada posição da tabela tem a mesma chance de receber uma chave (máximo espalhamento).

A implementação da função de hashing depende do conhecimento prévio da natureza e domínio da chave a ser utilizada. Por exemplo, utilizar apenas três dígitos do número de telefone de uma pessoa para armazená-lo na tabela. Neste caso, seria melhor usar os três últimos dígitos do que os três primeiros, pois os primeiros costumam se repetir com maior frequência e iriam gerar posições iguais na tabela. Assim, o ideal é usar um cálculo diferente de hash para cada tipo de chave.

A seguir, são mostrados alguns exemplos de função de hashing bastante utilizados.

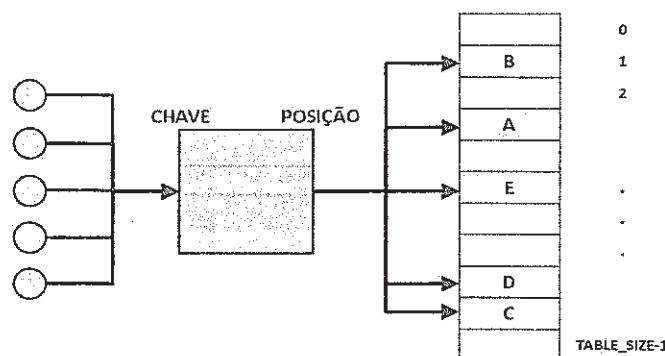


FIGURA 9.6

#### 9.4.1 Método da divisão

A função de hashing que utiliza o método da divisão (ou método da congruência linear) para espalhar os elementos é bastante simples e direta.



Basicamente, o método da divisão consiste em calcular o resto da divisão do valor inteiro que representa o elemento pelo tamanho da tabela, TABLE\_SIZE.

Ou seja, a posição é calculada utilizando uma simples operação de módulos, como mostra a Figura 9.7. Note que, antes da operação de módulo, realizamos uma operação de **E bit-a-bit** (`&`) com o valor `0x7FFFFFFF`. Isso é feito apenas para eliminar o bit de sinal do valor da chave, o que evita o risco de ocorrer um overflow e obtermos um número negativo.

##### Chave - método da divisão

```
01 int chaveDivisao(int chave, int TABLE_SIZE){
02     return (chave & 0x7FFFFFFF) % TABLE_SIZE;
03 }
```

FIGURA 9.7

Apesar de simples, o método da divisão apresenta alguns problemas.



Como trabalhamos com o resto da divisão, valores diferentes podem resultar na mesma posição.

Por exemplo, o resto da divisão de 11 por 10 e de 21 por 10 são o mesmo valor de posição: 1. Uma maneira de reduzir esse tipo de problema é utilizar como tamanho da tabela, TABLE\_SIZE, um número primo.

#### 9.4.2 Método da multiplicação

A função hash que utiliza o método da multiplicação (ou método da congruência linear multiplicativa) para espalhar os elementos é outra bastante simples e direta.



Basicamente, o método da multiplicação usa uma constante fracionária  $A$ ,  $0 < A < 1$ , para multiplicar o valor da chave que representa o elemento. Em seguida, a parte fracionária resultante é multiplicada pelo tamanho da tabela para calcular a posição do elemento.

Para entender esse processo, considere que queremos calcular a posição da chave 123456, usando a constante fracionária  $A = 0,618$  e que o tamanho da tabela seja 1024:

$$\begin{aligned} \text{Posição} &= \text{ParteInteira}(\text{TABLE\_SIZE} * \text{ParteFracionária}(\text{chave} * A)) \\ \text{Posição} &= \text{ParteInteira}(1024 * \text{ParteFracionária}(123456 * 0,618)) \\ \text{Posição} &= \text{ParteInteira}(1024 * \text{ParteFracionária}(762950,808)) \\ \text{Posição} &= \text{ParteInteira}(1024 * 0,808) \\ \text{Posição} &= \text{ParteInteira}(827,392) \\ \text{Posição} &= 827 \end{aligned}$$

Como se pode ver, a posição é calculada utilizando uma sequência de operações simples, como mostra a sua implementação na Figura 9.8.

##### Chave - método da multiplicação

```
01 int chaveMultiplicacao(int chave, int TABLE_SIZE){
02     float A = 0,6180339887; // constante: 0 < A < 1
03     float val = chave * A;
04     val = val - (int) val;
05     return (int) (TABLE_SIZE * val);
06 }
```

FIGURA 9.8

#### 9.4.3 Método da dobraria

Diferentes dos outros métodos para espalhar os elementos na tabela, o método da dobraria utiliza um esquema de dobrar e somar os dígitos do valor para calcular a sua posição.



O método da dobraria considera o valor inteiro que representa o elemento uma sequência de dígitos escritos em um pedaço de papel. Enquanto este valor for maior que o tamanho da tabela, o papel é dobrado e os dígitos sobrepostos são somados, desconsiderando-se as dezenas.

Esse processo de dobrar e somar os dígitos sobrepostos é mais bem explicado na Figura 9.9. Note que ele deve ser repetido enquanto os dígitos formarem um número maior que o tamanho da tabela.

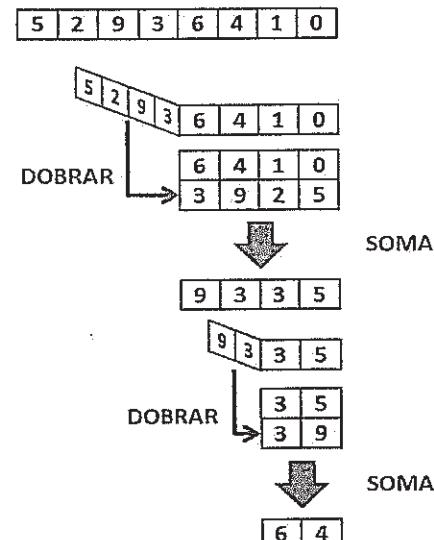


FIGURA 9.9



O método da dobra também pode ser usado com valores binários. Neste caso, em vez da soma, devemos utilizar a operação de "ou exclusivo". Não se usa as operações de "e" e "ou" binário, pois estas produzem resultados menores e maiores, respectivamente, que os operandos.

No caso de valores binários, a dobra é realizada de  $k$  em  $k$  bits, o que resulta em um valor de posição entre 0 e  $2^k + 1$ . Para entender esse processo, considere que queremos calcular a posição do valor 71 (0001000111 em binário) usando  $k = 5$ :

$$\text{Posição} = 00010 \text{ "ou exclusivo"} 00111$$

$$\text{Posição} = 00101$$

$$\text{Posição} = 5$$

A Figura 9.10 mostra a implementação do método da dobra para valores binários.

### Chave: método da dobra

```
01 int chaveDobra(int chave, int TABLE_SIZE){
02     int num_bits = 10;
03     int parte1 = chave >> num_bits;
04     int parte2 = chave & (TABLE_SIZE-1);
05     return (parte1 ^ parte2);
06 }
```

FIGURA 9.10

#### 9.4.4 Tratando uma string como chave

Sempre que desejamos inserir ou buscar um elemento da tabela hash, utilizamos parte deste elemento com chave para calcular a sua posição na tabela. No caso da **struct aluno** anteriormente definida, podemos utilizar o seu número de matrícula como a chave para calcular a posição.



E se meus dados fossem compostos apenas de **strings** e eu não tivesse um valor numérico disponível?

No caso dos elementos serem constituídos unicamente de **strings**, podemos optar por calcular um valor numérico a partir destas **strings**. Este valor pode ser facilmente calculado somando os valores ASCII dos caracteres que compõem a **string**, como mostra a Figura 9.11. O resultado dessa função pode então ser utilizado como parâmetro para um função de hashing.

### Calculando o valor da string

```
01 int valorString(char *str){
02     int i, valor = 7;
03     int tam = strlen(str);
04     for(i=0; i < tam; i++)
05         valor = 31 * valor + (int) str[i];
06     return valor;
07 }
```

FIGURA 9.11



Por que não devemos simplesmente somar os valores ASCII dos caracteres da **string**?

Quando trabalhamos com **strings**, não é bom apenas somar os valores ASCII dos caracteres, porque palavras com letras trocadas irão produzir o mesmo valor e, consequentemente, uma colisão:

Cama:  $99 + 97 + 109 + 97 = 402$   
 Maca:  $109 + 97 + 99 + 97 = 402$

Assim, para strings, é importante considerar a posição da letra, como feito na função `valorString()`. Nela, cada novo caractere é somado ao produto da soma dos caracteres anteriores pelo número 31. Desse modo, a posição dos caracteres terá influência no valor produzido. A escolha do número 31 se deve ao fato de ele ser um número primo, o que reduz a probabilidade de produzir valores iguais.

## 9.5 INSERÇÃO E BUSCA SEM COLISÃO

Tanto a inserção quanto a busca são tarefas muito simples e diretas de se realizar em uma tabela hash.



Basicamente, tanto na inserção quanto na busca, o que temos que fazer é calcular a posição dos dados no array a partir de parte dos dados (**chave**) a ser inseridos ou buscados.

Comecemos pela inserção, como mostra a sua implementação na Figura 9.12. Primeiramente, verificamos se o ponteiro `Hash* ha` é igual a `NULL` ou se a quantidade de elementos na tabela é igual ao tamanho da tabela (linha 2). A condição seria verdadeira se houvesse ocorrido um problema na criação da tabela, e, neste caso, não teríamos uma tabela válida para trabalhar, ou se a tabela estivesse cheia e não pudéssemos inserir um novo elemento nela. Assim, optamos por retornar o valor **ZERO** para indicar erro na inserção. Porém, se a tabela foi criada com sucesso e existe espaço vago dentro dela, podemos calcular a posição em que os dados serão inseridos.

Para calcular a posição do elemento, vamos considerar que a matrícula é sua chave (linha 5). Perceba que o mesmo poderia ser feito com o nome (linha 6), sendo apenas necessário converter a `string` para um valor inteiro. Em seguida, calcularmos a posição dessa chave utilizando o método da divisão (linha 8). Note que a posição poderia ser calculada com qualquer função de hashing. Na sequência, tentamos alocar memória para um novo elemento (linhas 9-10). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 11-12). Tendo a função `malloc()` retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 13). Uma vez copiados os dados, basta armazenar o ponteiro do novo elemento nessa posição do array `itens`, que foi calculada com a função de hashing, e incrementar a quantidade de elementos na tabela (linhas 14-15). Por fim, retornamos o valor **UM** (linha 16), indicando sucesso na operação de inserção.

```
01 int insereHash_SemColisao(Hash* ha, struct aluno al){
02     if(ha == NULL || ha->qtd == ha->TABLE_SIZE)
03         return 0;
04
05     int chave = al.matricula;
06     //int chave = valorString(al.nome);
07
08     int pos = chaveDivisao(chave, ha->TABLE_SIZE);
09
10    struct aluno* novo;
11    novo = (struct aluno*) malloc(sizeof(struct aluno));
12    if(novo == NULL)
13        return 0;
14    *novo = al;
15    ha->itens[pos] = novo;
16    ha->qtd++;
17    return 1;
}
```

FIGURA 9.12

Já a operação de busca, mostrada na Figura 9.13, é uma tarefa quase imediata. Como na operação de inserção, precisamos verificar se o ponteiro `Hash* ha` é igual a `NULL`, ou seja, se temos uma tabela válida. Caso essa condição seja verdadeira, optamos por retornar o valor **ZERO** para indicar erro na busca (linhas 2-3). Porém, se for uma tabela válida, utilizamos o número de matrícula passado por parâmetro para calcular a posição em que se encontra o aluno buscado (linha 5).



Note que a posição do elemento procurado poderia ser calculada com qualquer função de hashing. É necessário apenas que a função de hashing usada na busca seja a mesma usada na inserção.

```
01 int buscaHash_SemColisao(Hash* ha, int mat,
                           struct aluno* al){
02     if(ha == NULL)
03         return 0;
04
05     int pos = chaveDivisao(mat, ha->TABLE_SIZE);
06     if(ha->itens[pos] == NULL)
07         return 0;
08     *al = *(ha->itens[pos]);
09
10 }
```

FIGURA 9.13

Caso o valor armazenado na posição calculada pela função de hashing seja igual a **NULL**, a função retorna **ZERO**, por se tratar de uma posição em que nenhum dado de aluno está armazenado (linhas 6-7). No entanto, se a posição for válida, seu conteúdo é copiado para o conteúdo do ponteiro passado por referência (`al`) para a função e retornamos o valor **UM** (linhas 8-9), indicando sucesso na operação de busca.

## 9.6 HASHING UNIVERSAL

É importante lembrar que uma função de hashing está sujeita ao problema de gerar posições iguais para chaves diferentes. Por se tratar de uma função determinística, ela pode ser manipulada de forma indesejada. Conhecendo a função de hashing, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca para  $O(N)$ .



O **hashing universal** é uma estratégia que busca minimizar o problema de colisões. Basicamente, a proposta do **hashing universal** é escolher de modo aleatório (em tempo de execução) a função de hashing que será utilizada a partir de um conjunto de funções de hashing previamente definido.

Existem várias maneiras de se construir um conjunto (ou família) de funções de hashing. Uma família de funções pode ser facilmente obtida da seguinte forma:

Escolha um número primo  $p$  de tal modo que o valor de qualquer chave  $k$  a ser inserida na tabela seja menor do que  $p$  e maior ou igual a zero,  $0 \leq k < p$ . Note que o valor de  $p$  será obrigatoriamente maior do que o tamanho da tabela, `TABLE_SIZE`.

Escolha, aleatoriamente, dois números inteiros,  $a$  e  $b$ , de tal modo que  $a$  seja maior do que zero e menor ou igual a  $p$ ,  $0 < a \leq p$ , e  $b$  seja maior ou igual a zero e menor ou igual a  $p$ ,  $0 \leq b \leq p$ .

Dados os valores  $p$ ,  $a$  e  $b$ , definimos a função de hashing universal como sendo:

$$h(k)_{a,b} = ((ak + b) \% p) \% TABLE\_SIZE$$

Esse tipo de função de hashing universal permite que o tamanho da tabela, `TABLE_SIZE`, não seja necessariamente primo. Além disso, como existem  $p - 1$  valores diferentes para o valor de  $a$  e  $p$  valores possíveis para  $b$ , é possível gerar  $p(p - 1)$  funções de hashing diferentes.

## 9.7 HASHING PERFEITO E IMPERFEITO

A depender do tamanho da tabela, `TABLE_SIZE`, e dos valores inseridos nela, podemos classificar uma função de hashing como **imperfeita** ou **perfeita**.



Uma função de hashing é dita **imperfeita** se, para duas chaves diferentes, a saída da função é a mesma posição na tabela.

Ou seja, no hashing imperfeito podem ocorrer colisões das chaves armazenadas. A colisão de chaves na tabela não é algo exatamente ruim, é apenas indesejável, pois diminui o desempenho do sistema. De modo geral, muitas tabelas hash fazem uso de outra estrutura de dados para lidar com o problema da colisão, como veremos adiante.



Uma função de hashing é dita **perfeita** se nunca ocorre colisão.

Em outras palavras, o hashing perfeito garante que não haverá colisão das chaves dentro da tabela, ou seja, chaves **diferentes** irão sempre produzir posições **diferentes** na tabela. Desse modo, no pior caso, as operações de busca e inserção são sempre executadas em tempo constante,  $O(1)$ . Esse tipo de hashing é utilizado quando a colisão não é tolerável. Trata-se de um tipo de aplicação muito específica; por exemplo, o conjunto de palavras reservadas de uma linguagem de programação. Neste caso, conhecemos previamente o conteúdo a ser armazenado na tabela.

## 9.8 TRATAMENTO DE COLISÕES

Em um mundo ideal, uma função de hashing irá sempre fornecer posições diferentes para cada uma das chaves inseridas, obtendo assim o **hashing perfeito**. Porém, independentemente da função de hashing utilizada, ela vai retornar a mesma posição para duas chaves **diferentes**. A esse fenômeno se dá o nome de **colisão**.



Uma colisão é a ocorrência de duas ou mais chaves na tabela hash com o mesmo valor de posição.

Desse modo, a criação de uma tabela hash consiste em duas coisas: uma função de hashing e uma abordagem para o tratamento de colisões.



Uma escolha adequada da função de hashing e do tamanho da tabela pode minimizar as colisões.

Um dos motivos das colisões ocorrerem é porque temos mais chaves para armazenar do que o tamanho da tabela suporta. Como não há espaço suficiente para todas as chaves na tabela, colisões irão se dar. Com relação à função de hashing, a escolha de uma função que produza um espalhamento uniforme das chaves pode reduzir o número de colisões. Infelizmente, não se pode garantir que as funções de hashing possuam um bom potencial de distribuição (espalhamento), porque as colisões também são distribuídas de modo uniforme. Assim, independentemente da função, algumas colisões sempre irão ocorrer.



Colisões são teoricamente inevitáveis. Por isso, devemos sempre ter uma abordagem para tratá-las.

Independentemente da qualidade de nossa função de hashing, devemos ter um método para resolver o problema das colisões quando elas ocorrerem. Existem diversas formas de se tratar das colisões nas tabelas hash. Nas próximas seções, veremos duas técnicas bastante comuns: **endereçamento aberto** e **encadeamento separado**.

### 9.8.1 Endereçamento aberto

A ideia do endereçamento aberto (também conhecido como open addressing ou rehash) é que todos os elementos sejam armazenados na própria tabela hash, evitando assim o uso de listas encadeadas. Quando um colisão ocorre, essa estratégia irá procurar posições vagas (valor **NULL**) dentro do array que define a tabela hash até encontrar um lugar onde aquele elemento poderá ser inserido.



Basicamente, a ideia da estratégia de endereçamento aberto é, no caso de uma colisão, percorrer (sonhar) a tabela hash buscando uma posição ainda não ocupada.

Essa abordagem para o tratamento de colisões tem uma série de vantagens:

- Maior número de posições na tabela para a mesma quantidade de memória usada no encadeamento separado: a memória utilizada para armazenar os ponteiros da lista encadeada no encadeamento separado pode ser usada para aumentar o tamanho da tabela, diminuindo o número de colisões.
- A busca é realizada dentro da própria tabela, o que permite a recuperação mais rápida de elementos.
- É voltada para aplicações com restrições de memória.
- Em vez de acessarmos ponteiros, calculamos a sequência de posições a ser armazenadas.



Uma desvantagem importante no tratamento de colisões por endereçamento aberto é o maior esforço de processamento no cálculo das posições.

Esse maior esforço de processamento se deve ao fato de que, quando uma colisão ocorre, devemos calcular uma nova posição da tabela. Porém, se esta nova posição também estiver ocupada, devemos calcular uma nova posição e assim por diante. Ou seja, o cálculo da posição é refeito até que uma posição vaga seja encontrada. No pior caso, o custo da inserção se torna  $O(N)$ , quando todos os elementos inseridos colidem.

Para a realização deste cálculo, existem três estratégias muito utilizadas, como veremos a seguir:

#### Sondagem linear

Na estratégia de sondagem linear (também conhecida como tentativa linear, espalhamento linear ou rehash linear), o algoritmo tenta espalhar os elementos de forma sequencial a partir da posição calculada utilizando a função de hashing, como mostra a sua implementação na

Figura 9.14. Assim, o primeiro elemento ( $i = 0$ ) é colocado na posição obtida pela função de hashing (**pos**), o segundo (no caso de uma colisão) é colocado na posição **pos+1** (se possível) e assim por diante. Note que a nova posição é calculada utilizando uma simples operação de soma e módulo e que, antes da operação de módulo, realizamos uma operação de **E bit-a-bit (&)** com o valor **0x7FFFFFFF** para eliminar o bit de sinal do valor da chave. Isso evita o risco de ocorrer um overflow e obtermos um número negativo. A Figura 9.15 mostra um exemplo de tratamento de colisão usando sondagem linear.

```
Sondagem linear
```

```
01 int sondagemLinear(int pos, int i, int TABLE_SIZE){
02     return ((pos + i) & 0x7FFFFFFF) % TABLE_SIZE;
03 }
```

FIGURA 9.14

 Apesar de simples, essa abordagem de espalhamento apresenta um problema conhecido como **agrupamento primário**: à medida que a tabela hash fica cheia, o tempo para incluir ou buscar um elemento aumenta.

A medida que os elementos são inseridos na tabela hash usando a técnica de sondagem linear, começam a surgir longas sequências de posições ocupadas. A ocorrência desses agrupamentos aumenta o tempo de pesquisa na tabela, diminuindo o seu desempenho. Além disso, quanto maior for o agrupamento primário, maior a probabilidade de aumentá-lo ainda mais com a inserção de um novo elemento.

NULL	0	CHAVE	POSIÇÃO	INSCRIÇÃO
NULL	1	A	2	Posição 2 vazia. Insere elemento
NULL	2	B	6	Posição 6 vazia. Insere elemento
NULL	3			Posição 2 ocupada, procura na próxima posição: 3
NULL	4	C	2	Posição 3 vazia. Insere elemento
NULL	5			Posição 10 vazia. Insere elemento
NULL	6	D	10	Posição 10 ocupada, procura na próxima posição. Como a posição 10 é a última, volta para o início: 0
NULL	7			Posição 0 vazia. Insere elemento
NULL	8			
NULL	9			
NULL	10	E	10	

E	0
NULL	1
A	2
C	3
NULL	4
NULL	5
B	6
NULL	7
NULL	8
NULL	9
D	10

FIGURA 9.15

#### Sondagem quadrática

Na estratégia de sondagem quadrática (também conhecida como tentativa quadrática, espalhamento quadrático ou rehash quadrático), o algoritmo tenta espalhar os elementos utilizando

uma equação do segundo grau de forma  $pos + (c_1 * i) + (c_2 * i^2)$ , em que  $pos$  é a posição obtida pela função de hashing,  $i$  é tentativa atual e  $c_1$  e  $c_2$  são os coeficientes da equação.

A Figura 9.16 mostra a implementação dessa estratégia de tratamento de colisão. Nela, o primeiro elemento ( $i = 0$ ) é colocado na posição obtida pela função de hashing ( $pos$ ), o segundo (no caso de uma colisão) é colocado na posição  $pos + (c_1 * 1) + (c_2 * 1^2)$  (se possível), o terceiro (no caso de uma nova colisão) é colocado na posição  $pos + (c_1 * 2) + (c_2 * 2^2)$  (se possível) e assim por diante.

### Sondagem quadrática

```
01 int sondagemQuadratica(int pos, int i, int TABLE_SIZE) {
02     pos = pos + 2*i + 5*i*i;
03     return (pos & 0x7FFFFFFF) % TABLE_SIZE;
04 }
```

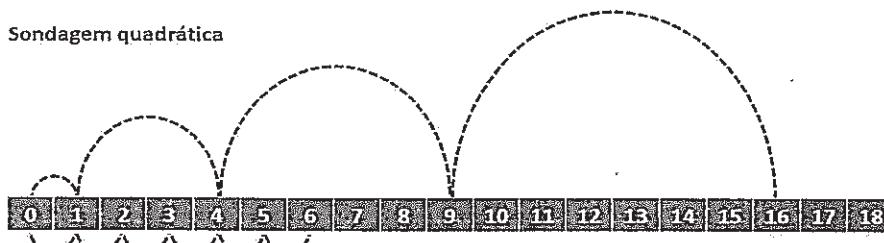
FIGURA 9.16



A sondagem quadrática resolve o problema de **agrupamento primário**. Porém, gera outro problema, conhecido como **agrupamento secundário**.

O problema do **agrupamento secundário** ocorre porque todas as chaves que produzem a mesma posição inicial na tabela hash também produzem as mesmas posições na sondagem quadrática. Felizmente, a degradação produzida na tabela hash pelos agrupamentos secundários ainda é menor que a produzida pelos agrupamentos primários. A Figura 9.17 compara as abordagens de sondagem linear e quadrática.

### Sondagem quadrática



### Sondagem linear

FIGURA 9.17

### Duplo hash

Na estratégia de duplo hash (também conhecida como espalhamento duplo) o algoritmo tenta espalhar os elementos utilizando duas funções de hashing:

- A primeira função de hashing,  $H1$ , é utilizada para calcular a **posição inicial** do elemento.
- A **segunda função de hashing,  $H2$** , é utilizada para calcular os **deslocamentos** em relação à posição inicial (no caso de uma colisão).

Desse modo, a posição de um novo elemento na tabela hash é obtida como sendo  $H1 + i * H2$ , em que  $i$  é tentativa atual de inserção do elemento. A Figura 9.18 mostra a implementação dessa estratégia de tratamento de colisão. Nela, o primeiro elemento ( $i = 0$ ) é colocado na posição obtida pela primeira função de hashing ( $H1$ ), o segundo (no caso de uma colisão) é colocado na posição  $H1 + 1 * H2$  (se possível), o terceiro (no caso de uma nova colisão) é colocado na posição  $H1 + 2 * H2$  (se possível) e assim por diante. Esse tipo de estratégia diminui a ocorrência de agrupamentos, o que faz dele um dos melhores métodos para tratamento de colisões em endereçamento aberto.

### Duplo hash

```
01 int duploHash(int H1, int chave, int i, int TABLE_SIZE) {
02     int H2 = chaveDivisao(chave, TABLE_SIZE-1) + 1;
03     return ((H1 + i*H2) & 0x7FFFFFFF) % TABLE_SIZE;
04 }
```

FIGURA 9.18



Para o duplo hash funcionar corretamente, é necessário que as duas funções de hashing sejam diferentes. Além disso, a segunda função de hashing não pode resultar em um valor igual a **ZERO**, pois, neste caso, não haveria deslocamento.

Na nossa implementação, a segunda função de hashing é calculada usando o método da divisão com um tamanho de tabela um pouco menor (é muito comum utilizar **TABLE\_SIZE-1** ou **TABLE\_SIZE-2**). Além disso, somamos **+1** ao valor da posição para garantir que a posição retornada não seja **0**.

### Inserção e busca com tratamento de colisão

Vimos, anteriormente, que a inserção e a busca são tarefas muito simples e diretas de se realizar em uma tabela hash. O mesmo vale para quando temos que tratar da colisão de dados.



Basicamente, tanto na inserção quanto na busca, o que temos que fazer é calcular a posição dos dados no array a partir de parte dos dados (**chave**) a ser inseridos ou buscados. Caso ocorra uma colisão, devemos calcular a nova posição de acordo com alguma estratégia de tratamento de colisão.

Comecemos pela inserção, como mostra a sua implementação na Figura 9.19. Primeiramente, verificamos se o ponteiro **Hash\*** é igual a **NULL** ou se a quantidade de elementos na

**Inserindo um elemento na tabela hash**

```

01 int insereHash_EnderAberto(Hash* ha, struct aluno al) {
02     if(ha == NULL || ha->qtd == ha->TABLE_SIZE)
03         return 0;
04
05     int chave = al.matricula;
06     //int chave = valorString(al.nome);
07
08     int i, pos, newPos;
09     pos = chaveDivisao(chave,ha->TABLE_SIZE);
10    for(i=0; i < ha->TABLE_SIZE; i++) {
11        newPos = sondagemLinear(pos,i,ha->TABLE_SIZE);
12        //newPos = sondagemQuadratica(pos,i,
13        //                                ha->TABLE_SIZE);
14        //newPos = duploHash(pos,chave,i,ha->TABLE_SIZE);
15        if(ha->itens[newPos] == NULL) {
16            struct aluno* novo;
17            novo = (struct aluno*) malloc(
18                sizeof(struct aluno));
19            if(novo == NULL)
20                return 0;
21            *novo = al;
22            ha->itens[newPos] = novo;
23            ha->qtd++;
24            return 1;
25        }
26    }
27    return 0;
}

```

FIGURA 9.19

tabela é igual ao tamanho da tabela (linha 2). A condição seria verdadeira se houvesse ocorrido um problema na criação da tabela e, neste caso, não teríamos uma tabela válida para trabalhar, ou se a tabela estivesse cheia e não pudéssemos inserir um novo elemento nela. Neste caso, optamos por retornar o valor **ZERO** para indicar erro na inserção. Porém, se a tabela foi criada com sucesso e existe espaço vago dentro dela, podemos calcular a posição em que os dados serão inseridos.

Para calcular a posição do elemento, vamos considerar que a matrícula é sua chave (linha 5). Perceba que o mesmo poderia ser feito com o nome (linha 6), sendo apenas necessário converter a **string** para um valor inteiro. Em seguida, calculamos a posição dessa chave utilizando o método da divisão (linha 9). Note que a posição poderia ser calculada com qualquer função de hashing. Como estamos prevendo que uma colisão poderá acontecer, vamos considerar certo número de tentativas antes de desistir de buscar esse elemento. Isso é feito usando um comando de repetição, que será executado um número de vezes igual ao tamanho da tabela (linha 10). Se o comando de repetição chegar a ser concluído, iremos retornar o valor **ZERO** (linha 25), indicando falha na operação de inserção.

Dentro do comando de repetição, vamos calcular a nova posição do elemento com base em uma estratégia de tratamento de colisão. Vamos usar a sondagem linear (linha 11), mas as sondagens quadrática ou duplo hash também poderiam ser utilizadas (linhas 12-13). Perceba que, na primeira tentativa ( $i = 0$ ), essas funções irão retornar o valor obtido pela função de hashing (linha 9). Em seguida, verificamos se a posição obtida está vazia (NULL). Se a afirmação for falsa, o algoritmo irá realizar uma nova tentativa para achar um lugar para esse elemento. Do contrário, tentamos alocar memória para um novo elemento (linhas 15-16). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 17-18). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse elemento (linha 19). Uma vez copiados os dados, basta armazenar o ponteiro do novo elemento nessa posição do array **itens** que foi calculada com a função de hashing e incrementar a quantidade de elementos na tabela (linhas 20-21). Por fim, retornamos o valor **UM** (linha 22), indicando sucesso na operação de inserção.

Assim como a operação de inserção, a operação de busca, mostrada na Figura 9.20, exige que tratemos da colisão. Neste caso, quando ocorre uma colisão não procuramos uma posição vazia, mas uma que contenha a porção dos dados que foi usada no cálculo da chave.

Primeiramente, precisamos verificar se o ponteiro **Hash\*** **ha** é igual a **NULL**, ou seja, se temos uma tabela válida. Caso essa condição seja verdadeira, optamos por retornar o valor **ZERO** para indicar erro na busca (linhas 2-3). Porém, se essa é uma tabela válida, utilizamos o número de matrícula passado por parâmetro para calcular a posição em que se encontra o aluno buscado (linha 6).



Note que a posição do elemento procurado poderia ser calculada com qualquer função de hashing. É necessário apenas que a função de hashing usada na busca seja a mesma usada na inserção.

Como estamos prevendo que uma colisão poderá acontecer, vamos considerar certo número de tentativas antes de desistir de buscar esse elemento. Isso é feito usando um comando de repetição, que será executado um número de vezes igual ao tamanho da tabela (linha 7). Se o comando de repetição chegar a ser concluído, iremos retornar o valor **ZERO** (linha 19), indicando falha na operação de busca.

Dentro do comando de repetição, vamos calcular a nova posição do elemento com base em uma estratégia de tratamento de colisão. Vamos usar a sondagem linear (linha 8), mas as sondagens quadrática ou duplo hash também poderiam ser utilizadas (linhas 9-10). Perceba que, na primeira tentativa ( $i = 0$ ), essas funções irão retornar o valor obtido pela função de hashing (linha 6). Caso o valor armazenado na posição calculada pela estratégia de tratamento de colisão seja igual a **NULL**, a função retorna **ZERO**, pois trata-se de uma posição em que nenhum dado de aluno está armazenado (linhas 11-12). No entanto, se existe algum elemento armazenado naquela posição, então compararmos seu campo matrícula com o informado no parâmetro da função. Se os valores forem diferentes, o algoritmo irá realizar uma nova tentativa de busca em outra posição da tabela. Do contrário, o conteúdo da posição é copiado para o conteúdo do ponteiro passado por referência (**al**) e retornamos o valor **UM** (linhas 15-16), indicando sucesso na operação de busca.

### Buscando um elemento da tabela hash

```

01 int buscaHash_EnderAberto(Hash* ha, int mat,
                           struct aluno* al){
02     if(ha == NULL)
03         return 0;
04
05     int i, pos, newPos;
06     pos = chaveDivisao(mat, ha->TABLE_SIZE);
07     for(i=0; i < ha->TABLE_SIZE; i++){
08         newPos = sondagemLinear(pos, i, ha->TABLE_SIZE);
09         //newPos = sondagemQuadratica(pos, i,
10           //ha->TABLE_SIZE);
11         //newPos = duploHash(pos, mat, i, ha->TABLE_SIZE);
12         if(ha->itens[newPos] == NULL)
13             return 0;
14
15         if(ha->itens[newPos]->matricula == mat){
16             *al = *(ha->itens[newPos]);
17             return 1;
18         }
19     }
20 }

```

FIGURA 9.20

#### 9.8.2 Encadeamento separado

O encadeamento separado (ou separate chaining) é uma maneira um pouco diferente de se tratar da colisão em uma tabela hash. Em vez de procurar posições vagas (valor `NULL`) dentro do array que define a tabela, esse tipo de tratamento de colisões armazena dentro de cada posição do array o início de uma **lista dinâmica encadeada**. É dentro desta lista que serão armazenadas as colisões (elementos com chaves iguais) para aquela posição do array, como mostra a Figura 9.21.

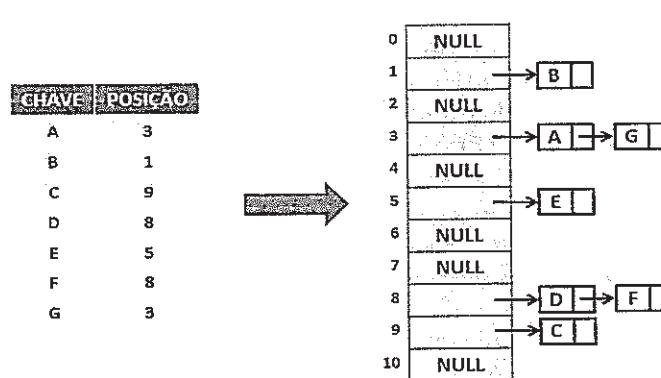


FIGURA 9.21

A **lista dinâmica encadeada** mantida em cada posição da tabela pode ser ordenada ou não. Ao usarmos uma lista não ordenada, é fácil perceber que essa estratégia de tratamento de colisão tem complexidade  $O(1)$  no pior caso: basta acessar a posição da tabela correspondente à chave daquele elemento e inserir o elemento no início da lista. Já a busca leva um tempo proporcional ao número de elementos dentro da lista armazenada naquela posição da tabela, ou seja, é preciso percorrer a lista procurando aquele elemento.



Uma desvantagem importante desse tipo de tratamento de colisão diz respeito à quantidade de memória consumida: gastamos mais memória para manter os ponteiros que ligam os diferentes elementos dentro de cada lista.

#### 9.9 EXERCÍCIOS

- 1) Defina, usando as suas palavras, o que é uma tabela hash e como ela funciona.
- 2) Cite duas características desejáveis para o bom funcionamento de uma função hash.
- 3) O *paradoxo do aniversário* afirma que, em uma sala contendo mais de 23 pessoas, a chance de duas pessoas fazerem aniversário no mesmo dia é maior do que 50%. Explique por que este paradoxo é um exemplo do maior problema das tabelas hash.
- 4) Descreva de que forma podemos detectar quando todas as posições possíveis vagas da tabela hash foram acessadas durante o reespalhamento.
- 5) Descreva, usando as suas palavras, o que é hashing universal.
- 6) Durante a inserção e a busca em tabelas hash, pode ocorrer colisão. Explique o que é uma colisão.
- 7) Descreva de que forma podemos tratar da colisão em tabelas hash.
- 8) Descreva, usando as suas palavras, como funciona o método de divisão. Cite um possível problema deste método.
- 9) Descreva as vantagens e desvantagens do método de endereçamento aberto.
- 10) Considere um conjunto de  $n$  chaves,  $x$ , formado pelos  $n$  primeiros múltiplos do número 7. Quantas colisões seriam obtidas para cada função hashing sugerida?
  - $h(x) = x \% 7$
  - $h(x) = x \% 14$
  - $h(x) = x \% 5$
- 11) Dada uma tabela hash de tamanho  $m = 9$ , função hash  $h(k) = k \% m$ , com encadeamento separado, mostre a tabela após a inserção das chaves 4, 23, 29, 15, 21, 43, 11, 5 e 10. Mostre também como ficaria a tabela após as inserções se seu tamanho fosse  $m = 11$ .
- 12) Dada uma tabela hash de tamanho  $m = 11$  função hash  $h(k) = (2k + 5)\%m$ , com encadeamento separado, mostre a tabela após a inserção das chaves 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 e 5.

13) Dada uma tabela hash de tamanho  $m = 10$ , com endereçamento aberto, mostre a tabela após a inserção das chaves 371, 121, 173, 203, 11, 24 para as seguintes funções de hash:

- Sondagem linear, função hash  $h(k) = k \% m + i$
- Sondagem quadrática, função hash  $h(k) = k \% m + i^2$
- Sondagem quadrática, função hash  $h(k) = k \% m + 2i + i^2$
- Hash duplo, função hash  $h1(k) = k \% m$ ; função hash  $h2(k) = 7 - (k \% 7)$

14) Insira a seguinte relação de 12 chaves em uma tabela hash com 3 cadeias de encadeamento. Considere uma busca pela chave J, cuja hash é 2. Qual é a sequência de chaves comparada com J?

chave	hash
D	2
Q	0
B	0
I	1
M	2
H	0
G	2
U	1
A	2
C	1
R	1
S	2

## CAPÍTULO 10

# Grafos

### 10.1 DEFINIÇÃO

Diversos tipos de aplicações necessitam que se represente um conjunto de objetos e as suas relações. Nestes casos, é interessante fazer uso de estruturas chamadas grafos.



Um grafo é um modelo matemático que representa as relações entre objetos de determinado conjunto.

Um grafo  $G(V, A)$  é definido em termos de dois conjuntos, como mostra a Figura 10.1:

- Um conjunto  $V$  de vértices, que são os itens representados em um grafo.
- Um conjunto  $A$  de arestas, que são utilizadas para conectar qualquer par de vértices. Neste caso, dois vértices são conectados segundo critério previamente estabelecido.



A teoria dos grafos fornece um extenso conjunto de ferramentas para a modelagem de um problema na forma de um grafo, de modo que essa modelagem se torna totalmente dependente da natureza do problema e dos objetivos que se pretende alcançar.

Praticamente qualquer objeto (seja ele um pixel de uma imagem, uma pessoa de um grupo de amigos ou mesmo uma cidade em um mapa) pode ser representado como um vértice em um grafo. Consequentemente, as ligações entre os vértices, isto é, as arestas do grafo, podem ser estabelecidas de acordo com alguma medida que represente adequadamente o relacionamento existente

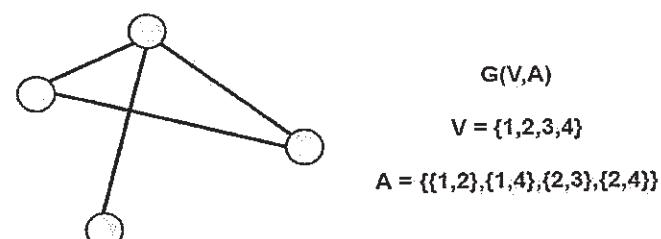


FIGURA 10.1

entre os pares de vértices. A grande flexibilidade dos grafos permite, por exemplo, a construção de grafos com diferentes tipos de vértices e arestas, isto é, vértices que representem entidades diferentes e arestas ligando uma mesma classe ou classes diferentes, como mostra a Figura 10.2.

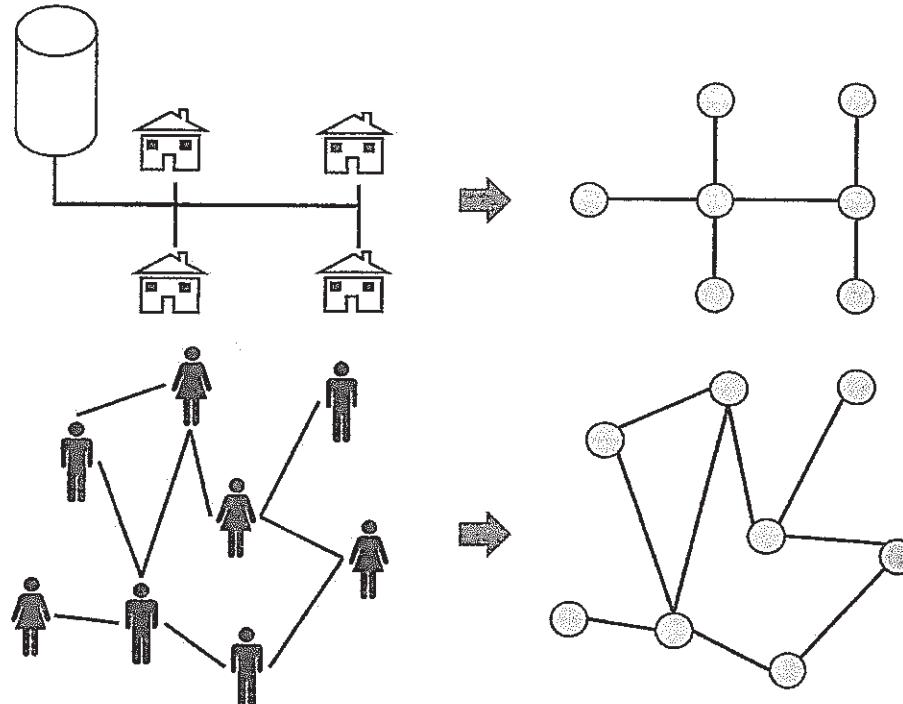


FIGURA 10.2

## 10.2 CONCEITOS BÁSICOS

### 10.2.1 Vértice

Um vértice é cada uma das entidades representadas em um grafo. O seu significado dentro do grafo depende da aplicação na qual o grafo é usado, ou seja, depende da natureza do problema modelado. Podem ser:

- Pessoas.
- Uma tarefa em um projeto.
- Lugares em um mapa.

Um exemplo de vértice é mostrado na Figura 10.3.

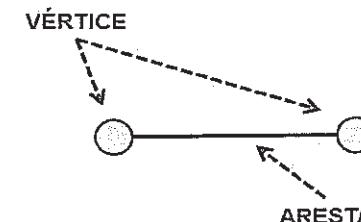


FIGURA 10.3

### 10.2.2 Arestas: conectando os vértices

Uma aresta (também chamada arco) está sempre associada a dois vértices. Ela é responsável por fazer a ligação entre esses dois vértices, ou seja, diz qual a relação que existe entre eles.



Dois vértices são considerados **adjacentes** se existir uma aresta ligando-os.

O significado da aresta dentro do grafo depende da aplicação na qual o grafo é usado, ou seja, depende da natureza do problema modelado. Pode ser, em um grafo de:

- Pessoas: parentesco entre elas ou amizade.
- Tarefas de um projeto: pré-requisito entre as tarefas.
- Lugares de um mapa: estradas que existem ligando os lugares.

Um exemplo de aresta é mostrado na Figura 10.3.

### 10.2.3 Direção das arestas: grafos e digrafos

Dependendo da aplicação na qual um grafo é usado, as arestas podem ou não ter uma direção associada a cada uma delas. Em um **grafo direcionado** ou **digrafo**, existe uma orientação quanto ao sentido da aresta, ou seja, se uma aresta liga os vértices  $A$  a  $B$ , isso significa que podemos ir de  $A$  para  $B$ , mas não o contrário. Já em um **grafo não direcionado** (ou simplesmente **grafo**), não existe nenhuma orientação quanto ao sentido da aresta, ou seja, se uma aresta liga os vértices  $A$  a  $B$ , isso significa que podemos ir de  $A$  para  $B$  ou de  $B$  para  $A$ . Um exemplo de grafo e grafo direcionado é mostrado na Figura 10.4.

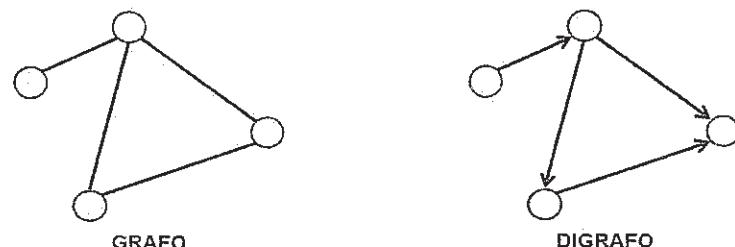


FIGURA 10.4

#### 10.2.4 Grau de um vértice

O grau de um vértice corresponde ao número de arestas que conectam aquele vértice a outro vértice do grafo. Em outras palavras, trata-se do número de vizinhos que aquele vértice possui no grafo. No caso dos digrafos, temos dois tipos de grau:

- **Grau de entrada:** corresponde ao número de arestas que chegam ao vértice partindo de outro.
- **Grau de saída:** corresponde ao número de arestas que partem do vértice em direção a outro.

A Figura 10.5 mostra o grau de cada vértice para um grafo e um digrafo.

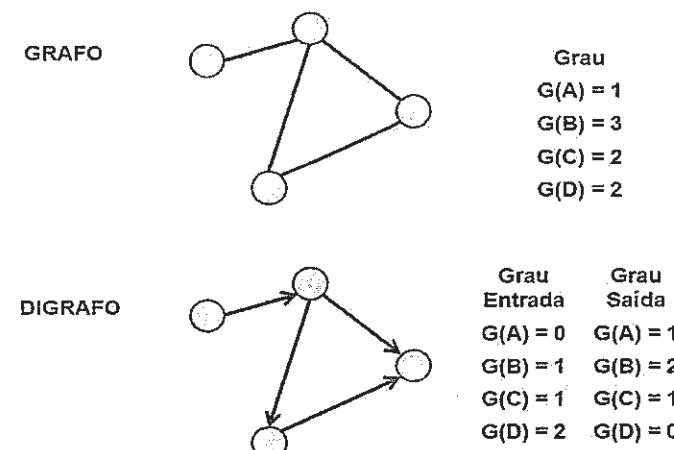


FIGURA 10.5

#### 10.2.5 Laços

Uma aresta é chamada **laço** se seu vértice de partida é o mesmo que o de chegada, ou seja, a aresta conecta o vértice a ele mesmo. A Figura 10.6 mostra um exemplo de laço em um grafo.

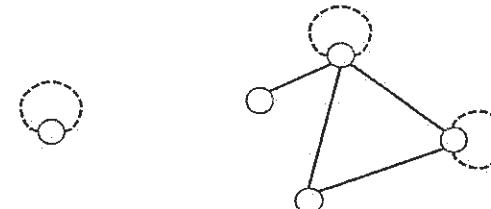


FIGURA 10.6

#### 10.2.6 Caminhos e ciclos

É comum em um grafo que dados dois vértices,  $v_1$  e  $v_5$ , não exista uma aresta conectando-os, isto é, eles não são adjacentes. De fato, a grande maioria dos grafos são esparsos, ou seja, apresentam apenas uma pequena fração de todas as arestas possíveis. Isso significa que um vértice se conecta com poucos vértices vizinhos. No entanto, dois vértices não adjacentes podem ser conectados por uma sequência de arestas, como  $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5)$ .



Um **caminho** entre dois vértices é uma sequência de vértices em que cada vértice está conectado ao vértice seguinte por meio de uma aresta. Neste caso, o número de vértices que precisamos percorrer de um vértice até o outro é denominado **comprimento do caminho**.

Desse modo, podemos dizer que o primeiro vértice (chamado vértice inicial) está conectado ao último vértice da sequência (chamado vértice final) se existir ao menos um caminho conectando-os. A Figura 10.7 mostra dois exemplos de caminhos em um grafo.



Um **caminho** é chamado **caminho simples** se nenhum dos vértices se repetir ao longo dele.

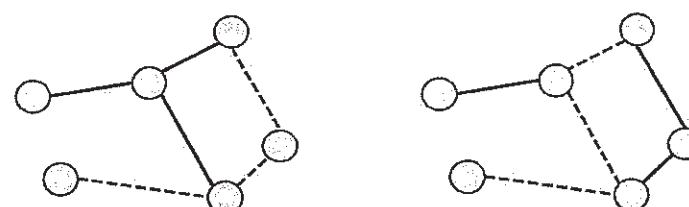


FIGURA 10.7

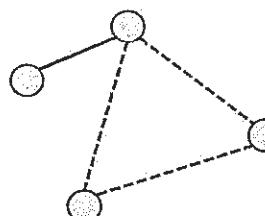


Um **ciclo** é um caminho em que os vértices inicial e final são o mesmo vértice. Neste caso, o **comprimento do ciclo** é o número de vértices que precisamos percorrer do vértice inicial até o final, onde o vértice final não é contado pois ele já foi contado como inicial.

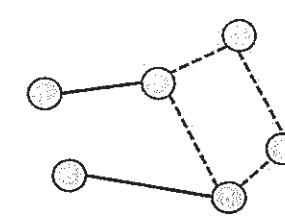
Note que um ciclo é um **caminho fechado** sem vértices repetidos e que a escolha do vértice inicial em um ciclo é uma tarefa arbitrária. Note também que um laço é um ciclo de comprimento 1. A Figura 10.8 mostra dois exemplos de ciclos em um grafo.



Um **grafo acíclico** é um grafo que não contém ciclos simples. Um ciclo é dito simples se cada vértice aparece apenas uma vez no ciclo.



CICLO: 2-3-4



CICLO: 2-3-4-5

FIGURA 10.8

### 10.2.7 Arestas múltiplas e multigrafo

Um grafo que possui arestas múltiplas é chamado **multigrafo**. Trata-se de um tipo de grafo especial que permite mais de uma aresta conectando o mesmo par de vértices. Neste caso, as arestas são ditas paralelas. A Figura 10.9 mostra um exemplo de **multigrafo**.



Para entender o conceito de arestas múltiplas, considere duas pessoas representadas por vértices em um grafo. No caso destes vértices estarem conectados por duas arestas, temos que uma aresta pode indicar uma relação de amizade entre as pessoas, enquanto uma segunda aresta poderia indicar uma relação de hierarquia dentro de uma empresa (por exemplo, fulano é chefe de beltrano).

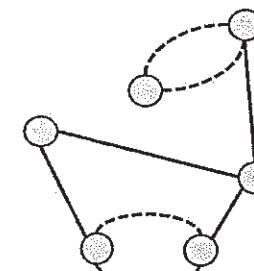


FIGURA 10.9

## 10.3 REPRESENTAÇÃO DE GRAFOS

Ao se modelar um problema utilizando um grafo, surge a questão: como representar este grafo no computador? Existem duas abordagens muito utilizadas para representar um grafo no computador. São elas:

- Matriz de adjacência.
- Lista de adjacência.



E que representação deve ser utilizada?

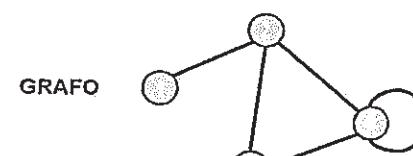
A representação escolhida para um grafo depende da aplicação. Não existe uma representação que seja melhor que a outra em todos os casos. A seguir, veremos como funciona cada um dos tipos de representação de grafo.

### 10.3.1 Matriz de adjacência

A representação de um grafo por **matriz de adjacência** faz uso de uma simples matriz para descrever as relações entre os vértices. Neste tipo de representação, um grafo contendo  $N$  vértices utiliza uma matriz com  $N$  linhas e  $N$  colunas para armazenar o grafo. Uma aresta ligando dois vértices é representada por uma **marca** (por exemplo, 1 existe aresta, 0 não existe) na posição  $(i,j)$  da matriz, sendo  $i$  o vértice inicial e  $j$  o vértice final da aresta, como mostra a Figura 10.10.

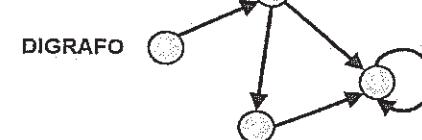


A representação de um grafo por **matriz de adjacência** possui um alto custo computacional,  $O(N^2)$ . Além disso, não é indicada para um grafo que possui muitos vértices mas poucas arestas ligando-os.



GRAFO

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



DIGRAFO

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

FIGURA 10.10

No entanto, se a matriz de adjacências armazenar somente a conectividade dos vértices (arestas), apenas um bit será necessário para cada posição da matriz. Isso torna essa representação bastante compacta. Por outro lado, operações como encontrar todos os vértices adjacentes a um vértice exigem que se pesquise em toda a linha da matriz.

### 10.3.2 Listas de adjacência

A representação de um grafo por **lista de adjacência** faz uso de uma lista de vértices para descrever as relações entre os vértices. Neste tipo de representação, um grafo contendo  $N$  vértices utiliza um array de ponteiros de tamanho  $N$  para armazenar os vértices do grafo. Em seguida, para cada vértice é criada uma lista de arestas, em que cada posição da lista armazena o índice do vértice e as quais vértices ele se conecta, como mostra a Figura 10.11.



A representação de um grafo por **lista de adjacência** possui um custo computacional  $O(N + M)$ , em que  $N$  é o número de vértices e  $M$  é o número de arestas no grafo.

Como se pode notar, a representação por **lista de adjacência** é mais indicada para um grafo que possui muitos vértices mas poucas arestas ligando-os. À medida que o número de arestas cresce, e não havendo nenhuma informação associada à aresta (por exemplo, seu peso), o uso de uma matriz de adjacência se torna mais eficiente. Além disso, descobrir se dois vértices estão conectados implica percorrer todas as arestas de um deles, enquanto na matriz de adjacência essa tarefa é imediata.

A seguir, veremos como implementar um TAD para representar um grafo usando uma lista de adjacência.

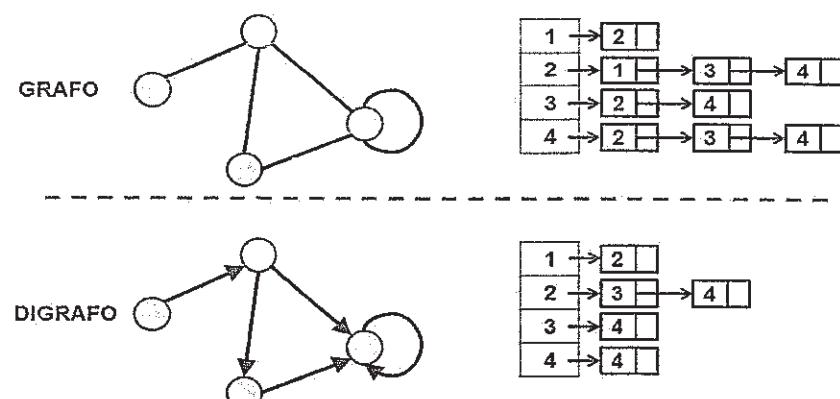


FIGURA 10.11

### 10.3.3 Criando o TAD grafo

#### Definindo o tipo grafo

Antes de começar a implementar o nosso grafo, é preciso definir algumas características, como o número de vértices que ele terá, se será um digrafo etc. Como estamos trabalhando com modularização, precisamos também definir o tipo opaco que representa nosso grafo. Este tipo será um ponteiro para a estrutura que define o grafo. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **Grafo.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **Grafo.h**, ilustrado na Figura 10.12. Como vamos modelar um grafo simples, temos que definir apenas o tipo opaco que será usado sempre que se desejar trabalhar com um grafo (linha 1) e as funções disponíveis para se trabalhar com esse grafo em especial (linhas 2-5) e que serão implementadas no arquivo **Grafo.c**.



No arquivo **Grafo.c**, iremos definir tudo aquilo que deve ficar oculto da nossa biblioteca e implementar as funções definidas em **Grafo.h**.

Basicamente, o arquivo **Grafo.c** (Figura 10.12) contém apenas:

- As chamadas às bibliotecas necessárias à implementação do grafo (linhas 1-3).
- A definição do tipo que descreve o funcionamento do grafo, **struct grafo** (linhas 5-12).
- As implementações das funções definidas no arquivo **Grafo.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que o nosso tipo grafo é uma estrutura contendo vários campos:

- **eh\_ponderado**: define se as arestas têm ou não peso (grafo ponderado ou não).
- **nro\_vert**: define o número de vértices que o grafo irá ter.
- **Gmax**: define o número de arestas com as quais um vértice poderá se conectar.
- **Arestas**: ponteiro no qual será alocada a matriz de arestas do grafo.
- **Pesos**: ponteiro no qual será alocada a matriz de pesos das arestas do grafo, se o grafo for ponderado.
- **Grau**: ponteiro no qual será alocado um vetor que irá armazenar o número de arestas já associadas a um vértice.

Por estarem definidos dentro do arquivo .c, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo `main()`, apenas o seu outro nome definido no arquivo `Grafo.h` (linha 1), que pode somente declarar um ponteiro para ele, da seguinte forma:

```
Grafo *gr;
```

#### Arquivo Grafo.h

```
01 typedef struct grafo Grafo;
02 Grafo* cria_Grafo(int nro_vert,int Gmax,int eh_ponderado);
03 void libera_Grafo(Grafo* gr);
04 int insereAresta(Grafo* gr,int orig,int dest,
                    int digrafo,float peso);
05 int removeAresta(Grafo* gr,int orig,int dest,int digrafo);
```

#### Arquivo Grafo.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Grafo.h" //inclui os Protótipos
04 //Definição do tipo Grafo
05 struct grafo{
06     int eh_ponderado;
07     int nro_vert;
08     int Gmax;
09     int** arestas;
10     float** pesos;
11     int* grau;
12 };
```

FIGURA 10.12

#### Criando e destruindo um grafo

Para utilizar um grafo em seu programa, a primeira coisa a fazer é criar um grafo vazio. Essa tarefa é executada pela função descrita na Figura 10.13. Basicamente, o que esta função faz é a alocação de uma área de memória para armazenar a estrutura que representa o grafo (linha 3). Caso essa operação seja feita com sucesso (linha 3), a função aloca a lista de adjacências (arestas) e configura outros campos da estrutura de acordo com o número de vértices (nro\_vert) e o número máximo de arestas que ele pode ter (Gmax), informados pelo usuário (linhas 6-13). Além disso, o usuário também informa se o grafo será ponderado ou não. Em caso afirmativo (linha 15), uma segunda estrutura (pesos), similar à lista de adjacências (arestas), é alocada para guardar o peso associado àquela aresta (linhas 16-18).

#### Criando um grafo

```
01 Grafo* cria_Grafo(int nro_vert,int Gmax,int eh_ponderado) {
02     Grafo *gr;
03     gr = (Grafo*) malloc(sizeof(struct grafo));
04     if(gr != NULL){
05         int i;
06         gr->nro_vert = nro_vert;
07         gr->Gmax = Gmax;
08         gr->eh_ponderado = (eh_ponderado != 0)?1:0;
09         gr->grau=(int*)calloc(nro_vert,sizeof(int));
10
11         gr->arestas=(int**)malloc(nro_vert*sizeof(int*));
12         for(i=0; i<nro_vert; i++)
13             gr->arestas[i] = (int*)malloc(Gmax * sizeof(int));
14
15         if(gr->eh_ponderado){
16             gr->pesos=(float**)malloc(nro_vert*sizeof(float*));
17             for(i=0; i<nro_vert; i++)
18                 gr->pesos[i]=(float*)malloc(Gmax*sizeof(float));
19         }
20     }
21     return gr;
22 }
```

FIGURA 10.13

O código que realiza a destruição do nosso grafo é mostrado na Figura 10.14. Inicialmente, verificamos se o grafo é válido, ou seja, se a tarefa de criação dele foi realizada com sucesso (linha 2). Em seguida, liberamos a lista de adjacências (arestas) anteriormente alocada (linhas 6-6). Caso o grafo seja ponderado (linha 8), o mesmo processo é realizado para os pesos (linhas 11-11). Por fim, liberamos a memória associada ao array de graus e a estrutura que representa o grafo (linhas 13-14).

**Descrição do código**

```

01 void libera_Grafo(Grafo* gr){
02     if(gr != NULL){
03         int i;
04         for(i=0; i<gr->nro_vert; i++)
05             free(gr->arestas[i]);
06         free(gr->arestas);
07
08         if(gr->eh ponderado){
09             for(i=0; i<gr->nro_vert; i++)
10                 free(gr->pesos[i]);
11             free(gr->pesos);
12         }
13         free(gr->grau);
14         free(gr);
15     }
16 }
```

FIGURA 10.14

**Inserindo uma aresta no grafo**

Uma vez criado o grafo, podemos começar a inserir as suas arestas, como mostra a sua implementação na Figura 10.15. Primeiramente, a função verifica se o ponteiro **Grafo \*gr** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação do grafo e, neste caso, não teríamos um grafo válido para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar um grafo inválido (linha 3). Porém, se o grafo foi criado com sucesso, precisamos verificar se os índices dos vértices inicial (**orig**) e final (**dest**) da aresta são valores válidos (linhas 4 e 6). Caso um deles não seja um valor válido, a função irá retornar o valor **ZERO** (linhas 5 e 7).

Tendo um grafo e vértices válidos, inserimos o vértice **dest** após o último vértice da lista de adjacências do vértice **orig** (linha 9). Caso o grafo seja ponderado, fazemos isso também para o seu peso (linhas 10-11). Terminada a inserção da aresta, incrementamos o total de arestas do vértice **orig** (linha 12).

Em seguida, verificamos se o grafo não é um digrafo. Em caso afirmativo, chamamos a função recursivamente, invertendo a ordem dos vértices, para que seja inserida a aresta no outro sentido (linhas 14-15). Por fim, retornamos o valor **UM** (linha 16), indicando sucesso na operação de inserção da aresta.

**Inserindo uma aresta**

```

01 int insereAresta(Grafo* gr,int orig,int dest,
02                   int digrafo,float peso){
03     if(gr == NULL)
04         return 0;
05     if(orig < 0 || orig >= gr->nro_vert)
06         return 0;
07     if(dest < 0 || dest >= gr->nro_vert)
08         return 0;
09
10     gr->arestas[orig][gr->grau[orig]] = dest;
11     if(gr->eh ponderado)
12         gr->pesos[orig][gr->grau[orig]] = peso;
13     gr->grau[orig]++;
14
15     if(digrafo == 0)
16         insereAresta(gr,dest,orig,1,peso);
17 }
```

FIGURA 10.15

**Removendo uma aresta do grafo**

Apesar de simples, remover uma aresta de um grafo é uma tarefa trabalhosa.



Isso ocorre porque precisamos procurar a aresta a ser removida na lista de adjacências, a qual pode estar no início, no meio ou no final da lista.

Basicamente, o que temos que fazer é procurar essa aresta na lista de adjacências e movimentar todas as arestas que estão à frente na lista uma posição para trás dentro do array. Isso sobrescreve a aresta a ser removida, ao mesmo tempo que diminui o número de arestas daquele vértice, como mostra a sua implementação na Figura 10.16. Primeiro, a função verifica se o ponteiro **Grafo \*gr** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação do grafo e, neste caso, não teríamos um grafo válido para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar um grafo inválido (linha 3). Porém, se o grafo foi criado com sucesso, precisamos verificar se os índices dos vértices inicial (**orig**) e final (**dest**) da aresta a ser removida são valores válidos (linhas 4 e 6). Caso um deles não seja um valor válido, a função irá retornar o valor **ZERO** (linhas 5 e 7).

Tendo um grafo e vértices válidos, temos que percorrer as arestas do vértice **orig** enquanto não chegarmos ao seu final, e enquanto o valor armazenado não for o vértice **dest** (linhas 9-11). Terminado o processo de busca, verificamos se estamos no final da lista de arestas ou não (linha 12). Em caso afirmativo, essa aresta não existe e a remoção não é possível (linha 13). Caso contrário, diminuímos em uma unidade a quantidade de arestas do vértice **orig** (linha 14) e copiamos o último elemento para a posição do elemento a ser removido (linhas 15). Desse

modo, não é preciso deslocar as arestas que estão à frente da aresta removida. Caso o grafo seja ponderado, fazemos isso também para o seu peso (linhas 16-17).

Em seguida, verificamos se o grafo não é um digrafo. Em caso afirmativo, chamamos a função recursivamente, invertendo a ordem dos vértices, para que seja removida a aresta no outro sentido (linhas 18-19). Por fim, retornamos o valor UM (linha 16), indicando sucesso na operação de remoção da aresta.

#### Removendo uma aresta

```

01 int removeAresta(Grafo* gr, int orig, int dest,
                     int eh_digrafo){
02     if(gr == NULL)
03         return 0;
04     if(orig < 0 || orig >= gr->nro_vertices)
05         return 0;
06     if(dest < 0 || dest >= gr->nro_vertices)
07         return 0;
08
09     int i = 0;
10    while(i<gr->grau[orig] && gr->arestas[orig][i]!=dest)
11        i++;
12    if(i == gr->grau[orig])//elemento não encontrado
13        return 0;
14    gr->grau[orig]--;
15    gr->arestas[orig][i]=gr->arestas[orig][gr->grau[orig]];
16    if(gr->eh_ponderado)
17        gr->pesos[orig][i]=gr->pesos[orig][gr->grau[orig]];
18    if(eh_digrafo == 0)
19        removeAresta(gr,dest,orig,i);
20    return 1;
21 }
```

FIGURA 10.16

## 10.4 TIPOS DE GRAFOS

### 10.4.1 Grafo trivial e simples

Um **grafo trivial** é a forma mais simples de grafo que existe. Trata-se de um grafo que possui um único vértice e nenhuma aresta ou laço. Já um **grafo simples** é a forma mais comum de grafo que existe. Trata-se de um grafo não direcionado, sem laços e sem arestas paralelas. A Figura 10.17 mostra um exemplo para cada um desses grafos.

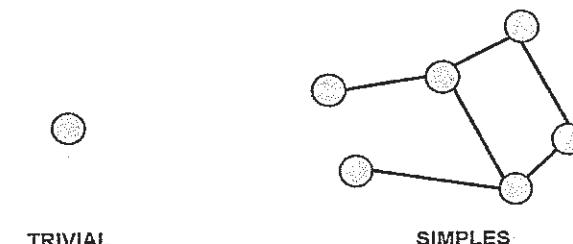


FIGURA 10.17

### 10.4.2 Grafo completo

Um **grafo completo** consiste em um grafo **simples** (ou seja, um grafo não direcionado, sem laços e sem arestas paralelas), onde cada vértice se conecta a todos os outros vértices do grafo. A Figura 10.18 mostra dois exemplos desse tipo de grafo.

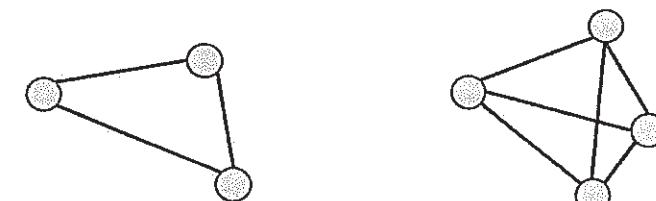


FIGURA 10.18

### 10.4.3 Grafo regular

Dá-se o nome de **grafo regular** a todo grafo cujos vértices todos possuem o mesmo grau (número de arestas ligadas a ele). A Figura 10.19 mostra alguns exemplos desse tipo de grafo.



Todo grafo completo é também regular, mas nem todo grafo regular é dito completo.

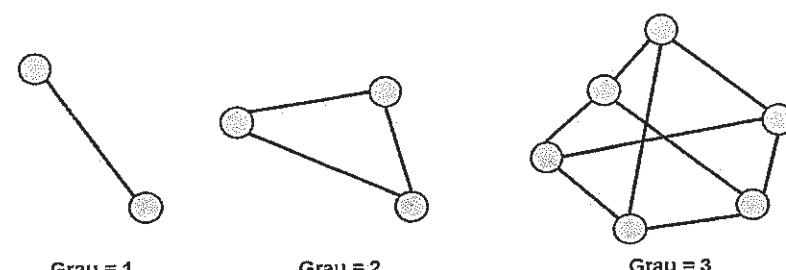


FIGURA 10.19

#### 10.4.4 Subgrafo

Dados dois grafos  $G(V, A)$  e  $G_S(V_S, A_S)$ , temos que  $G_S(V_S, A_S)$  é um **subgrafo** de  $G(V, A)$  se o conjunto de vértices  $V_S$  for um subconjunto de  $V$ ,  $V_S \subseteq V$ , e se o conjunto de arestas  $A_S$  for um subconjunto de  $A$ ,  $A_S \subseteq A$ . A Figura 10.20 mostra um exemplo de um grafo e seus subgrafos.

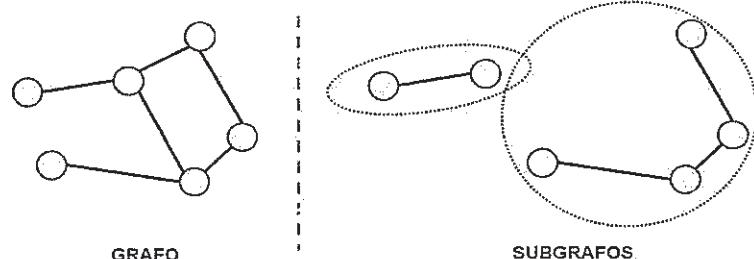


FIGURA 10.20

#### 10.4.5 Grafo bipartido

Um grafo  $G(V, A)$  é chamado **grafo bipartido** se o seu conjunto de vértices puder ser dividido em dois subconjuntos  $V_1$  e  $V_2$  sem intersecção. Já as arestas conectam apenas os vértices que estão em subconjuntos diferentes, ou seja, uma aresta sempre conecta um vértice de  $V_1$  a  $V_2$  ou vice-versa, porém ela nunca conecta vértices do mesmo subconjunto entre si. A Figura 10.21 mostra um exemplo de um grafo bipartido.

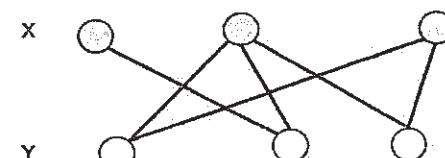


FIGURA 10.21

**Em um grafo bipartido, todo ciclo tem comprimento par.**

Assim, para verificar se um grafo é bipartido, basta checar se ele possui ao menos um ciclo de comprimento ímpar.

#### 10.4.6 Grafo conexo e desconexo

Chama-se de **grafo conexo** todo grafo em que, para quaisquer dois vértices distintos, sempre existe um caminho que os une. Quando isso não acontece, temos um **grafo desconexo**. Um **grafo desconexo** contém no mínimo duas partes, cada uma delas chamada **componente conexa**. A Figura 10.22 mostra exemplos de grafos conexo e desconexo.



Um grafo é **totalmente desconexo** quando possui mais de um vértice, mas nenhuma aresta.

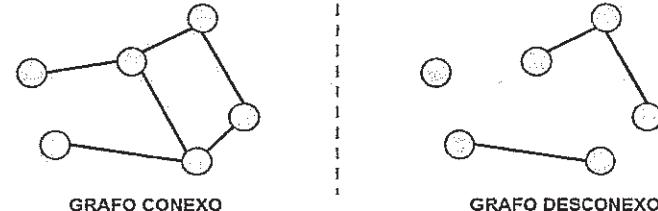


FIGURA 10.22

#### 10.4.7 Grafos isomorfos

Dois grafos  $G_1(V_1, A_1)$  e  $G_2(V_2, A_2)$  são ditos **isomorfos** se existir uma função que faça o mapeamento de vértices e arestas de modo que os dois grafos se tornem coincidentes. Em outras palavras, dois grafos são isomorfos se houver uma função  $f$  tal que, para cada dois vértices  $a$  e  $b$  adjacentes no grafo  $G_1$ ,  $f(a)$  e  $f(b)$  também sejam adjacentes no grafo  $G_2$ .

Condições mínimas para que dois grafos sejam isomorfos:

- Possuírem o mesmo número de vértices.
- Possuírem o mesmo número de arestas.
- Possuírem o mesmo número de vértices de grau  $n$ , para qualquer valor  $n$  entre 0 e o número de vértices que o grafo contém.

A Figura 10.23 mostra dois exemplos de grafos isomorfos.

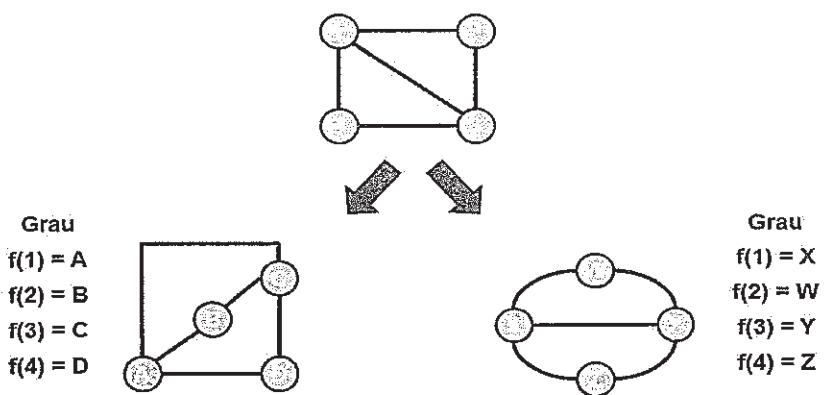


FIGURA 10.23

#### 10.4.8 Grafo ponderado

Dependendo da aplicação na qual um grafo é usado, as arestas podem ou não ter um peso (valor numérico) associado a cada uma delas. Quando isso ocorre, dizemos que o grafo é **ponderado**. Um exemplo de grafo ponderado é mostrado na Figura 10.24.



Muitos problemas de engenharia são modelados como grafos ponderados. Nesses casos, os valores associados às arestas representam grandezas como distâncias, altitudes, capacidades ou fluxos.

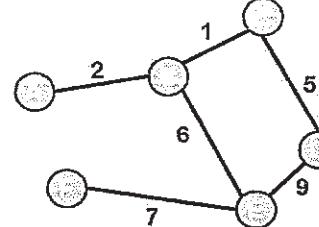


FIGURA 10.24

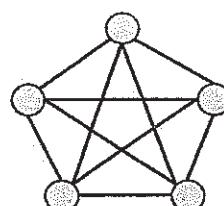
#### 10.4.9 Grafo hamiltoniano

Um **grafo hamiltoniano** é um tipo especial de grafo que possui um caminho que visita todos os seus vértices apenas uma vez. A esse caminho dá-se o nome de **caminho hamiltoniano**. Um **ciclo hamiltoniano** é um ciclo no qual cada vértice é visitado exatamente uma vez, retornando ao seu ponto de partida (esse é o único vértice que se repete). Exemplos de caminho e ciclo hamiltonianos são mostrados na Figura 10.25.

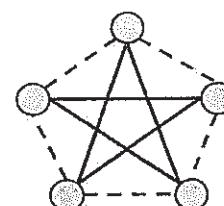


A detecção de um **grafo hamiltoniano** é uma tarefa extremamente árdua.

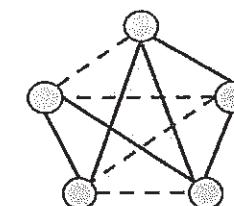
O problema de verificar se existe um caminho (ou ciclo) hamiltoniano em um grafo é NP-completo, ou seja, é pouco provável que exista um algoritmo polinomial para resolver isso.



GRAFO HAMILTONIANO



CICLO HAMILTONIANO



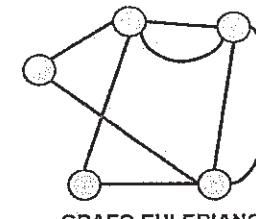
CAMINHO HAMILTONIANO

FIGURA 10.25

#### 10.4.10 Grafo euleriano

Um **grafo euleriano** é um tipo especial de grafo que possui um **ciclo** que visita todas as suas arestas apenas uma vez, iniciando e terminando no mesmo vértice. A esse **ciclo** dá-se o nome de **ciclo euleriano**. Um exemplo de ciclo euleriano é mostrado na Figura 10.26.

Um **grafo semieuleriano** é um tipo especial de grafo que possui um **caminho** aberto (não é um ciclo) que visita todas as suas arestas apenas uma vez. A esse **caminho** dá-se o nome de **caminho euleriano**. Um exemplo de caminho euleriano é mostrado na Figura 10.27.



GRAFO EULERIANO

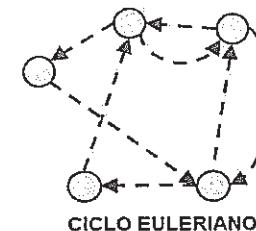
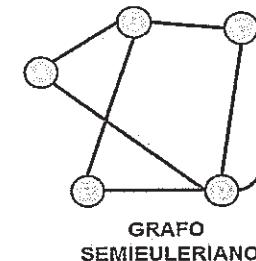
CICLO EULERIANO  
C-D-C-B-A-D-E-B-C

FIGURA 10.26



GRAFO SEMIEULERIANO

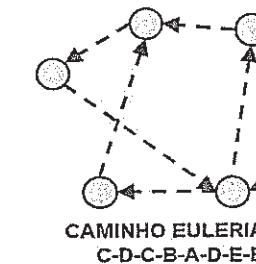
CAMINHO EULERIANO  
C-D-C-B-A-D-E-B

FIGURA 10.27

#### 10.5 ALGORITMOS DE BUSCA

A operação de busca consiste em explorar o grafo de uma maneira bem específica. Trata-se de um processo sistemático para caminhar por seus vértices e arestas.



De modo geral, as operações de busca dependem do vértice inicial.

O ponto de partida de uma busca é um aspecto bastante importante da própria busca. Por exemplo, em uma busca pelo menor caminho, temos que saber qual é o ponto de partida deste caminho.

As operações de busca são utilizadas para resolver uma série de problemas em grafos. Para alguns tipos de problemas, a busca pode precisar visitar todos os vértices, enquanto para outros,

apenas um subconjunto dos vértices precisa ser visitado. Existem vários tipos de busca que podemos realizar em um grafo. Nas seções seguintes, iremos abordar os três principais:

- Busca em profundidade.
- Busca em largura.
- Busca pelo menor caminho.

Para entender o funcionamento destes três tipos de busca, o grafo apresentado na Figura 10.28 será utilizado em todos os exemplos.

```
Gráfico para teste das buscas
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Grafo.h"
04 int main(){
05     int eh_digrafo = 1;
06     Grafo* gr = cria_Grafo(5, 5, 0);
07     insereAresta(gr, 0, 1, eh_digrafo, 0);
08     insereAresta(gr, 1, 3, eh_digrafo, 0);
09     insereAresta(gr, 1, 2, eh_digrafo, 0);
10     insereAresta(gr, 2, 4, eh_digrafo, 0);
11     insereAresta(gr, 3, 0, eh_digrafo, 0);
12     insereAresta(gr, 3, 4, eh_digrafo, 0);
13     insereAresta(gr, 4, 1, eh_digrafo, 0);
14
15     //realizar a busca aqui
16
17     libera_Grafo(gr);
18     system("pause");
19     return 0;
20 }
```

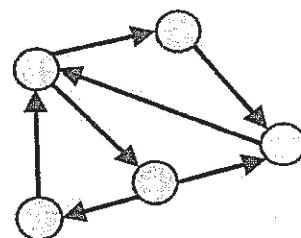


FIGURA 10.28

### 10.5.1 Busca em profundidade

O algoritmo de busca em profundidade pode ter o seu funcionamento assim descrito:



Partindo de um vértice inicial, a busca explora o máximo possível cada um dos vizinhos de um vértice antes de retroceder (**backtracking**).

Em outras palavras, esse tipo de busca se inicia em um vértice e se aprofunda nos vértices vizinhos até encontrar um dos dois casos: o alvo da busca ou um vértice sem vizinhos que possam ser visitados.



O algoritmo de busca em profundidade está relacionado com o conceito de **backtracking**.

Durante a busca, o grafo é percorrido de maneira sistemática. Esse processo se repete até que a busca falhe, ou se encontre um vértice sem vizinhos. Nesse momento, entra em funcionamento o mecanismo de **backtracking**: a busca retorna pelo mesmo caminho percorrido com o objetivo de encontrar um caminho alternativo. Trata-se de um mecanismo usado em linguagens de programação, como Prolog.

O código que realiza a busca em profundidade é mostrado na Figura 10.29. O algoritmo trabalha com o conceito de recursividade: ele usa uma função para realizar a busca, **buscaProfundidade** (linhas 1-8), e outra para inicializar a busca, **buscaProfundidade\_Grafo** (linhas 9-14).

A função que inicializa a busca, **buscaProfundidade\_Grafo**, recebe três parâmetros: o grafo (**gr**), o vértice inicial da busca (**ini**) e um array (**visitado**), cujo tamanho é o número de vértices do grafo. O array será usado para marcar a ordem em que cada vértice será visitado e este será o resultado da nossa função. Basicamente, a função marca todos os vértices como não visitados (linhas 11-12) e, em seguida, chama a função **buscaProfundidade** para o vértice inicial, **ini**, com o contador de visitação, **cont**, inicializado em 1.

Já a função **buscaProfundidade** recebe como parâmetros o grafo (**gr**), o vértice inicial da busca (**ini**), o array (**visitado**) e o contador de visitação (**cont**). A busca se inicia marcando o vértice inicial como visitado (linha 3). Em seguida, para cada vizinho de **ini** (linha 4) a função verifica se ele já foi marcado como visitado (linha 5). Caso ele não tenha sido visitado, a função **buscaProfundidade** é novamente chamada, tendo esse vizinho como vértice inicial da busca e o contador de visitação incrementado em uma unidade (linha 6). A busca termina quando não houver mais vizinhos a serem visitados. O array **visitado** contém agora a ordem em que cada vértice do grafo foi visitado, a partir do vértice inicial **ini**. Esse processo é mais bem ilustrado pela Figura 10.30, que considera o vértice 0 como o vértice inicial.

```

Busca em profundidade
01 void buscaProfundidade(Grafo *gr, int ini,
                           int *visitado, int cont){
02     int i;
03     visitado[ini] = cont;
04     for(i=0; i<gr->grau[ini]; i++){
05         if(!visitado[gr->arestas[ini][i]])
06             buscaProfundidade(gr, gr->arestas[ini][i],
07                                 visitado, cont+1);
08     }
09 void buscaProfundidade_Grafo(Grafo *gr, int ini,
                               int *visitado){
10     int i, cont = 1;
11     for(i=0; i<gr->nro_vertices; i++)
12         visitado[i] = 0;
13     buscaProfundidade(gr, ini, visitado, cont);
14 }
```

FIGURA 10.29

Considerando um grafo  $G(V, A)$ , em que  $|V|$  é o número de vértices e  $|A|$  é o número de arestas, a complexidade da busca em profundidade, no pior caso, tem:

- Custo para ir para cada vértice proporcional a  $|V|$ .
- Custo para transitar em cada aresta proporcional  $|A|$ .
- Complexidade da busca, no pior caso,  $O(|V| + |A|)$ .

Existem várias aplicações que fazem uso da busca em profundidade. Ela pode ser utilizada para:

- Encontrar componentes conectados e fortemente conectados.
- Ordenação topológica de um grafo.
- Procurar a saída de um labirinto.
- Verificar se um grafo é completamente conexo (por exemplo, se a rede de computadores está funcionando direito ou não).
- Implementar a ferramenta de preenchimento do Photoshop (balde de pintura).

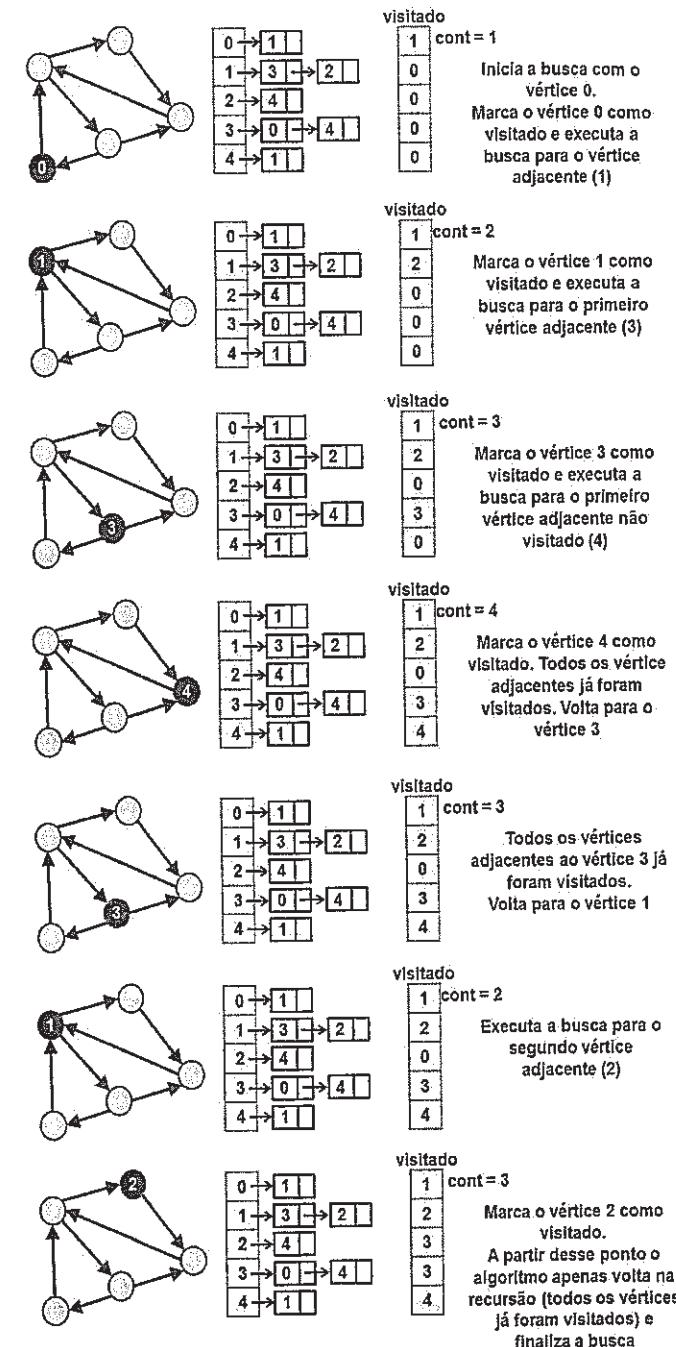


FIGURA 10.30

### 10.5.2 Busca em largura

O algoritmo de busca em largura pode ter o seu funcionamento assim descrito:



Partindo de um vértice inicial, a busca explora todos os vizinhos de um vértice. Em seguida, para cada vértice vizinho, ela repete esse processo, visitando os vértices ainda inexplorados.

Em outras palavras, esse tipo de busca se inicia em um vértice e, então, visita todos os seus vizinhos antes de se aprofundar na busca. Esse processo continua até que o alvo da busca seja encontrado ou não existam mais vértices a ser visitados.



O algoritmo de busca em largura faz uso do conceito de fila.

Durante a busca, o grafo é percorrido de maneira sistemática: primeiro, ela marca como “visitados” todos os vizinhos de um vértice e, em seguida, começa a visitar os vizinhos de cada vértice na ordem em que eles foram marcados. Para realizar essa tarefa, uma fila é utilizada para administrar a visitação dos vértices: o primeiro vértice marcado (ou marcado há mais tempo) é o primeiro a ser visitado.

O código que realiza a busca em largura é mostrado na Figura 10.31. A função recebe três parâmetros: o grafo (gr), o vértice inicial da busca (ini) e um array (visitado), cujo tamanho é o número de vértices do grafo. O array será usado para marcar a ordem em que cada vértice será visitado e esse será o resultado da nossa função. Inicialmente, marcamos todos os vértices como não visitados (linhas 4-5). Em seguida, criamos um array auxiliar fila (linhas 7-8). Como o próprio nome diz, este array será utilizado como sendo nossa fila estática, com IF e FF o início e o final da fila, respectivamente.

Depois, inserimos o vértice inicial no final da fila e o marcamos como visitado (linhas 9-11). Tem então início a busca. Enquanto houver vértices na fila (linha 12), o seguinte conjunto de passos será realizado:

- Remove-se um vértice da fila, vert (linhas 13-14).
- Incrementa-se o contador de visitação, cont (linha 15).
- Para cada vizinho de vert (linha 16):
  - Verifica-se se ele já foi marcado como visitado (linha 17).
  - Caso ele não tenha sido visitado, ele é inserido no final da fila e marcado como visitado (linhas 18-20).

Uma vez que a fila fique vazia (linha 12), a busca termina e a fila pode ser destruída (linha 24). O array visitado contém agora a ordem em que cada vértice do grafo foi visitado, a partir do vértice inicial ini. Esse processo é mais bem ilustrado pela Figura 10.32, que considera o vértice 0 como o vértice inicial.

```

Busca em largura
01 void buscaLargura_Grafo(Grafo *gr,int ini,int *visitado){
02     int i,vert,NV,cont=1;
03     int *fila, IF = 0, FF = 0;
04     for(i=0; i<gr->nro_vertices; i++)
05         visitado[i] = 0;
06
07     NV = gr->nro_vertices;
08     fila = (int*) malloc(NV * sizeof(int));
09     FF++;
10     fila[FF] = ini;
11     visitado[ini] = cont;
12     while(IF != FF){
13         IF = (IF + 1) % NV;
14         vert = fila[IF];
15         cont++;
16         for(i=0; i<gr->grafo[vert]; i++){
17             if(!visitado[gr->arestas[vert][i]]){
18                 FF = (FF + 1) % NV;
19                 fila[FF] = gr->arestas[vert][i];
20                 visitado[gr->arestas[vert][i]] = cont;
21             }
22         }
23     }
24     free(fila);
25 }
```

FIGURA 10.31

Considerando um grafo  $G(V, A)$ , em que  $|V|$  é o número de vértices e  $|A|$  é o número de arestas, a complexidade da busca em largura, no pior caso, tem:

- Custo de inserção e remoção em fila constante.
- Custo de enfileirar e remover todos os vértices uma vez  $O(|V|)$ .
- Custo de utilizar todas as arestas  $O(|A|)$ .
- Complexidade da busca, no pior caso,  $O(|V| + |A|)$ .

Existem várias aplicações que fazem uso da busca em largura. Ela pode ser utilizada para:

- Achar todos os vértices conectados a apenas um componente.
- Achar o menor caminho entre dois vértices.
- Testar se um grafo é bipartido.
- Roteamento: encontrar um número mínimo de hops em uma rede. Os hops são os vértices intermediários no caminho correspondente à conexão.
- Encontrar um número mínimo de intermediários entre duas pessoas.

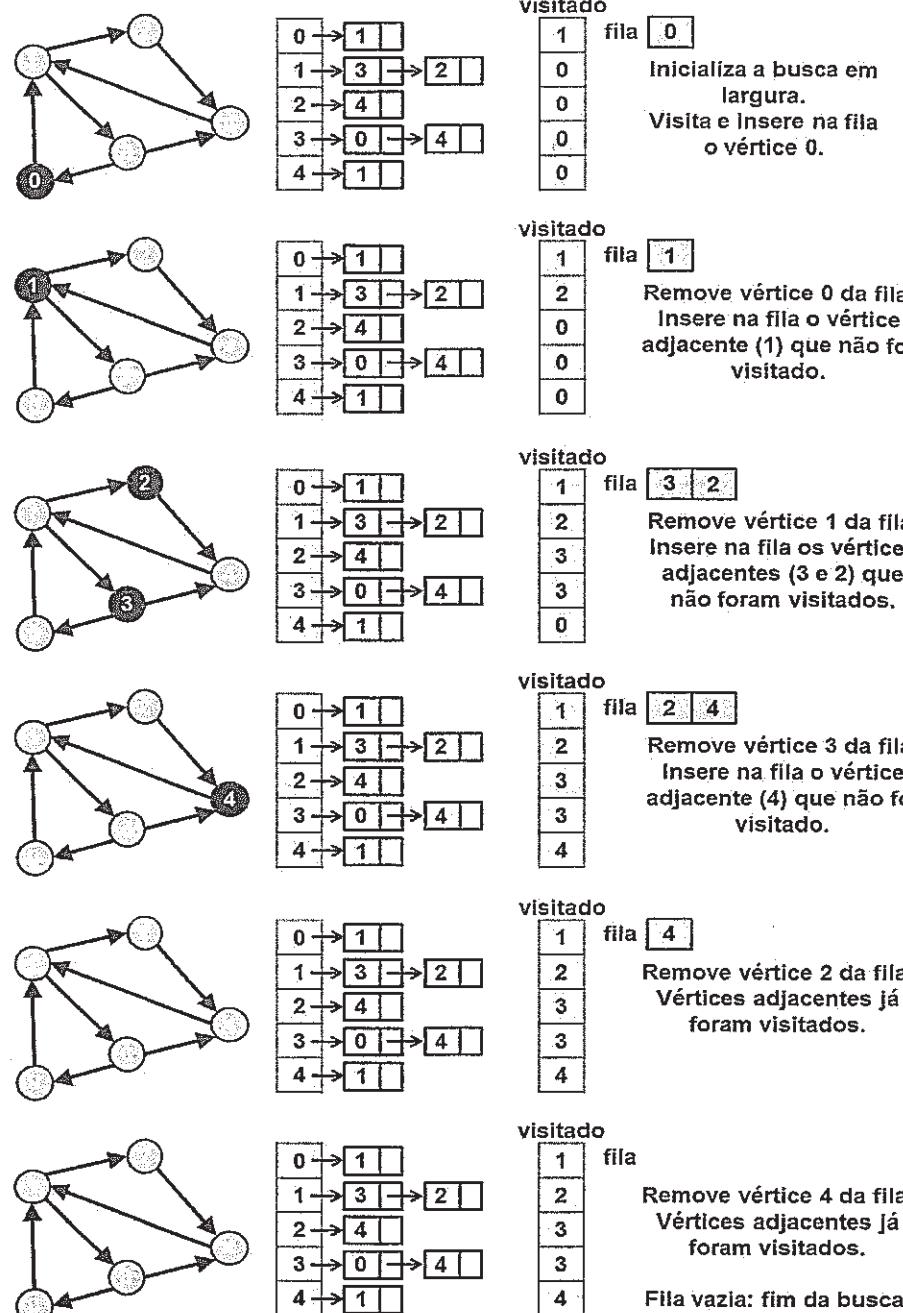


FIGURA 10.32

### 10.5.3 Menor caminho entre dois vértices

O menor caminho entre dois vértices é a aresta que os conecta. No entanto, é muito comum em um grafo não existir uma aresta conectando dois vértices  $v_1$  e  $v_5$ , isto é, eles não são adjacentes. Apesar disso, dois vértices que não são adjacentes podem ser conectados por uma sequência de arestas, como  $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5)$ . Caso essa seja a menor sequência de aresta que liga os dois vértices, dizemos que é o **menor caminho** ou **caminho mais curto** ou **caminho geodésico** entre eles.

O menor caminho ou caminho geodésico entre dois vértices é o caminho que apresenta o menor comprimento dentre todos os possíveis que conectam esses vértices.

No caso, o comprimento se refere ao número de arestas que conectam os dois vértices. Considerando um grafo ponderado, podemos também calcular o comprimento do caminho como sendo a soma dos pesos das arestas que compõem esse caminho.

Uma das maneiras de achar o menor caminho é utilizando o algoritmo de Dijkstra.

O algoritmo de Dijkstra é talvez o mais conhecido algoritmo para resolver esse problema. Ele resolve o problema do menor caminho para grafos e digrafos, ponderados ou não. Sua limitação é que, no caso de um grafo ponderado, as arestas não podem ter pesos negativos. Seu funcionamento pode ser assim brevemente descrito:

Partindo de um vértice inicial, o algoritmo de Dijkstra calcula a menor distância deste vértice a todos os demais (desde que exista um caminho entre eles).

O código que realiza a busca pelo menor caminho é mostrado na Figura 10.33. O algoritmo utiliza uma função para achar o vizinho mais próximo de um vértice que ainda não tenha sido visitado, **procuraMenorDistancia** (linhas 1-15), e outra para realizar a busca pelo caminho, **menorCaminho\_Grafo** (linhas 16-47).

A função **procuraMenorDistancia** recebe três parâmetros: o array de distâncias (**dist**), o array de vértices visitados (**visitado**) e o número de vértices do grafo (**NV**). Basicamente, a função testa todos os vértices (linha 3) à procura daquele que tiver a menor distância **não negativa** e que ainda não tenha sido visitado (linhas 4-12). O índice do vértice que satisfaz essas condições é o retorno da função.

Já a função **menorCaminho\_Grafo** recebe como parâmetros o grafo (**gr**), o vértice inicial (**ini**) e dois arrays: **ant**, para armazenar o antecessor do vértice dentro do caminho, e **dist**, que armazenará a distância do vértice inicial até ele. Esses arrays têm tamanho igual ao número de vértices do grafo.

```

    Menor caminho entre dois vértices
01 int procuraMenorDistancia(float *dist, int *visitado,
                                int NV){
02     int i, menor = -1, primeiro = 1;
03     for(i=0; i < NV; i++){
04         if(dist[i] >= 0 && visitado[i] == 0){
05             if(primeiro){
06                 menor = i;
07                 primeiro = 0;
08             }else{
09                 if(dist[menor] > dist[i])
10                     menor = i;
11             }
12         }
13     }
14     return menor;
15 }
16 void menorCaminho_Grafo(Grafo *gr, int ini,
                           int *ant, float *dist){
17     int i, cont, NV, ind, *visitado, vert;
18     cont = NV = gr->nro_vertices;
19     visitado = (int*) malloc(NV * sizeof(int));
20     for(i=0; i < NV; i++){
21         ant[i] = -1;
22         dist[i] = -1;
23         visitado[i] = 0;
24     }
25     dist[ini] = 0;
26     while(cont > 0){
27         vert = procuraMenorDistancia(dist, visitado, NV);
28         if(vert == -1)
29             break;
30
31         visitado[vert] = 1;
32         cont--;
33         for(i=0; i<gr->grau[vert]; i++){
34             ind = gr->arestas[vert][i];
35             if(dist[ind] < 0){
36                 dist[ind] = dist[vert] + 1;
37                 ant[ind] = vert;
38             }else{
39                 if(dist[ind] > dist[vert] + 1){
40                     dist[ind] = dist[vert] + 1;
41                     ant[ind] = vert;
42                 }
43             }
44         }
45     }
46     free(visitado);
47 }

```

FIGURA 10.33

Inicialmente, criamos uma variável para guardar o número de vértices que faltam ser visitados, **cont**, e um array auxiliar **visitado** para gerenciar os vértices que ainda precisam ser visitados (linhas 18-19). Em seguida, marcamos todos os vértices como não possuindo um antecessor (-1), distância inválida (-1) e não visitados (0) (linhas 20-24). Por fim, marcamos que a distância até o vértice inicial é 0 (linha 25). Tem então início a busca. Enquanto houver vértices a visitar (linha 26), o seguinte conjunto de passos será realizado:

- Use a função **procuraMenorDistancia** para achar o vértice com a menor distância e que ainda não foi visitado, **vert** (linha 27). Caso o índice do vértice seja inválido (-1), a busca termina (linhas 28-29).
- Marque esse vértice como visitado e diminua o número de vértices a serem visitados (linhas 31-32).
- Para cada vizinho de **vert** (linha 33):
  - Se a distância até ele for negativa (linha 35), a distância passa a ser a distância de **vert** mais uma unidade (ou seja, mais uma aresta) e **vert** se torna o antecessor desse vértice no caminho (linhas 36-37).
  - Se a distância até ele for positiva (linha 38), verifique se a distância de **vert** mais uma unidade é menor do que a distância atual (linha 39). Em caso afirmativo, essa passa a ser a nova distância até o vértice e **vert** se torna o antecessor desse vértice no caminho (linhas 40-41).

Uma vez que não existam mais vértices a ser visitados (linha 26), a busca termina e o array auxiliar pode ser destruído (linha 46). Os arrays **ant** e **dist** contêm agora os antecessores e a distância de cada vértice do grafo no caminho traçado a partir do vértice inicial **ini**. Esse processo é mais bem ilustrado pela Figura 10.34, que considera o vértice 0 como o vértice inicial.

Existem várias aplicações que fazem uso do cálculo do menor caminho. Ele pode ser utilizado:

- Para achar o grau de separação entre duas pessoas em uma rede social.
- Para achar um trajeto em um mapa rodoviário.
- Para programar robôs para explorar áreas.
- Em algoritmos de roteamento.

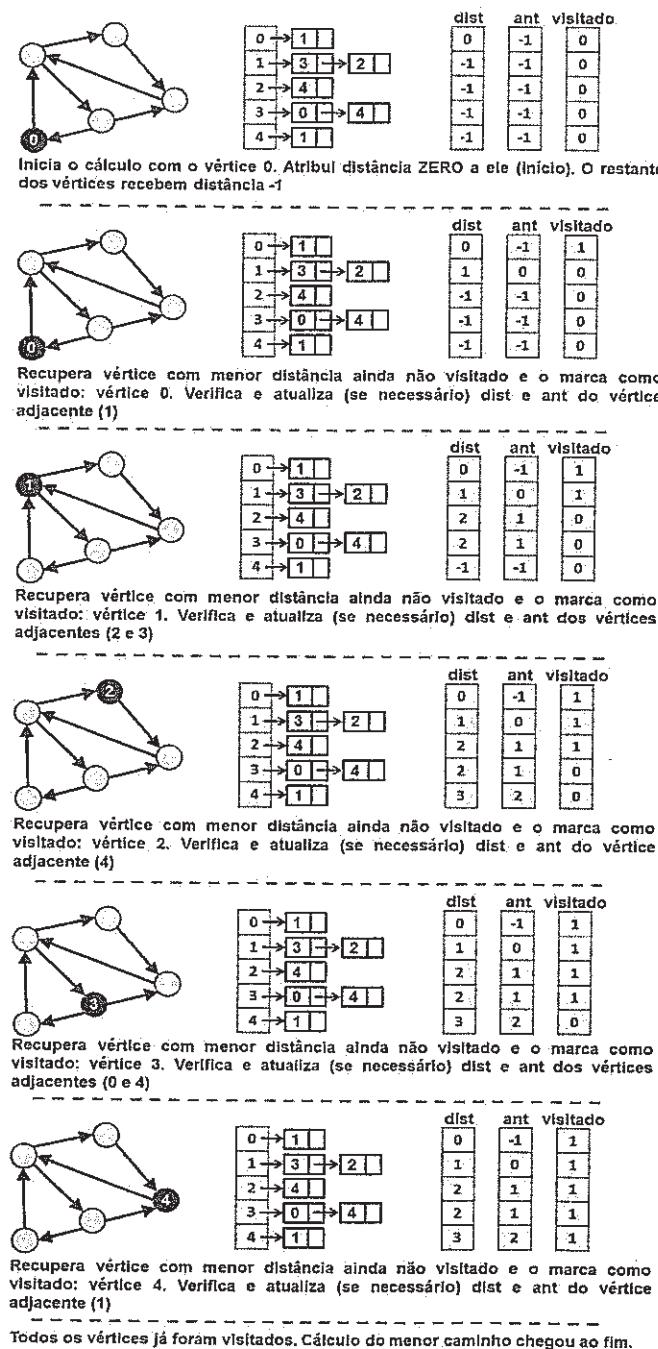


FIGURA 10.34

## 10.6 ÁRVORE GERADORA MÍNIMA

Dado um grafo  $G(V, A)$ , uma árvore geradora (spanning tree) é um subgrafo que contém todos os vértices do grafo original e um conjunto de arestas que permite conectar todos esses vértices na forma de uma árvore. Em outras palavras, uma árvore geradora é a menor estrutura que conecta todos os vértices do grafo. A árvore geradora possui:

Todos os vértices  $V$ .

Um total de arestas igual ao **número de vértices menos um** ( $|V| - 1$  arestas).

Se considerarmos que nosso grafo é ponderado (ou seja, suas arestas possuem um peso), podemos querer encontrar o conjunto de arestas de menor custo que conecte todos os vértices do grafo. Neste caso, teremos uma árvore geradora mínima (minimum spanning tree), como mostrado na Figura 10.35.

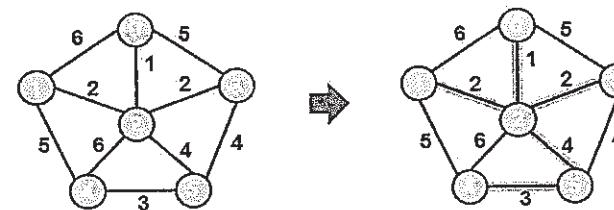


FIGURA 10.35



De modo geral, para um grafo possuir uma árvore geradora mínima ele deve satisfazer as seguintes propriedades: não direcionado, conexo e ponderado.

Como a árvore geradora conecta todos os vértices do grafo, espera-se que ele seja conexo, ou seja, para quaisquer dois vértices distintos, sempre existirá um caminho que os une. Esse tipo de situação é mais fácil de ocorrer em um grafo no qual as arestas apenas conectam os vértices, sem informação de direção (não direcionado). Como todos os vértices estão conectados, calcular a árvore geradora não depende do vértice inicial. No caso do grafo não ser conexo, podemos usar o vértice inicial para achar a árvore geradora de cada **componente conexa**. Por fim, o peso da aresta nos permite encontrar uma árvore geradora específica dentre todas possíveis: a árvore geradora mínima ou árvore geradora de custo mínimo.

Existem várias aplicações que fazem uso da árvore geradora mínima. Ela pode ser utilizada para:

- Transporte aéreo: mapa de conexões de voo.
- Transporte terrestre: infraestrutura das rodovias com o menor uso de material.
- Redes de computadores: conectar uma série de computadores com a menor quantidade de fibra ótica possível.
- Redes elétricas e telefônicas: unir um conjunto de localidades com menor gasto.

- Circuitos integrados.
- Análise de clusters.
- Armazenamento de informações.

O problema de encontrar uma árvore geradora mínima pode ser resolvido usando uma estratégia gulosa que constrói a árvore incrementalmente. Nas seções seguintes, iremos abordar dois algoritmos clássicos capazes de obter soluções ótimas, sendo que a diferença entre eles está na regra utilizada para encontrar a aresta que fará parte da árvore:

- Algoritmo de Prim.
- Algoritmo de Kruskal.

Para entender o funcionamento dos dois algoritmos, o grafo apresentado nas Figuras 10.35 e 10.36 será utilizado para o cálculo da árvore geradora mínima.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Grafo.h"
04 int main(){
05     int eh_digrafo = 0; //não é direcionado
06     Grafo* gr = cria_Grafo(6, 6, 1); //é ponderado
07     insereAresta(gr, 0, 1, eh_digrafo, 6);
08     insereAresta(gr, 0, 2, eh_digrafo, 1);
09     insereAresta(gr, 0, 3, eh_digrafo, 5);
10     insereAresta(gr, 1, 2, eh_digrafo, 2);
11     insereAresta(gr, 1, 4, eh_digrafo, 5);
12     insereAresta(gr, 2, 3, eh_digrafo, 2);
13     insereAresta(gr, 2, 4, eh_digrafo, 6);
14     insereAresta(gr, 2, 5, eh_digrafo, 4);
15     insereAresta(gr, 3, 5, eh_digrafo, 4);
16     insereAresta(gr, 4, 5, eh_digrafo, 3);
17     int i, pai[6];
18     //algoritmoPRIM_Grafo(gr, 0, pai);
19     //algoritmoKruskal_Grafo(gr, 0, pai);
20     for(i=0; i<6; i++)
21         printf("%d: %d\n", pai[i], i);
22     libera_Grafo(gr);
23     system("pause");
24     return 0;
25 }
```

FIGURA 10.36

### 10.6.1 Algoritmo de Prim

O algoritmo de Prim é um algoritmo clássico capaz de obter uma solução ótima para o problema da árvore geradora mínima. Ele pode ter o seu funcionamento assim descrito:



Partindo de um vértice inicialmente na árvore, o algoritmo procura a aresta de **menor peso** que conecta um vértice da árvore a outro que ainda não esteja na árvore. Este vértice é então adicionado na árvore e o processo se repete até que todos os vértices façam parte da árvore, ou quando não se possa encontrar uma aresta que satisfaça essa condição.

Em outras palavras, o algoritmo de Prim inicia a árvore com um vértice qualquer e adiciona um novo vértice à árvore a cada iteração. Esse processo continua até que todos os vértices do grafo façam parte da árvore, ou não seja possível achar uma aresta de menor peso conectando os vértices (grafo desconexo).

O algoritmo de Prim é mostrado na Figura 10.37. Essa função recebe três parâmetros: o grafo (**gr**), o vértice que será o ponto de partida para crescer a árvore (**orig**), e um array (**pai**), cujo tamanho é o número de vértices do grafo. O array será usado para marcar quem é o pai de cada vértice na árvore resultante.

Inicialmente, marcamos todos os vértices como sem pai (-1) (linhas 5-6). Em seguida, marcamos que o pai do vértice inicial é ele mesmo (linha 8). Têm então início o algoritmo. Enquanto for possível achar um vértice sem pai (linha 9), o seguinte conjunto de passos será realizado:

- Verifique quais vértices do grafo já possuem pai (linhas 11-12).
- Para cada vértice com pai, percorra suas arestas procurando vértices vizinhos que não possuem pai (linhas 13-14).
- Caso esse seja o primeiro vértice vizinho visitado que não possui pai, considere que ele possui o **menor peso** e guarde os vértices que formam essa aresta do grafo: **orig** e **dest** (linhas 15-19).
- Caso esse **não** seja o primeiro vértice vizinho visitado que não possui pai, verifique se o peso da aresta é **menor** do que a da aresta armazenada e guarde os vértices que formam essa aresta do grafo (**orig** e **dest**), se necessário (linhas 20-25).

Uma vez que se tenha percorrido todos os vértices do grafo, verifique se foi possível achar uma aresta de menor peso (linha 31). Caso não tenha sido possível, o processo termina (linha 32). Do contrário, definimos o vértice **orig** como pai do vértice **dest** na árvore e o algoritmo continua sua execução (linha 34). Ao final, o array **pai** irá conter os antecessores de cada vértice do grafo na árvore montada a partir do vértice inicial **orig** (o qual é pai de si mesmo). Esse processo é mais bem ilustrado pela Figura 10.38, que considera o vértice 0 o vértice inicial.

**Algoritmo de Prim**

```

01 void algoritmoPRIM_Grafo(Grafo *gr,int orig,int *pai){
02     int i, j, dest, NV, primeiro;
03     double menorPeso;
04     NV = gr->nro_vertices;
05     for(i=0; i < NV; i++)
06         pai[i] = -1;// sem pai
07
08     pai[orig] = orig;
09     while(1){
10         primeiro = 1;
11         for(i=0; i < NV; i++){
12             if(pai[i] != -1){
13                 for(j=0; j<gr->grau[i]; j++){
14                     if(pai[gr->arestas[i][j]] == -1){
15                         if(primeiro){
16                             menorPeso = gr->pesos[i][j];
17                             orig = i;
18                             dest = gr->arestas[i][j];
19                             primeiro = 0;
20                         }else{
21                             if(menorPeso > gr->pesos[i][j]){
22                                 menorPeso = gr->pesos[i][j];
23                                 orig = i;
24                                 dest = gr->arestas[i][j];
25                             }
26                         }
27                     }
28                 }
29             }
30             if(primeiro == 1)
31                 break;
32
33             pai[dest] = orig;
34         }
35     }
36 }
```

FIGURA 10.37

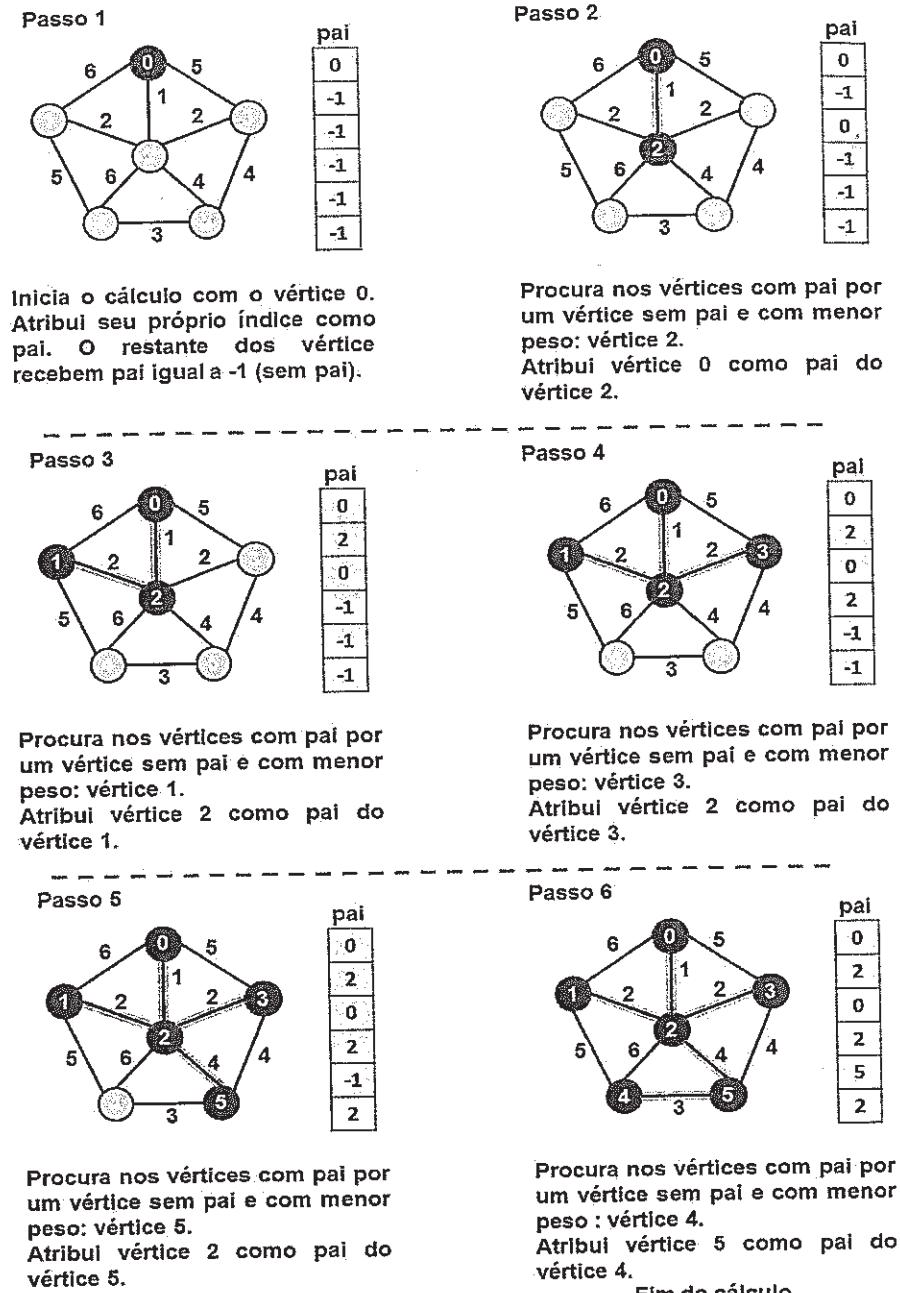


FIGURA 10.38



Essa versão do algoritmo de Prim deve ser usada apenas para fins didáticos, pois é muito ineficiente.

Considerando um grafo  $G(V, A)$ , em que  $|V|$  é o número de vértices e  $|A|$  o número de arestas, a complexidade dessa versão do algoritmo de Prim, no pior caso, é  $O(|V| * |A|)$ . Como o valor de  $|A|$  é proporcional a  $|V|^2$ , seu custo no pior caso é  $O(|V|^3)$ .



A eficiência do algoritmo de Prim depende da forma usada para encontrar a aresta de menor peso. Usando uma **fila de prioridades** para achar a aresta, o custo pode ser reduzido para  $O(|A| \log |V|)$ , no pior caso.

### 10.6.2 Algoritmo de Kruskal

O algoritmo de Kruskal é outro algoritmo clássico capaz de obter uma solução ótima para o problema da árvore geradora mínima. Ele pode ter o seu funcionamento assim descrito:



Considerando cada vértice uma árvore independente, o algoritmo procura a aresta de menor peso que conecta duas árvores diferentes. Os vértices das árvores selecionadas passam a fazer parte de uma mesma árvore. O processo se repete até que todos os vértices façam parte de uma mesma árvore, ou quando não se possa encontrar uma aresta que satisfaça essa condição.

Em outras palavras, o algoritmo de Kruskal se inicia com uma floresta, ou seja, várias árvores. A cada iteração, duas árvores são selecionadas para ser unidas em uma mesma árvore. Esse processo continua até que todos os vértices do grafo façam parte da mesma árvore, ou não seja possível achar uma aresta de menor peso conectando os vértices (grafo desconexo).

O algoritmo de Kruskal é mostrado na Figura 10.39. Essa função recebe três parâmetros: o grafo (*gr*), o vértice que será o ponto de partida para crescer a árvore (*orig*) e um array (*pai*), cujo tamanho é o número de vértices do grafo. O array será usado para marcar quem é o pai de cada vértice na árvore resultante.

Inicialmente, criamos um array auxiliar *arv* para gerenciar a qual árvore cada vértice pertence (linha 5). Em seguida, marcamos todos os vértices como pertencentes à própria árvore e sem pai (-1) (linhas 6-9). Também marcamos que o pai do vértice inicial é ele mesmo (linha 10). Têm então início o algoritmo. Enquanto for possível achar uma aresta ligando dois vértices de árvores diferentes (linha 11), o seguinte conjunto de passos será realizado:

- Para cada vértice, percorra suas arestas procurando por vértices vizinhos que estejam em uma árvore diferente (linhas 13-15).
- Caso essa seja a primeira aresta encontrada conectando árvores diferentes, considere que ela possui o **menor peso** e guarde os vértices que formam essa aresta do grafo: *orig* e *dest* (linhas 16-20).
- Caso essa **não** seja a primeira aresta encontrada conectando árvores diferentes, verifique se o peso da aresta é **menor** do que a da aresta armazenada e guarde os vértices que formam essa aresta do grafo (*orig* e *dest*), se necessário (linhas 21-26).

```

Algoritmo de Kruskal
01 void algoritmoKruskal_Grafo(Grafo *gr,int orig,int *pai){
02     int i, j, dest, NV, primeiro, *arv;
03     double menorPeso;
04     NV = gr->nro_vertices;
05     arv = (int*) malloc(NV * sizeof(int));
06     for(i=0; i < NV; i++){
07         arv[i] = i;
08         pai[i] = -1;// sem pai
09     }
10    pai[orig] = orig;
11    while(1){
12        primeiro = 1;
13        for(i=0; i < NV; i++){
14            for(j=0; j<gr->grau[i]; j++){
15                if(arv[i] != arv[gr->arestas[i][j]]){
16                    if(primeiro){
17                        menorPeso = gr->pesos[i][j];
18                        orig = i;
19                        dest = gr->arestas[i][j];
20                        primeiro = 0;
21                    }else{
22                        if(menorPeso > gr->pesos[i][j]){
23                            menorPeso = gr->pesos[i][j];
24                            orig = i;
25                            dest = gr->arestas[i][j];
26                        }
27                    }
28                }
29            }
30        }
31        if(primeiro == 1)
32            break;
33        if(pai[orig] == -1)
34            pai[orig] = dest;
35        else
36            pai[dest] = orig;
37        for(i=0; i < NV; i++)
38            if(arv[i] == arv[dest])
39                arv[i] = arv[orig];
40        free(arv);
41    }
42 }
```

FIGURA 10.39

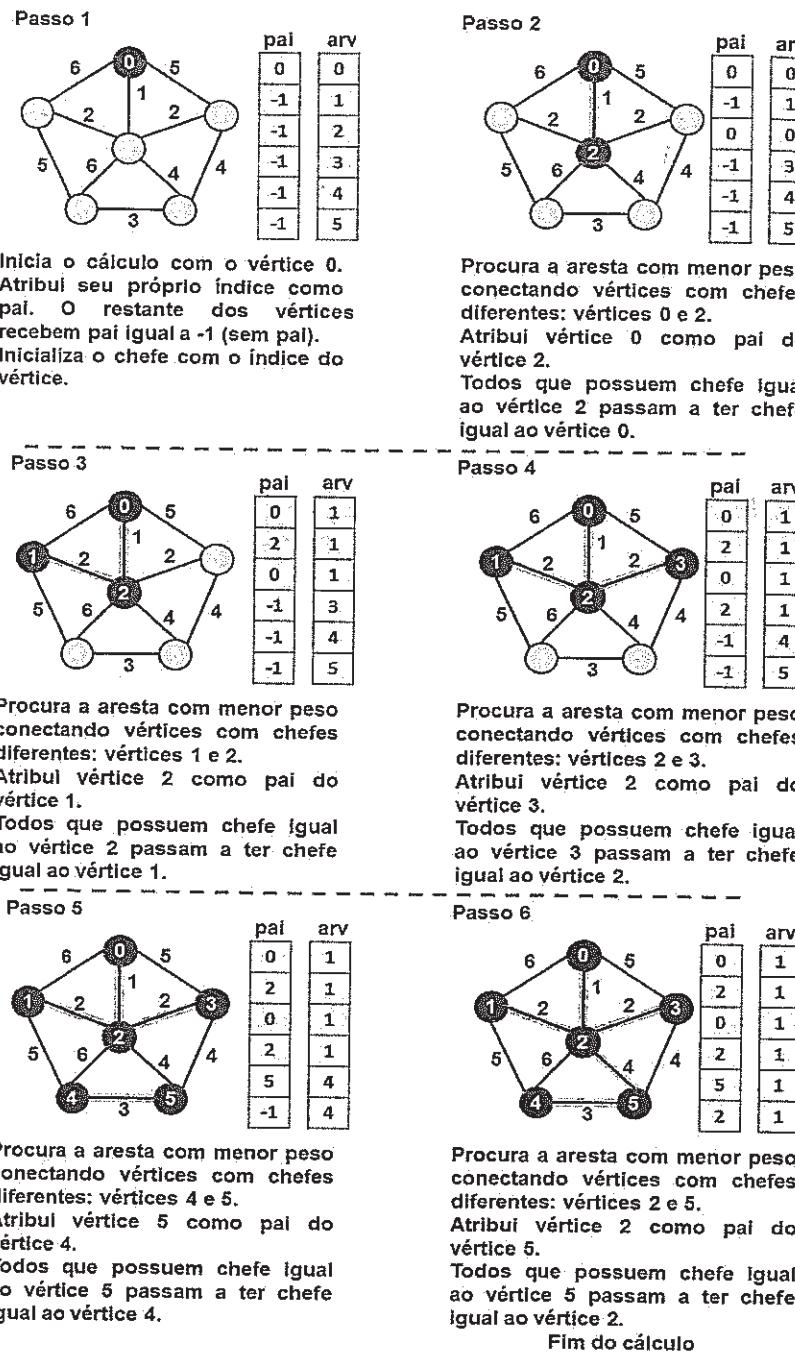


FIGURA 10.40

Uma vez que se tenha percorrido todas as arestas do grafo, verifique se foi possível achar uma aresta de menor peso (linha 31). Caso não tenha sido possível, o processo termina (linha 32). Do contrário, definimos um vértice como pai do vértice que ainda não possui pai (linhas 34-37). Em seguida, unimos as duas árvores da aresta selecionada (todos os vértices que fazem parte da árvore do vértice **dest** passam a fazer parte da árvore do vértice **orig**) e o algoritmo continua sua execução (linhas 39-42). Ao final, liberamos o array auxiliar **arv** e o array **pai** irá conter os antecessores de cada vértice do grafo na árvore montada a partir do vértice inicial **orig** (que é pai de si mesmo). Esse processo é mais bem ilustrado pela Figura 10.40, que considera o vértice 0 o vértice inicial.



Essa versão do algoritmo de Kruskal deve ser usada apenas para fins didáticos, pois é muito ineficiente.

Considerando um grafo  $G(V, A)$ , em que  $|V|$  é o número de vértices e  $|A|$ , o número de arestas, a complexidade dessa versão do algoritmo de Kruskal, no pior caso, é  $O(|V| * |A|)$ . Como o valor de  $|A|$  é proporcional a  $|V|^2$ , seu custo, no pior caso, é  $O(|V|^3)$ .

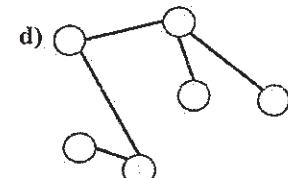
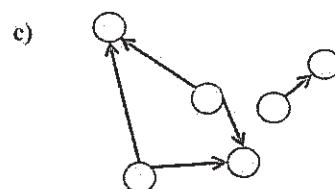
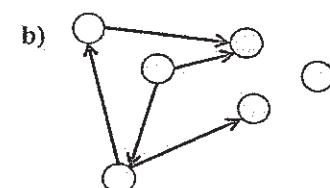
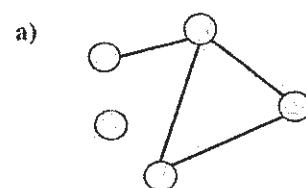


A eficiência do algoritmo de Kruskal depende da forma usada para encontrar a aresta de menor peso. Usando uma estrutura de dados **união-e-desunião** (**union&find**) para achar a aresta, o custo pode ser reduzido para  $O(|A| \log |V|)$ , no pior caso.

## 10.7 EXERCÍCIOS

- 1) Descreva, usando as suas palavras, o que é um grafo e dê exemplos de suas aplicações.
- 2) Descreva, usando as suas palavras, o que é um subgrafo.
- 3) Descreva, usando as suas palavras, o que é um grafo bipartido.
- 4) Descreva, usando as suas palavras, o que é um grafo conexo. É um desconexo?
- 5) Descreva, usando as suas palavras, o que são grafos isomorfos. Dê exemplos.
- 6) Descreva, usando as suas palavras, o que é um grafo hamiltoniano.
- 7) Descreva, usando as suas palavras, o que é um grafo euleriano.
- 8) Desenhe as versões orientada e não orientada do grafo  $G(V, E)$ , em que  $V = \{1, 2, 3, 4, 5, 6\}$  e  $E = \{(6,1), (5,2), (3,4), (2,1), (3,6)\}$ .

9) Defina os conjuntos  $V$  e  $E$  dos grafos ilustrados:



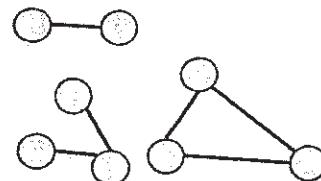
10) Defina os conjuntos  $V$  e  $E$  e desenhe os grafos não orientados completos contendo 4, 5 e 6 vértices.

11) Dado um grafo com três vértices de grau 3 e um vértice de grau 5, diga quantas arestas ele possui.

12) Escreva uma função para obter todos os nós adjacentes (vizinhos) de um nó do grafo. Considere que o grafo é representado por uma matriz de adjacências.

13) Escreva uma função para obter todos os nós adjacentes (vizinhos) de um nó do grafo. Considere que o grafo é representado por uma lista de adjacências.

14) Quantas componentes conexas tem o grafo?



15) Descreva, com suas palavras, o funcionamento de um algoritmo de busca em profundidade. Dê dois exemplos de aplicação real deste algoritmo.

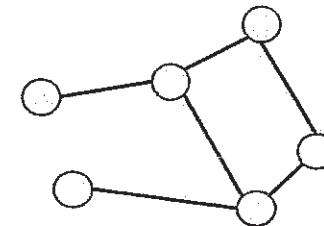
16) Descreva, com suas palavras, o funcionamento de um algoritmo de busca em largura. Dê dois exemplos de aplicação real deste algoritmo.

17) Descreva, com suas palavras, o funcionamento de um algoritmo de busca pelo menor caminho. Dê dois exemplos de aplicação real deste algoritmo.

18) Escreva um algoritmo para verificar se um grafo é acíclico. Para isso, use o algoritmo de busca em profundidade.

19) Dado o algoritmo de busca em profundidade, escreva uma versão não recursiva dele.

20) Dado o grafo, mostre o resultado da busca em largura e em profundidade. Considere o vértice 1 o início da busca.



21) Considere um grafo contendo  $N$  vértices e representado por uma matriz de adjacências. Escreva um algoritmo que diga se o grafo é ou não orientado.

# Árvores

## 11.1 DEFINIÇÃO

Diversas aplicações necessitam que se represente um conjunto de objetos e as suas relações hierárquicas. Nestes casos, é interessante fazer uso de estruturas chamadas árvores.



Uma árvore é uma abstração matemática usada para representar estruturas hierárquicas não lineares dos objetos modelados.

Várias são as aplicações das árvores. Basicamente, qualquer problema em que exista algum tipo de hierarquia pode ser representado por uma árvore. Um exemplo disso é a estrutura de diretórios do computador (Figura 11.1). Outros exemplos:

- Relações de descendência (pai, filho etc.).
- Diagrama hierárquico de uma organização.
- Campeonatos de modalidades desportivas.
- Taxonomia.
- Busca de dados armazenados no computador.
- Representação de espaço de soluções (ex: jogo de xadrez).
- Modelagem de algoritmos.

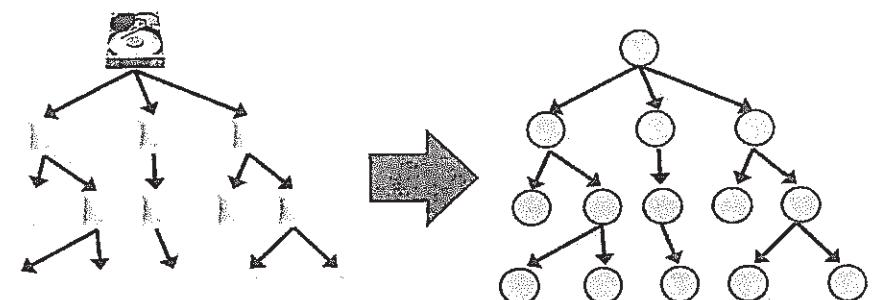


FIGURA 11.1

Como nos grafos, uma árvore também é definida usando um conjunto de **nós** (ou **vértices**) (que são os itens representados na árvore e dependem da natureza do problema modelado) e **arestas** (que são utilizadas para conectar qualquer par de nós ou vértices).



Uma árvore é um tipo especial de **grafo**. Porém, ela deve ser sempre um grafo não direcionado, **conexo** e **acíclico**.

O fato de a árvore ser um grafo **conexo** significa que existe exatamente um caminho entre quaisquer dois de seus nós. Já o fato do grafo ser **acíclico** garante que a árvore não possui ciclos. Em termos de representação, podemos representar uma árvore utilizando um grafo ou um diagrama de Venn (diagrama de inclusão ou conjuntos aninhados), como mostra a Figura 11.2.

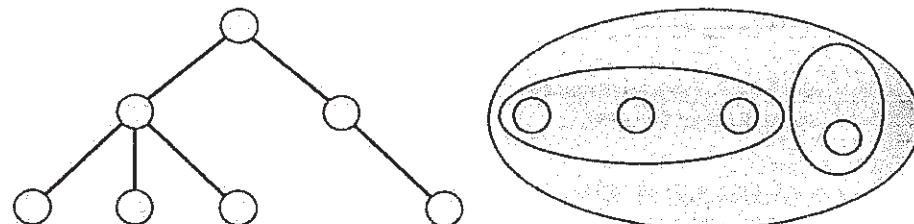


FIGURA 11.2

## 11.2 CONCEITOS BÁSICOS

### 11.2.1 Notação

A Figura 11.3 apresenta os principais conceitos relativos às árvores. Uma árvore é formada por um conjunto de nós ligados por arestas de forma hierárquica, simulando as árvores encontradas na natureza. Porém, enquanto as árvores na natureza crescem de baixo para cima, com a raiz localizada na parte inferior (no solo) e as folhas no topo, em computação, as árvores crescem de cima para baixo, ou seja, a raiz se encontra no topo e as folhas na parte mais baixa. Na sequência, podemos ver a nomenclatura usada para se trabalhar com uma árvore:

- **Raiz:** é o nó localizado na parte mais alta da árvore, o único que não possui pai.
- **Pai:** também chamado ancestral, é o nó antecessor imediato de outro nó.
- **Filho:** é o nó sucessor imediato de outro nó.
- **Nó folha:** também chamado nó terminal, é qualquer nó que não possui filhos.
- **Nó interno:** também chamado nó não terminal, é qualquer nó que possui ao menos um filho.

- **Caminho:** é uma sequência de nós de modo que existe sempre uma aresta ligando o nó anterior com o seguinte. Note que existe exatamente um caminho entre a raiz e cada um dos nós da árvore.

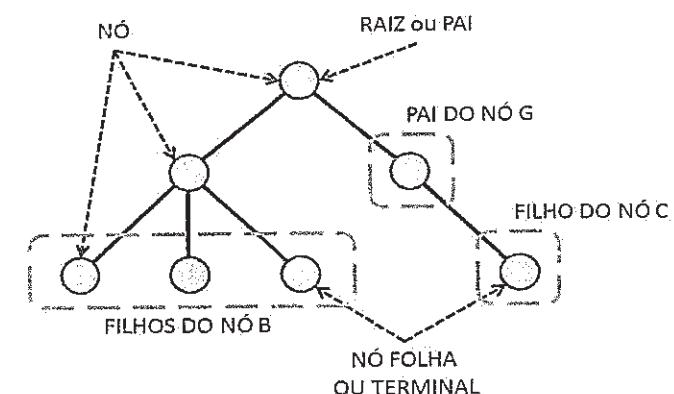


FIGURA 11.3

### 11.2.2 Grau do nó e subárvore

Dado um nó da árvore, cada filho seu é considerado a raiz de uma nova subárvore. Ou seja, qualquer nó é a raiz de uma subárvore constituindo-se dele e dos nós abaixo dele. Já o grau de um nó é dado pelo número de subárvore que ele possui. Na Figura 11.3, temos que o nó A possui duas subárvore (B e C), portanto o nó A possui grau 2. Já o nó B possui três subárvore (D, E e F). Nós folhas (D, E, F e G) possuem grau 0.

### 11.2.3 Altura e nível da árvore

Em uma árvore, os nós são classificados em diferentes níveis. O nível é dado pelo número de nós que existem no caminho entre esse nó e a raiz (nível 0). Já a altura (ou profundidade) da árvore é o número total de níveis de uma árvore, ou seja, é o comprimento do caminho mais longo da raiz até uma das suas folhas. A Figura 11.4 mostra uma árvore com três níveis.

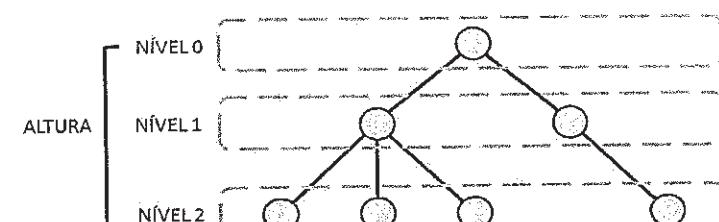


FIGURA 11.4

### 11.3 TIPOS DE ÁRVORE

Na computação, assim como na natureza, existem vários tipos de árvores. Cada uma delas foi desenvolvida pensando diferentes tipos de aplicações. Algumas delas:

- Árvore binária de busca.
- Árvore AVL.
- Árvore rubro-negra.
- Árvore B, B+ e B\*.
- Árvore 2-3.
- Árvore 2-3-4.
- Quadtree.
- Octree.
- etc.

### 11.4 ÁRVORE BINÁRIA

Uma árvore binária é um tipo especial de árvore em que cada nó pode possuir nenhuma, uma ou no máximo duas subárvores: a subárvore da esquerda e a da direita. Caso o nó possua nenhuma subárvore, este será um nó folha. Um exemplo de árvore binária é mostrado na Figura 11.5.



Árvores binárias são muito úteis para modelar situações nas quais, a cada ponto do processo, é preciso tomar uma decisão entre duas direções.

As árvores binárias também são úteis em banco de dados, compiladores, compressão de dados (código de Huffman) e na representação de expressões aritméticas. Outra aplicação em que árvores binárias são importantes é na manutenção de estruturas nas quais a ordem é importante.

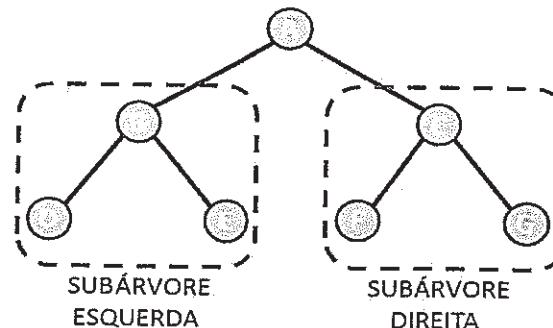


FIGURA 11.5

#### 11.4.1 Tipos de árvore binária

Existem três tipos de árvores binárias: **estritamente binária**, **cheia** e **completa**. Basicamente, elas diferem entre si pelo número de subárvores de um nó e pelo posicionamento do nó na árvore.

Uma árvore **estritamente binária** é aquela na qual cada nó possui sempre ou nenhuma (no caso de nó folha) ou duas subárvores. Não existe nenhum nó interno com apenas um filho, todos têm sempre dois, como mostra a Figura 11.6.

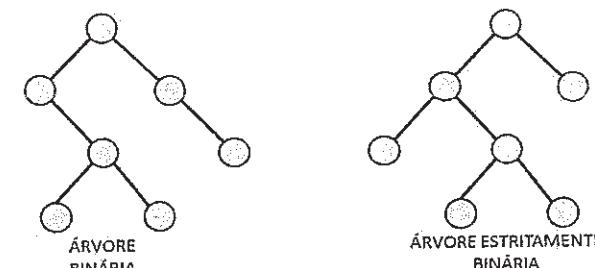


FIGURA 11.6

Uma árvore binária **completa** é uma árvore **estritamente binária** na qual todos os seus nós folhas estão no mesmo nível. Neste tipo de árvore, é possível calcular o número de nós por nível, assim como o número total de nós da árvore:

- Um nível  $n$  possui exatamente  $2^n$  nós.
- Se um nível  $n$  possui  $m$  nós, o nível  $n + 1$  possuiirá  $2m$  nós.
- Uma árvore de altura  $H$  possui  $2^H - 1$  nós.

A Figura 11.7 mostra uma árvore binária cheia de altura  $H = 4$ , ou seja, que possui 15 nós.

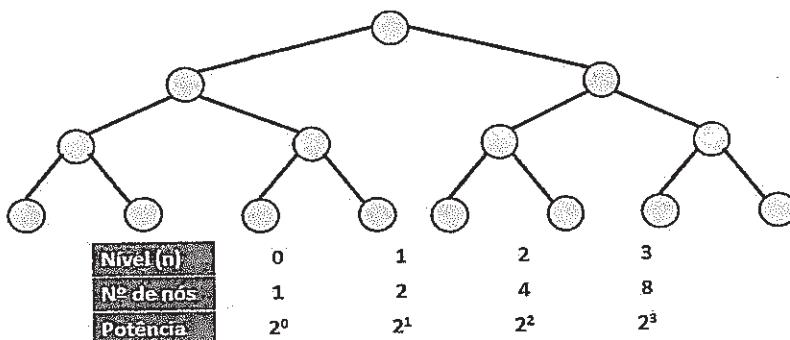


FIGURA 11.7

Por fim, uma árvore binária **quase completa** é uma árvore em que a diferença de altura entre as subárvore de qualquer nó é, no máximo, de 1. Ou seja, cada nó folha da árvore deve estar no nível  $D$  ou  $D - 1$ , como mostra a Figura 11.8.

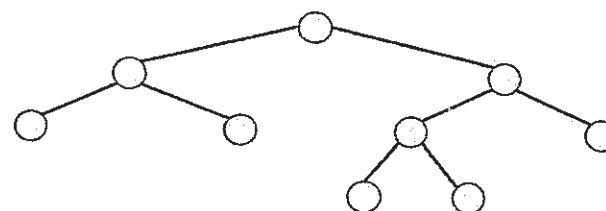


FIGURA 11.8

#### 11.4.2 Tipos de implementação

Ao se modelar um problema utilizando uma árvore binária, surge a questão: como implementar essa árvore no computador? Existem duas abordagens muito utilizadas. São elas:

- Usando um array ( alocação estática).
- Usando uma lista encadeada ( alocação dinâmica).

A implementação escolhida para a árvore depende da aplicação. Não existe uma implementação que seja melhor que a outra em todos os casos. Independentemente do tipo de alocação usado na implementação da árvore, as seguintes operações básicas são possíveis:

- Criação da árvore.
- Inserção de um elemento.
- Remoção de um elemento.
- Busca por um elemento.
- Destrução da árvore.
- Além de informações com número de nós, altura ou se está vazia.

A seguir, veremos como funciona cada um dos tipos de implementação de árvore.

##### Implementação usando um array (heap)

A ideia básica é utilizar um array como uma estrutura de dados do tipo heap. Uma heap permite simular uma árvore binária **completa** ou **quase completa** (a exceção é o seu último nível). Neste tipo de implementação, é necessário definir o número máximo de elementos que a árvore irá possuir para podermos definir o tamanho do array utilizado para representar a heap. Ou seja, essa abordagem tem a desvantagem de que toda a estrutura da árvore deve ser previamente conhecida.

O uso de um array como uma heap faz com que cada posição do array passe a ser considerado o pai de duas outras posições, chamadas filhos. Com isso, o acesso aos elementos da árvore se dá por meio de duas funções:

- **FILHO\_ESQ(PAI) =  $2 * PAI + 1$ .**
- **FILHO\_DIR(PAI) =  $2 * PAI + 2$ .**

A função **FILHO\_ESQ** retorna o índice do filho à esquerda do nó armazenado na posição **PAI**, enquanto a função **FILHO\_DIR** retorna o índice do filho à direita. A Figura 11.9 exemplifica a construção da árvore e o uso destas funções.



A implementação usando arrays é utilizada quando todos os elementos são conhecidos *a priori*. Seu uso é mais adequado quando a operação mais comum é a busca por um elemento na árvore.

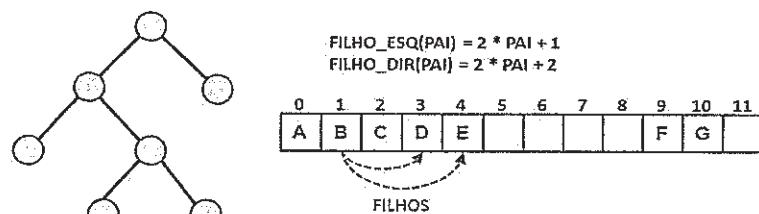


FIGURA 11.9

##### Implementação usando acesso encadeado

Neste tipo de implementação, o espaço de memória é alocado em tempo de execução. A árvore cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos. Cada nó da árvore é um ponteiro para uma estrutura contendo três campos de informação: um campo **dado**, utilizado para armazenar a informação inserida na árvore, e dois ponteiros: **esq**, que aponta para a subárvore da esquerda e **dir**, que aponta para a subárvore da direita. A Figura 11.10 exemplifica a construção de uma árvore e o uso desses ponteiros.



A implementação usando acesso encadeado não necessita que se conheça previamente todos os elementos a ser inseridos na árvore. Do mesmo modo, a árvore a ser representada não precisa ser uma árvore binária completa ou quase completa.

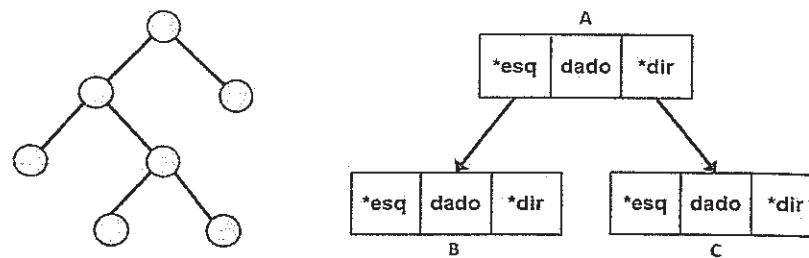


FIGURA 11.10

### 11.4.3 Criando a TAD árvore binária

Nesta seção, iremos ver como implementar um tipo abstrato de dado capaz de representar uma árvore binária usando alocação dinâmica e acesso encadeado. Neste tipo de implementação, cada nó da árvore é um ponteiro para uma estrutura contendo as informações relativas àquele nó. A ideia é utilizar um **ponteiro para ponteiro** para guardar o primeiro nó da árvore, como mostra a Figura 11.11. Assim, fica fácil mudar quem é a **raiz** da árvore (se necessário).

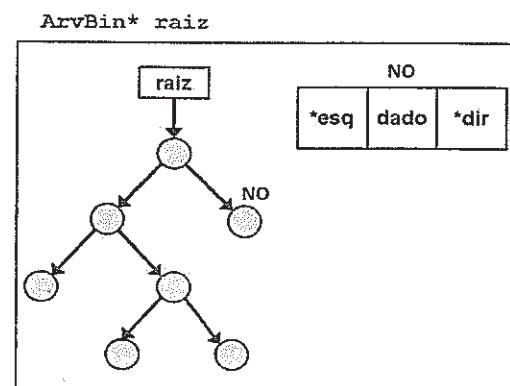


FIGURA 11.11

#### Definindo o tipo árvore binária

Antes de começar a implementar a nossa árvore binária, é preciso definir o tipo de dado que será armazenado em cada nó. Uma árvore binária pode armazenar qualquer tipo de informação em seus nós. Para tanto, é necessário que especifiquemos isso na sua declaração. Como estamos trabalhando com modularização, precisamos também definir o **tipo opaco** que representa nossa árvore binária. E, trabalhando com alocação dinâmica da árvore binária, esse tipo será um **ponteiro para ponteiro** da estrutura que define o nó da árvore. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.



No arquivo **ArvoreBinaria.h**, iremos declarar tudo aquilo que será visível para o programador.

Vamos começar definindo o arquivo **ArvoreBinaria.h**, ilustrado na Figura 11.12. Por se tratar de uma árvore com alocação dinâmica, temos que definir:

- Para fins de padronização, um novo nome para o **ponteiro** do tipo árvore (linha 1). Esse é o tipo que será usado sempre que se desejar trabalhar com uma árvore binária.
- As funções disponíveis para se trabalhar com essa árvore e que serão implementadas no arquivo **ArvoreBinaria.c** (linhas 3-13).



Por que colocamos um ponteiro no comando **typedef** quando criamos um novo nome para o tipo (linha 1)?

Por estarmos trabalhando com uma árvore com alocação dinâmica e encadeada, temos que trabalhar com um ponteiro para ponteiro a fim de poder fazer modificações na raiz da árvore, algo necessário em alguns tipos de árvores binárias. Por questões de modularização, podemos esconder um dos ponteiros do usuário para que ele não tenha que se preocupar com possíveis mudanças na raiz da árvore.



No arquivo **ArvoreBinaria.c**, iremos definir tudo aquilo que deve ficar oculto da biblioteca e implementar as funções definidas em **ArvoreBinaria.h**.

Basicamente, o arquivo **ArvoreBinaria.c** (Figura 11.12) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da árvore binária (linhas 1-3).
- A definição do tipo que descreve cada nó da árvore binária, **struct NO** (linhas 4-8).
- As implementações das funções definidas no arquivo **ArvoreBinaria.h**. As implementações dessas funções serão vistas nas seções seguintes.

Note que a nossa estrutura **NO** nada mais é do que uma estrutura contendo três campos:

- Um ponteiro **esq**, que indica o filho da esquerda daquele nó (também do tipo **struct NO**).
- Um ponteiro **dir**, que indica o filho da direita daquele nó (também do tipo **struct NO**).
- E um campo **info** do tipo **int**, que é o tipo de dado a ser armazenado no nó da árvore.

Neste exemplo, optamos por armazenar em cada nó apenas um valor inteiro (**info**).

Por estarem definidos dentro do arquivo **.c**, os campos dessa estrutura não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome definido no arquivo **ArvoreBinaria.h** (linha 1), que pode somente declarar um ponteiro para ele da seguinte forma:

```
Arquivo ArvoreBinaria.h
01 typedef struct NO* ArvBin;
02
03 ArvBin* cria_ArvBin();
04 void libera_ArvBin(ArvBin *raiz);
05 int insere_ArvBin(ArvBin* raiz, int valor);
06 int remove_ArvBin(ArvBin *raiz, int valor);
07 int estaVazia_ArvBin(ArvBin *raiz);
08 int altura_ArvBin(ArvBin *raiz);
09 int totalNO_ArvBin(ArvBin *raiz);
10 int consulta_ArvBin(ArvBin *raiz, int valor);
11 void preOrdem_ArvBin(ArvBin *raiz);
12 void emOrdem_ArvBin(ArvBin *raiz);
13 void posOrdem_ArvBin(ArvBin *raiz);
```

```
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreBinaria.h" //inclui os Protótipos
struct NO{
    int info;
    struct NO *esq;
    struct NO *dir;
};
```

FIGURA 11.12

### Entendendo a notação de ponteiros da árvore

Muitas vezes, torna-se difícil entender a notação de ponteiros de uma árvore. Isso ocorre pelo fato de estarmos utilizando um **ponteiro para ponteiro** para representar a raiz da árvore e ponteiros para cada um dos dois filhos de determinado nó. Além disso, algumas operações da árvore envolvem a passagem do endereço dos nós filhos, os quais já são ponteiros, resultando novamente em um **ponteiro para ponteiro**. Para evitar confusões nos códigos que serão apresentados nas próximas seções, a Figura 11.13 traz uma relação das principais notações de ponteiros (endereçamento e acesso a conteúdo) utilizados na manipulação de uma árvore binária e seus respectivos significados.

raiz	Ponteiro do tipo ArvBin*
	Equivale a um ponteiro do tipo struct NO**
PAI	Ponteiro do tipo struct NO*
*raiz	Conteúdo da raiz (PAI). Equivale a um ponteiro do tipo struct NO*
(*raiz)->info	Informação contida no elemento apontado pela raiz (50)
(*raiz)->esq	Ponteiro para o filho à esquerda da raiz (FE). Equivale a um ponteiro do tipo struct NO*
&(*raiz)->esq	Endereço do filho à esquerda da raiz (FE). Equivale a um ponteiro do tipo ArvBin* ou struct NO**
PAI->dir	Ponteiro para o filho à direita de PAI (FD). Equivale a um ponteiro do tipo struct NO*
PAI->info	Informação contida no elemento apontado por PAI (50)
PAI->dir->info	Informação contida no filho à direita do elemento apontado por PAI (99)

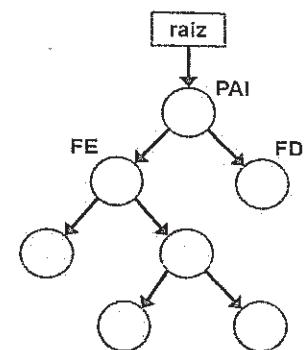


FIGURA 11.13

### Criando e destruindo uma árvore

Para utilizar o tipo árvore binária em seu programa, a primeira coisa a fazer é criar uma nova árvore. Essa tarefa é executada pela função descrita na Figura 11.14. Basicamente, o que essa função faz é a alocação de uma área de memória para armazenar o endereço da raiz da árvore (linha 2), que é um **ponteiro para ponteiro**. Esta área de memória corresponde à memória necessária para armazenar o endereço de um nó da árvore, `sizeof(ArvBin)` ou `sizeof(struct NO*)`. Em seguida, a função inicializa o conteúdo desse **ponteiro para ponteiro** com a constante `NONE`. Esta constante é utilizada na árvore binária para indicar que não existe nenhum nó alocado após o atual, ou seja, que este nó não possui filhos. Como a raiz da árvore aponta para tal constante, isso significa que a árvore está vazia. A Figura 11.15 indica o conteúdo do nosso ponteiro `raiz` após a chamada da função que cria a árvore.

```
Arquivo CriandoArvoreBinaria.c
01 ArvBin* cria_ArvBin(){
02     ArvBin* raiz = (ArvBin*) malloc(sizeof(ArvBin));
03     if(raiz != NULL)
04         *raiz = NULL;
05     return raiz;
06 }
```

FIGURA 11.14

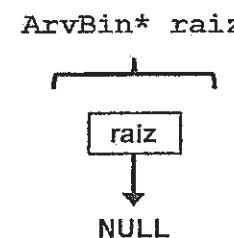


FIGURA 11.15



Para liberar uma árvore binária implementada com alocação dinâmica, e acesso encadeado, é preciso percorrer toda a árvore liberando a memória alocada para cada nó inserido nela.

A Figura 11.16 mostra o código utilizado para liberar uma árvore binária. Este algoritmo utiliza uma função para percorrer recursivamente todos os nós da árvore e liberá-los, *libera\_NO* (linhas 1-8), e outra para inicializar a destruição da árvore e liberar a memória alocada para a raiz, *libera\_ArvBin* (linhas 9-14).

A função *libera\_ArvBin* recebe como parâmetro a árvore a ser liberada e verifica se ela é válida, ou seja, se a tarefa de criação da árvore foi realizada com sucesso (linha 10). Em seguida, essa função chama a função *libera\_NO* para percorrer e liberar recursivamente todos os nós da árvore, começando pelo conteúdo da raiz (linha 12). Por fim, a memória associada à raiz da árvore é liberada (linha 13).

Já a função *libera\_NO* recebe como parâmetro o endereço de um nó da árvore e verifica se este é um nó válido, ou seja, se a tarefa de alocação e inserção do nó na árvore foi realizada com sucesso (linha 2). Em seguida, ela é chamada recursivamente para o filho da esquerda (linha 4),

#### Destrutando uma árvore binária

```

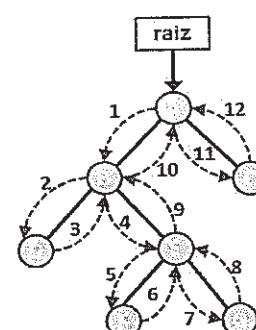
01 void libera_NO(struct NO* no) {
02     if(no == NULL)
03         return;
04     libera_NO(no->esq);
05     libera_NO(no->dir);
06     free(no);
07     no = NULL;
08 }
09 void libera_ArvBin(ArvBin* raiz) {
10     if(raiz == NULL)
11         return;
12     libera_NO(*raiz); //libera cada nó
13     free(raiz); //libera a raiz
14 }
  
```

FIGURA 11.16

ira o da direita (linha 5) e, por fim, libera a memória do nó recebido por parâmetro (linha 7). Esse processo é mais bem ilustrado pela Figura 11.17.



Perceba que, para liberar um nó da árvore, devemos antes liberar a sua subárvore esquerda e direita. Desse modo, este nó se torna um nó folha e pode ser liberado sem prejuízos para o restante da árvore.



#### Inicia no nó A

- 1 visita B
- 2 visita D
- 3 libera D, volta para B
- 4 visita E
- 5 visita F
- 6 libera F, volta para E
- 7 visita G
- 8 libera G, volta para E
- 9 libera E, volta para B
- 10 libera B, volta para A
- 11 visita C
- 12 libera C, volta para A e libera A
- libera raiz

FIGURA 11.17

#### Informações básicas sobre a árvore

Na sequência, veremos como implementar algumas funções para retornar três informações importantes sobre a árvore binária: se ela está vazia, o número de nós que possui e a sua altura. Comecemos por saber se a árvore está vazia.



Uma árvore binária será considerada vazia sempre que o conteúdo da sua "raiz" apontar para a constante **NULL**.

A implementação da função que retorna se a árvore binária está vazia é mostrada na Figura 11.18. Note que essa função, em primeiro lugar, verifica se o ponteiro *ArvBin \*raiz* é igual a *NULL*. A condição seria verdadeira se houvesse ocorrido um problema na criação da árvore, neste caso, não teríamos uma árvore válida para trabalhar. Assim, optamos por retornar o valor **UM** para indicar uma árvore inválida (linha 3). Porém, se a árvore foi criada com sucesso, então é possível acessar o conteúdo da sua "raiz" (*\*raiz*) e comparar o seu valor com a constante *NULL*, que é o valor inicial do conteúdo da "raiz" quando criamos a árvore. Se os valores forem

iguais (ou seja, nenhum nó contido na árvore), a função irá retornar o valor **UM** (linha 5). Caso contrário, irá retornar o valor **ZERO** (linha 6).

### Retornando se a árvore binária está vazia

```
01 int estaVazia_ArvBin(ArvBin *raiz) {
02     if(raiz == NULL)
03         return 1;
04     if(*raiz == NULL)
05         return 1;
06     return 0;
07 }
```

FIGURA 11.18

A Figura 11.19 mostra a implementação da função que retorna o total de nós de uma árvore. Note que essa função, em primeiro lugar, verifica se o ponteiro **ArvBin \*raiz** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da árvore e, neste caso, não teríamos uma árvore válida para trabalhar. Assim, optamos por retornar o valor **ZERO** para indicar uma árvore vazia (linha 3). Porém, se a árvore foi criada com sucesso, então é possível acessar o conteúdo de sua raiz (ou seja, o primeiro nó da árvore) e comparar o seu valor com a constante **NULL**, que é o valor inicial do conteúdo do raiz quando criamos a árvore. Se os valores forem iguais (ou seja, nenhum nó contido dentro da árvore), a função irá retornar o valor **ZERO** (linha 5).

Uma vez que temos uma árvore com nós para ser contados, chamamos recursivamente a função para a subárvore da esquerda (linha 6) e para a subárvore da direita (linha 7).



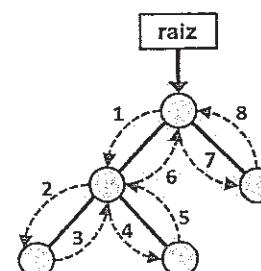
Note que a função **totalNO\_ArvBin** recebe como parâmetro um ponteiro para ponteiro (**ArvBin\*** ou **struct NO\*\***). Como os nós da árvore são ponteiros simples (**struct NO\***) é preciso passar o endereço deste ponteiro na chamada recursiva da função (linhas 6-7).

Cada chamada recursiva da função vai descer na subárvore até encontrar um nó folha. Um nó que seja folha não possui subárvores da esquerda nem da direita. Desse modo, as suas chamadas recursivas (linhas 6-7) irão retornar **ZERO**, por causa do segundo **if** (linhas 4-5). Neste caso, onde a **raiz** é um nó folha, a função irá retornar o valor **UM**, pois não existem subárvores (**total\_esq** e **total\_dir** são zero), o que indica que existe apenas um nó. Ao voltarmos na recursão até a raiz, os valores calculados para cada nó folha são passados para seus respectivos pais, que somam mais um nó ao processo e repassam para seus pais, até que o valor retorne para a raiz da árvore. Esse processo é mais bem ilustrado pela Figura 11.20.

### Retornando o número de nós da árvore binária

```
01 int totalNO_ArvBin(ArvBin *raiz) {
02     if(raiz == NULL)
03         return 0;
04     if(*raiz == NULL)
05         return 0;
06     int total_esq = totalNO_ArvBin(&(*raiz)->esq);
07     int total_dir = totalNO_ArvBin(&(*raiz)->dir);
08     return(total_esq + total_dir + 1);
09 }
```

FIGURA 11.19



inicia no nó A

- 1 visita B
  - 2 visita D
  - 3 D é nó folha: conta como 1 nó.  
Volta para B
  - 4 visita E
  - 5 E é nó folha: conta como 1 nó.  
Volta para B
  - 6 Número de nós em B é 3: total de nós a direita (1) + total de nós a esquerda (1) + 1.  
Volta para A
  - 7 visita C
  - 8 C é nó folha: conta como 1 nó.  
Volta para A
- Número de nós em A é 5: total de nós a direita (1) + total de nós a esquerda (3) + 1.

FIGURA 11.20

A Figura 11.21 mostra a implementação da função que retorna a altura de uma árvore. Note que essa função, como no caso de contar o total de nós da árvore, verifica se a árvore é válida (linhas 2-3) e se ela não está vazia (linhas 4-5). Em ambos os casos, a função retorna o valor **ZERO** caso as afirmações sejam verdadeiras. Uma vez que temos uma árvore com nós, é possível calcular a sua altura. Para tanto, chamamos recursivamente a função para a subárvore da esquerda (linha 6) e para a subárvore da direita (linha 7).



Note que a função **altura\_ArvBin** recebe como parâmetro um ponteiro para ponteiro (**ArvBin\*** ou **struct NO\*\***). Como os nós da árvore são ponteiros simples (**struct NO\***) é preciso passar o endereço deste ponteiro na chamada recursiva da função (linhas 6-7).

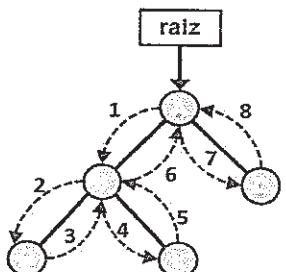
Como no cálculo do número de nós da árvore, cada chamada recursiva da função vai descer na subárvore até encontrar um nó folha. Um nó que seja folha não possui subárvores da esquerda nem da direita. Desse modo, as suas chamadas recursivas (linhas 6-7) irão retornar **ZERO** por causa do segundo **if** (linhas 4-5). Neste caso, onde a **raiz** é um nó folha, a função irá retornar o valor **UM** como sendo a altura da árvore, pois não existem subárvores (**alt\_esq e alt\_dir** são zeros), o que indica que existe apenas um nó. Ao voltarmos na recursão até a raiz, cada nó pai calcula as alturas de suas subárvores e as compara (linhas 6-8). Em seguida, o nó pai retorna para o nó mais acima na árvore a altura da maior subárvore somada de mais uma unidade, que é a altura deste nó na árvore (linhas 8-11). Esse processo segue até que o valor retorne para a raiz da árvore. Ele é mais bem ilustrado pela Figura 11.22.

```

Retornando a altura da árvore binária

01 int altura_ArvBin(ArvBin *raiz){
02     if (raiz == NULL)
03         return 0;
04     if (*raiz == NULL)
05         return 0;
06     int alt_esq = altura_ArvBin(&(*raiz)->esq);
07     int alt_dir = altura_ArvBin(&(*raiz)->dir);
08     if (alt_esq > alt_dir)
09         return (alt_esq + 1);
10     else
11         return (alt_dir + 1);
12 }
```

FIGURA 11.21



- inicia no nó A
- 1 visita B
  - 2 visita D
  - 3 D é nó folha: altura é 1. Volta para B
  - 4 visita E
  - 5 E é nó folha: altura é 1. Volta para B
  - 6 altura de B é 2: maior altura dos filhos + 1. Volta para A
  - 7 visita C
  - 8 C é nó folha: altura é 1. Volta para A
- altura de A é 3: maior altura dos filhos + 1.

FIGURA 11.22

### Percorrendo uma árvore

Uma operação muito comum em árvores binárias é percorrer todos os seus nós, executando alguma ação em cada nó. Esta ação pode ser mostrar (imprimir) o valor do nó, modificar esse valor etc.



Não existe uma ordem "natural" para se percorrer todos os nós de uma árvore binária.

Apesar disso, existem algumas formas muito utilizadas para se percorrer uma árvore. São elas:

- **Percorso pré-ordem:** visitar a raiz, o filho da esquerda e o filho da direita.
- **Percorso em-ordem:** visitar o filho da esquerda, a raiz e o filho da direita.
- **Percorso pós-ordem:** visitar o filho da esquerda, o filho da direita e a raiz.

Note que a diferença entre eles está na ordem em que cada nó é visitado ao longo do percurso. As três formas de percurso possuem implementação intuitiva via algoritmos recursivos, que serão mostrados a seguir.

A Figura 11.23 mostra a função que realiza o percurso **pré-ordem**. Esta função recebe como parâmetro uma árvore (raiz) e, caso ela não seja inválida (linhas 2-3) e não seja vazia (linha 4), irá executar o percurso. No percurso **pré-ordem** (linhas 5-7), primeiro se imprime (linha 5) o valor associado ao nó e, em seguida, chama-se o algoritmo recursivamente para percorrer a subárvore da esquerda (linha 6) e a da direita (linha 7). Perceba que nas chamadas recursivas temos que passar o endereço do nó, pois a função tem como parâmetro um **ponteiro para ponteiro**. O processo de imprimir os nós de uma árvore usando o percurso **pré-ordem** é mais bem ilustrado pela Figura 11.24.

### Percorso pré-ordem

```

Percorso pré-ordem

01 void preOrdem_ArvBin(ArvBin *raiz){
02     if(raiz == NULL)
03         return;
04     if(*raiz != NULL){
05         printf("%d\n", (*raiz)->info);
06         preOrdem_ArvBin(&(*raiz)->esq);
07         preOrdem_ArvBin(&(*raiz)->dir);
08     }
09 }
```

FIGURA 11.23

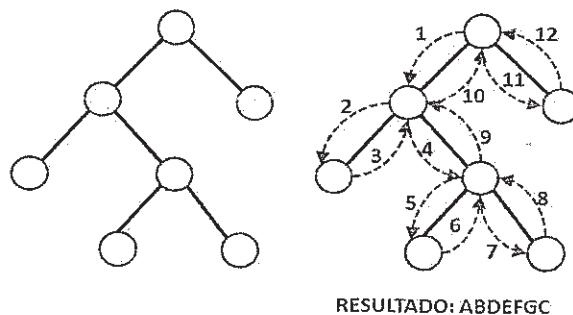


FIGURA 11.24

```

inicia no nó A
1 imprime A, visita B
2 imprime B, visita D
3 imprime D, volta para B
4 visita E
5 imprime E, visita F
6 imprime F, volta para E
7 visita G
8 imprime G, volta para E
9 volta para B
10 volta para A
11 visita C
12 imprime C, volta para A
  
```

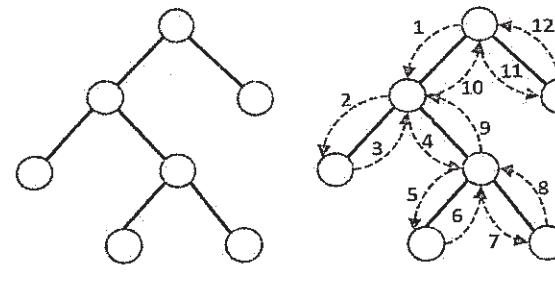


FIGURA 11.26

```

inicia no nó A
1 visita B
2 visita D
3 imprime D, volta para B
4 imprime B, visita E
5 visita F
6 imprime F, volta para E
7 imprime E, visita G
8 imprime G, volta para E
9 volta para B
10 volta para A
11 imprime A, visita C
12 imprime C, volta para A
  
```

A Figura 11.25 mostra a função que realiza o percurso **em-ordem**. Esta função recebe como parâmetro uma árvore (raiz) e, caso ela não seja inválida (linhas 2-3) e não seja vazia (linha 4), irá executar o percurso. No percurso **em-ordem** (linhas 5-7), primeiro chama-se o algoritmo recursivamente para percorrer a subárvore da esquerda (linha 5) e, em seguida, imprime-se (linha 6) o valor associado ao nó e visita-se a subárvore da direita (linha 7). Note que nas chamadas recursivas temos que passar o endereço do nó, pois a função tem como parâmetro um **ponteiro para ponteiro**. O processo de imprimir os nós de uma árvore usando o percurso **em-ordem** é mais bem ilustrado pela Figura 11.26.

```

Percorso em-ordem
01 void emOrdem_ArvBin(ArvBin *raiz){
02     if(raiz == NULL)
03         return;
04     if(*raiz != NULL){
05         emOrdem_ArvBin(&((*raiz)->esq));
06         printf("%d\n", (*raiz)->info);
07         emOrdem_ArvBin(&((*raiz)->dir));
08     }
09 }
```

FIGURA 11.25

Por fim, a Figura 11.27 mostra a função que realiza o percurso **pós-ordem**. Esta função recebe como parâmetro uma árvore (raiz) e, caso ela não seja inválida (linhas 2-3) e não seja vazia (linha 4), irá executar o percurso. No percurso **pós-ordem** (linhas 5-7), primeiro chama-se o algoritmo recursivamente para percorrer a subárvore da esquerda (linha 5) e a da direita (linha 6). Em seguida, imprime-se (linha 7) o valor associado ao nó. Note que nas chamadas recursivas temos que passar o endereço do nó, pois a função tem como parâmetro um **ponteiro para ponteiro**. O processo de imprimir os nós de uma árvore usando o percurso **pós-ordem** é mais bem ilustrado pela Figura 11.28.

```

Percorso pós-ordem
01 void posOrdem_ArvBin(ArvBin *raiz){
02     if(raiz == NULL)
03         return;
04     if(*raiz != NULL){
05         posOrdem_ArvBin(&((*raiz)->esq));
06         posOrdem_ArvBin(&((*raiz)->dir));
07         printf("%d\n", (*raiz)->info);
08     }
09 }
```

FIGURA 11.27

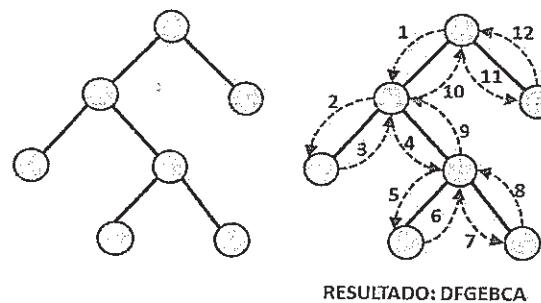


FIGURA 11.28

- inicia no nó A  
1 visita B  
2 visita D  
3 imprime D, volta para B  
4 visita E  
5 visita F  
6 imprime F, volta para E  
7 visita G  
8 imprime G, volta para E  
9 imprime E, volta para B  
10 imprime B, volta para A  
11 visita C  
12 imprime C, volta para A e imprime A

### Inserindo ou removendo um nó da árvore

As operações de inserção ou remoção de um nó na árvore dependem da aplicação a que se destina a árvore. Por tal motivo, essas operações serão vistas nas próximas seções.

## 11.5 ÁRVORE BINÁRIA DE BUSCA

### 11.5.1 Definição

Uma árvore binária de busca é um tipo especial de árvore binária, ou seja, ela possui as mesmas propriedades de uma árvore binária. No entanto, nesta nova árvore, cada nó possui um **valor** (chave) associado a ele, e este **valor** determina a posição do nó na árvore.



A princípio, considera-se que não existem valores repetidos na árvore.

Em uma árvore binária de busca, temos a seguinte regra para posicionamento dos valores na árvore, para cada nó pai:

- Todos os valores da subárvore esquerda são **menores** do que o nó pai.
- Todos os valores da subárvore direita são **maiores** do que o nó pai.

Um exemplo de árvore binária de busca é mostrado na Figura 11.29.



A **inserção** e a **remoção** de nós na árvore binária de busca devem ser realizadas respeitando essa regra de posicionamento dos nós.

A árvore binária de busca é uma ótima alternativa ao uso de arrays para operações de busca binária, pois, além de permitir este tipo de busca, ela possui a vantagem de ser uma estrutura dinâmica: é muito mais fácil acrescentar um nó na árvore segundo a sua regra de posicionamento do que inserir um valor dentro de um vetor ordenado (o que envolveria deslocar vários elementos a todo momento). Além disso, esse tipo de árvore também é utilizada para análise de expressões algébricas: prefixa, infixa e pós-fixa.

No Quadro 11.1, podemos ver o custo para as principais operações em uma árvore binária de busca contendo N nós. Note que o pior caso ocorre quando a árvore não está balanceada.

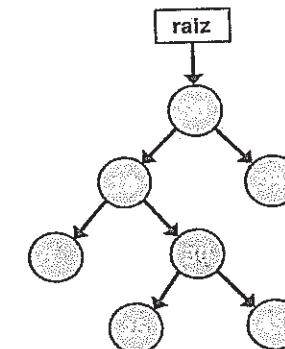


FIGURA 11.29

QUADRO 11.1

Operação	Caso médio	Pior caso
Inserção	$O(\log N)$	$O(N)$
Remoção	$O(\log N)$	$O(N)$
Consulta	$O(\log N)$	$O(N)$

### 11.5.2 Inserindo um nó na árvore

Inserir um novo nó em uma árvore binária de busca é uma tarefa bastante simples. Basicamente, o que temos que fazer é alocar espaço para o novo nó e procurar a sua posição na árvore usando o seguinte conjunto de passos:

- Primeiro, compare o **valor** a ser inserido com a **raiz**.
- Se o **valor** for **menor** do que a **raiz**: vá para a subárvore da **esquerda**.
- Se o **valor** for **maior** do que a **raiz**: vá para a subárvore da **direita**.
- Aplique o método recursivamente (pode ser feito sem recursão) até chegar a um **nó folha**.

Também existe o caso em que a inserção é feita em uma árvore que está vazia, como mostrado na Figura 11.30. Nesta situação, a raiz da árvore, que inicialmente apontava para NULL, passa a apontar para o único elemento inserido até então.

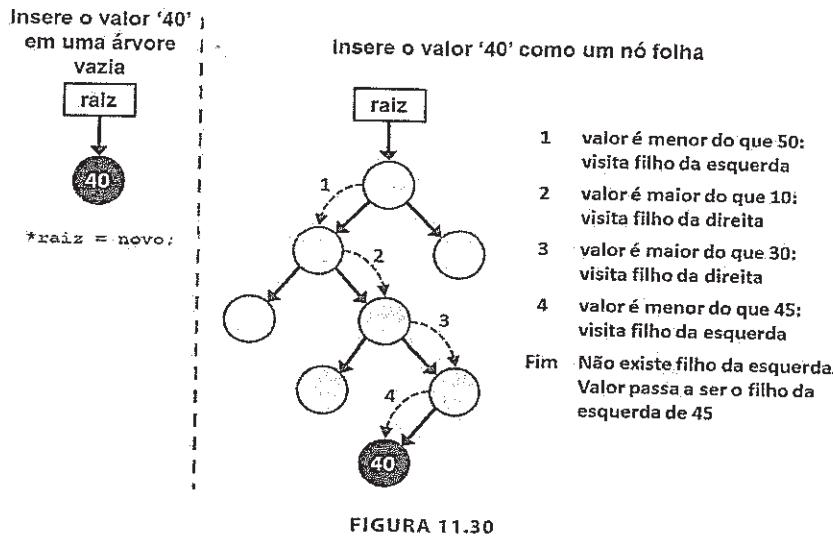


FIGURA 11.30

A Figura 11.31 mostra a implementação da função de inserção. Primeiramente, a função verifica se o ponteiro **ArvBin\*** **raiz** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da árvore. Assim, optamos por retornar o valor **ZERO** para indicar uma árvore inválida (linha 3). Porém, se a árvore foi criada com sucesso, podemos tentar alocar memória para um novo nó (linhas 4 e 5). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linhas 6 e 7). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse nó (linha 8). Repare também que o nó não possui filhos à esquerda nem à direita (linhas 9-10).



Todo nó inserido em uma árvore binária de busca é um **nó folha ou terminal**.

Em seguida, temos que considerar que a árvore pode ou não estar vazia (linha 12):

- No caso de ser uma árvore vazia, mudamos o conteúdo da “raiz” da árvore (**\*raiz**) para que ele passe a ser o nosso **novo** (linha 13).
- No caso de NÃO ser uma árvore vazia, temos que percorrer a árvore procurando o nó folha onde o novo nó será inserido. Para isso, criamos um **nó atual** começando na **raiz**

```

Inserindo um elemento na ÁRB:
01 int insere_ArvBin(ArvBin* raiz, int valor){
02     if(raiz == NULL)
03         return 0;
04     struct NO* novo;
05     novo = (struct NO*) malloc(sizeof(struct NO));
06     if(novo == NULL)
07         return 0;
08     novo->info = valor;
09     novo->dir = NULL;
10    novo->esq = NULL;
11
12    if(*raiz == NULL)
13        *raiz = novo;
14    else{
15        struct NO* atual = *raiz;
16        struct NO* ant = NULL;
17        while(atual != NULL){
18            ant = atual;
19            if(valor == atual->info){
20                free(novo);
21                return 0;//elemento já existe
22            }
23            if(valor > atual->info)
24                atual = atual->dir;
25            else
26                atual = atual->esq;
27        }
28        if(valor > ant->info)
29            ant->dir = novo;
30        else
31            ant->esq = novo;
32    }
33    return 1;
34 }
35 }
```

FIGURA 11.31

(linha 15). Note que, além do nó **atual**, também armazenamos seu nó pai (**ant**). Enquanto o nó **atual** for diferente de **NULL**:

- O nó **ant** recebe o valor de **atual** (linha 18).
- Se o valor do nó (**info**) for igual ao valor a ser inserido (**valor**), o valor já existe e a inserção termina retornando **ZERO** (linhas 19-22).
- Se o valor a ser inserido (**valor**) for maior do que o **atual**, o nó **atual** recebe o seu filho da **direita**. Do contrário, ele recebe o filho da **esquerda** (linhas 24-27);
- Uma vez que o nó **atual** seja igual a **NULL**, basta verificar se o novo nó será filho da **esquerda** ou da **direita** do nó **ant** (linhas 29-32).

Ao final do processo, retornamos o valor **UM** para indicar sucesso na operação de inserção (linha 34). Esse processo é mais bem ilustrado pela Figura 11.30.

### 11.5.3 Removendo um nó da árvore

Remover um nó de uma árvore binária de busca não é uma tarefa tão simples quanto uma inserção.



Isso ocorre porque precisamos procurar o nó a ser removido da árvore, o qual pode ser um nó **folha** ou um **nó interno** (que pode ser a raiz) com um ou dois filhos. Se este for um nó interno, é preciso reorganizar a árvore para que ela continue sendo uma árvore binária de busca.

Além disso, precisamos verificar se a árvore é vazia (quando a remoção não é possível), e se a remoção desse nó não gera uma árvore vazia.

A Figura 11.32 mostra o código utilizado para remover um nó de uma árvore binária de busca. Esse algoritmo utiliza uma função para procurar o nó a ser removido, **remove\_ArvBin** (linhas 22-46), e outra para remover e reorganizar a árvore (se necessário), **remove\_atual** (linhas 1-21).

A função **remove\_ArvBin** recebe como parâmetros a árvore (**ArvBin \*raiz**) e um número inteiro (**valor**) do nó que será removido. Primeiramente, a função verifica se o ponteiro **ArvBin \*raiz** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da árvore. Assim, optamos por retornar o valor **ZERO** para indicar uma árvore inválida (linhas 23-24). Porém, se a árvore é válida, criamos dois nós auxiliares: **atual**, começando na **raiz**, e seu nó pai, **ant** (linhas 25-26). Enquanto o nó **atual** for diferente de **NULL**:

- Se o valor do nó **atual** (**info**) for igual ao valor buscado (**valor**), verificamos se o nó **atual** é a **raiz** da árvore ou um dos filhos do nó **ant**. Para todos estes casos, chamamos a função **remove\_atual** e retornamos o valor **UM**, indicando sucesso na operação de remoção (linhas 28-38).
- Se o valor do nó **atual** (**info**) for diferente do valor buscado (**valor**), o nó **ant** recebe o **atual**. Se o valor buscado for maior do que o **atual**, o nó **atual** recebe o seu filho da **direita**. Do contrário, recebe o filho da **esquerda** (linhas 39-43).

Uma vez que o nó **atual** seja igual a **NULL**, retornamos o valor **ZERO** para indicar que o valor procurado não se encontra na árvore (linha 45).

Já a função **remove\_atual** recebe como parâmetro o endereço de um nó da árvore (**atual**) a ser removido e retorna qual deverá ser o seu nó substituto na árvore. Primeiro, a função verifica se o nó a ser removido não possui filho à esquerda (linha 3). Se a condição for verdadeira, a função copia o filho da direita para um nó auxiliar (**no2**), libera o nó **atual**, e retorna o filho da direita (**no2**) como substituto do nó **atual** na árvore (linhas 4-6). O processo, ilustrado na Figura 11.33, permite remover um nó que seja **folha** ou que possua apenas a subárvore da **direita**.

### Removendo um elemento da ABB

```

01 struct NO* remove_atual(struct NO* atual) {
02     struct NO *nol, *no2;
03     if(atual->esq == NULL){
04         no2 = atual->dir;
05         free(atual);
06         return no2;
07     }
08     nol = atual;
09     no2 = atual->esq;
10     while(no2->dir != NULL){
11         nol = no2;
12         no2 = no2->dir;
13     }
14     if(nol != atual){
15         nol->dir = no2->esq;
16         no2->esq = atual->esq;
17     }
18     no2->dir = atual->dir;
19     free(atual);
20     return no2;
21 }
22 int remove_ArvBin(ArvBin *raiz, int valor){
23     if(raiz == NULL)
24         return 0;
25     struct NO* ant = NULL;
26     struct NO* atual = *raiz;
27     while(atual != NULL){
28         if(valor == atual->info){
29             if(atual == *raiz)
30                 *raiz = remove_atual(atual);
31             else{
32                 if(ant->dir == atual)
33                     ant->dir = remove_atual(atual);
34                 else
35                     ant->esq = remove_atual(atual);
36             }
37             return 1;
38         }
39         ant = atual;
40         if(valor > atual->info)
41             atual = atual->dir;
42         else
43             atual = atual->esq;
44     }
45     return 0;
46 }
```

FIGURA 11.32

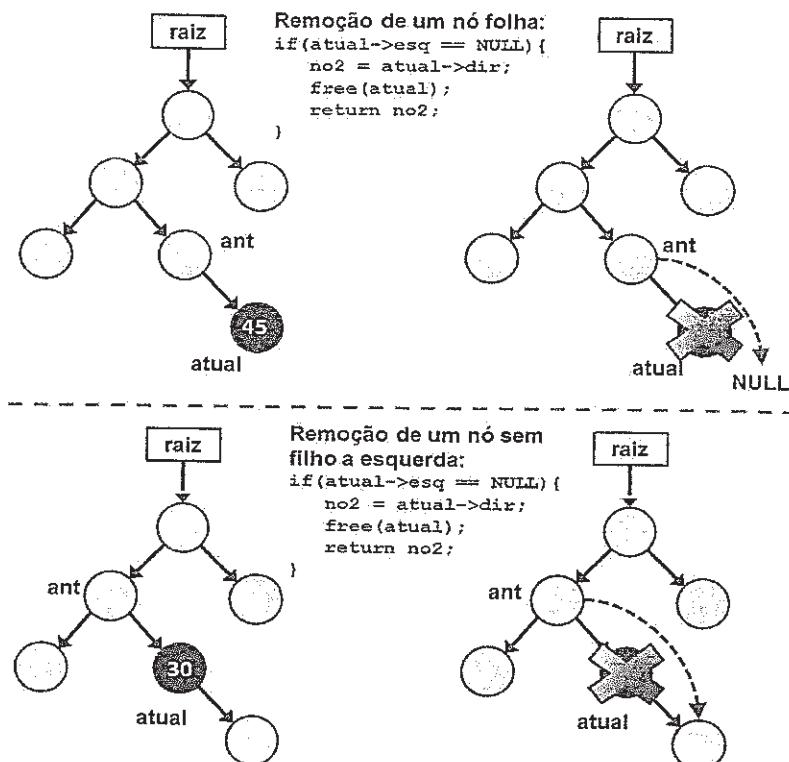


FIGURA 11.33

A parte mais difícil é remover um nó que possua os dois filhos. Neste caso, precisamos procurar o filho mais à direita da subárvore da esquerda (**no2**) do nó **atual** (linhas 8-13). Uma vez encontrado este filho, verificamos se o pai dele (**no1**) é diferente do nó a ser removido (**atual**). Se forem diferentes, a subárvore da esquerda de **no2** (se existir) se torna a subárvore da direita de **no1**, e a subárvore da esquerda de **atual** se torna a subárvore da esquerda de **no2** (linhas 14-17). Em seguida, fazemos com que **no2** receba a subárvore da direita do nó **atual** como sendo sua (linha 18). Por fim, liberamos o nó **atual** e retornamos **no2** como sendo o seu substituto (linhas 19-20). Esse processo é mais bem ilustrado pela Figura 11.34.

#### 11.5.4 Consultando um nó da árvore

Consultar se um nó existe em uma árvore binária de busca é uma tarefa similar à inserção de um novo nó. Basicamente, o que temos que fazer é percorrer os nós da árvore usando o seguinte conjunto de passos:

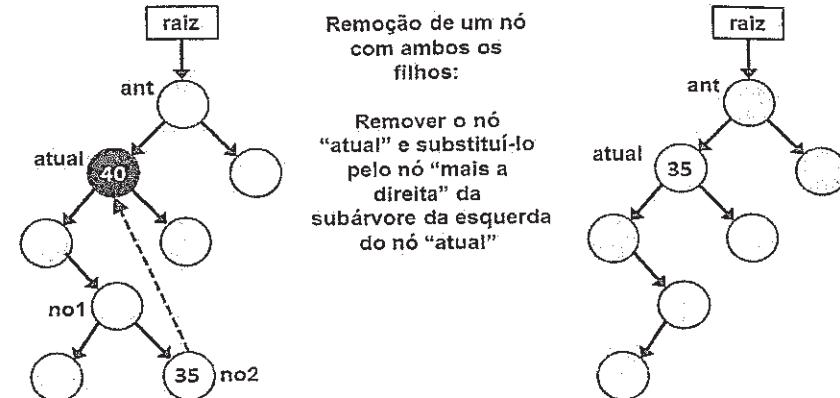


FIGURA 11.34

- Primeiro, compare o **valor** buscado com a **raiz**.
- Se o **valor** for **menor** do que a **raiz**: vá para a subárvore da **esquerda**.
- Se o **valor** for **maior** do que a **raiz**: vá para a subárvore da **direita**.
- Aplique o método recursivamente (pode ser feito sem recursão) até que a **raiz** seja igual ao **valor** buscado.

A Figura 11.35 mostra a implementação da função de busca. Neste caso, estamos procurando um nó pelo seu valor. Primeiro, a função verifica se o ponteiro **ArvBin\*** **raiz** é igual a **NULL**. A condição seria verdadeira se houvesse ocorrido um problema na criação da árvore. Assim, optamos por retornar o valor **ZERO** para indicar uma árvore inválida (linha 3). No entanto, se a árvore é válida, criamos um nó auxiliar (**atual**) para receber o conteúdo da **raiz** (linha 4). Este nó será utilizado para percorrer a árvore. Enquanto o nó **atual** for diferente de **NULL**:

- Se o valor do nó **atual** (**info**) for igual ao valor buscado (**valor**), a função retorna o valor **UM** para indicar sucesso na operação de busca (linhas 6-8).
- Se o valor buscado for **maior** do que o **atual**, o nó **atual** recebe o seu filho da **direita**. Do contrário, ele recebe o filho da **esquerda** (linhas 9-12).

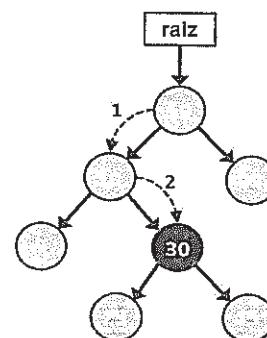
Uma vez que o nó **atual** seja igual a **NULL**, retornamos o valor **ZERO** para indicar que o valor procurado não se encontra na árvore (linha 14). O processo de busca por um elemento que existe ou não na árvore é mais bem ilustrado pela Figura 11.36.

### Consultando um elemento da ABE

```

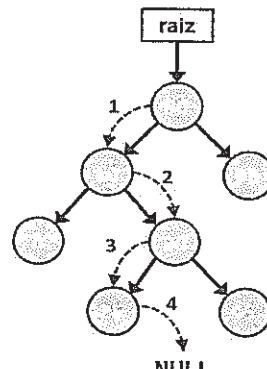
01 int consulta_ArvBin(ArvBin *raiz, int valor) {
02     if(raiz == NULL)
03         return 0;
04     struct NO* atual = *raiz;
05     while(atual != NULL){
06         if(valor == atual->info){
07             return 1;
08         }
09         if(valor > atual->info)
10             atual = atual->dir;
11         else
12             atual = atual->esq;
13     }
14     return 0;
15 }
```

FIGURA 11.35



Valor procurado: 30

- 1 valor procurado é menor do que 50: visita filho da esquerda
- 2 valor procurado é maior do que 10: visita filho da direita
- valor procurado é igual ao do nó: retornar dados do nó



Valor procurado: 28

- 1 valor procurado é menor do que 50: visita filho da esquerda
  - 2 valor procurado é maior do que 10: visita filho da direita
  - 3 valor procurado é menor do que 30: visita filho da esquerda
  - 4 valor procurado é maior do que 25: visita filho da direita
- Filho da direita de 25 não existe:  
a busca falhou

FIGURA 11.36

### 1.6 ÁRVORE BINÁRIA DE BUSCA BALANCEADA

Uma árvore binária de busca balanceada é uma árvore binária em que as alturas das subárvore **esquerda** e **direita** de cada nó da árvore diferem de, no máximo, uma unidade.



Essa diferença das alturas das subárvore **esquerda** e **direita** é chamada **fator de balanceamento** (ou **fb**) do nó.

Infelizmente, os algoritmos de inserção e remoção em árvores binárias não garantem que a árvore gerada a cada passo esteja balanceada. Dependendo da ordem em que os dados são inseridos na árvore, podemos criar uma árvore na forma de uma escada. Um exemplo desse tipo de árvore ocorre quando os valores (1, 2, 3, 10, 4, 5, 9, 7, 8 e 6) são inseridos, nessa ordem, em uma árvore, como mostra a Figura 11.37.



A eficiência da busca em uma árvore binária depende do seu balanceamento.

Dado o mesmo número de nós  $N$  na árvore, quanto maior a altura, maior o custo da operação. Assim, o custo das principais operações em árvores (inserção, remoção e busca) é, no pior caso:

$O(\log N)$ , se a árvore está balanceada.

$O(N)$ , se a árvore não está balanceada.



Como então resolver o problema de balanceamento da árvore?

A solução para o problema de balanceamento da árvore consiste em modificar as operações de **inserção** e **remoção** da árvore binária para balancear a árvore a cada nova inserção ou remoção.

Na computação, existem vários tipos de árvores balanceadas. Cada uma delas foi desenvolvida pensando diferentes tipos de aplicações. Algumas delas:

- Árvore AVL.
- Árvore Red-Black (também conhecida como vermelho-preto ou rubro-negra).
- Árvore 2-3.
- Árvore 2-3-4.
- Árvore B, B+ e B\*.

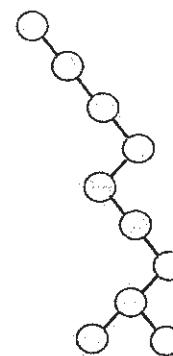


FIGURA 11.37

## 11.7 ÁRVORE AVL

### 11.7.1 Definição

A árvore AVL é um tipo de árvore binária balanceada com relação à altura das suas subárvore. Ela foi criada por Adelson-Velskii e Landis, de quem recebeu a sua nomenclatura, em 1962.

Trata-se de um tipo de árvore que permite o rebalanceamento local da árvore, ou seja, apenas a parte afetada pela **inserção** ou **remoção** é rebalanceada.



Para manter o balanceamento, a árvore AVL faz uso de rotações simples ou duplas na etapa de rebalanceamento, o qual ocorre a cada inserção ou remoção.

Por meio destas rotações, a árvore AVL busca manter-se como uma árvore binária quase completa. Assim, o custo máximo de qualquer algoritmo é  $O(\log N)$ .



O objetivo das operações de rotação é acertar o **fator de平衡amento** (ou **fb**) de cada nó, restituindo assim o equilíbrio da árvore. O **fator de平衡amento** nada mais é do que a diferença das alturas das subárvore esquerda e direita de um nó.

A Figura 11.38 mostra como é realizado o cálculo do **fator de平衡amento** de determinado nó (no caso, o nó pai da árvore). Este cálculo deve ser realizado para cada um dos nós da árvore. A Figura 11.38 mostra o cálculo do **fator de平衡amento** para cada nó da árvore.



Caso uma das subárvore de um nó não exista, então a altura desta subárvore será igual a -1.

### Fator de Balanceamento

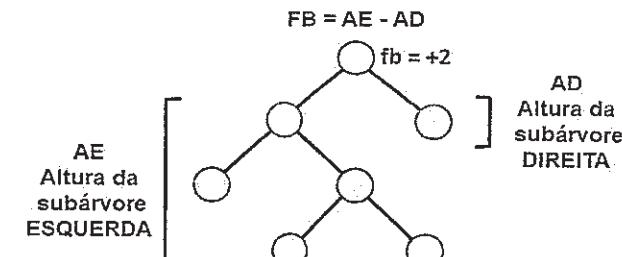


FIGURA 11.38

Na árvore AVL, as alturas das subárvore esquerda e direita de cada nó diferem de, no máximo, uma unidade. Ou seja, o **fator de平衡amento** deve ser +1, 0 ou -1.



Se o **fator de平衡amento** for maior do que +1 ou menor do que -1 em um nó da árvore AVL, então a árvore deve ser balanceada naquele nó.

Um exemplo de árvore que precisa de balanceamento é mostrado na Figura 11.39. Perceba que, no nó que precisa ser rebalanceado, a altura da subárvore da esquerda é muito maior que a altura da subárvore da direita, assim como a quantidade de nós em cada subárvore.

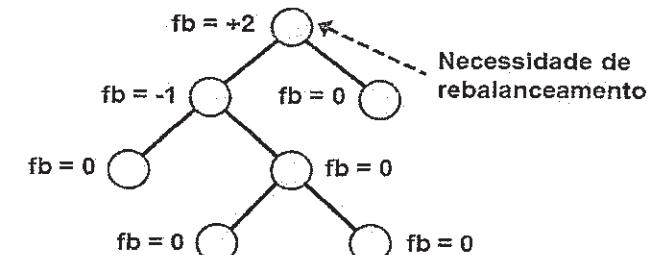


FIGURA 11.39

A Figura 11.40 mostra uma árvore AVL e o fator de balanceamento de cada nó. Esta árvore é obtida ao se inserir os valores (1, 2, 3, 10, 4, 5, 9, 7, 8 e 6), nessa ordem, na árvore. Para ver a mesma árvore sem balanceamento, basta ver a Figura 11.37.

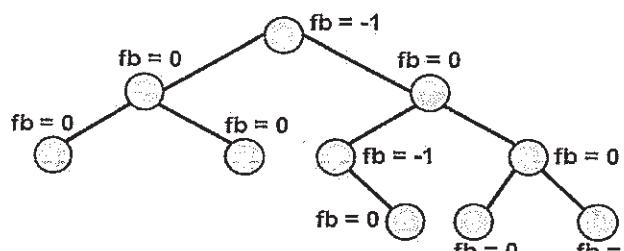


FIGURA 11.40

### 11.7.2 Implementando uma árvore AVL

A implementação da TAD de uma árvore AVL é idêntica à da árvore binária mostrada na Seção 11.4.3. Ou seja, aqui também iremos utilizar um **ponteiro para ponteiro** para guardar o primeiro nó da árvore. Isso é especialmente importante na árvore AVL, pois o uso de um **ponteiro para ponteiro** permite mudar mais facilmente quem é a **raiz** no caso de uma rotação (se necessário).

Como na árvore binária, o arquivo **ArvoreAVL.h**, ilustrado na Figura 11.41, define:

- Para fins de padronização, um novo nome para o **ponteiro do tipo árvore** (linha 1). Esse é o tipo que será usado sempre que se desejar trabalhar com uma árvore AVL.
- As funções disponíveis para se trabalhar com essa árvore, e que serão implementadas no arquivo **ArvoreAVL.c** (linhas 3-13).

Já o arquivo **ArvoreAVL.c** (Figura 11.41) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da árvore AVL (linhas 1-3).
- A definição do tipo que descreve cada nó da árvore AVL, **struct NO** (linhas 4-9).
- As implementações das funções definidas no arquivo **arvoreav1.h**. As implementações dessas funções serão vistas nas seções seguintes.

Perceba que a nossa estrutura **NO** possui um campo a mais na árvore AVL do que na árvore binária. O campo **alt** será utilizado para armazenar a altura da subárvore considerando aquele nó como a raiz. Desse modo, torna-se mais rápido e direto o cálculo do **fator de平衡amento** durante o balanceamento da árvore.

#### Arquivo ArvoreAVL.h

```

01 typedef struct NO* ArvAVL;
02
03 ArvAVL* cria_ArvAVL();
04 void libera_ArvAVL(ArvAVL *raiz);
05 int insere_ArvAVL(ArvAVL* raiz, int valor);
06 int remove_ArvAVL(ArvAVL *raiz, int valor);
07 int estaVazia_ArvAVL(ArvAVL *raiz);
08 int altura_ArvAVL(ArvAVL *raiz);
09 int totalNO_ArvAVL(ArvAVL *raiz);
10 int consulta_ArvAVL(ArvAVL *raiz, int valor);
11 void preOrdem_ArvAVL(ArvAVL *raiz);
12 void emOrdem_ArvAVL(ArvAVL *raiz);
13 void posOrdem_ArvAVL(ArvAVL *raiz);
  
```

#### Arquivo ArvoreAVL.c

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ArvoreAVL.h" //inclui os Protótipos
04 struct NO{
05     int info;
06     int alt;
07     struct NO *esq;
08     struct NO *dir;
09 };
  
```

FIGURA 11.41



Com respeito à implementação, das funções, com exceção das funções de **inserção** e **remoção**, as demais funções da árvore AVL são implementadas de forma idêntica às da árvore binária.

Além dessas funções, vamos definir algumas funções auxiliares (Figura 11.42). O propósito delas é agilizar a tarefa de cálculo do balanceamento da árvore.

### Funções auxiliares

```

01 // retorna a altura de uma árvore
02 int alt_NO(struct NO* no) {
03     if(no == NULL)
04         return -1;
05     else
06         return no->alt;
07 }
08
09 // retorna o fator de balanceamento de um nó
10 int fatorBalanceamento_NO(struct NO* no) {
11     return labs(alt_NO(no->esq) - alt_NO(no->dir));
12 }
13
14 // retorna o maior dentre dois valores
15 int maior(int x, int y) {
16     if(x > y)
17         return x;
18     else
19         return y;
20 }

```

FIGURA 11.42

Por fim, para utilizarmos nossa árvore AVL no arquivo `main()`, basta declarar um ponteiro para ele (definido no arquivo `ArvoreAVL.h`), da seguinte forma:

`ArvAVL *raiz;`

### 11.7.3 Rotações

A operação básica usada para balancear uma árvore AVL é a rotação. É por meio dela que a árvore AVL busca manter-se como uma árvore binária **quase completa**. Ao todo, existem dois tipos de rotação: **simples** e **dupla**. Os dois tipos diferem entre si pelo sentido da inclinação entre o nó **pai** e **filho**:

- **Rotação simples:** o nó desbalanceado (pai), seu filho e o seu neto estão todos no mesmo sentido de inclinação.
- **Rotação dupla:** o nó desbalanceado (pai) e seu filho estão inclinados no sentido inverso ao neto. Equivale a duas rotações simples.

Além disso, existem duas rotações **simples** e duas **duplas**:

- **Rotação simples à direita** ou **rotação LL**.
- **Rotação simples à esquerda** ou **rotação RR**.
- **Rotação dupla à direita** ou **rotação LR**.
- **Rotação dupla à esquerda** ou **rotação RL**.



As rotações são aplicadas no ancestral mais próximo do nó inserido cujo fator de balanceamento passa a ser +2 ou -2.

A seguir são apresentadas as implementações desses quatro tipos de rotação. O parâmetro das funções implementadas é o ancestral mais próximo do nó inserido.



Dependendo do desbalanceamento a ser tratado na árvore, uma única rotação pode não ser suficiente. Neste caso, várias rotações são realizadas em diferentes nós da árvore.

#### Rotação LL

A rotação LL, ou rotação simples à direita, ocorre quando um novo nó é inserido na subárvore da **esquerda** do filho **esquerdo** de A, que é o nó desbalanceado. Neste caso, como houve dois movimentos para a esquerda em relação ao nó A, é necessário fazer uma **rotação à direita**, de modo que o nó intermediário B ocupe o lugar de A, e A se torne a subárvore direita de B, como mostra a Figura 11.43. Nesta figura, o novo nó inserido pode ser o próprio nó C ou estar em uma das subárvore do nó C.

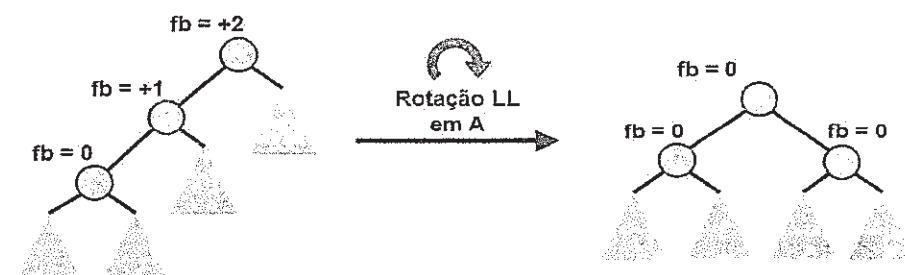


FIGURA 11.43

A Figura 11.44 mostra a implementação da função que realiza a **rotação LL**. A função recebe como parâmetro o nó da árvore que está desbalanceado (A) como se este fosse a **raiz** de uma árvore AVL (`ArvAVL *`). Primeiramente, a função associa ao nó B o filho da esquerda do nó A (linhas 2-3) e coloca o filho à direita de B (se este existir) como sendo o novo filho à esquerda de A (linha 4). Na sequência, o nó A se torna o filho à direita do nó B (linha 5). Como os nós A e B tiveram suas posições alteradas, é necessário recalcular a altura de cada um deles (linhas

6-7). Por fim, o conteúdo da raiz, que antes continha o nó A, passa a apontar para o nó B. Essa última operação só é possível porque o nó A foi passado para a função como um ponteiro para ponteiro (ArvAVL \*).

```
Rotação LL
01 void RotacaoLL(ArvAVL *A) {
02     struct NO *B;
03     B = (*A)->esq;
04     (*A)->esq = B->dir;
05     B->dir = *A;
06     (*A)->altura = maior(altura_NO((*A)->esq),
07                             altura_NO((*A)->dir)) + 1;
08     B->altura = maior(altura_NO(B->esq), (*A)->altura) + 1;
09 }
```

FIGURA 11.44

A Figura 11.45 mostra um exemplo passo a passo desse tipo de rotação.

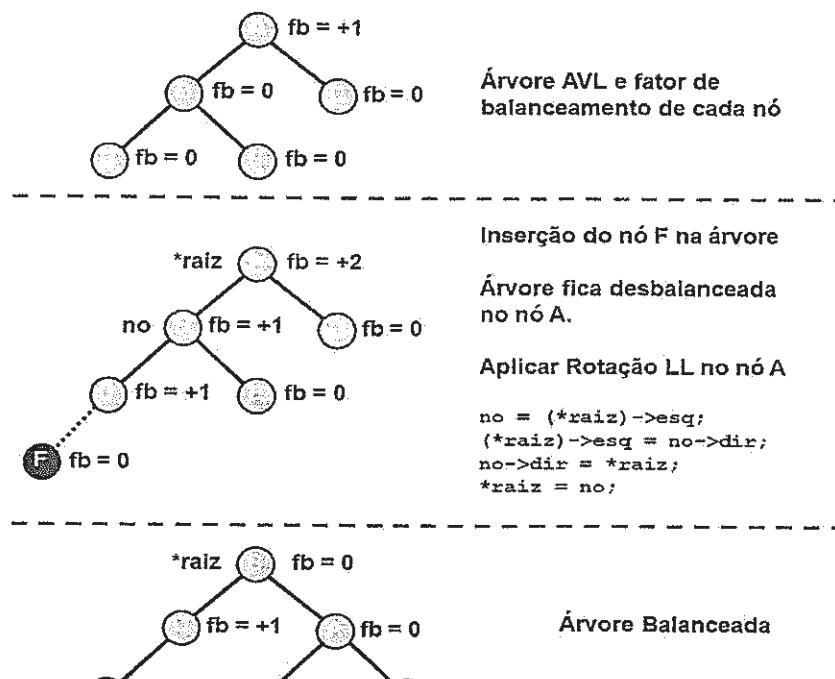


FIGURA 11.45

### Rotação RR

A rotação RR, ou **rotação simples à esquerda**, ocorre quando um novo nó é inserido na subárvore da **direita** do filho **direito** de A, que é o nó desbalanceado. Neste caso, como houve dois movimentos para a direita em relação ao nó A, é necessário fazer uma **rotação à esquerda**, de modo que o nó intermediário B ocupe o lugar de A, e A se torne a subárvore esquerda de B, como mostra a Figura 11.46. Nesta figura, o novo nó inserido pode ser o próprio nó C ou estar em uma das subárvore do nó C.



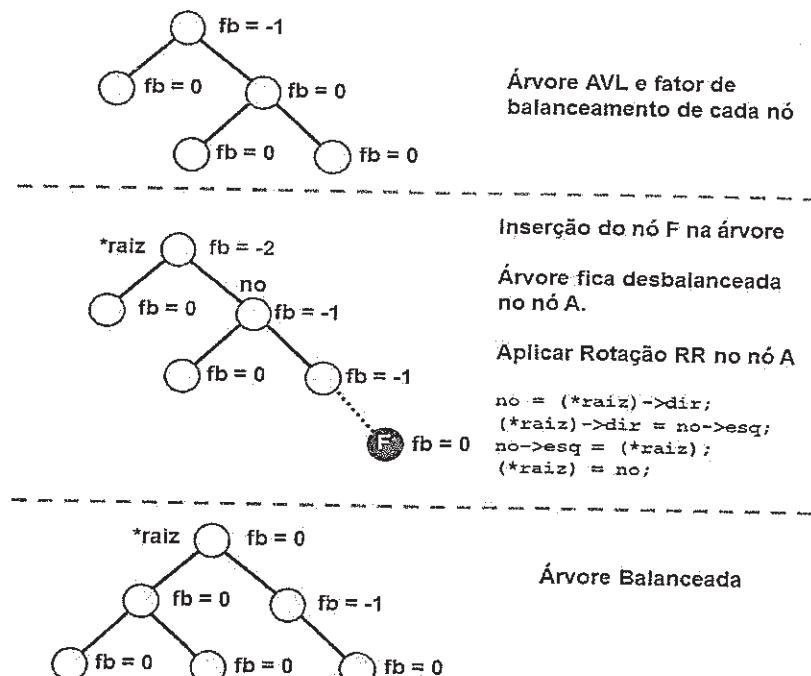
FIGURA 11.46

A Figura 11.47 mostra a implementação da função que realiza a **rotação RR**. A função recebe como parâmetro o nó da árvore que está desbalanceado (A) como se este fosse a **raiz** de uma árvore AVL (ArvAVL \*). Primeiramente, a função associa ao nó B o filho da **direita** do nó A (linhas 2-3) e coloca o filho à **esquerda** de B (se este existir) como sendo o novo filho à **direita** de A (linha 4). Na sequência, o nó A se torna o filho à **esquerda** do nó B (linha 5). Como os nós A e B tiveram suas posições alteradas, é necessário recalcular a altura de cada um deles (linhas 6-7). Por fim, o conteúdo da **raiz**, que antes continha o nó A, passa a apontar para o nó B. Essa última operação só é possível porque o nó A foi passado para a função como um ponteiro para ponteiro (ArvAVL \*).

```
Rotação RR
01 void RotacaoRR(ArvAVL *A) {
02     struct NO *B;
03     B = (*A)->dir;
04     (*A)->dir = B->esq;
05     B->esq = (*A);
06     (*A)->altura = maior(altura_NO((*A)->esq),
07                           altura_NO((*A)->dir)) + 1;
08     B->altura = maior(altura_NO(B->dir), (*A)->altura) + 1;
09 }
```

FIGURA 11.47

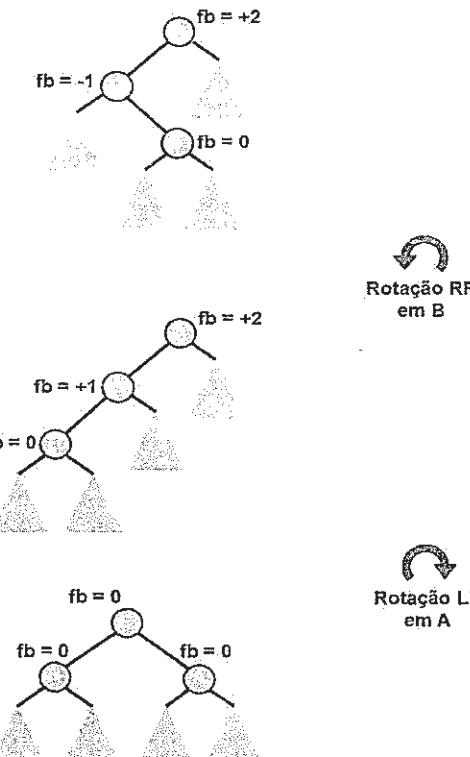
A Figura 11.48 mostra um exemplo passo a passo desse tipo de rotação.



**FIGURA 11.48**

## Rotação LR

A **rotação LR**, ou **rotação dupla à direita**, ocorre quando um novo nó é inserido na subárvore da direita do filho **esquerdo** de A, que é o nó desbalanceado. Neste caso, é necessário fazer uma **rotação dupla**, de modo que o nó C se torne o pai dos nós A (filho da direita) e B (filho da esquerda), como mostra a Figura 11.49. Nesta figura, o novo nó inserido pode ser o próprio nó C ou estar em uma das subárvores do nó C.



**FIGURA 11.49**



As rotações **duplas** podem ser implementadas utilizando duas rotações **simples**.

Como houve, partindo do nó A, um movimento para a **esquerda** e outro para a **direita**, a rotação **dupla** pode ser substituída por duas rotações **simples**: uma **rotação RR** aplicada no filho da esquerda do nó A (nó B) e uma **rotação LL** aplicada no nó A, como mostra a sua implementação na Figura 11.50.

## Rotacao LR

```
01 void RotacaoLR(ArvAVL *A) {
02     RotacaoRR(&(*A)->esq);
03     RotacaoLL(A);
04 }
```

**FIGURA 11.50**

A Figura 11.51 mostra um exemplo passo a passo desse tipo de rotação.

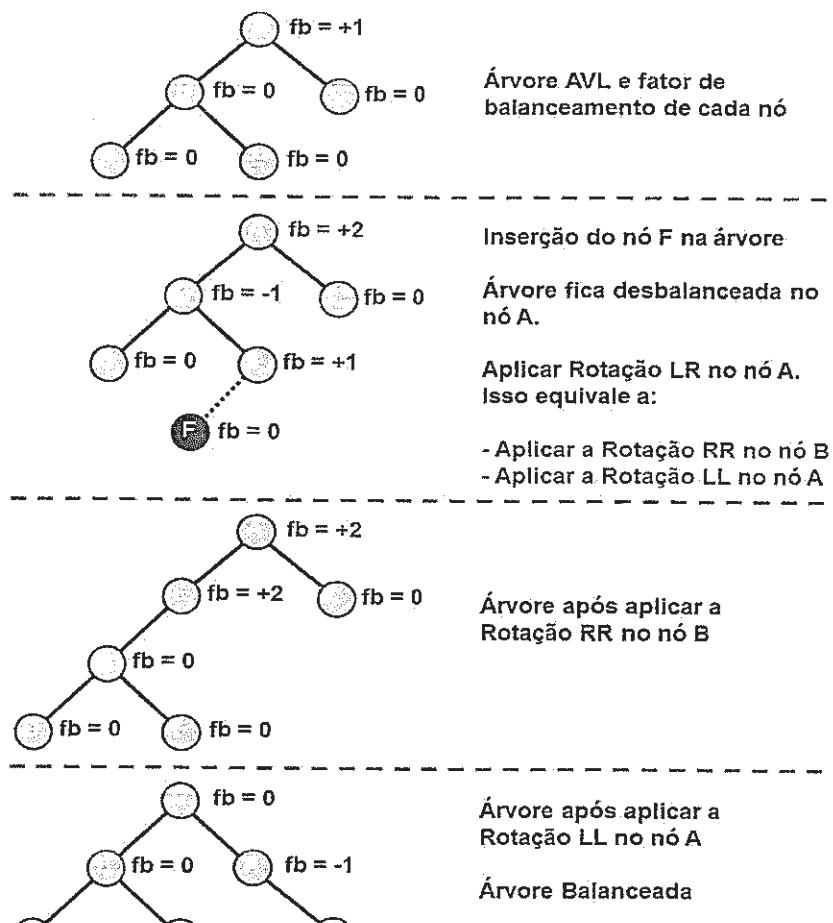


FIGURA 11.51

### Rotação RL

A rotação RL, ou rotação dupla à esquerda, ocorre quando um novo nó é inserido na subárvore da esquerda do filho direito de A, que é o nó desbalanceado. Neste caso, é necessário fazer uma rotação dupla, de modo que o nó C se torne o pai dos nós A (filho da esquerda) e B (filho da direita), como mostra a Figura 11.52. Nesta figura, o novo nó inserido pode ser o próprio nó C ou estar em uma das subárvores do nó C.

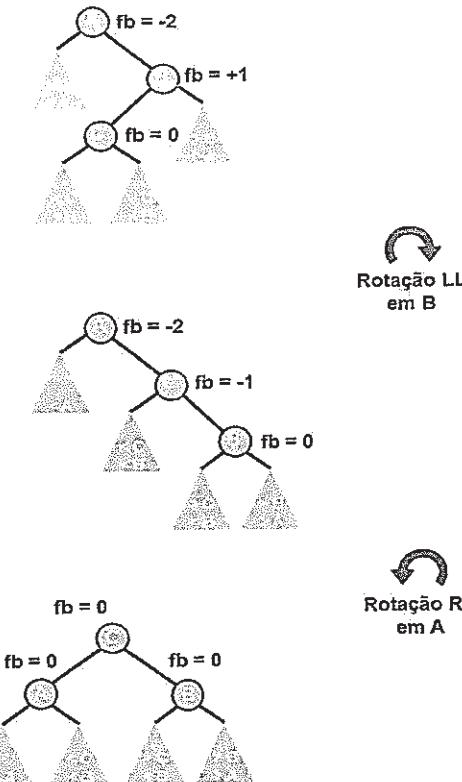


FIGURA 11.52



As rotações duplas podem ser implementadas utilizando duas rotações simples.

Como houve, partindo do nó A, um movimento para a direita e outro para a esquerda, a rotação dupla pode ser substituída por duas rotações simples: uma rotação LL aplicada no filho da direita do nó A (nó B) e uma rotação RR aplicada no nó A, como mostra a sua implementação na Figura 11.53.

Rotação RL	
<pre> 01 void RotacaoRL(ArvAVL *raiz){ 02     RotacaoLL(&amp;(*raiz)-&gt;dir); 03     RotacaoRR(raiz); 04 }</pre>	

FIGURA 11.53

A Figura 11.54 mostra um exemplo passo a passo desse tipo de rotação.

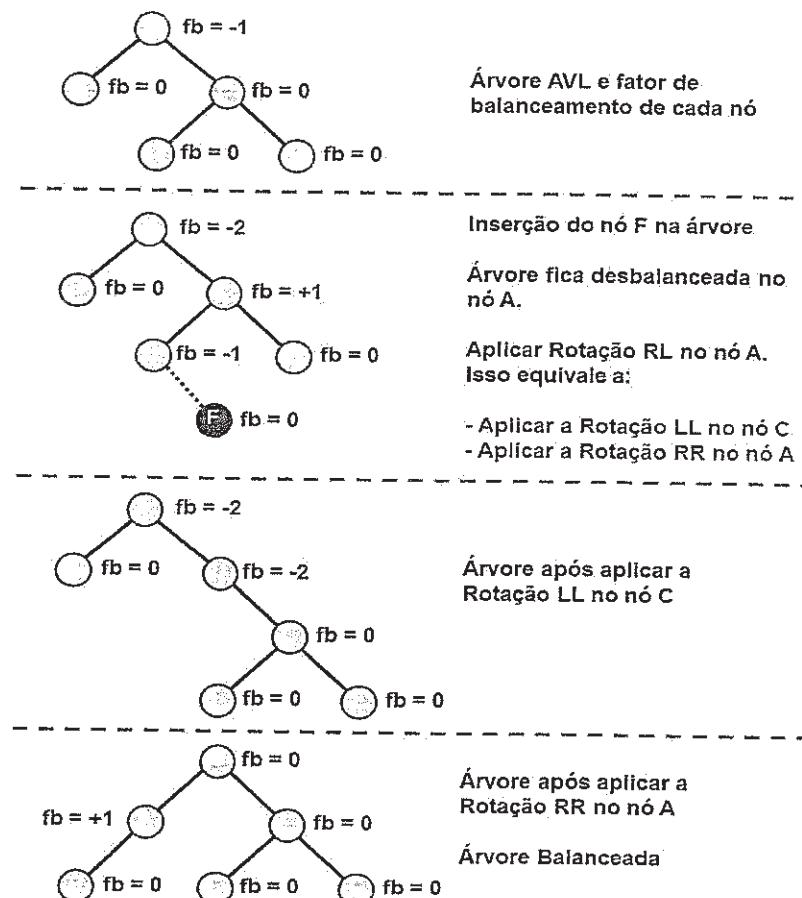


FIGURA 11.54

Quando usar cada tipo de rotação?

Uma dúvida muito comum na implementação da árvore AVL diz respeito a quando utilizar cada uma das quatro rotações. O Quadro 11.2 mostra os casos em que cada rotação é utilizada. É fácil perceber que, quando os sinais do nó A e do nó B são iguais, a rotação é **simples**. Além disso, se os sinais são iguais e negativos, a rotação é à esquerda (RR); senão, é uma rotação à direita (LL).

Nas rotações duplas, os sinais do nó A e do nó B são diferentes. Se o sinal de A é positivo, a rotação é dupla à direita (LR); senão, é uma rotação dupla à esquerda (RL).



As rotações LL e RR são simétricas entre si, assim como as rotações LR e RL.

E, como as funções de rotação apenas atualizam ponteiros, a sua complexidade é  $O(1)$ .

QUADRO 11.2

Fator de balanceamento de A	Fator de balanceamento de B	Posições dos nós B e C em relação ao nó A	Rotação
+2	+1	B é filho à esquerda de A C é filho à esquerda de B	LL
-2	-1	B é filho à direita de A C é filho à direita de B	RR
+2	-1	B é filho à esquerda de A C é filho à direita de B	LR
-2	+1	B é filho à direita de A C é filho à esquerda de B	RL

#### 11.7.4 Inserindo um nó na árvore

Inserir um novo nó em uma árvore AVL é uma tarefa bastante simples. Basicamente, o que temos que fazer é alocar espaço para o novo nó e procurar a sua posição na árvore usando o seguinte conjunto de passos:

- Primeiro, compare o **valor** a ser inserido com a **raiz**.
- Se o **valor** for **menor** do que a **raiz**: vá para a subárvore da **esquerda**.
- Se o **valor** for **maior** do que a **raiz**: vá para a subárvore da **direita**.
- Aplique o método recursivamente (pode ser feito sem recursão) até chegar a um **nó folha**.

Também existe o caso em que a inserção é feita em uma árvore que está vazia. Nesta situação, a raiz da árvore, que inicialmente apontava para **NULL**, passa a apontar para o único elemento inserido até então.



A inserção de um novo nó na árvore AVL é exatamente igual à inserção na árvore binária de busca. No entanto, uma vez inserido o novo nó na árvore, começam a surgir as diferenças entre uma simples árvore binária de busca e uma árvore AVL.

Para inserir um nó, temos que percorrer um conjunto de nós da árvore até chegar ao nó folha que irá se tornar o pai do novo nó. Uma vez inserido este nó, devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós e, se necessário, aplicar uma das quatro rotações para restabelecer o balanceamento da árvore.

A Figura 11.55 mostra a implementação da função de inserção. Primeiramente, a função verifica se o conteúdo do ponteiro **ArvBin\*** **raiz** é igual a **NULL** (linha 3). A condição é verdadeira em dois casos:

- Era uma árvore vazia.
- Ao descer, recursivamente, na árvore, o nó de onde viemos era um nó folha.

Seja qual for o caso, devemos alocar memória para inserir o **novo** nó (linhas 5-6). Caso a alocação de memória não seja possível, a função irá retornar o valor **ZERO** (linha 7). Tendo a função **malloc()** retornado um endereço de memória válido, podemos copiar os dados que vamos armazenar para dentro desse nó (linha 9). Repare também que o nó não possui filhos à esquerda nem à direita e que sua altura é **ZERO** (linhas 10-12). Por fim, copiamos o **novo** nó para o conteúdo da raiz da árvore e retornamos **UM** para indicar sucesso na operação de inserção (linhas 13-14).

Vamos imaginar que, inicialmente, nossa árvore não esteja vazia. Neste caso, temos que percorrer a árvore até achar o nó folha em que o novo nó será inserido. Primeiro, copiamos o conteúdo da raiz para um nó auxiliar, **atual** (linha 17). Em seguida, consideramos três casos:

- Se o valor do nó **atual** for **maior** do que o valor a ser inserido (**valor**), devemos ir, recursivamente, para a subárvore **esquerda** do nó **atual** (linhas 18-19).
- Se o valor do nó **atual** for **menor** do que o valor a ser inserido (**valor**), devemos ir, recursivamente, para a subárvore **direita** do nó **atual** (linhas 28-29).
- Se o valor do nó **atual** for **igual** ao valor a ser inserido (**valor**), o valor já existe na árvore e sua inserção termina retornando **ZERO**, o que indica que houve uma falha na inserção (linhas 37-38).

Esse conjunto de passos permite caminhar na árvore em busca do ponto de inserção do novo nó. Falta agora rebalancear a árvore. Note que as chamadas recursivas são feitas dentro de um comando condicional (**if**) (linhas 19 e 29). Se a resposta da função, armazenada na variável **res**, for igual a **UM** (1), a inserção foi realizada com sucesso e devemos verificar o balanceamento do nó **atual** (linhas 20 e 30). Devemos então considerar as quatro possíveis rotações, se o nó estiver desbalanceado:

- Se a chamada recursiva da função foi feita para a subárvore **esquerda** do nó **atual**, devemos escolher entre a **rotação LL** e a **rotação LR** (linha 19):
  - Se o **valor** inserido for **menor** do que o valor do filho à esquerda de **atual**, isso significa que o **valor** foi inserido na subárvore esquerda do filho esquerdo de **atual**: **rotação LL** (linhas 21-22).
  - Caso contrário, o **valor** foi inserido na subárvore direita do filho esquerdo de **atual**: **rotação LR** (linhas 23-24).
- Se a chamada recursiva da função foi feita para a subárvore **direita** do nó **atual**, devemos escolher entre a **rotação RR** e a **rotação RL** (linha 29):
  - Se o **valor** inserido for **maior** do que o valor do filho à direita de **atual**, isso significa que o **valor** foi inserido na subárvore direita do filho direito de **atual**: **rotação RR** (linhas 31-32).
  - Caso contrário, o **valor** foi inserido na subárvore esquerda do filho direito de **atual**: **rotação RL** (linhas 33-34).

Perceba que esse conjunto de passos responsável pela rotação é feito sempre que voltamos de uma chamada recursiva. Por fim, a função de inserção atualiza a altura do nó **atual** e retorna o valor **res**, que é o valor retornado pelas chamadas recursivas da função de inserção (linhas 40-41). O processo de inserção é mais bem ilustrado pelas Figuras 11.56, 11.57 e 11.58.

**Inserindo um elemento na AVL**

```

01 int insere_ArvAVL(ArvAVL *raiz, int valor){
02     int res;
03     if(*raiz == NULL) { //árvore vazia ou nó folha
04         struct NO *novo;
05         novo = (struct NO*)malloc(sizeof(struct NO));
06         if(novo == NULL)
07             return 0;
08
09         novo->info = valor;
10        novo->alt = 0;
11        novo->esq = NULL;
12        novo->dir = NULL;
13        *raiz = novo;
14        return 1;
15    }
16
17    struct NO *atual = *raiz;
18    if(valor < atual->info){
19        if((res = insere_ArvAVL(&(atual->esq), valor)) == 1){
20            if(fatorBalanceamento_NO(atual) >= 2){
21                if(valor < (*raiz)->esq->info)
22                    RotacaoLL(raiz);
23                else
24                    RotacaoLR(raiz);
25            }
26        }
27    }else{
28        if(valor > atual->info){
29            if((res = insere_ArvAVL(&(atual->dir), valor)) == 1){
30                if(fatorBalanceamento_NO(atual) >= 2){
31                    if((*raiz)->dir->info < valor)
32                        RotacaoRR(raiz);
33                    else
34                        RotacaoRL(raiz);
35                }
36            }
37        }
38    }
39    //Valor duplicado!!
40    atual->alt=maior(alt_NO(atual->esq), alt_NO(atual->dir))+1;
41    return res;
42 }
```

FIGURA 11.55

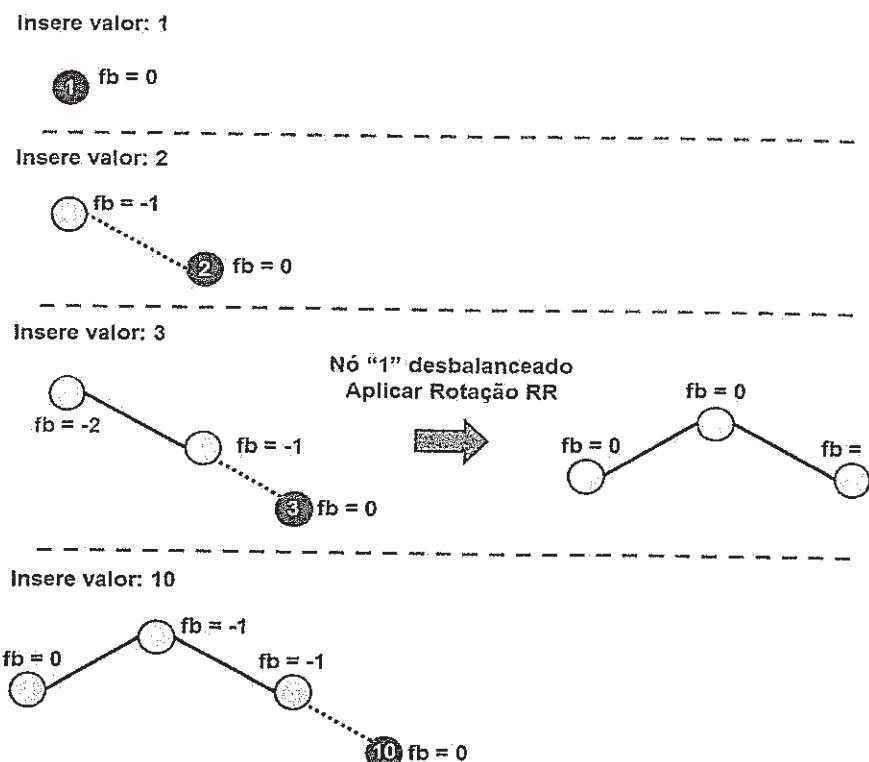


FIGURA 11.56

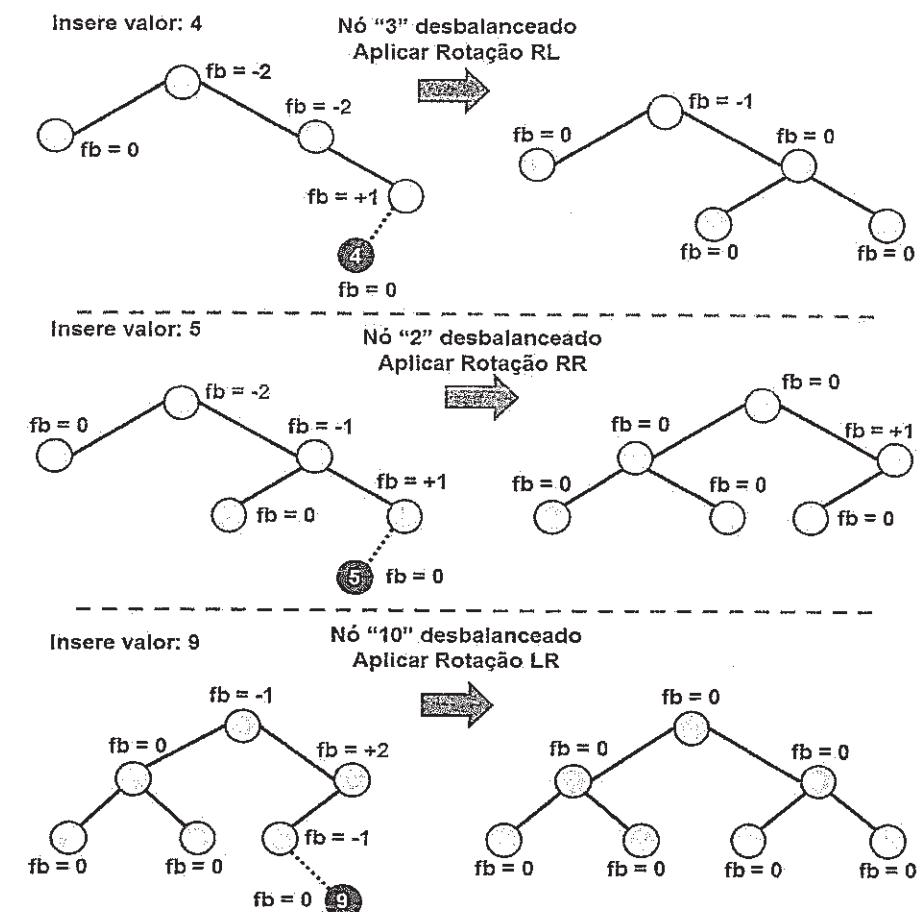
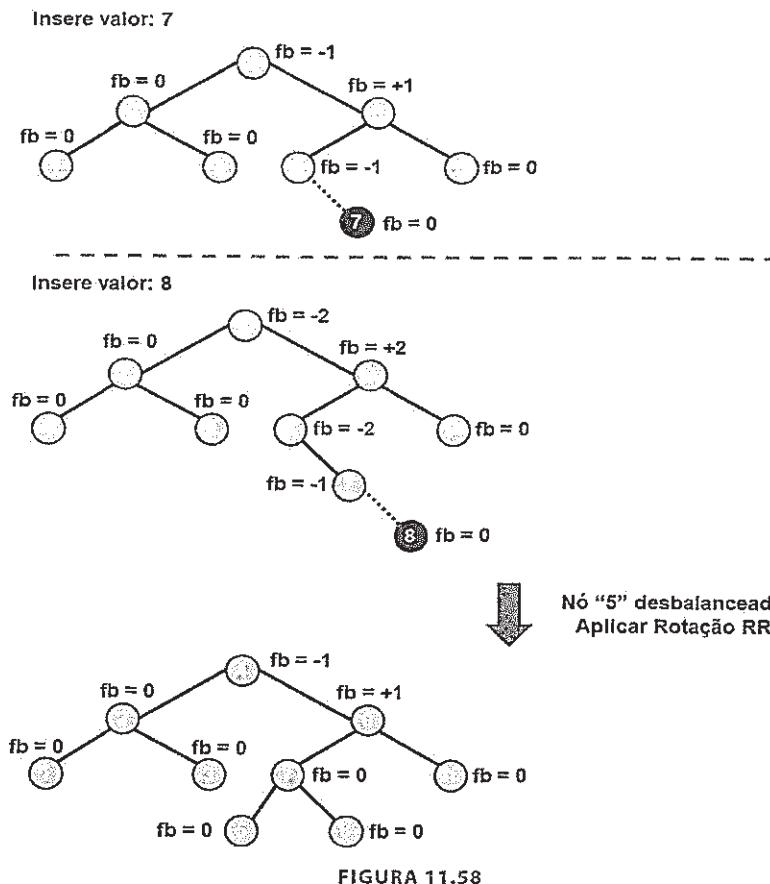


FIGURA 11.57



- Se o **valor** for **maior** do que a **raiz**: vá para a subárvore da **direita**.
- Aplique o método recursivamente (pode ser feito sem recursão) até encontrar o nó que contenha esse valor ou um ponteiro **NULL** (valor não existe na árvore).

Além disso, precisamos verificar se a árvore é vazia (neste caso, a remoção não é possível) e a remoção desse nó não gera uma árvore vazia.



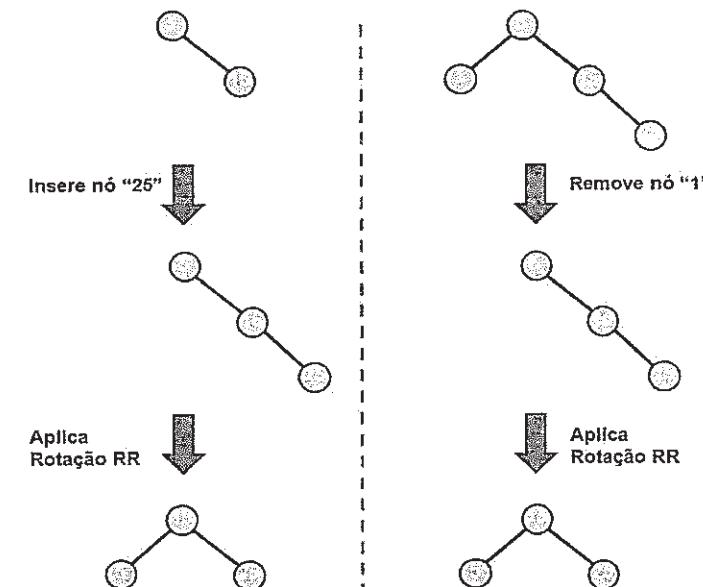
A remoção de um nó da árvore AVL é exatamente igual à remoção na árvore binária de busca. No entanto, uma vez removido o nó, começam a surgir as diferenças entre uma simples árvore binária de busca e uma árvore AVL.

Para remover um nó, temos que percorrer um conjunto de nós da árvore até chegar ao nó que será removido. Uma vez removido, devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós e, se necessário, aplicar uma das quatro rotações para restabelecer o balanceamento da árvore.



Para balancear a árvore após a remoção de um nó, valem as mesmas regras da inserção, com uma diferença: **remover** um nó da subárvore da **direita** equivale a **inserir** um nó na subárvore da **esquerda**.

A Figura 11.59 mostra um exemplo em que a remoção em uma árvore AVL produz exatamente o mesmo tipo de rotação que a inserção em outra árvore.



### 1.7.5 Removendo um nó da árvore

Remover um novo nó de uma árvore AVL é uma tarefa um pouco complicada, se comparada com a inserção.

**Ícone de informação** Isso ocorre porque precisamos procurar o nó a ser removido da árvore, o qual pode ser um **nó folha** ou um **nó interno** (que pode ser a **raiz**), com um ou dois filhos. Se este for um **nó interno**, é preciso reorganizar a árvore para que ela continue sendo uma árvore binária de busca. Terminada a remoção, é preciso também corrigir o balanceamento da árvore.

Podemos descrever o processo de busca e remoção de um nó com o seguinte conjunto de passos:

- Primeiro, compare o **valor** a ser removido com a **raiz**.
- Se o **valor** for **menor** do que a **raiz**: vá para a subárvore da **esquerda**.

A Figura 11.60 mostra o código utilizado para procurar e remover um nó de uma árvore AVL, `remove_ArvAVL`. Note que o algoritmo faz uso de outra função: `procuraMenor` (linha 35), cuja implementação é mostrada na Figura 11.61. Esta função é usada para procurar o **nó mais à esquerda** a partir de certo nó da árvore, **atual**.

Primeiramente, a função de remoção (Figura 11.60) verifica se o conteúdo do ponteiro `ArvAVL* raiz` é igual a `NULL` (linha 2). A condição é verdadeira em dois casos:

- Era uma árvore vazia.
- Ao descer, recursivamente, na árvore, o nó de onde viemos era um nó folha.

Seja qual for o caso, se o conteúdo da raiz for igual a `NULL`, retornamos o valor **ZERO** para indicar que o valor procurado não se encontra na árvore (linha 3).

Vamos imaginar que, inicialmente, nossa árvore não esteja vazia. Neste caso, temos que percorrer a árvore até achar o nó que será removido. Devemos, então, considerar três casos:

- Se o valor do nó **raiz** for **maior** do que o valor a ser removido (**valor**), devemos ir, recursivamente, para a subárvore **esquerda** do nó **raiz** (linhas 5-6).
- Se o valor do nó **raiz** for **menor** do que o valor a ser removido (**valor**), devemos ir, recursivamente, para a subárvore **direita** do nó **raiz** (linhas 15-16).
- Se o valor do nó **raiz** for **igual** ao valor a ser removido (**valor**), encontramos o nó a ser removido (linha 25).

Esse conjunto de passos permite caminhar na árvore em busca do nó a ser removido. Falta agora tratar da remoção e rebalancear a árvore. A remoção do nó é tratada nas linhas 25-48. Primeiro, verificamos se o nó a ser removido (**raiz**) é um nó folha (sem filhos) ou se ele possui apenas um dos filhos (linha 26).

- Se esta afirmação for **verdadeira**, a função irá copiar a **raiz** para um nó auxiliar (linha 28) e o nó filho (se for nó folha, irá copiar a constante `NULL`) para o lugar do nó pai (linhas 29-32). Em seguida, a função irá liberar a memória do nó auxiliar (linha 33).
- Se esta afirmação for **falsa**, significa que o nó **raiz** possui dois filhos. Neste caso, usamos a função `procuraMenor` para encontrar o **nó mais à esquerda** a partir do filho à direita do nó **raiz**, e copiamos a informação deste nó para o nó a ser removido, **raiz** (linhas 35-36). Em seguida, chamamos novamente a função `remove_ArvAVL`, desta vez para remover o nó retornado pela função `procuraMenor` (linha 37). Por fim, temos que tratar do balanceamento dessa remoção (linha 38), o qual poderá ser uma **rotação LL**, se a altura da árvore direita do filho esquerdo for menor ou igual a da árvore esquerda do filho esquerdo (linhas 39-40), ou uma **rotação LR**, no caso contrário (linhas 41-42).

Terminado o processo de remoção, atualizamos a altura do nó e retornamos o valor **UM**, indicando sucesso na operação de remoção (linhas 45-47). Falta agora tratar do balanceamento da árvore, à medida que voltarmos na recursão. Repare que as chamadas recursivas da função

Removendo um elemento da AVL

```

01 int remove_ArvAVL(ArvAVL *raiz, int valor){
02   if(*raiz == NULL)// valor não existe.
03     return 0;
04   int res;
05   if(valor < (*raiz)->info){
06     if((res = remove_ArvAVL(&(*raiz)->esq,valor)) == 1){
07       if(fatorBalanceamento_NO(*raiz) >= 2){
08         if(alt_NO((*raiz)->dir->esq) <=
09             alt_NO((*raiz)->dir->dir))
10           RotacaoRR(raiz);
11         else
12           RotacaoRL(raiz);
13       }
14     }
15   if((*raiz)->info < valor){
16     if((res = remove_ArvAVL(&(*raiz)->dir,valor)) == 1){
17       if(fatorBalanceamento_NO(*raiz) >= 2){
18         if(alt_NO((*raiz)->esq->dir) <=
19             alt_NO((*raiz)->esq->esq))
20           RotacaoLL(raiz);
21         else
22           RotacaoLR(raiz);
23       }
24     }
25   if((*raiz)->info == valor){
26     if(((*raiz)->esq == NULL || (*raiz)->dir == NULL)){
27       //nó tem 1 filho ou nenhum
28       struct NO *oldNode = (*raiz);
29       if((*raiz)->esq != NULL)
30         *raiz = (*raiz)->esq;
31       else
32         *raiz = (*raiz)->dir;
33       free(oldNode);
34     }else// nó tem 2 filhos
35     struct NO* temp = procuraMenor((*raiz)->dir);
36     (*raiz)->info = temp->info;
37     remove_ArvAVL(&(*raiz)->dir, (*raiz)->info);
38     if(fatorBalanceamento_NO(*raiz) >= 2){
39       if(alt_NO((*raiz)->esq->dir) <=
40           alt_NO((*raiz)->esq->esq))
41         RotacaoLL(raiz);
42       else
43         RotacaoLR(raiz);
44     }
45     if(*raiz != NULL)
46       (*raiz)->altura = maior(altura_NO((*raiz)->esq),
47                               altura_NO((*raiz)->dir))+1;
48   }
49   (*raiz)->altura=maior(altura_NO((*raiz)->esq),
50                           altura_NO((*raiz)->dir))+1;
51   return res;
52 }
  
```

FIGURA 11.60

de remoção são feitas dentro de um comando condicional (`if`) (linhas 6 e 16). Se a resposta da função, armazenada na variável `res`, for igual a `UM`, a remoção foi realizada com sucesso e devemos verificar o balanceamento do nó (linhas 7 e 17). Devemos então considerar as quatro possíveis rotações, se o nó estiver desbalanceado:

- Se a chamada recursiva da função foi feita para a subárvore esquerda do nó raiz, devemos escolher entre a **rotação RR** e a **rotação RL** (linha 6):
  - Se a altura da árvore esquerda do filho direito for menor ou igual a da árvore direita do filho direito: **rotação RR** (linhas 8-9).
  - Caso contrário: **rotação RL** (linhas 10-11).
- Se a chamada recursiva da função foi feita para a subárvore direita do nó raiz, devemos escolher entre a **rotação LL** e a **rotação LR** (linha 16):
  - Se a altura da árvore direita do filho esquerdo for menor ou igual a da árvore esquerda do filho esquerdo: **rotação LL** (linhas 18-19).
  - Caso contrário: **rotação LR** (linhas 20-21).

Perceba que esse conjunto de passos responsável pela rotação é feito sempre que voltamos a uma chamada recursiva. Por fim, a função de remoção atualiza a altura do nó **raiz** e retorna o valor `res`, que é o valor retornado pelas chamadas recursivas da função de inserção (linhas 50-51). O processo de inserção é mais bem ilustrado pela Figura 11.62.

#### Procurando o menor elemento da AVL

```

01 struct NO* procuraMenor(struct NO* atual){
02     struct NO *no1 = atual, *no2 = atual->esq;
03     while(no2 != NULL) {
04         no1 = no2;
05         no2 = no2->esq;
06     }
07     return no1;
08 }
```

FIGURA 11.61

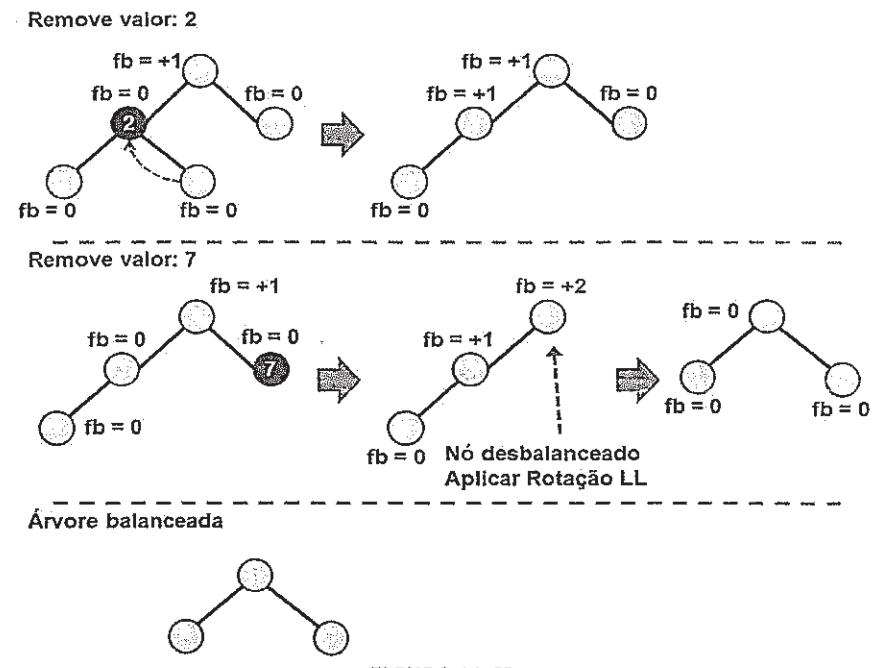


FIGURA 11.62

## 1.8 ÁRVORE RUBRO-NEGRA

### 1.8.1 Definição

Como a árvore AVL, a árvore rubro-negra (conhecida como vermelho-preto ou red-black) também é um tipo de árvore binária balanceada. Porém, diferente da árvore AVL, que usa a altura das suas subárvores, a árvore rubro-negra utiliza um esquema de coloração dos nós para controlar o balanceamento da árvore. Ela foi originalmente criada por Rudolf Bayer, em 1972, usando árvores binárias simétricas. Posteriormente, em um trabalho de Leonidas J. Guibas e Robert Sedgewick, de 1978, adquiriu o seu nome atual.

A árvore rubro-negra possui esse nome pois cada nó possui um atributo de cor (além dos dois ponteiros para seus filhos), que pode ser o **vermelho** ou o **preto**. Além disso, a árvore deve satisfazer o seguinte conjunto de propriedades:

- Todo nó da árvore é **vermelho** ou **preto**.
- A raiz é sempre **preta**.
- Todo nó folha (`NULL`) é **preto**.
- Se um nó é **vermelho**, então os seus filhos são **pretos** (ou seja, não existem nós **vermelhos** consecutivos).
- Para cada nó, todos os caminhos deste nó para os nós folhas descendentes contêm o mesmo número de nós **pretos**.

A Figura 11.63 apresenta um exemplo de árvore rubro-negra. A terceira propriedade diz que todos os nós **NULL** (representados por pequenos quadrados pretos) devem ser pretos. Como todo nó folha termina com dois ponteiros para **NULL**, eles podem ser ignorados na representação da árvore, para fins de didática.

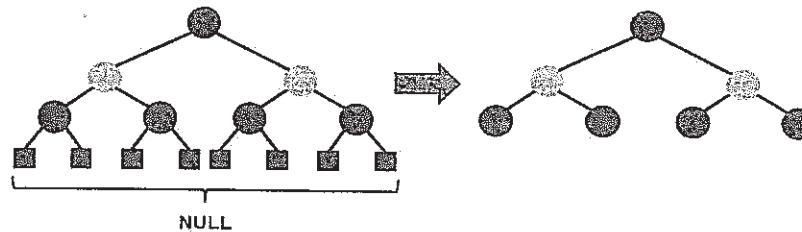


FIGURA 11.63



Para manter o balanceamento, a árvore rubro-negra faz uso de **rotações** e **ajuste de cores** na etapa de rebalanceamento, que ocorre a cada **inserção ou remoção**.

Por meio dessas operações, a árvore rubro-negra busca manter-se como uma árvore binária quase completa. Assim, o custo máximo de qualquer algoritmo é  $O(\log N)$ .



O objetivo das operações de **rotação** e de **ajuste de cores** é garantir que suas propriedades não sejam violadas durante a inserção ou remoção de um nó, restituindo o equilíbrio da árvore.

Assim, caso alguma das propriedades que definem a árvore rubro-negra não seja satisfeita, são realizadas rotações e/ou ajustes de cores, de forma que a árvore permaneça balanceada.

### 11.8.2 Diferença entre as árvores AVL e rubro-negra

Apesar de ambas (AVL e rubro-negra) serem árvores平衡adas de busca, existem algumas diferenças entre elas que devem ser consideradas antes de sua utilização.



Na teoria, as duas árvores possuem a mesma complexidade computacional em suas operações de inserção, remoção e busca:  $O(\log N)$ . Na prática, a árvore AVL é mais rápida na operação de busca, e mais lenta nas operações de inserção e remoção.

Isso se deve ao fato da árvore AVL ser mais balanceada do que as árvores rubro-negras, o que acelera a operação de busca. No entanto, possuir um balanceamento mais rígido exige um custo maior na operação de inserção e remoção: no pior caso, uma operação de remoção pode exigir  $O(\log N)$  rotações na árvore AVL, mas apenas três rotações na árvore rubro-negra.

Logo, se sua aplicação realiza de forma intensa a operação de busca, é melhor usar uma árvore AVL. Se a operação mais usada for a inserção ou a remoção, adote uma árvore rubro-negra.



Árvores rubro-negras são de uso mais geral do que árvores AVL.

A árvore rubro-negra trabalha melhor que a AVL nas operações de inserção e remoção. Isso é com que elas sejam utilizadas em diversas aplicações e bibliotecas de linguagens de programação, como:

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: `map`, `multimap`, `multiset`.
- Linux kernel: `completely fair scheduler`, `linux/rbtree.h`.

### 1.8.3 Árvore rubro-negra caída para a esquerda

Desenvolvida por Robert Sedgewick em 2008, a árvore rubro-negra caída para a esquerda (*left-leaning red-black tree*) é uma variante da árvore rubro-negra. Como a árvore original, ela garante a mesma complexidade de operações, mas possui um implementação mais simples na inserção e na remoção de nós.



Além de satisfazer todas as propriedades da árvore rubro-negra convencional, a árvore rubro-negra caída para a esquerda possui uma propriedade extra que deve ser respeitada: se um nó é **vermelho**, ele é o **filho esquerdo** do seu pai.

É essa propriedade extra da árvore que confere o seu aspecto de **caída para a esquerda**: os nós vermelhos sempre são filhos à esquerda, como mostra a Figura 11.64.

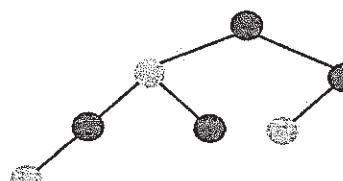


FIGURA 11.64

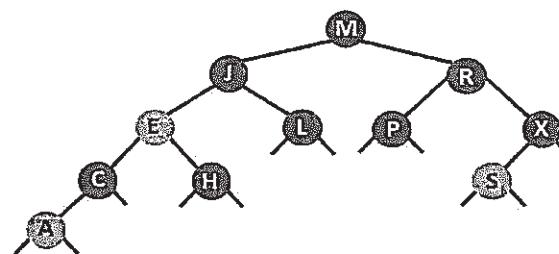


A implementação da árvore rubro-negra caída para a esquerda corresponde à implementação de uma árvore 2-3.

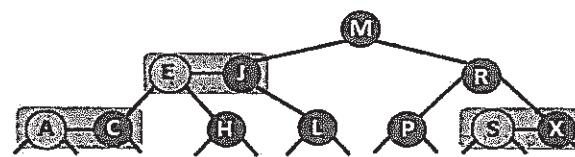
Diferente das árvores vistas até o momento, a árvore 2-3 não é uma árvore binária. Neste tipo de árvore, cada nó interno pode armazenar um ou dois valores e, dependendo da quantidade de valores armazenados, ter dois (um valor) ou três (dois valores) filhos. Seu funcionamento é o mesmo da árvore binária de busca. No caso de um nó com dois valores e três subárvores, os elementos da subárvore esquerda são sempre menores do que o primeiro valor, enquanto os elementos da subárvore direita são sempre maiores do que o segundo valor. Na subárvore do

meio, encontram-se os elementos que são maiores do que o primeiro, mas menores do que o segundo valor do nó pai.

No caso da árvore rubro-negra caída para a esquerda, esta corresponde a uma árvore 2-3 se considerarmos que o nó vermelho será sempre o valor menor de um nó contendo dois valores e três subárvore, como mostra a Figura 11.65. Assim, balancear a árvore rubro-negra equivale a manipular uma árvore 2-3, uma tarefa muito mais simples do que manipular uma árvore AVL ou uma rubro-negra convencional.



ÁRVORE RUBRO-NEGRA CAÍDA PARA A ESQUERDA



ÁRVORE 2-3

FIGURA 11.65

#### 11.8.4 Implementando uma árvore rubro-negra

Como a árvore AVL, a implementação da TAD de uma árvore rubro-negra é idêntica à da árvore binária mostrada na Seção 11.4.3. Ou seja, aqui também iremos utilizar um **ponteiro para ponteiro** para guardar o primeiro nó da árvore. Como visto anteriormente, o uso de um **ponteiro para ponteiro** permite mudar com mais facilidade quem é a **raiz** no caso de alguma operação na árvore (como rotação) alterar o nó da raiz.

Como na árvore binária, o arquivo **ArvoreLLRB.h**, ilustrado na Figura 11.66, define:

- Para fins de padronização, um novo nome para o **ponteiro do tipo árvore** (linha 1). Esse é o tipo que será usado sempre que se desejar trabalhar com uma árvore rubro-negra.
- As funções disponíveis para se trabalhar com essa árvore e que serão implementadas no arquivo **ArvoreLLRB.c** (linhas 3-13).

#### Arquivo ArvoreLLRB.h

```

01 typedef struct NO* ArvLLRB;
02
03 ArvLLRB* cria_ArvLLRB();
04 void libera_ArvLLRB(ArvLLRB* raiz);
05 int insere_ArvLLRB(ArvLLRB* raiz, int valor);
06 int remove_ArvLLRB(ArvLLRB* raiz, int valor);
07 int estaVazia_ArvLLRB(ArvLLRB *raiz);
08 int totalNO_ArvLLRB(ArvLLRB *raiz);
09 int altura_ArvLLRB(ArvLLRB *raiz);
10 int consulta_ArvLLRB(ArvLLRB *raiz, int valor);
11 void preOrdem_ArvLLRB(ArvLLRB *raiz);
12 void emOrdem_ArvLLRB(ArvLLRB *raiz);
13 void posOrdem_ArvLLRB(ArvLLRB *raiz);

```

#### Arquivo ArvoreLLRB.c

```

01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ArvoreLLRB.h" //inclui os Protótipos
04
05 #define RED 1
06 #define BLACK 0
07
08 struct NO{
09     int info;
10     struct NO *esq;
11     struct NO *dir;
12     int cor;
13 };

```

FIGURA 11.66

Já o arquivo **ArvoreLLRB.c** (Figura 11.66) contém apenas:

- As chamadas às bibliotecas necessárias à implementação da árvore rubro-negra (linhas 1-3).
- A definição de duas constantes para representar as cores dos nós da árvore (linhas 5-6).
- A definição do tipo que descreve cada nó da árvore rubro-negra, **struct NO** (linhas 8-13).
- As implementações das funções definidas no arquivo **ArvoreLLRB.h**. As implementações dessas funções serão vistas nas seções seguintes.

Perceba que, assim como na árvore AVL, nossa estrutura **NO** na árvore rubro-negra possui um campo a mais do que na árvore binária. Este campo, **cor**, será utilizado para armazenar a cor daquele nó da árvore. A informação de cor é utilizada durante o平衡amento da árvore.



Com respeito à implementação das funções, com exceção de **inserção** e **remoção**, as demais funções da árvore rubro-negra são implementadas de forma idêntica às da árvore binária.

Por fim, para utilizarmos nossa árvore rubro-negra no arquivo **main()**, basta declarar um ponteiro para ele (definido no arquivo **ArvoreLLRB.h**) da seguinte forma:

```
ArvLLRB *raiz;
```

### 11.8.5 Acessando e mudando a cor dos nós

Cada nó da árvore rubro-negra tem associado a ele uma cor: ou **vermelha** ou **preta**. Essa cor é utilizada para controlar o balanceamento da árvore. Além disso, todo ponteiro **NULL** é considerado de cor **preta**. Assim, é interessante possuir uma função capaz de retornar a cor do nó, como mostra a Figura 11.67.

#### Acessando a cor de um nó

```
01 int cor(struct NO* H) {
02     if(H == NULL)
03         return BLACK;
04     else
05         return H->cor;
06 }
```

FIGURA 11.67

Durante o balanceamento da árvore, pode ser necessário mudar a cor de um nó e de seus filhos de vermelho para preto, ou vice-versa. Isso ocorre quando, por exemplo, um nó possui dois filhos vermelhos. Neste caso, temos uma violação de uma das propriedades da árvore que precisa ser corrigida. A Figura 11.68 mostra a função responsável por fazer a mudança nas cores de um nó da árvore e de seus filhos. Basicamente, ela inverte a cor do nó pai com uma operação de negação e verifica se cada um de seus filhos (o da esquerda e o da direita) existe. Se algum nó filho existir, a função também inverte a sua cor. Perceba que é uma operação “administrativa”, já que não altera a estrutura ou o conteúdo da árvore. A Figura 11.69 mostra um exemplo de mudança de cores dos nós.



Uma operação de mudança de cor não altera o número de nós pretos da raiz até os nós folhas. No entanto, pode introduzir dois nós consecutivos vermelhos na árvore, o que deve ser corrigido com outras operações.

#### Mudando a cor de um nó e de seus filhos

```
01 void trocaCor(struct NO* H) {
02     H->cor = !H->cor;
03     if(H->esq != NULL)
04         H->esq->cor = !H->esq->cor;
05     if(H->dir != NULL)
06         H->dir->cor = !H->dir->cor;
07 }
```

FIGURA 11.68

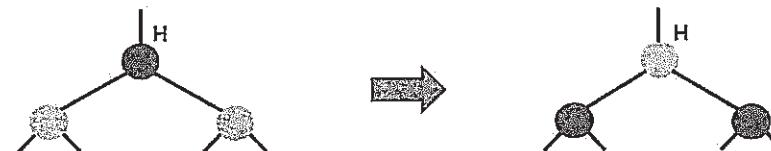


FIGURA 11.69

### 8.6 Rotações

Ao estudarmos a árvore AVL, vimos que ela utiliza quatro funções de rotação para rebalancear a árvore. A árvore rubro-negra possui apenas duas funções de rotação.



As operações de rotação da árvore rubro-negra são mais simples de implementar e de depurar em comparação com as rotações da árvore AVL.

A rotação é uma operação que, dado um conjunto de três nós, visa deslocar um nó vermelho e esteja à esquerda para a direita, ou vice-versa. Dependendo do caso, podemos usar uma rotação à esquerda ou uma rotação à direita.



As operações de rotação apenas atualizam ponteiros, de modo que a sua complexidade é O(1).

A Figura 11.70 mostra a implementação da função que rotaciona à esquerda. A função recebe como parâmetro um nó **A**, que possui um nó **B** como filho direito (linha 2). Basicamente, a função movimenta o nó **B** para o lugar do nó **A**, de modo que o nó **A** se torne o nó esquerdo do nó **B** (linhas 3-4). Junto com essa mudança de posições, o nó **B** passa a ter a cor do nó **A**, enquanto o nó **A** passa a ter a cor **vermelha** (linhas 5-6). A Figura 11.71 mostra um exemplo de rotação à esquerda.

**Rotação à esquerda**

```

01 struct NO* rotacionaEsquerda(struct NO* A) {
02     struct NO* B = A->dir;
03     A->dir = B->esq;
04     B->esq = A;
05     B->cor = A->cor;
06     A->cor = RED;
07     return B;
08 }
```

FIGURA 11.70

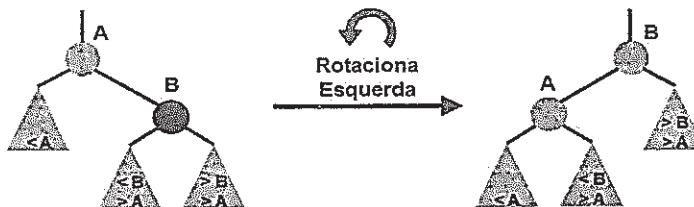


FIGURA 11.71

A função que rotaciona à direita é mostrada na Figura 11.72. Como podemos notar, ela se parece muito com a de rotação à esquerda. A função recebe como parâmetro um nó A, que possui um nó B como filho **esquerdo** (linha 3). Basicamente, a função movimenta o nó B para o lugar do nó A, de modo que o nó A se torne o filho direito do nó B (linhas 3-4). Junto com essa mudança de posições, o nó B passa a ter a cor do nó A, enquanto o nó A passa a ter a cor vermelha (linhas 5-6). A Figura 11.73 mostra um exemplo de rotação à direita.

**Rotação à direita**

```

01 struct NO* rotacionaDireita(struct NO* A) {
02     struct NO* B = A->esq;
03     A->esq = B->dir;
04     B->dir = A;
05     B->cor = A->cor;
06     A->cor = RED;
07     return B;
08 }
```

FIGURA 11.72

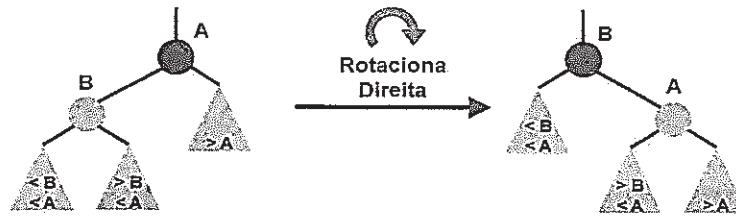


FIGURA 11.73

**1.8.7 Movendo os nós vermelhos**

Algumas operações da árvore rubro-negra, quando utilizadas, podem causar uma violação s propriedades da árvore. Por exemplo, a função `trocaCor()` pode introduzir sucessivos nós vermelhos à direita, o que viola uma das propriedades da árvore. Para resolver esse problema, a árvore rubro-negra possui outras funções (além das funções de rotação) que ajudam a restabelecer o balanceamento da árvore e garantir que as suas propriedades são respeitadas.

De modo geral, dado um conjunto de três nós, essas funções têm como objetivo movimentar um nó vermelho para a subárvore esquerda ou direita, dependendo da situação na qual o conjunto de nós se encontra.

A Figura 11.74 mostra a implementação da função que movimenta um nó vermelho para a esquerda. Basicamente, a função recebe como parâmetro um nó H (representado pelo ponteiro ). Ela troca as cores dele e de seus filhos (linha 2). Em seguida, verifica se o filho à esquerda do nó direito de H, B, é vermelho (linha 3). Em caso afirmativo, a função aplica uma rotação direita no nó C (filho direito de A), e uma rotação à esquerda no nó A (linhas 4-5). Desse modo, o nó B se torna o pai dos nós A e C, sendo o nó A o seu filho à esquerda. Por fim, ela troca as cores do nó B e de seus filhos (linha 6), e retorna o nó B como o nó ocupando o lugar A. Esse processo é mais bem ilustrado pela Figura 11.75.

**Movendo um nó vermelho para a esquerda**

```

01 struct NO* move2EsqRED(struct NO* H) {
02     trocaCor(H);
03     if(cor(H->dir->esq) == RED) {
04         H->dir = rotacionaDireita(H->dir);
05         H = rotacionaEsquerda(H);
06         trocaCor(H);
07     }
08     return H;
09 }
```

FIGURA 11.74

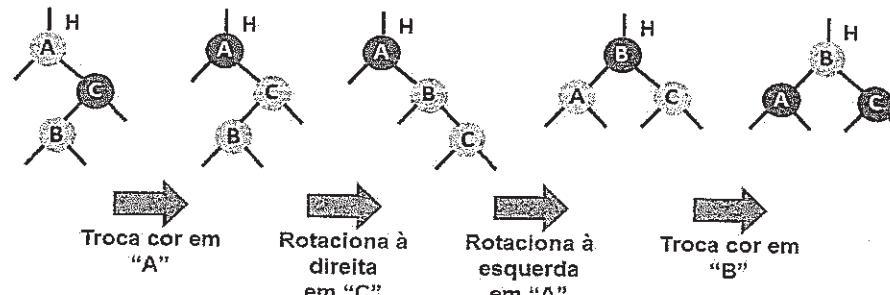


FIGURA 11.75

A função que movimenta um nó vermelho para a direita é mostrada na Figura 11.76. É possível notar que ela é similar, mas mais simples do que a que movimenta para a esquerda. Basicamente, a função recebe como parâmetro um nó C (representado pelo ponteiro H) e troca as cores dele e de seus filhos (linha 2). Em seguida, verifica se o filho à esquerda do filho esquerdo de C, A, é vermelho (linha 3). Em caso afirmativo, a função aplica uma rotação à direita no nó C (linha 4). Desse modo, o nó B se torna o pai dos nós A e C, sendo o nó C o seu filho à direita. Por fim, ela troca as cores do nó B e de seus filhos (linha 5), e retorna o nó B como o nó ocupando o lugar de C. Esse processo é mais bem ilustrado pela Figura 11.77.

```
01 struct NO* move2DirRED(struct NO* H) {
02     trocaCor(H);
03     if(cor(H->esq->esq) == RED) {
04         H = rotacionaDireita(H);
05         trocaCor(H);
06     }
07     return H;
08 }
```

FIGURA 11.76

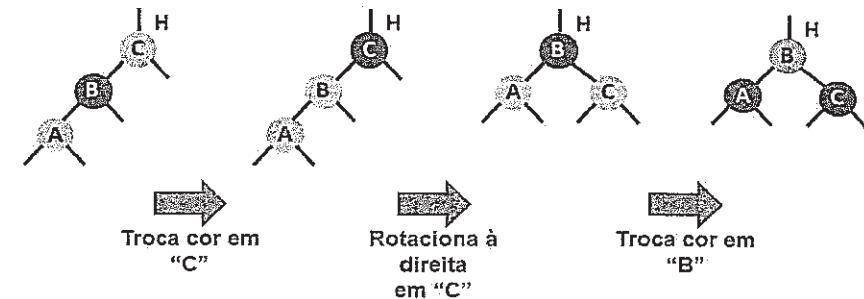


FIGURA 11.77

Com isso, temos uma função que verifica, em sequência, se alguma das propriedades da árvore rubro-negra foi violada e as corrige. A implementação desta função é mostrada na Figura 11.78. Assim, dado um nó H, a função verifica e corrige as seguinte violações:

- O filho direito é da cor vermelha: fazer uma **rotação à esquerda** (linhas 3-4).
- O filho direito e o neto da esquerda são vermelhos: fazer uma **rotação à direita** (linhas 7-8).
- Ambos os filhos são vermelhos: **trocar a cor** do pai e dos filhos (11-12).

```
01 struct NO* balancear(struct NO* H) {
02     //nó vermelho é sempre filho à esquerda
03     if(cor(H->dir) == RED)
04         H = rotacionaEsquerda(H);
05
06     //Filho da direita e neto da esquerda são vermelhos
07     if(H->esq != NULL && cor(H->dir) == RED &&
08         cor(H->esq->esq) == RED)
09         H = rotacionaDireita(H);
10
11     //2 filhos vermelhos: troca cor!
12     if(cor(H->esq) == RED && cor(H->dir) == RED)
13         trocaCor(H);
14
15 }
```

FIGURA 11.78

A Figura 11.79 mostra as situações e o efeito produzido por cada uma dessas correções nos nós da árvore.

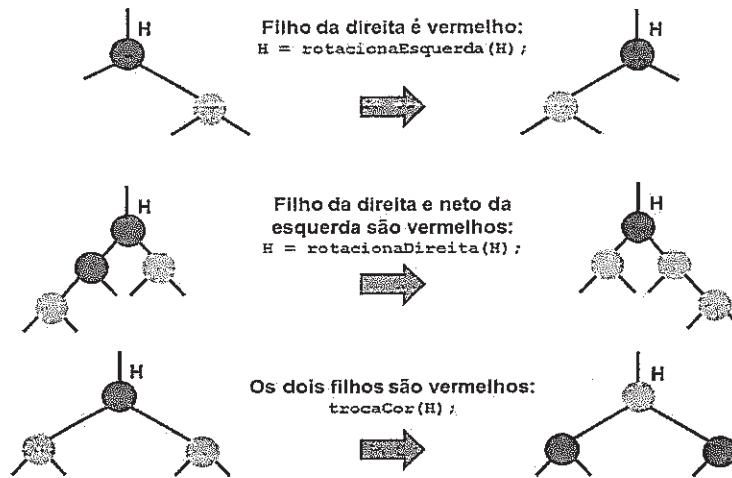


FIGURA 11.79

### 11.8.8 Inserindo um nó na árvore

Inserir um novo nó em uma árvore rubro-negra é uma tarefa similar à inserção na árvore AVL. Primeiramente, temos que percorrer um conjunto de nós da árvore até chegar ao nó folha que irá se tornar o pai do novo nó, alocar memória para este nó e copiar os dados inseridos para dentro dele. Uma vez inserido o nó, devemos voltar pelo caminho percorrido, verificar se ocorreu a violação de alguma das propriedades da árvore e, se necessário, aplicar uma das rotações ou mudança de cores para restabelecer o balanceamento da árvore.

A Figura 11.80 mostra a implementação da função de inserção. Note que esse algoritmo utiliza duas funções:

- `insereNO` (linhas 1-36), responsável por percorrer a árvore procurando o ponto de inserção do novo nó e seu posterior rebalanceamento.
- `insere_ArvLLRB` (linhas 37-44), responsável por fazer a interface com o usuário e corrigir a raiz da árvore.

A função `insere_ArvLLRB` (linhas 37-44) recebe como parâmetros a raiz da árvore onde será feita a inserção e o valor a ser inserido. Ela é responsável por chamar a função `insereNO`, que irá inserir e rebalancear a árvore após a inserção (linha 39). Note que a função `insereNO` irá retornar a nova raiz dessa árvore. Essa função também recebe um parâmetro por

```

Inserindo um elemento na árvore rubro-negra
01 struct NO* insereNO(struct NO* H, int valor, int *resp) {
02     if(H == NULL){
03         struct NO *novo
04         novo = (struct NO*)malloc(sizeof(struct NO));
05         if(novo == NULL){
06             *resp = 0;
07             return NULL;
08         }
09         novo->info = valor;
10         novo->cor = RED;
11         novo->dir = NULL;
12         novo->esq = NULL;
13         *resp = 1;
14         return novo;
15     }
16
17     if(valor == H->info)
18         *resp = 0;// Valor duplicado
19     else{
20         if(valor < H->info)
21             H->esq = insereNO(H->esq, valor, resp);
22         else
23             H->dir = insereNO(H->dir, valor, resp);
24     }
25
26     if(cor(H->dir) == RED && cor(H->esq) == BLACK)
27         H = rotacionaEsquerda(H);
28
29     if(cor(H->esq) == RED && cor(H->esq->esq) == RED)
30         H = rotacionaDireita(H);
31
32     if(cor(H->esq) == RED && cor(H->dir) == RED)
33         trocaCor(H);
34
35     return H;
36 }
37 int insere_ArvLLRB(ArvLLRB* raiz, int valor){
38     int resp;
39     *raiz = insereNO(*raiz, valor, &resp);
40     if((*raiz) != NULL)
41         (*raiz)->cor = BLACK;
42
43     return resp;
44 }
```

FIGURA 11.80

referência (`resp`), que permitirá saber se a inserção foi realizada com sucesso ou não. Após o retorno da função `insereNO`, verificamos se a raiz da árvore é diferente de `NULL`. Se esta afirmação for verdadeira, definimos a cor da raiz como `preta` (linhas 40-41). Mesmo que a inserção falhe, a cor da raiz sempre será `preta`. Por fim, retornamos o valor de `resp` com a resposta do processo de inserção.

Também devemos ter em mente que a corrigão de uma violação das propriedades é avante do novo nó na árvore. Ela recebe como parâmetros o ponteiro para um nó da árvore (**struct *No*\* *H***), o valor a ser inserido e o ponteiro para um nó da árvore (**struct *No*\* *H***). Note que, inicialmente, o ponteiro **H** possui o conteúdo da raiz da árvore.

Para inserir o novo nó na árvore, devemos alterar o conteúdo da estrutura **No** que o ponteiro aponta para o nó da árvore. Isso pode gerar uma nova violação de suas propriedades. Assim, para corrigir essas duas propriedades, uma sequência de três passos deve ser executada:

- Se o filho da direita é vermelho e o filho da esquerda é vermelho
- Se ambos os filhos são vermelhos, temos uma propriedade violada (dois nós vermelhos em sequência).
- Se o operador de inserção não é igual ao valor de memória para inserir o novo nó (linhas 3-4). Caso a alocação de memória não seja possível, a função irá retornar o valor **NULL** e colocar o valor **ZERO** no parâmetro resp, indicando falta na operação de inserção (linhas 5-8). Tendo a função malloco() retornado um endereço de memória válido, podemos copiar os dados que retornamos o valor **UM** no parâmetro resp para indicar sucesso da função (linhas 10-12). Por fim, a sequência para dentro desse nó (linha 9). Repare também que o nó não possui filhos mas armazena para dentro desse nó (linhas 21 e 23). O processo de inserção é mais bem ilustrado pelas Figuras 11.82 e 11.83.

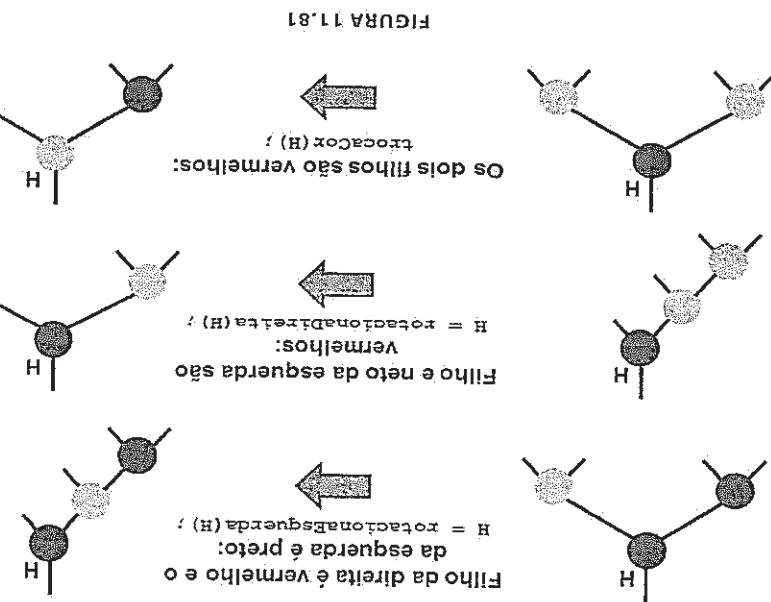


FIGURA 11.81

A Figura 11.81 mostra as situações e o efeito produzido por cada uma dessas correções nos nós da árvore. Por fim, a função de inserção retorna o nº **H** (linha 35) como sendo um dos filhos do nó pai que fez a respectiva chamada recursiva (linhas 21 e 23). O processo de inserção é mais bem ilustrado pelas Figuras 11.82 e 11.83.

- Se um nº é vermelho, ele é o filho esquerdo do seu pai.
- Se um nº é vermelho, os seus filhos são pretos (ou seja, não existem nós vermelhos consecutivos).

Essa combinação de passos (linhas 17-24) permite caminhar na árvore em busca do ponto a inserção, para a subárvore dirigida do nº **H** (linhas 22-23).

- Se o valor do nº **H** for menor do que o valor de inserido (valor), devemos ir, recurrivamente, para a subárvore esquerda do nº **atual** (linhas 20-21).
- Se o valor do nº **H** for maior do que o valor de inserido (valor), devemos ir, recurrivamente, para a subárvore direita do nº **H** (linhas 22-23).
- Se a inserção que houve numa folha na inserção (valor) indica que existe na árvore (resp), o que é a inserção Preenchendo com **ZERO** o parâmetro de resposta (resp), o que indica que houve uma folha na inserção (valor), o valor já existe na árvore.

Agora, vamos imaginar que nossa árvore não está vazia. Nesse caso, temos que precorrer recursivamente até achar o nº folha em que o novo nº será inserido (linhas 17-24). Isso é feito, recursivamente, considerando três casos:

• Se o valor de memória para inserir o nº folha em que o novo nº folha é igual ao nº folha em que o novo nº folha é igual ao nº folha inserido (linhas 1-4). Tendo a árvore atingido esse nível, podemos imprimi-la. Nesse caso, temos que precorrer recursivamente até achar o nº folha em que o novo nº folha é igual ao nº folha inserido (linhas 13-14).

• Se o valor de memória para inserir o nº folha em que o novo nº folha é igual ao nº folha inserido nem à direita e que sua cor é inicialmente vermelha (linhas 9-12). Por fim, a sequência para dentro desse nó (linha 9). Repare também que o nó não possui filhos mas armazena para dentro desse nó (linhas 21 e 23). O processo de inserção é mais bem ilustrado pelas Figuras 11.82 e 11.83.

• Se o valor de memória para inserir o nº folha em que o novo nº folha é igual ao nº folha inserido nem à esquerda e que sua cor é inicialmente preta (linhas 10-12). Por fim, a sequência para dentro desse nó (linha 10). Repare também que o nó não possui filhos mas armazena para dentro desse nó (linhas 21 e 23). O processo de inserção é mais bem ilustrado pelas Figuras 11.82 e 11.83.

- Bra uma árvore vazia.
- Ao descer, recursivamente, na árvore, o nº de onde viemos era um nº folha.

A condição é verdadeira em dois casos:

• Primeiro, a função verifica se o conteúdo ponteiro **No\* H** é igual a **NULL** (linha 2).

• Segundo, se o conteúdo ponteiro **No\* H** possuir o conteúdo da raiz da árvore.

A função de inserção (**resp**) Note que, inicialmente, o ponteiro **H** possui o conteúdo da raiz da árvore. Isso pode gerar uma nova violação de suas propriedades. Assim, para corrigir essas duas propriedades, uma sequência de três passos deve ser executada:

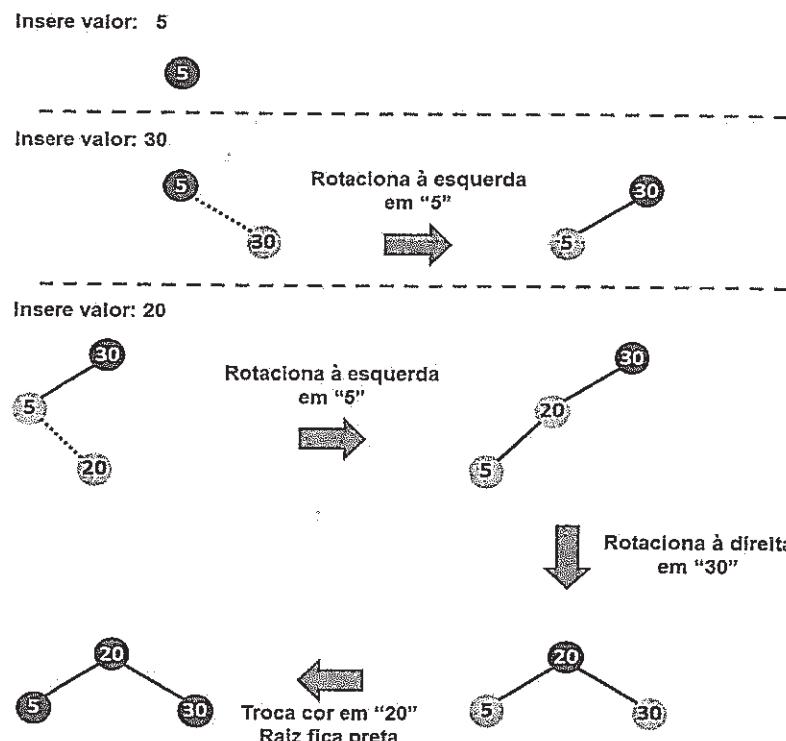


FIGURA 11.82

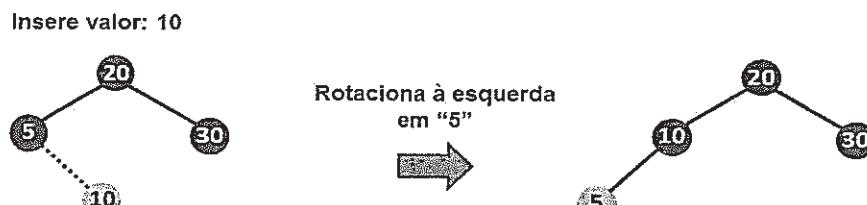


FIGURA 11.83

### 11.8.9 Removendo um nó da árvore

Remover um novo nó de uma árvore rubro-negra é uma tarefa similar à remoção na árvore AVL. Primeiro, temos que percorrer um conjunto de nós da árvore até chegar ao nó que será removido (se ele existir), o qual pode ser um nó folha ou um nó **interno** (que pode ser a raiz) com um ou dois filhos. Se este for um nó interno, é preciso reorganizar a árvore para que ela continue sendo uma árvore binária de busca. Além disso, precisamos verificar se a árvore é vazia (neste caso, a remoção não é possível) e se a remoção do nó não gera uma árvore vazia.

Diferentemente da árvore AVL, a remoção na árvore rubro-negra corrige o平衡amento da árvore tanto na ida quanto na volta da recursão. Isso significa que o próprio processo de busca pelo nó a ser removido já prevê possíveis violações das propriedades da árvore. Desse modo, somente devemos executar a remoção se o nó a ser removido realmente existe na árvore.

A Figura 11.84 mostra a implementação da função de remoção. Note que esse algoritmo utiliza duas funções:

- **remove\_NO** (linhas 1-27), responsável por procurar o nó a ser removido e rebalancear a árvore.
- **remove\_ArvLLRB** (linhas 28-37), responsável por fazer a interface com o usuário, verificar se o nó a ser removido existe e corrigir a raiz da árvore.

A função **remove\_ArvLLRB** (linhas 28-37) recebe como parâmetros a raiz da árvore onde será feita a remoção e o valor a ser removido. Ela é responsável por chamar a função **remove\_NO**, que irá remover e rebalancear a árvore (linha 31).

Primeiramente, precisamos verificar se o nó **valor** existe na árvore. O teste é feito com a função **consulta\_ArvLLRB**, que nada mais é do que uma função que percorre a árvore para saber se o nó **valor** existe na árvore (linha 29). Se o nó não existir, retornamos o valor **ZERO** para indicar falha no processo de remoção (linha 36). Caso o nó exista, executamos a função **remove\_NO**. Note que essa função irá retornar a nova raiz da árvore depois da remoção (linhas 30-31). Após o retorno da função **remove\_NO**, verificamos se a raiz da árvore é diferente de **NONE**. Se esta afirmação for verdadeira, definimos a cor da raiz como **preta** (linhas 32-33). Por fim, retornamos o valor **UM** para indicar sucesso na operação de remoção (linha 34).

A função **remove\_NO** (linhas 1-27) é uma função recursiva responsável por realmente fazer a remoção do nó da árvore. Ela recebe como parâmetros o ponteiro para um nó da árvore (**struct NO\* H**) e o **valor** a ser removido. Note que, inicialmente, o ponteiro **H** possui o conteúdo da **raiz** da árvore (linhas 30-31).

Primeiro, a função verifica se o conteúdo do ponteiro **struct NO\* H** é maior do que o **valor** a ser removido (linha 2). Se esta afirmação for verdadeira, a função irá verificar se o nó

### Removendo um elemento da árvore rubro-negra

```

01 struct NO* remove_NO(struct NO* H, int valor){
02     if(valor < H->info){
03         if(cor(H->esq) == BLACK &&
04             cor(H->esq->esq) == BLACK)
05             H = move2EsqRED(H);
06
07         H->esq = remove_NO(H->esq, valor);
08     }else{
09         if(cor(H->esq) == RED)
10             H = rotacionaDireita(H);
11
12         if(valor == H->info && (H->dir == NULL)){
13             free(H);
14             return NULL;
15         }
16
17         if(cor(H->dir) == BLACK &&
18             cor(H->dir->esq) == BLACK)
19             H = move2DirRED(H);
20
21         if(valor == H->info){
22             struct NO* x = procuraMenor(H->dir);
23             H->info = x->info;
24             H->dir = removerMenor(H->dir);
25         }else
26             H->dir = remove_NO(H->dir, valor);
27     }
28     return balancear(H);
29 }
30 int remove_ArvLLRB(ArvLLRB *raiz, int valor){
31     if(consulta_ArvLLRB(raiz, valor)){
32         struct NO* h = *raiz;
33         *raiz = remove_NO(h, valor);
34         if(*raiz != NULL)
35             (*raiz)->cor = BLACK;
36         return 1;
37     }else
38         return 0;
39 }
```

FIGURA 11.84

possui filho e neto da cor **preta à esquerda** e, se necessário, irá chamar a função que move nó **vermelho para a esquerda** (linhas 3-4). Em seguida, a função irá descer, recursivamente, para o nó da esquerda (linha 6).

No caso do conteúdo do ponteiro **struct NO\* H** não ser maior do que o **valor** a ser removido (linha 2), uma série de ajustes devem ser feitos para preparar a árvore para a remoção:

- Se o nó da esquerda for **vermelho**, devemos aplicar uma **rotação à direita** (linhas 8-9).
- Se o nó a ser removido não possuir filho à direita (nó folha), ele deverá ser removido e a recursão termina (linhas 11-14).
- Se o filho da direita é **preto** e o filho à esquerda do filho direito também é **preto**, devemos chamar a função que move nó vermelho para a direita (linhas 16-17).

Terminados os ajustes, temos que verificar se o conteúdo do ponteiro **H** é **igual** ao **valor** a ser removido. Caso não seja, a função irá descer, recursivamente, para o nó da direita (linha 24). No entanto, se este for o nó a ser removido, iremos usar a função **procuraMenor** para encontrar o **nó mais à esquerda** a partir do filho à direita do nó **H** (linha 20). Em seguida, copiamos informação deste nó para o nó a ser removido, **H**, e chamamos a função **removeMenor** para mover o nó retornado pela função **procuraMenor** (linhas 21-22).

A implementação das funções **procuraMenor** e **removeMenor** são apresentadas na Figura 11.85. Basicamente, a função **procuraMenor** permite encontrar o **nó mais à esquerda** de determinado nó. Já a função **removeMenor** remove o **nó mais à esquerda** de determinado nó, usando a árvore nesse processo.

Por fim, a função de remoção executa um balanceamento sempre que volta da recursão (linha 26). Isso é feito com a função **balancear**, mostrada na Figura 11.78. O processo de rotação é mais bem ilustrado pela Figura 11.86.

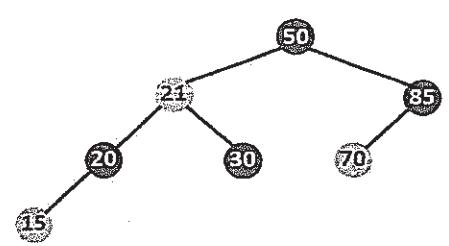
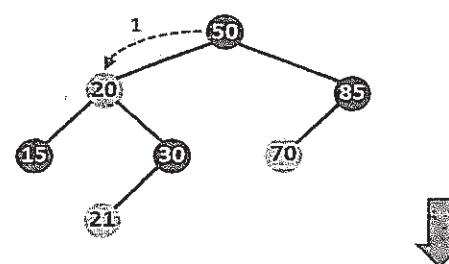
### Procurando e removendo o menor elemento da rubro-negra

```

01 struct NO* removerMenor(struct NO* H) {
02     if(H->esq == NULL){
03         free(H);
04         return NULL;
05     }
06     if(cor(H->esq) == BLACK && cor(H->esq->esq) == BLACK)
07         H = move2EsqRED(H);
08
09     H->esq = removerMenor(H->esq);
10    return balancear(H);
11 }
12 struct NO* procuraMenor(struct NO* atual) {
13     struct NO *no1 = atual;
14     struct NO *no2 = atual->esq;
15     while(no2 != NULL){
16         no1 = no2;
17         no2 = no2->esq;
18     }
19    return no1;
20 }
```

FIGURA 11.85

Remove valor: 15



- 1 Inicia a busca pelo nó a ser removido a partir do nó "50".  
Nó procurado é menor do que 50. Visita nó "20".  
Nó "20" tem filho e neto (NULL) da cor preta à ESQUERDA. Chama a função move2EsqRED()

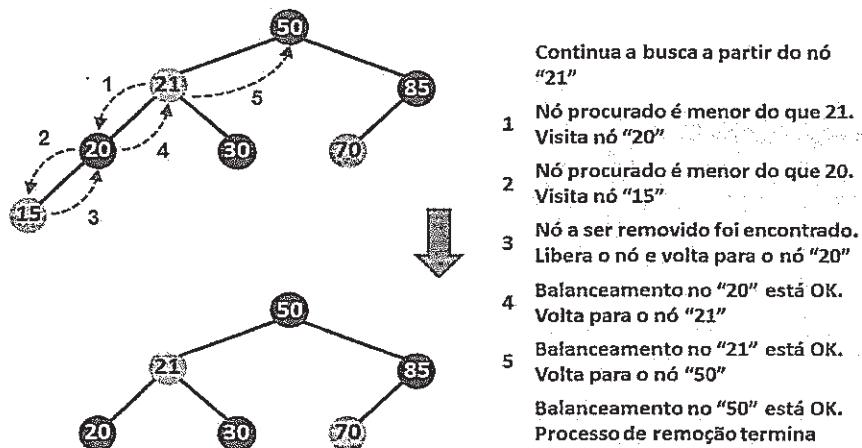


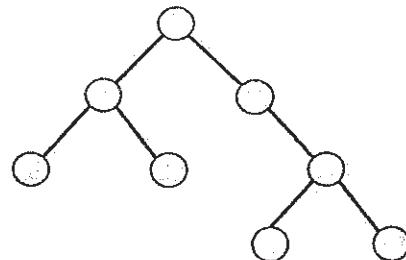
FIGURA 11.86

## 11.9 EXERCÍCIOS

- 1) Descreva, usando as suas palavras, o que é uma árvore. Utilize os conceitos de grafo.
- 2) Escreva uma função que percorra uma árvore binária e conte o seu número de nós.
- 3) Escreva uma função que percorra uma árvore binária e conte o seu número de nós não folhas.

- 4) Escreva uma função que percorra uma árvore binária e conte o seu número de nós folhas.
- 5) Escreva uma função que percorra uma árvore binária e exclua todos os nós com valor par.
- 6) Escreva uma função que retorne se uma árvore é binária de busca ou não.
- 7) Escreva uma função que encontre o maior valor existente em uma árvore binária de busca.
- 8) Escreva uma função que encontre o menor valor existente em uma árvore binária de busca.
- 9) Desenhe a árvore binária de busca resultante da inserção dos seguintes elementos (na ordem):
  - A, B, C, L, H, X, R, U, F, M, I.
  - I, D, T, U, A, F, E, N, Z, C, B, L.
- 0) Desenhe a árvore AVL resultante da inserção dos seguintes elementos (na ordem):
  - A, B, C, L, H, X, R, U, F, M, I.
  - I, D, T, U, A, F, E, N, Z, C, B, L.
- 1) Desenhe a árvore rubro-negra resultante da inserção dos seguintes elementos (na ordem):
  - A, B, C, L, H, X, R, U, F, M, I.
  - I, D, T, U, A, F, E, N, Z, C, B, L.
- 2) Duas árvores binárias são *similares* se possuem a mesma estrutura de nós (os valores podem ser diferentes). Implemente uma função para verificar se duas árvores são similares.
- 3) Duas árvores binárias são *iguais* se possuem a mesma estrutura de nós e os valores nas mesmas posições da árvore. Implemente uma função para verificar se duas árvores são iguais.
- 4) Explique, usando as suas palavras, as vantagens e desvantagens de usar árvores binárias平衡adas.
- 5) Descreva, usando as suas palavras, o que é uma árvore AVL e como funciona.
- 6) Descreva, usando as suas palavras, a diferença entre uma árvore binária de busca e uma AVL.
- 7) Descreva, usando as suas palavras, a diferença entre uma árvore AVL e uma árvore rubro-negra?
- 8) Qual a diferença na operação de busca em uma árvore binária de busca e em uma AVL? Explique.
- 9) Implemente funções não recursivas para realizar os três tipos de percurso (in-ordem, pré-ordem e pós-ordem) em uma árvore binária.
- 0) No percurso em largura, os nós de uma árvore são listados por nível, da esquerda para a direita. Dada uma árvore rubro-negra cujo percurso em largura é (67, 51, 87, 23, 53, 82, 90, 17, 31, 52, 60, 16 e 21), liste as chaves em nós rubros em ordem crescente.

- 21) Dada a árvore binária, mostre o resultado dos três tipos de percurso (in-ordem, pré-ordem e pós-ordem).



## APÍTULO 12

# Ios e aplicações das estruturas de dados

Quando estamos desenvolvendo uma aplicação, pode ser necessário utilizar uma estrutura de dados. Porém, nem todas as estruturas de dados são adequadas para resolver determinado problema. Assim, neste capítulo iremos ver quando e por que usar cada uma das estruturas de dados vistas ao longo deste livro.



**Estruturas (structs)** devem ser utilizadas sempre que se tiver uma coleção de dados que precisam ser manipulados de forma conjunta.

Uma **estrutura** pode ser vista como um conjunto de variáveis sob o mesmo nome, e cada uma delas pode ter qualquer tipo (ou o mesmo tipo). A ideia básica por trás da estrutura é que é apenas um tipo de dado que contém vários membros, que nada mais são do que outras variáveis. Em outras palavras, uma estrutura é uma variável que contém dentro de si outras variáveis, o que permite uma manipulação mais conveniente dos dados. Além disso, as estruturas possuem alto desempenho no acesso à memória e são fundamentais para a construção de quase todas as estruturas de dados conhecidas, como listas, filas, pilhas, árvores, grafos etc.



**Arrays** devem ser utilizados sempre que se souber a quantidade de dados que será armazenada e precisar de acesso aleatório a esses dados.

Nunca subestime o poder dos arrays. Basicamente, um array é uma estrutura de dados linearmente utilizada para armazenar e organizar dados em um computador. Trata-se de uma estrutura de dados muito útil quando sabemos de modo antecipado a quantidade de dados que devemos armazenar. O acesso aos elementos é extremamente rápido,  $O(1)$ , permitindo que seja feito de forma sequencial ou aleatória através do índice. Além disso, ele ocupa menos memória do que as estruturas dinâmicas por não possuir ponteiros ligando os dados. Sua desvantagem é o alto custo das operações de inserção e remoção, pois envolve o deslocamento de elementos. Arrays são estruturas de dados com alto desempenho no acesso à memória e fundamentais para a construção de outras estruturas de dados, como:

- Em uma lista sequencial estática.
- Em uma fila estática.
- Em uma pilha estática.
- Em uma tabela hash.



**Listas** devem ser utilizadas sempre que se desejar armazenar dados como em um array, mas não se souber a quantidade e não se precisar de acesso aleatório aos dados.

Uma lista é uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador. Trata-se de uma estrutura muito similar a um **array**. No entanto, nas listas, principalmente nas dinâmicas, não precisamos saber antecipadamente a quantidade de dados que serão armazenados: a lista pode crescer ou diminuir à medida que os dados são inseridos ou removidos. Ademais, podemos inserir dados no meio da lista sem nenhum custo extra, diferentemente dos arrays, para os quais esse tipo de inserção envolve o deslocamento de elementos. Infelizmente, todos esses benefícios vêm com uma desvantagem: não temos acesso aleatório às posições da lista. Desse modo, não podemos acessar diretamente uma posição do meio, nem fazer uma busca binária.

Listas são utilizadas de forma indireta com outras aplicações ou estruturas de dados, como:

- Na implementação de filas e pilhas.
- Para representar um baralho de cartas.
- Para representar um polinômio e implementar operações entre dois polinômios.
- Tratamento de colisões em tabelas hash (encadeamento separado ou separate chaining).
- Representação de um grafo por lista de adjacência.
- Sistema de arquivos FAT.
- Matriz esparsa: uso de listas encadeadas contendo somente os elementos não nulos.
- Representação de números muito grandes.



**Filas** devem ser utilizadas sempre que se desejar processar itens de acordo com a ordem de chegada.

Nas filas a inserção de um item é feita de um lado, enquanto a remoção é feita do outro. Ou seja, se quisermos acessar determinado elemento da fila, deveremos remover todos os que estiverem à frente dele. Desse modo, filas podem ser usadas para armazenar e controlar o fluxo de dados em um computador ou o acesso aos seus recursos, como:

- Gerenciamento de documentos enviados para a impressora.
- Processamento de tarefas em multiprogramação.

Filas também são utilizadas de forma indireta com outras aplicações ou estruturas de dados, como:

- Busca em largura em grafos e árvores.
- Fila de prioridade.
- Soma de inteiros (super) longos.
- Manipulação de uma sequência de caracteres.
- Transferência de dados assíncrona.
- Lista de reproduções em tocadores de MP3.



**Pilhas** devem ser utilizadas sempre que se necessitar voltar em tarefas ou itens na ordem inversa da que foram percorridas.

Nas pilhas, a inserção e a remoção são realizadas sempre no topo. Ou seja, se quisermos acessar determinado elemento da pilha, devemos remover todos os que estiverem sobre ele. Desse modo, pilhas podem ser usadas para manter um histórico de tarefas ou ações como:

- O histórico do seu navegador de internet.
- A lista de formatações feitas em um editor de texto e que podem ser desfeitas com o comando desfazer.

Essa funcionalidade da pilha também permite a criação de algoritmos de backtracking, ou seja, ao desempilhar os elementos da pilha esses algoritmos são capazes de percorrer tarefas ou tópicos na ordem inversa da que foram colocados na pilha. Uma aplicação simples de backtracking é um algoritmo para achar a solução de um labirinto: à medida que se avança no labirinto, podemos guardar o caminho que já foi percorrido. Sempre que encontrarmos um beco sem saída, podemos voltar no caminho desempilhando a pilha e escolher uma nova rota.

Pilhas também são utilizadas de forma indireta com outras aplicações ou estruturas de dados, como:

- O gerenciamento de memória em ambientes Java, C++, FORTRAN etc.
- A chamada e o retorno de funções.
- O processamento de estruturas aninhadas de profundidade imprevisível.
- O processamento de expressões aritméticas (prefixa, infixa e posfixa).
- A conversão de números decimais em binários.
- A implementação do quick sort sem recursão.



**Tabelas hash** devem ser utilizadas sempre que se necessitar de velocidade no acesso a pares de dados do tipo **valor/chave** e a ordenação dos dados não for necessária.

Nas tabelas hash, a inserção e a busca são realizadas em tempo constante,  $O(1)$ . Isso ocorre porque esse tipo de estrutura usa uma chave para acessar de forma rápida determinada posição no array onde os valores estão espalhados. Desse modo, tabelas hash podem ser usadas para:

- Construção de dicionários e arrays associativos (por exemplo, listas de telefones, catálogo de produtos etc.).
- Recuperação rápida de informação.
- Verificação de exclusividade (se aquele item é único).

Tabelas hash também são utilizadas de forma indireta com outras aplicações ou estruturas de dados, como:

- Busca de elementos em base de dados: estruturas de dados em memória, bancos de dados e mecanismos de busca na internet.
- Verificação de integridade de dados e autenticação de mensagens: os dados são enviados juntamente com o resultado da função de hashing. Quem receber os dados recalcula a função de hashing usando os dados recebidos e compara o resultado obtido com o que recebeu. Se os resultados forem diferentes, houve erro de transmissão.
- Armazenamento de senhas com segurança: a senha não é armazenada no servidor, mas sim o resultado da função de hashing.
- Implementação da tabela de símbolos dos compiladores.
- Criptografia: MD5 e família SHA (Secure Hash Algorithm).
- Tabelas de roteamento.
- Reconhecimento de palavras e detecção de erros de ortografia (dicionário de palavras).



**Grafos** devem ser utilizados sempre que se necessitar representar um conjunto de objetos e explorar as suas relações.

Um grafo é um modelo matemático que representa os objetos (vértices) de determinado conjunto e as relações (arestas) que existem entre eles. Grafos são ideais para se estudar as relações entre objetos e possuem uma grande quantidade de algoritmos desenvolvidos para essa finalidade. Desse modo, grafos podem ser usados para modelar e estudar:

- Redes sociais: relações de parentesco ou amizade entre pessoas.
- Redes de transporte, tecnológicas e de serviços: melhor trajeto em uma rodovia, como a informação flui em uma rede de computadores, se existem falhas em uma rede elétrica etc.
- Organização de dados: tarefas de um projeto e o pré-requisito entre elas.
- Lugares de um mapa: estradas que existem ligando os lugares, qual o melhor caminho.
- Sociologia: medir o prestígio de atores, explorar como boatos se espalham.

Grafos são virtualmente capazes de modelar qualquer tipo de problema. Por esse motivo, existem inúmeros algoritmos desenvolvidos para os mais diversos fins. Além disso, os grafos são utilizados de forma indireta com outras aplicações ou estruturas de dados, como:

- Procurar a saída de um labirinto.
- Verificar se uma rede de computadores está funcionando direito ou não (grafo completamente conexo).
- Implementar a ferramenta de preenchimento do Photoshop (balde de pintura).
- Roteamento: encontrar um número mínimo de hops em uma rede. Os hops são os vértices intermediários no caminho correspondente à conexão.
- Achar o menor caminho entre duas cidades vértices (melhor trajeto).
- Em algoritmos de roteamento: protocolos de roteamento dinâmicos (OSPF, IS-IS), algoritmo A\* (planejamento de rota).
- Programar robôs para explorar áreas.
- Procurar a rede celular GSM na vizinhança (coloração de grafos).
- Achar o melhor meio de conectar um conjunto de prédios por fibras ópticas (árvore geradora de custo mínimo).



Árvores devem ser utilizadas sempre que se necessitar representar um conjunto de objetos e explorar as suas relações hierárquicas.

Uma árvore é um tipo especial de **grafo**. Ou seja, como o grafo, a árvore também é um modelo matemático que representa os objetos de determinado conjunto. No entanto, enquanto grafo permite modelar as relações entre objetos de um conjunto, a árvore modela as relações hierárquicas entre esses objetos. Assim, qualquer problema em que exista algum tipo de hierarquia pode ser representado com uma árvore:

- Relações de descendência (pai, filho etc.).
- Diagrama hierárquico de uma organização.
- Campeonatos de modalidades desportivas.
- Taxonomia.
- Busca de dados armazenados no computador.
- Representação de espaço de soluções (ex: jogo de xadrez).
- Modelagem de algoritmos.

Além disso, as árvores possuem um tempo de execução rápido ( $O(\log N)$ ), se a árvore estiver balanceada) para todas as operações: inserção, remoção e busca. Isso permite encontrar facilmente valores mínimos e máximos, percorrer os elementos e inserir elementos aleatoriamente. Além disso, árvores são úteis na manutenção de estruturas nas quais a ordem é importante, pois permite armazenar dados ordenados de forma eficiente e com rápido acesso. Por esse motivo, existem vários tipos de árvores desenvolvidas para os mais diversos fins: árvore binária, árvore L, árvore B etc. Ademais, as árvores são utilizadas de forma indireta com outras aplicações estruturas de dados, como:

- Processamento de expressões aritméticas (prefixa, infixa e posfixa).
- Codificação de Huffman (código de compressão de dados).

- Processos de decisão (árvore de decisão).
- Detecção de colisão em jogos (quadtree e octree).
- Particionamento de espaço para indexação espacial (quadtree e octree).
- Indexação de informação de uma posição geográfica (árvores R).
- Gerenciamento de banco de dados e sistemas de arquivos (árvores B).
- Representação das possibilidades de jogadas a partir do estado do jogo (game tree).

