Técnicas de Programação

TP1101

Prof. Giovane Barcelos
giovane_barcelos@uniritter.edu.br

Plano de Ensino
Conteúdo programático

1. Introdução à programação em C 2. Desenvolvimento estruturado de programas em C 3. Controle de programa 4. Funções 5. Arrays 6. Ponteiros	**N1**
7. Caracteres e strings 8. Entrada/Saída formatada 9. Estruturas, uniões, manipulações de bits e enumerações 10. Processamento de arquivos 11. Estruturas de dados 12. O pré-processador 13. Outros tópicos sobre C	**N2**

Objetivos

- A criar, ler, gravar e atualizar arquivos.
- Processar arquivos por acesso sequencial.
- Processar arquivos por acesso aleatório.

Objetivos

- **A alocar e liberar memória de forma dinâmica para objetos de dados.**
- **A formar estruturas de dados encadeadas usando ponteiros, estruturas autorreferenciadas e recursão.**
- **A criar e manipular listas encadeadas, filas, pilhas e árvores binárias.**
- **Várias aplicações importantes das estruturas de dados encadeadas.**

Introdução

- Até aqui, estudamos estruturas de dados de tamanho fixo, tais como arrays unidimensionais, arrays bidimensionais e structs.
- Este capítulo introduz as **estruturas dinâmicas de dados**, que crescem e encolhem durante a execução do programa.
- **Listas encadeadas** são coleções de itens de dados 'alinhadas em uma fila' — inserções e exclusões são feitas em qualquer lugar em uma lista encadeada.
- **Pilhas** (stacks) são importantes em compiladores e sistemas operacionais — inserções e exclusões são feitas somente em uma extremidade da pilha: seu **topo**.

Introdução

- **Filas** representam filas de espera; as inserções são feitas no fim (também chamado de **cauda**) da fila, e as exclusões são feitas na frente (também chamada de cabeça) da fila.
- **Árvores binárias** facilitam a busca e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistemas de arquivos e a compilação de expressões em linguagem de máquina.
- Essas estruturas de dados têm muitas outras aplicações interessantes.

Introdução

- **Discutiremos os principais tipos de estruturas de dados e implementaremos programas que criam e manipulam essas estruturas de dados.**
- **Na próxima parte do livro — introdução à C++ e programação orientada a objeto —, estudaremos a abstração de dados.**
- **Essa técnica nos permitirá criar essas estruturas de dados de uma maneira totalmente diferente, que foi desenvolvida para produzir um software muito mais fácil de manter e reutilizar.**
- **Os programas abordam especialmente a manipulação de ponteiros, um assunto que muitas pessoas consideram ser um dos tópicos mais difíceis de C.**

Estruturas autorreferenciadas

- Uma estrutura autorreferenciada contém um membro ponteiro que aponta para uma estrutura do mesmo tipo de estrutura.
- Por exemplo, a definição
  - ```
    struct node {
       int data;
       struct node *nextPtr;
    };
    ```

  define um tipo, struct node.
- A estrutura do tipo struct node tem dois membros — o membro inteiro data e o membro ponteiro nextPtr.
    ```

Estruturas autorreferenciadas

- O membro nextPtr aponta para uma estrutura do tipo struct node — o mesmo tipo de estrutura que está sendo declarada aqui, daí o termo 'classe autorreferenciada'.
- O membro nextPtr é chamado de link — ou seja, nextPtr pode ser usado para 'vincular' uma estrutura do tipo struct node a outra estrutura do mesmo tipo.
- As estruturas autorreferenciadas podem ser vinculadas para formar estruturas de dados úteis, tais como listas, filas, pilhas e árvores.

Estruturas autorreferenciadas

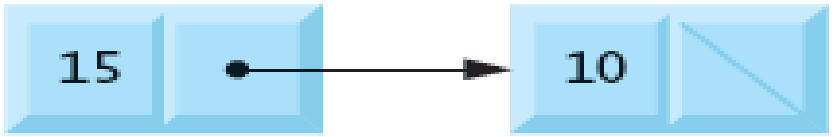

Figura 12.1 ■ Estruturas autorreferenciadas vinculadas.

Estruturas autorreferenciadas

- **A Figura 12.1 ilustra dois objetos de estrutura autorreferenciada unidos para formar uma lista.**
- **Uma barra invertida (\), que representa um ponteiro NULL, é colocada no membro do link da segunda estrutura autorreferenciada para indicar que o link não aponta para outra estrutura.**
- **[*Nota*: a barra é usada somente para fins de ilustração; ela não corresponde ao caractere de barra invertida usado em C.]**
- **Um ponteiro NULL indica o fim de uma estrutura de dados, da mesma maneira que o caractere nulo indica o fim de uma string.**

Estruturas autorreferenciadas

Erro comum de programação 12.1

Não inicializar o link no último nó de uma lista com NULL *pode ocasionar erros no tempo de execução.*

Alocação dinâmica de memória

- Criar e manter estruturas dinâmicas de dados exige uma alocação dinâmica de memória — a capacidade que um programa tem de obter mais espaço de memória durante a execução para manter novos nós, e para liberar espaço do qual não se precisa mais.
- O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador, ou a quantidade de memória virtual disponível em um sistema de memória virtual.
- Frequentemente, os limites são muito baixos, porque a memória disponível deve ser compartilhada entre muitas aplicações.
- As funções malloc e free, e o operador sizeof, são essenciais para a alocação dinâmica de memória.

Alocação dinâmica de memória

- A função malloc usa o número de bytes a serem alocados como argumento, e retorna um ponteiro do tipo void * (ponteiro para void) para a memória alocada.
- Um ponteiro void * pode ser atribuído a uma variável de qualquer tipo de ponteiro.
- A função malloc geralmente é usada com o operador sizeof.

Alocação dinâmica de memória

- **Por exemplo a instrução**
  - **newPtr = malloc(sizeof(struct node));**
- **avalia sizeof(struct node) para determinar o tamanho em bytes de uma estrutura do tipo struct node, aloca uma nova área na memória desse número de bytes e armazena um ponteiro para a memória alocada na variável newPtr.**

Alocação dinâmica de memória

Dica de portabilidade 12.1
O tamanho de uma estrutura não é necessariamente a soma dos tamanhos de seus membros. Isso ocorre por causa
dos vários requisitos de alinhamento de limites de endereços dependentes de máquina (ver Capítulo 10).

Erro comum de programação 12.2
Supor que o tamanho de uma estrutura é simplesmente a soma dos tamanhos de seus dados-membros é um erro lógico.

Alocação dinâmica de memória

Boa prática de programação 12.1
Use o operador sizeof para determinar o tamanho de uma estrutura.

Dica de prevenção de erro 12.1
Ao usar malloc, teste um valor de retorno de ponteiro NULL, que indica que a memória não foi alocada.

Alocação dinâmica de memória

Erro comum de programação 12.3
Não retornar a memória alocada dinamicamente quando ela não é mais necessária, pode fazer com que o sistema
fique sem memória prematuramente. Isso, às vezes, é chamado de 'vazamento de memória'.

Boa prática de programação 12.2
Quando a memória que foi alocada dinamicamente não for mais necessária, use free para liberar a memória
imediatamente para o sistema.

Alocação dinâmica de memória

Erro comum de programação 12.4
Liberar memória não alocada dinamicamente com malloc.

Erro comum de programação 12.5
Referir-se à memória que foi liberada é um erro que normalmente resulta na falha do programa.

Alocação dinâmica de memória

- A memória alocada não é inicializada.
- Se não houver memória disponível, malloc retorna NULL.
- A função free libera memória — ou seja, a memória é retornada ao sistema, de modo que possa ser realocada no futuro.
- Para liberar memória dinamicamente alocada pela chamada malloc anterior, use a instrução
  - free(newPtr);
- C também oferece funções calloc e realloc para criar e modificar arrays dinâmicos.

Listas encadeadas

- **Uma lista encadeada é uma coleção linear de objetos de uma classe autorreferenciadas, chamados de nós, conectados por links de ponteiros — daí o termo lista 'encadeada'.**
- **Uma lista encadeada é acessada por meio de um ponteiro para o primeiro nó da lista.**
- **Os nós subsequentes são acessados por meio do membro ponteiro de link armazenado em cada nó.**
- **Por convenção, o ponteiro de link do último nó de uma lista é inicializado em NULL, para marcar o fim da lista.**
- **Os dados são armazenados em uma lista encadeada dinamicamente — cada nó é criado de acordo com a necessidade.**

Listas encadeadas

- Um nó pode conter dados de qualquer tipo, inclusive outros objetos struct.
- Pilhas e filas são também estruturas de dados lineares, e, como veremos, são versões de listas encadeadas com restrições.
- As árvores são estruturas de dados não lineares.
- Listas de dados podem ser armazenadas em arrays, mas as listas encadeadas oferecem várias vantagens.
- Uma lista encadeada é apropriada quando é impossível predizer o número de elementos de dados a ser representado na estrutura de dados.
- As listas encadeadas são dinâmicas, assim o comprimento de uma lista pode aumentar ou diminuir conforme a necessidade.

Listas encadeadas

Dica de desempenho 12.1

Um array pode ser declarado para conter mais elementos do que o número de itens esperados, mas isso pode desperdiçar memória. As listas encadeadas podem fornecer uma forma mais adequada de utilização de memória nessas situações.

Listas encadeadas

- **O tamanho de um array, porém, não pode ser alterado depois de a memória ser alocada.**
- **Arrays podem ficar cheios.**
- **Listas encadeadas ficam cheias somente quando o sistema não tem memória suficiente para satisfazer solicitações dinâmicas de alocação de memória.**

Listas encadeadas

Dica de desempenho 12.2
A inserção e a exclusão em um array ordenado podem consumir muito tempo — todos os elementos a partir do elemento inserido ou excluído devem ser movidos de forma apropriada.

Dica de desempenho 12.3
Os elementos de um array são armazenados consecutivamente na memória. Isso permite o acesso imediato a qualquer elemento do array, porque o endereço de qualquer elemento pode ser calculado diretamente com base na posição de início do array. As listas encadeadas não dispõem de tal 'acesso direto' imediato a seus elementos.

Listas encadeadas

- **Listas encadeadas podem ser mantidas em ordem ao se inserir cada novo elemento no ponto apropriado da lista.**

Listas encadeadas

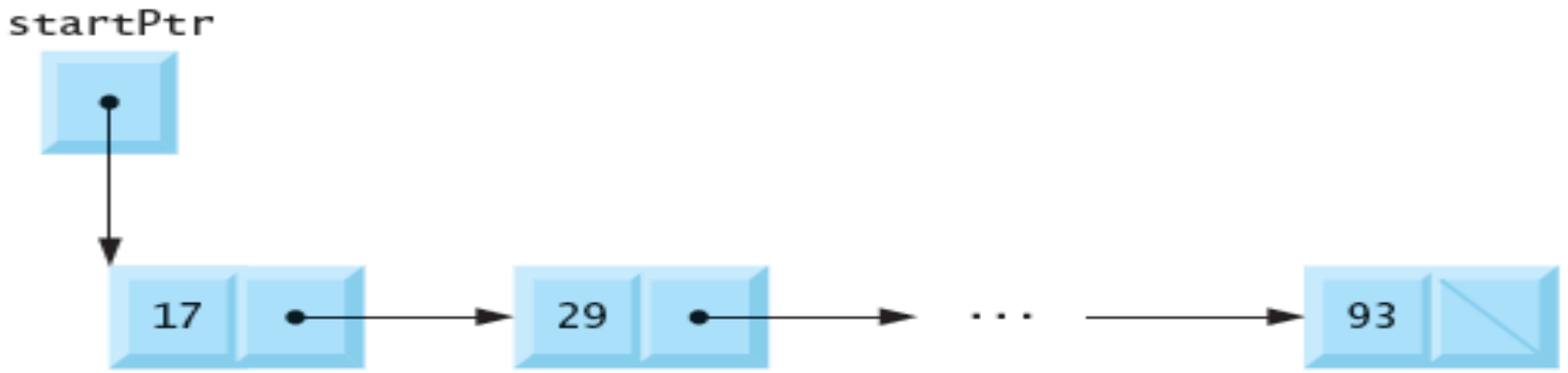

Figura 12.2 ■ Representação gráfica da lista encadeada.

Listas encadeadas

Dica de desempenho 12.4

Usar a alocação de memória dinâmica (em vez de arrays) em estruturas de dados que crescem e encolhem durante a execução pode economizar memória. Tenha em mente, porém, que ponteiros ocupam espaço, e que a alocação de memória dinâmica incorre na sobrecarga das chamadas de função.

Listas encadeadas

- **Os nós de uma lista encadeada não estão armazenados consecutivamente na memória.**
- **Logicamente, porém, os nós de uma lista encadeada parecem ser contíguos.**
- **A Figura 12.2 ilustra uma lista encadeada com vários nós.**

Listas encadeadas

- O programa da Figura 12.3 (cuja saída é mostrada na Figura 12.4) manipula uma lista de caracteres.
- O programa permite que você insira um caractere na lista em ordem alfabética (função insert) ou exclua um caractere da lista (função delete).

Listas encadeadas

```
1  /* Fig. 12.3: fig12_03.c
2     Operando e mantendo uma lista */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* estrutura autorreferenciada */
7  struct listNode {
8     char data; /* cada listNode contém um caractere */
9     struct listNode *nextPtr; /* ponteiro para próximo nó */
10 }; /* fim da estrutura listNode */
11
12 typedef struct listNode ListNode; /* sinônimo de struct listNode */
13 typedef ListNode *ListNodePtr; /* sinônimo de ListNode* */
14
15 /* protótipos */
16 void insert( ListNodePtr *sPtr, char value );
17 char delete( ListNodePtr *sPtr, char value );
18 int isEmpty( ListNodePtr sPtr );
19 void printList( ListNodePtr currentPtr );
20 void instructions( void );
21
22 int main( void )
23 {
24    ListNodePtr startPtr = NULL; /* inicialmente não existem nós */
25    int choice; /* escolha do usuário */
26    char item; /* char inserido pelo usuário */
27
28    instructions(); /* exibe o menu */
29    printf( "? " );
30    scanf( "%d", &choice );
31
```

Listas encadeadas

```
32    /* loop enquanto usuário não escolhe 3 */
33    while ( choice != 3 ) {
34
35       switch ( choice ) {
36          case 1:
37             printf( "Digite um caractere: " );
38             scanf( "\n%c", &item );
39             insert( &startPtr, item ); /* insere item na lista */
40             printList( startPtr );
41             break;
42          case 2: /* exclui um elemento */
43             /* se lista não está vazia */
44             if ( !isEmpty( startPtr ) ) {
45                printf( "Digite caractere a ser excluído: " );
46                scanf( "\n%c", &item );
47
48                /* se caractere for encontrado, é removido */
49                if ( delete( &startPtr, item ) ) { /* remove item */
50                   printf( "%c deleted.\n", item );
51                   printList( startPtr );
52                } /* fim do if */
53                else {
54                   printf( "%c não encontrado.\n\n", item );
55                } /* fim do else */
56             } /* fim do if */
57             else {
58                printf( "Lista está vazia.\n\n" );
59             } /* fim do else */
60
61             break;
62          default:
63             printf( "Escolha inválida.\n\n" );
```

Figura 12.3 ■ Inserção e exclusão de nós em uma lista. (Parte 1 de 3.)

Listas encadeadas

```
64                instructions();
65                break;
66             } /* fim do switch */
67
68             printf( "? " );
69             scanf( "%d", &choice );
70          } /* fim do while */
71
72          printf( "Fim da execução.\n" );
73          return 0; /* indica conclusão bem-sucedida */
74       } /* fim do main */
75
76       /* exibe instruções do programa ao usuário */
77       void instructions( void )
78       {
79          printf( "Digite sua escolha:\n"
80             "   1 para inserir um elemento na lista.\n"
81             "   2 para excluir um elemento da lista.\n"
82             "   3 para terminar.\n" );
83       } /* fim das instruções de função */
84
85       /* Insere novo valor na lista, na ordem classificada */
86       void insert( ListNodePtr *sPtr, char value )
87       {
88          ListNodePtr newPtr; /* ponteiro para novo nó */
89          ListNodePtr previousPtr; /* ponteiro para nó anterior na lista */
90          ListNodePtr currentPtr; /* ponteiro para nó atual na lista */
91
92       newPtr = malloc( sizeof( ListNode ) ); /* cria nó */
93
94       if ( newPtr != NULL ) { /* espaço está disponível */
95          newPtr->data = value; /* coloca valor no nó */
96          newPtr->nextPtr = NULL; /* nó não se une a outro nó */
97
```

Listas encadeadas

```
98       previousPtr = NULL;
99       currentPtr = *sPtr;
100
101      /* loop para achar o local correto na lista */
102      while ( currentPtr != NULL && value > currentPtr->data ) {
103         previousPtr = currentPtr;  /* caminha para ...*/
104         currentPtr = currentPtr->nextPtr; /* ... próximo nó */
105      } /* fim do while */
106
107      /* insere novo nó no início da lista */
108      if ( previousPtr == NULL ) {
109         newPtr->nextPtr = *sPtr;
110         *sPtr = newPtr;
111      } /* fim do if */
112      else { /* insere novo nó entre previousPtr e currentPtr */
113            previousPtr->nextPtr = newPtr;
114            newPtr->nextPtr = currentPtr;
115         } /* fim do else */
116      } /* fim do if */
117      else {
118         printf( "%c não inserido. Sem memória disponível.\n", value );
119      } /* fim do else */
120   } /* fim da função insert */
121
122   /* Exclui um elemento da lista */
123   char delete( ListNodePtr *sPtr, char value )
124   {
125      ListNodePtr previousPtr; /* ponteiro para nó anterior na lista */
126      ListNodePtr currentPtr; /* ponteiro para nó atual na lista */
```

Figura 12.3 ■ Inserção e exclusão de nós em uma lista. (Parte 2 de 3.)

Listas encadeadas

```
127     ListNodePtr tempPtr; /* ponteiro de nó temporário */
128
129     /* exclui primeiro nó */
130     if ( value == ( *sPtr )->data ) {
131        tempPtr = *sPtr; /* aponta para o nó que está sendo removido */
132        *sPtr = ( *sPtr )->nextPtr; /* retira thread do nó */
133        free( tempPtr ); /* libera o nó com thread retirado */
134        return value;
135     } /* fim do if */
136     else {
137        previousPtr = *sPtr;
138        currentPtr = ( *sPtr )->nextPtr;
139
140        /* loop para achar local correto na lista */
141        while ( currentPtr != NULL && currentPtr->data != value ) {
142           previousPtr = currentPtr;  /* caminha até ...   */
143           currentPtr = currentPtr->nextPtr; /* ... próximo nó */
144        } /* fim do while */
145
146        /* exclui nó em currentPtr */
147        if ( currentPtr != NULL ) {
148           tempPtr = currentPtr;
149           previousPtr->nextPtr = currentPtr->nextPtr;
150           free( tempPtr );
151           return value;
152        } /* fim do if */
153     } /* fim do else */
154
```

Listas encadeadas

```
155      return '\0';
156   } /* fim da função delete */
157
158   /* Retorna 1 se a lista estiver vazia, 0 se estiver cheia */
159   int isEmpty( ListNodePtr sPtr )
160   {
161      return sPtr == NULL;
162   } /* fim da função isEmpty */
163
164   /* Imprime a lista */
165   void printList( ListNodePtr currentPtr )
166   {
167      /* se lista estiver vazia */
168      if ( currentPtr == NULL ) {
169         printf( "Lista está vazia.\n\n" );
170      } /* fim do if */
171      else {
172         printf( "A lista é:\n" );
173
174         /* enquanto não chega ao final da lista */
175         while ( currentPtr != NULL ) {
176            printf( "%c --> ", currentPtr->data );
177            currentPtr = currentPtr->nextPtr;
178         } /* fim do while */
179
180         printf( "NULL\n\n" );
181      } /* fim do else */
182   } /* fim da função printList */
```

Figura 12.3 ■ Inserção e exclusão de nós em uma lista. (Parte 3 de 3.)

Listas encadeadas

```
Digite sua escolha:
  1 para inserir um elemento na lista.
  2 para excluir um elemento da lista.
  3 para terminar.
? 1
Digite um caractere: B
A lista é:
B --> NULL

? 1
Digite um caractere: A
A lista é:
A --> B --> NULL

? 1
Digite um caractere: C
A lista é:
A --> B --> C --> NULL

? 2
Digite caractere a ser excluído: D
D não encontrado.

? 2
Digite caractere a ser excluído: B
B excluído.
A lista é:
A --> C --> NULL
```

Listas encadeadas

```
? 2
Digite caractere a ser excluído: C
C excluído.
A lista é:
A --> NULL

? 2
Digite caractere a ser excluído: A
A excluído.
Lista está vazia.

? 4
Escolha inválida.

Digite sua escolha:
  1 para inserir um elemento na lista.
  2 para excluir um elemento da lista.
  3 para terminar.
? 3
Fim da execução.
```

Figura 12.4 ■ Exemplo de saída do programa da Figura 12.3.

Listas encadeadas

- As principais funções das listas encadeadas são insert (linhas 86-120) e delete (linhas 123-156).
- A função isEmpty (linhas 159-162) é chamada de função predicado — ela não altera a lista, mas, determina se a lista está vazia (ou seja, o ponteiro para o primeiro nó da lista é NULL).
- Se a lista estiver vazia, 1 é retornado; caso contrário, 0 é retornado.
- A função printList (linhas 165-182) imprime a lista.

Listas encadeadas

- Os caracteres são inseridos na lista em ordem alfabética.
- A função insert (linhas 86-120) recebe o endereço da lista e um caractere a ser inserido.
- O endereço da lista é necessário quando um valor deve ser inserido no início da lista.
- Fornecer o endereço da lista permite que ela (ou seja, o ponteiro para o primeiro nó da lista) seja modificada por meio de uma chamada por referência.
- Como a própria lista é um ponteiro (para seu primeiro elemento), a passagem do endereço da lista cria um ponteiro para um ponteiro (ou seja, indireção dupla).
- Esta é uma noção complexa, e exige uma programação cuidadosa.

Listas encadeadas

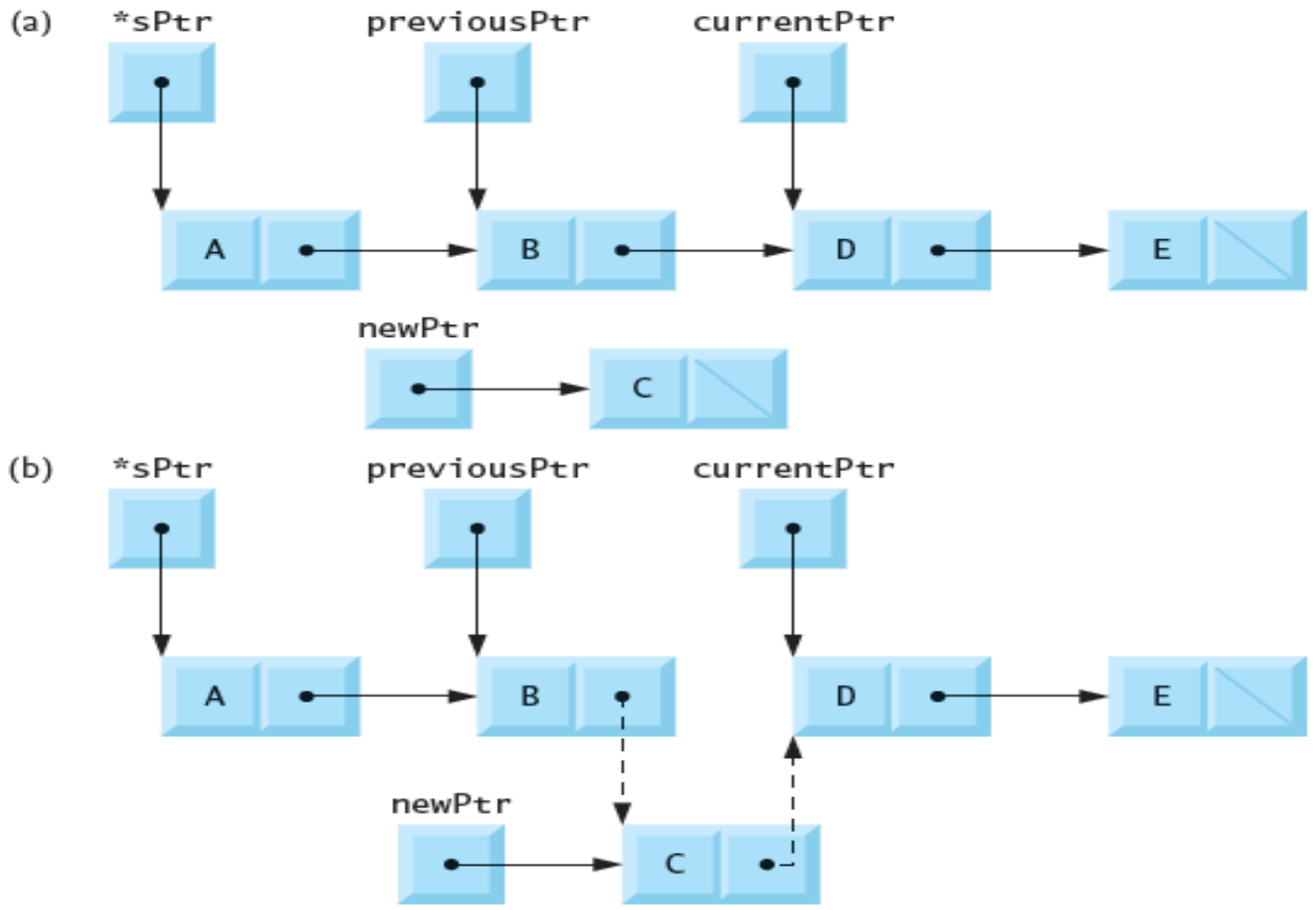

Figura 12.5 ■ Inserção de um nó em uma lista ordenada.

Listas encadeadas

- As etapas para inserir um caractere na lista são as seguintes (ver Figura 12.5):
  - Criar um nó chamando malloc, atribuindo a newPtr o endereço da memória alocada (linha 92), atribuindo o caractere a ser inserido a newPtr->data (linha 95) e atribuindo NULL a newPtr->nextPtr (linha 96).
  - Inicializar previousPtr a NULL (linha 198) e currentPtr a *sPtr (linha 99) — ponteiro para o início da lista. Os ponteiros previousPtr e currentPtr armazenam os locais do nó antes do ponto de inserção e do nó após o ponto de inserção.
  - Enquanto currentPtr não for NULL e o valor a ser inserido for maior que currentPtr->data (linha 102), atribua currentPtr a previousPtr (linha 103) e passe currentPtr para o próximo nó da lista (linha 104). Isso localiza o ponto de inserção para o valor.

Listas encadeadas

Dica de prevenção de erro 12.2

Atribua NULL ao membro de ligação de um novo nó. Os ponteiros devem ser inicializados antes de serem usados.

Listas encadeadas

- **Se previousPtr for NULL (linha 108), insira o novo nó como primeiro nó na lista (linhas 109-110). Atribua *sPtr to newPtr->nextPtr (o novo link de nó aponta para o primeiro nó), e atribua newPtr a *sPtr (*sPtr aponta para o novo nó). Caso contrário, se previousPtr não for NULL, o novo nó é inserido no local (linhas 113-114). Atribua newPtr a previousPtr->nextPtr (o nó anterior aponta para o novo nó) e atribua currentPtr a newPtr->nextPtr (o novo link de nó aponta para o nó atual).**

Listas encadeadas

- A Figura 12.5 ilustra a inserção de um nó que contém o caractere 'C' em uma lista ordenada.
- A parte (a) da figura mostra a lista e o novo nó antes da inserção.
- A parte (b) mostra o resultado da inserção do novo nó.
- Os ponteiros reatribuídos são setas tracejadas.
- Para simplificar, implementamos a função insert (e outras funções semelhantes neste capítulo) com um tipo de retorno void.
- É possível que a função malloc deixe de alocar a memória solicitada.
- Nesse caso, seria melhor que nossa função insert retornasse um status que indicasse se a operação foi bem-sucedida.

Listas encadeadas

- A função delete ((linhas 123-156) recebe o endereço do ponteiro para o início da lista e um caractere a ser excluído.
- As etapas n para se excluir um caractere da lista são as seguintes:
  - Se o caractere a ser excluído combina com o caractere no primeiro nó da lista (linha 130), atribua *sPtr a tempPtr (tempPtr será usado para liberar — free a memória indesejada, atribua(*sPtr)->nextPtr a *sPtr (*sPtr agora aponta para o segundo nó na lista), libere (free) a memória apontada por tempPtr e retorne o caractere que foi excluído.
  - Caso contrário, inicialize previousPtr com *sPtr e inicialize currentPtr com (*sPtr)->nextPtr (linhas 137–138).
  - Enquanto currentPtr não for NULL e o valor a ser excluído não for igual a currentPtr->data (linha 141) e atribua currentPtr a previousPtr (linha 142) e atribua currentPtr->nextPtr a currentPtr (linha 143). Isso localiza o caractere a ser excluído se ele estiver contido na lista.

Listas encadeadas

- **Se currentPtr não for NULL (linha 147), atribua currentPtr a tempPtr (linha 148), atribua currentPtr->nextPtr a previousPtr->nextPtr (linha 149), libere o nó apontado por tempPtr (linha 150) e retorne o caractere que foi excluído da lista (linha 151). Se currentPtr for NULL, retorne o caractere nulo ('\0') para indicar que o caractere a ser excluído não foi encontrado na lista (linha 155).**

Listas encadeadas

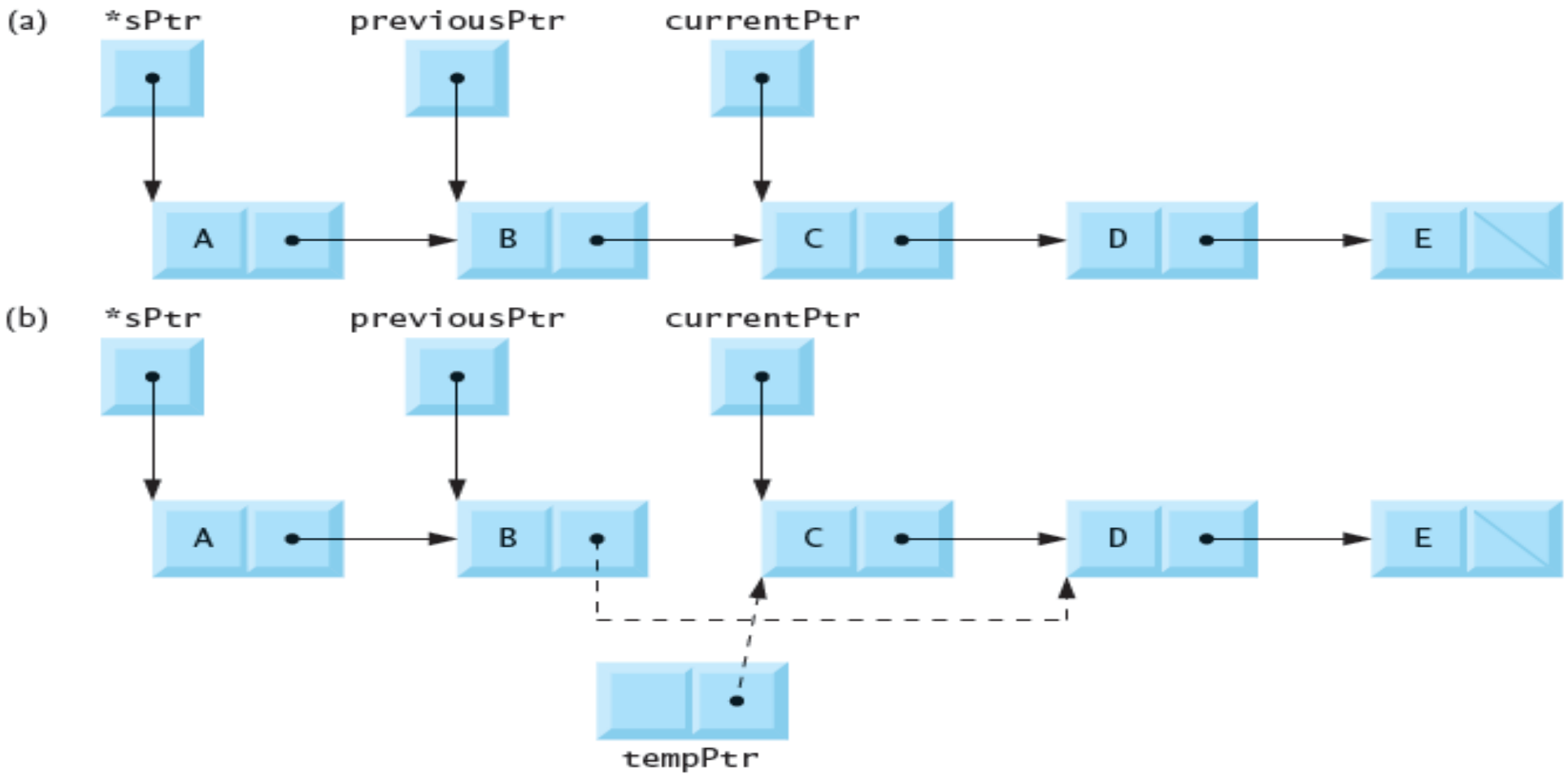

Figura 12.6 ■ Exclusão do nó de uma lista.

Listas encadeadas

- A Figura 12.6 ilustra a exclusão de um nó a partir de uma lista encadeada.
- A parte (a) da figura mostra a lista encadeada depois da operação de inserção anterior.
- A parte (b) mostra a reatribuição do elemento de link de previousPtr e a atribuição de currentPtr a tempPtr.
- O ponteiro tempPtr é usado para liberar a memória alocada para armazenar 'C'.

Listas encadeadas

- A função printList (linhas 165-182) recebe um ponteiro para o início da lista como um argumento, e se refere ao ponteiro como currentPtr.
- Primeiro, a função determina se a lista está vazia (linhas 168-170) e, se estiver, imprime "A lista está vazia." e termina.
- Caso contrário, ela imprime os dados da lista (linhas 171-181).

Listas encadeadas

- **Enquanto currentPtr não for NULL, o valor de currentPtr->data será impresso pela função, e currentPtr->nextPtr será atribuído a currentPtr.**
- **Se o link no último nó da lista não for NULL, o algoritmo de impressão tentará imprimir após o final da lista, e ocorrerá um erro.**
- **O algoritmo de impressão é idêntico para listas encadeadas, pilhas e filas.**

Pilhas

- Uma pilha é uma versão sujeita a restrições de uma lista encadeada.
- Somente pelo topo é que novos nós podem ser acrescentados ou removidos de uma pilha.
- Por essa razão, uma pilha é chamada de estrutura de dados último a entrar, primeiro a sair (LIFO — last-in, first-out).
- Uma pilha é referenciada por um ponteiro para o elemento do topo da pilha.
- O membro de link no último nó da pilha é inicializado com NULL para indicar a parte inferior da pilha.

Pilhas

Figura 12.7 ■ Representação gráfica da pilha.

Pilhas

- A Figura 12.7 ilustra uma pilha com vários nós.
- As pilhas e as listas encadeadas são representadas de formas idênticas.
- A diferença entre ambas é que inserções e exclusões podem ocorrer em qualquer lugar de uma lista encadeada, mas somente no topo de uma pilha.

Pilhas

Erro comum de programação 12.6
Não definir o link como NULL no último nó de uma pilha pode ocasionar erros de tempo de execução.

Pilhas

- As principais funções usadas para manipular uma pilha são push e pop.
- A função push cria um novo nó e o coloca no topo da pilha.
- A função pop remove um nó do topo da pilha, libera a memória que foi alocada para o nó removido e retorna o valor removido.
- A Figura 12.8 (saída mostrada na Figura 12.9) implementa uma pilha simples de inteiros.
- O programa oferece três opções: (1) colocar um valor na pilha (função push), (2) remover um valor da pilha (função pop) e (3) terminar o programa.

Pilhas

```
1  /* Fig. 12.8: fig12_08.c
2     Programa de pilha dinâmica */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* estrutura autorreferenciada */
7  struct stackNode {
8     int data; /* define dados como um int */
9     struct stackNode *nextPtr; /* ponteiro stackNode */
10 }; /* fim da estrutura stackNode */
11
12 typedef struct stackNode StackNode; /* sinônimo de struct stackNode */
13 typedef StackNode *StackNodePtr; /* sinônimo de StackNode* */
14
15 /* protótipos */
16 void push( StackNodePtr *topPtr, int info );
17 int pop( StackNodePtr *topPtr );
18 int isEmpty( StackNodePtr topPtr );
19 void printStack( StackNodePtr currentPtr );
20 void instructions( void );
21
22 /* função main inicia execução do programa */
23 int main( void )
24 {
25    StackNodePtr stackPtr = NULL; /* aponta para topo da pilha */
26    int choice; /* escolha do menu do usuário */
27    int value; /* entrada int pelo usuário */
28
29    instructions(); /* exibe o menu */
30    printf( "? " );
```

Figura 12.8 ■ Um programa simples de pilha. (Parte 1 de 3.)

Pilhas

```
31      scanf( "%d", &choice );
32
33      /* enquanto usuário não digita 3 */
34      while ( choice != 3 ) {
35
36         switch ( choice ) {
37            /* coloca valor na pilha */
38            case 1:
39               printf( "Digite um inteiro: " );
40               scanf( "%d", &value );
41               push( &stackPtr, value );
42               printStack( stackPtr );
43               break;
44            /* remove valor da pilha */
45            case 2:
46               /* se a pilha não está vazia */
47               if ( !isEmpty( stackPtr ) )  {
48                  printf( "O valor retirado é %d.\n", pop( &stackPtr ) );
49               } /* fim do if */
50
51               printStack( stackPtr );
52               break;
53            default:
54               printf( "Escolha inválida.\n\n" );
55               instructions();
56               break;
57         } /* fim do switch */
58
59         printf( "? " );
60         scanf( "%d", &choice );
61      } /* fim do while */
62
```

Pilhas

```
63       printf( "Fim da execução.\n" );
64       return 0; /* indica conclusão bem-sucedida */
65    } /* fim do main */
66
67    /* exibe informações do programa ao usuário */
68    void instructions( void )
69    {
70       printf( "Digite escolha:\n"
71          "1 para colocar um valor na pilha\n"
72          "2 para retirar um valor da pilha\n"
73          "3 para terminar programa\n" );
74    } /* fim da função instructions */
75
76    /* Insere um nó no topo da pilha */
77    void push( StackNodePtr *topPtr, int info )
78    {
79       StackNodePtr newPtr; /* ponteiro para novo nó */
80
81       newPtr = malloc( sizeof( StackNode ) );
82
83       /* insere o nó no topo da pilha */
84       if ( newPtr != NULL ) {
85          newPtr->data = info;
86          newPtr->nextPtr = *topPtr;
87          *topPtr = newPtr;
88       } /* fim do if */
89       else { /* nenhum espaço disponível */
90          printf( "%d não inserido. Nenhuma memória disponível.\n", info );
91       } /* fim do else */
```

Figura 12.8 ■ Um programa simples de pilha. (Parte 2 de 3.)

Pilhas

```
 92   } /* fim da função push */
 93
 94   /* Remove um nó do topo da pilha */
 95   int pop( StackNodePtr *topPtr )
 96   {
 97      StackNodePtr tempPtr; /* ponteiro de nó temporário */
 98      int popValue; /* node value */
 99
100      tempPtr = *topPtr;
101      popValue = ( *topPtr )->data;
102      *topPtr = ( *topPtr )->nextPtr;
103      free( tempPtr );
104      return popValue;
105   } /* fim da função pop */
106
107   /* Imprime a pilha */
108   void printStack( StackNodePtr currentPtr )
109   {
```

Pilhas

```
110      /* se a pilha está vazia */
111      if ( currentPtr == NULL ) {
112         printf( "A pilha está vazia.\n\n" );
113      } /* fim do if */
114      else {
115         printf( "A pilha é:\n" );
116
117         /* enquanto não chega final da pilha */
118         while ( currentPtr != NULL ) {
119            printf( "%d --> ", currentPtr->data );
120            currentPtr = currentPtr->nextPtr;
121         } /* fim do while */
122
123         printf( "NULL\n\n" );
124      } /* fim do else */
125   } /* fim da função printList */
126
127   /* Retorna 1 se a pilha está vazia, caso contrário, retorna 0 */
128   int isEmpty( StackNodePtr topPtr )
129   {
130      return topPtr == NULL;
131   } /* fim da função isEmpty */
```

Figura 12.8 ■ Um programa simples de pilha. (Parte 3 de 3.)

Pilhas

```
Digite escolha:
1 para colocar um valor na pilha
2 para retirar um valor da pilha
3 para terminar programa
? 1
Digite um inteiro: 5
A pilha é:
5 --> NULL

? 1
Digite um inteiro: 6
```

Figura 12.9 ■ Exemplo de saída do programa da Figura 12.8. (Parte 1 de 2.)

Pilhas

```
A pilha é:
6 --> 5 --> NULL

? 1
Digite um inteiro: 4
A pilha é:
4 --> 6 --> 5 --> NULL

? 2
O valor removido é 4.
A pilha é:
6 --> 5 --> NULL

? 2
O valor retirado é 6.
A pilha é:
5 --> NULL

? 2
O valor retirado é 5.
A pilha está vazia.

? 2
A pilha está vazia.

? 4
Escolha inválida.

Digite escolha:
1 para colocar um valor na pilha
2 para retirar um valor da pilha
3 para terminar programa
? 3
Fim da execução.
```

Figura 12.9 ■ Exemplo de saída do programa da Figura 12.8. (Parte 2 de 2.)

Pilhas

- A função push (linhas 77-92) coloca um novo nó no topo da pilha.
- A função consiste em três etapas:
  - Criar um novo nó chamando malloc e atribuir o local da memória alocada a newPtr (linha 81).
  - Atribuir a newPtr->data o valor a ser colocado na pilha (linha 85) e atribuir *topPtr (o ponteiro do topo da pilha) a newPtr->nextPtr (linha 86) — o membro de link de newPtr agora aponta para o nó do topo anterior.
  - Atribua newPtr a *topPtr (linha 87) — *topPtr agora aponta para o novo topo da pilha.

Pilhas

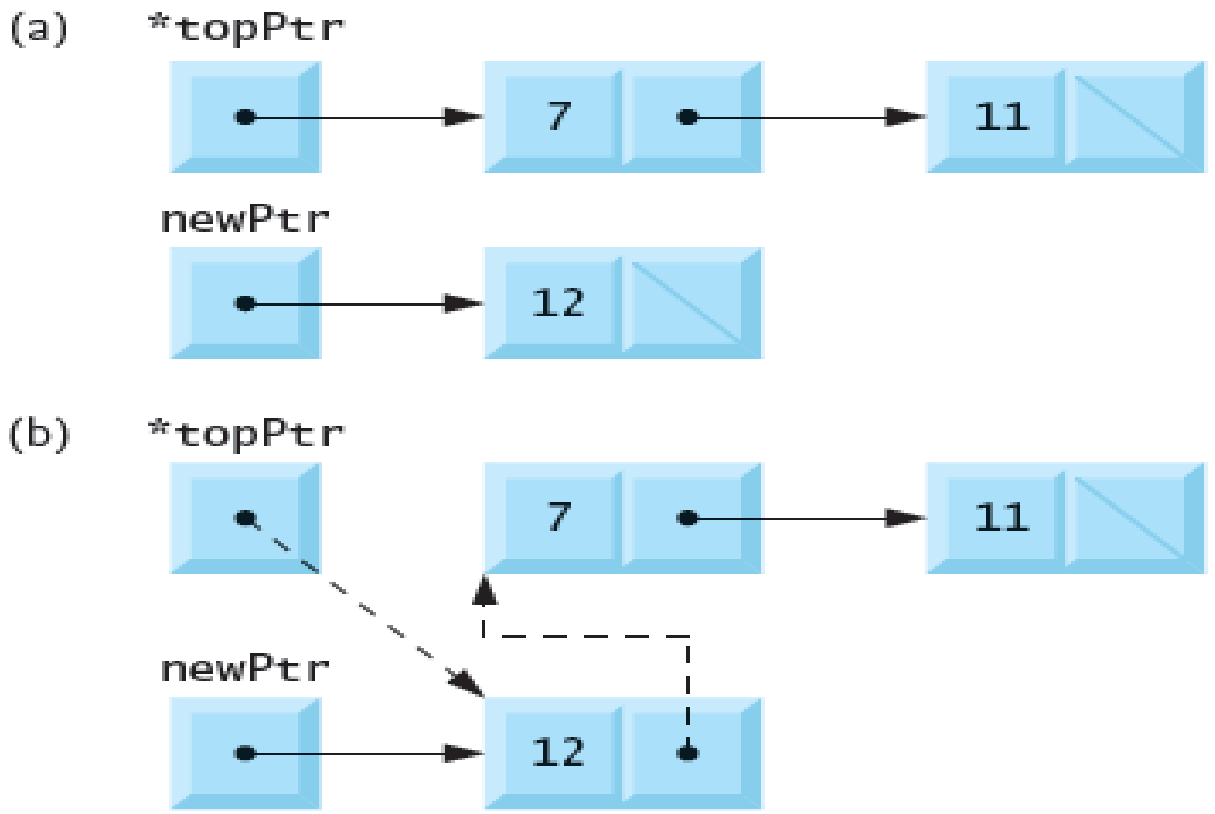

Figura 12.10 ■ Operação push.

Pilhas

- As manipulações que envolvem *topPtr mudam o valor de stackPtr em main.
- A Figura 12.10 ilustra a função push.
- A parte (a) da figura mostra a pilha e o novo nó antes da operação push.
- As setas tracejadas na parte (b) ilustram as *etapas* 2 e 3 da operação push que permitem que o nó contendo 12 se torne o novo topo da pilha.

Pilhas

- A função pop (linhas 95-105) remove um nó do topo da pilha.
- A função main determina se a pilha está vazia antes de chamar pop.
- A operação pop consiste em cinco etapas:
  - Atribuir *topPtr a tempPtr (linha100); tempPtr será usado para liberar a memória desnecessária.
  - Atribua (*topPtr)->data a popValue (linha 101) para salvar o valor no nó superior.
  - Atribua (*topPtr)->nextPtr a *topPtr (linha 102) para que *topPtr contenha o endereço do novo nó do topo.
  - Liberar a memória apontada por tempPtr (linha 103).
  - Retornar popValue para quem chamou (linha 104).

Pilhas

- A Figura 12.11 ilustra a função pop.
- A parte (a) mostra a pilha após a operação push anterior.
- A parte (b) mostra tempPtr apontando para o primeiro nó da pilha e topPtr apontando para o segundo nó da pilha.
- A função free é usada para liberar a memória apontada por tempPtr.

Pilhas

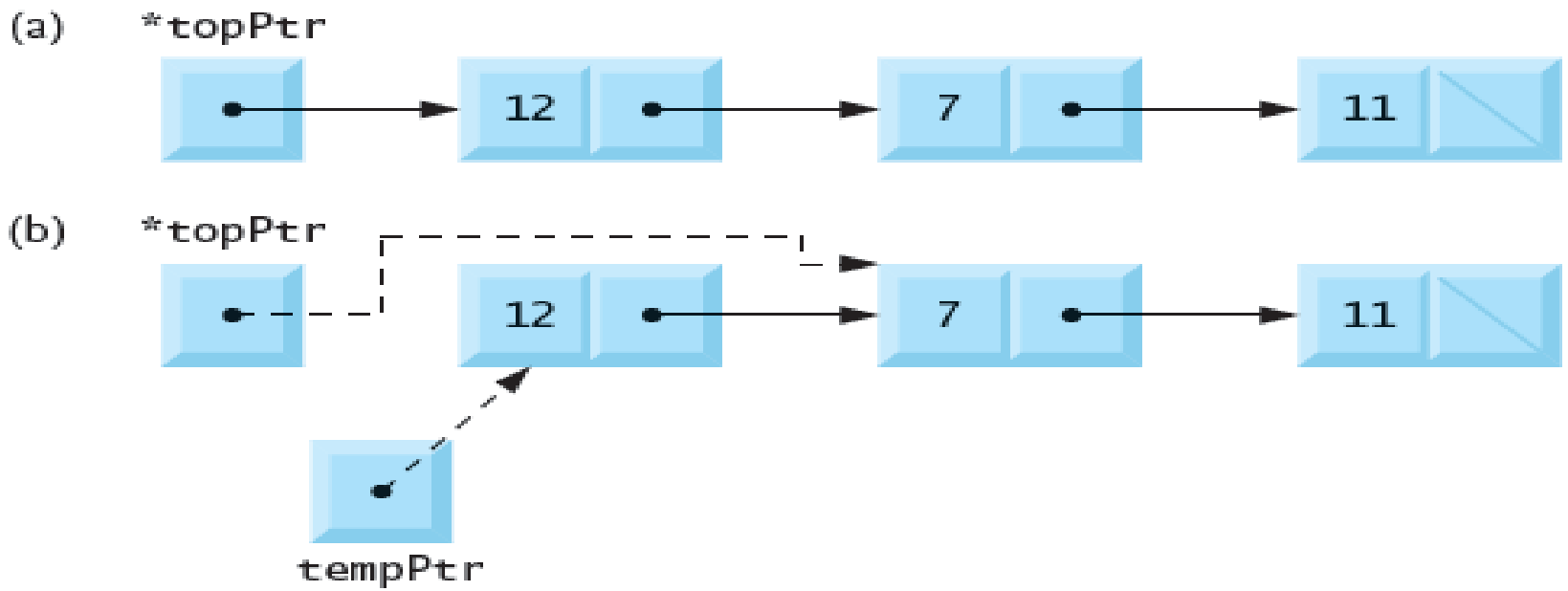

Figura 12.11 ■ Operação pop.

Pilhas

- As pilhas possuem muitas aplicações interessantes.
- Por exemplo, sempre que é feita uma chamada de função, a função chamada precisa saber como retornar para a função ou programa que a chamou, de modo que o endereço de retorno é colocado em uma pilha.
- Se ocorre uma série de chamadas de função, os valores de retorno sucessivos são colocados na pilha na ordem ‘último a entrar, primeiro a sair’, de modo que cada função pode retornar para a função ou programa que a chamouer.

Pilhas

- As pilhas dão suporte a chamadas de função recursivas da mesma maneira que as chamadas não recursivas convencionais.
- As pilhas contêm o espaço criado para variáveis automáticas em cada chamada de função.
- Quando a função retorna para a função ou programa que a chamou, o espaço para as variáveis automáticas dessa função é removido da pilha, e essas variáveis passam a ser desconhecidas pelo programa.
- As pilhas são usadas pelos compiladores no processo de avaliação de expressões e geração de código em linguagem de máquina.

Pilhas

- Outra estrutura de dados comum é a **fila**.
- Uma fila é semelhante a uma fila no caixa de um supermercado — a primeira pessoa da fila é atendida primeiro, e outros clientes entram no final da fila e esperam para serem atendidos.
- Os nós de fila são removidos somente a partir da **cabeça da fila** e são inseridos apenas na **cauda da fila**.
- Por essa razão, uma fila é chamada de estrutura de dados **primeiro a entrar, primeiro a sair** (**FIFO — first-in, first-out**).
- As operações de inserção e retirada são conhecidas como enqueue e dequeue.

Pilhas

- As filas têm muitas aplicações em sistemas de computadores.
- A maioria dos computadores tem um único processador e, portanto, somente um usuário pode ser atendido a cada vez.
- Os pedidos dos outros usuários são colocados em uma fila.
- Cada pedido avança gradualmente para a frente da fila à medida que os usuários são atendidos.
- O pedido no início da fila é o próximo a ser atendido.

Pilhas

- As filas também são usadas para suportar a impressão em spool.
- Um ambiente multiusuário pode ter somente uma impressora.
- Muitos usuários podem estar gerando saídas para serem impressas.
- Se a impressora estiver ocupada, outras saídas ainda poderão ser geradas.
- Elas são postas em um spool no disco, onde esperam em uma fila até que a impressora se torne disponível.

Pilhas

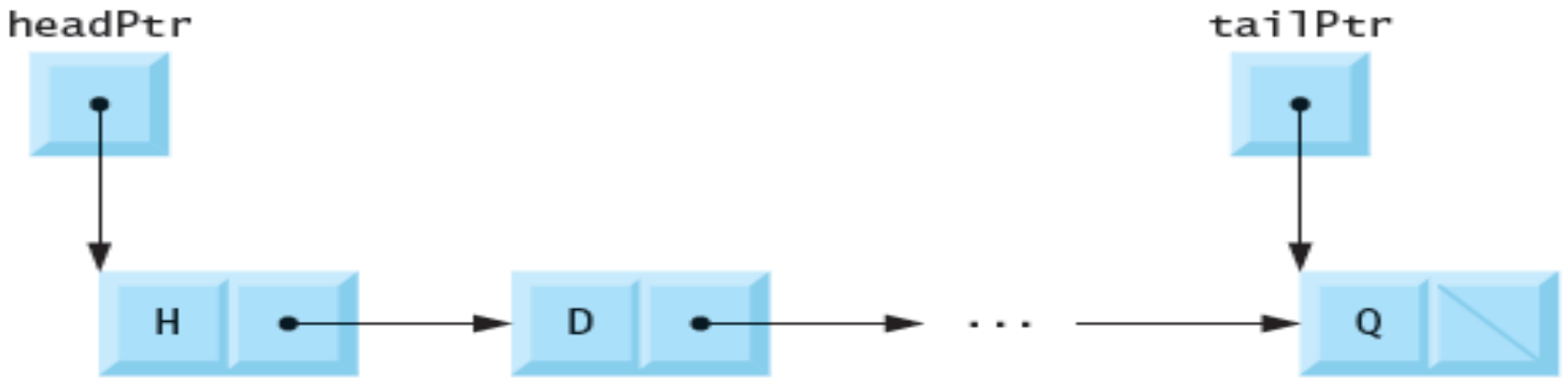

Figura 12.12 ■ Representação gráfica da fila.

Pilhas

- Pacotes de informações também esperam em filas em redes de computador.
- Toda vez que um pacote chega em um nó de rede, ele deve ser direcionado para o próximo nó na rede ao longo do caminho para o destino final do pacote.
- O nó de direcionamento encaminha um pacote de cada vez, de modo que pacotes adicionais são enfileirados até que o direcionador possa encaminhá-los.
- A Figura 12.12 ilustra uma fila com vários nós.
- Observe os ponteiros para a cabeça da fila e para a cauda da fila.

Erro comum de programação 12.7
Não inicializar o link no último nó de uma fila com NULL *pode provocar erros de runtime.*

Pilhas

- A Figura 12.13 (saída na Figura 12.14) realiza manipulações de fila.
- O programa oferece várias opções: inserir um nó na fila (função enqueue), remover um nó da fila (função dequeue) e terminar o programa.

Pilhas

```
1  /* Fig. 12.13: fig12_13.c
2     Operando e mantendo uma fila */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* estrutura autorreferenciada */
7  struct queueNode {
8     char data; /* define dados como char */
9     struct queueNode *nextPtr; /* queueNode pointer */
10 }; /* fim da estrutura queueNode */
11
12 typedef struct queueNode QueueNode;
13 typedef QueueNode *QueueNodePtr;
```

Figura 12.13 ■ Processamento de uma fila. (Parte 1 de 4.)

Pilhas

```
14
15  /* protótipos de função */
16  void printQueue( QueueNodePtr currentPtr );
17  int isEmpty( QueueNodePtr headPtr );
18  char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
19  void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
20     char value );
21  void instructions( void );
22
23  /* função main inicia execução do programa */
24  int main( void )
25  {
26     QueueNodePtr headPtr = NULL; /* inicializa headPtr */
27     QueueNodePtr tailPtr = NULL; /* inicializa tailPtr */
28     int choice; /* escolha de menu do usuário */
29     char item; /* entrada char pelo usuário */
30
31     instructions(); /* exibe o menu */
32     printf( "? " );
33     scanf( "%d", &choice );
34
35     /* enquanto usuário não digita 3 */
36     while ( choice != 3 ) {
37
38        switch( choice ) {
39           /* enfileira valor */
40           case 1:
41              printf( "Digite um caractere: " );
42              scanf( "\n%c", &item );
43              enqueue( &headPtr, &tailPtr, item );
44              printQueue( headPtr );
45              break;
```

Pilhas

```
46            /* desenfileira valor */
47            case 2:
48               /* se fila não estiver vazia */
49               if ( !isEmpty( headPtr ) ) {
50                  item = dequeue( &headPtr, &tailPtr );
51                  printf( "%c saiu da fila.\n", item );
52               } /* fim do if */
53
54               printQueue( headPtr );
55               break;
56            default:
57               printf( "Escolha inválida.\n\n" );
58               instructions();
59               break;
60         } /* fim do switch */
61
62         printf( "? " );
63         scanf( "%d", &choice );
64      } /* fim do while */
65
66      printf( "Fim da execução.\n" );
67      return 0; /* indica conclusão bem-sucedida */
68   } /* fim do main */
69
```

Figura 12.13 ■ Processamento de uma fila. (Parte 2 de 4.)

Pilhas

```
70  /* exibe instruções do programa ao usuário */
71  void instructions( void )
72  {
73     printf ( "Digite sua escolha:\n"
74              "   1 para incluir um item na fila\n"
75              "   2 para remover um item da fila\n"
76              "   3 para encerrar\n" );
77  } /* fim da função instructions */
78
79  /* insere um nó na cauda da fila */
80  void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
81     char value )
82  {
83     QueueNodePtr newPtr; /* ponteiro para novo nó */
84
85     newPtr = malloc( sizeof( QueueNode ) );
86
87     if ( newPtr != NULL ) { /* se houver espaço disponível */
88        newPtr->data = value;
89        newPtr->nextPtr = NULL;
90
91        /* se vazia, insere nó na cabeça */
92        if ( isEmpty( *headPtr ) ) {
93           *headPtr = newPtr;
94        } /* fim do if */
95        else {
96           ( *tailPtr )->nextPtr = newPtr;
97        } /* fim do else */
98
99        *tailPtr = newPtr;
100    } /* fim do if */
101    else {
```

```
102          printf( "%c não inserido. Não há memória disponível.\n", value );
103       } /* fim do else */
104    } /* fim da função enqueue */
105
106    /* remove nó da cabeça da fila */
107    char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
108    {
109       char value; /* valor do nó */
110       QueueNodePtr tempPtr; /* ponteiro de nó temporário */
111
112       value = ( *headPtr )->data;
113       tempPtr = *headPtr;
114       *headPtr = ( *headPtr )->nextPtr;
115
116       /* se a fila estiver vazia */
117       if ( *headPtr == NULL ) {
118          *tailPtr = NULL;
119       } /* fim do if */
120
121       free( tempPtr );
122       return value;
123    } /* fim da função dequeue */
124
125    /* Retorna 1 se a lista estiver vazia; caso contrário, retorna 0 */
```

Figura 12.13 ■ Processamento de uma fila. (Parte 3 de 4.)

Pilhas

```
126  int isEmpty( QueueNodePtr headPtr )
127  {
128     return headPtr == NULL;
129  } /* fim da função isEmpty */
130
131  /* Imprime a fila */
132  void printQueue( QueueNodePtr currentPtr )
133  {
134     /* se a fila estiver vazia */
135     if ( currentPtr == NULL ) {
136        printf( "A fila está vazia.\n\n" );
137     } /* fim do if */
138     else {
139        printf( "A fila é:\n" );
140
141        /* enquanto não for fim da fila */
142        while ( currentPtr != NULL ) {
143           printf( "%c --> ", currentPtr->data );
144           currentPtr = currentPtr->nextPtr;
145        } /* fim do while */
146
147        printf( "NULL\n\n" );
148     } /* fim do else */
149  } /* fim da função printQueue */
```

Figura 12.13 ■ Processamento de uma fila. (Parte 4 de 4.)

Pilhas

```
Digite sua escolha:
  1 para incluir um item na fila
  2 para remover um item da fila
  3 para encerrar
? 1
Digite um caractere: A
A fila é:
A --> NULL

? 1
Digite um caractere: B
A fila é:
A --> B --> NULL

? 1
Digite um caractere: C
A fila é:
A --> B --> C --> NULL

? 2
A saiu da fila.
A fila é:
B --> C --> NULL
```

Figura 12.14 ■ Exemplo de saída do programa da Figura 12.13. (Parte 1 de 2.)

Pilhas

```
? 2
B saiu da fila.
A fila é:
C --> NULL

? 2
C saiu da fila.
A fila está vazia.

? 2
A fila está vazia.

? 4
Escolha inválida.

Digite sua escolha:
  1 para incluir um item na fila
  2 para remover um item da fila
  3 para encerrar
? 3
Fim da execução.
```

Figura 12.14 ■ Exemplo de saída do programa da Figura 12.13. (Parte 2 de 2.)

Pilhas

- **A função enqueue (linhas 80-104) recebe três argumentos de main: o endereço do ponteiro para a cabeça da fila, o endereço do ponteiro para a cauda da fila e o valor a ser inserido na fila.**

Pilhas

- **A função consiste em três etapas:**
  - **Criação de um novo nó: chame malloc, atribua o local de memória alocado a newPtr (linha 85), atribua o valor a ser inserido na fila a newPtr->data (linha 88) e atribua NULL a newPtr->nextPtr (linha 89).**
  - **Se a fila estiver vazia (linha 92), atribua newPtr a *headPtr (linha 93); caso contrário, atribua ponteiro newPtr a (*tailPtr)->nextPtr (linha 96).**
  - **Atribua newPtr a *tailPtr (linha 99).**

Pilhas

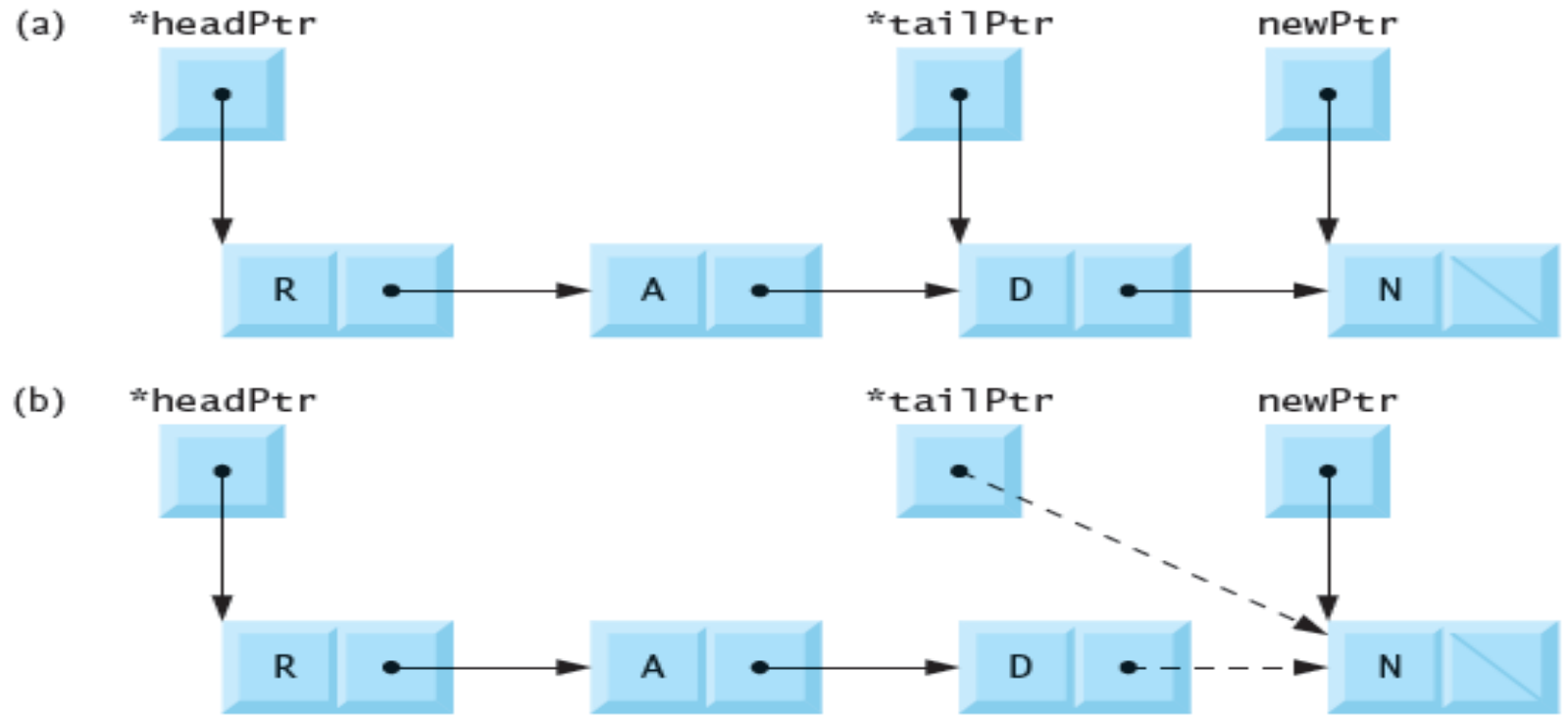

Figura 12.15 ■ Operação enqueue.

Pilhas

- A Figura 12.15 ilustra uma operação enqueue.
- A parte (a) mostra a fila e o novo nó antes da operação.
- As setas tracejadas na parte (b) ilustram as Etapas 2 e 3 da função enqueue que permitem que o novo nó seja incluído no final de uma fila que não está vazia.

Pilhas

- **A função dequeue (linhas 107-123) recebe o endereço do ponteiro para a cabeça da fila e o endereço do ponteiro para a cauda da fila como argumentos, e remove o primeiro nó da fila..**

Pilhas

- **A operação dequeue consiste em seis etapas:**
  - **Atribuir (*headPtr)->data a value para salvar os dados (linha 112).**
  - **Atribuir *headPtr a tempPtr (linha 113), que será usado para liberar (free) a memória desnecessária.**
  - **Atribuir(*headPtr)->nextPtr a *headPtr (linha 114), para que *headPtr agora aponte para o novo primeiro nó na fila.**
  - **Se *headPtr for NULL (linha 117), atribuir NULL a *tailPtr (linha 118).**
  - **Liberar a memória apontada por tempPtr (linha 121).**
  - **Retornar value a quem chamou (linha 122).**

Pilhas

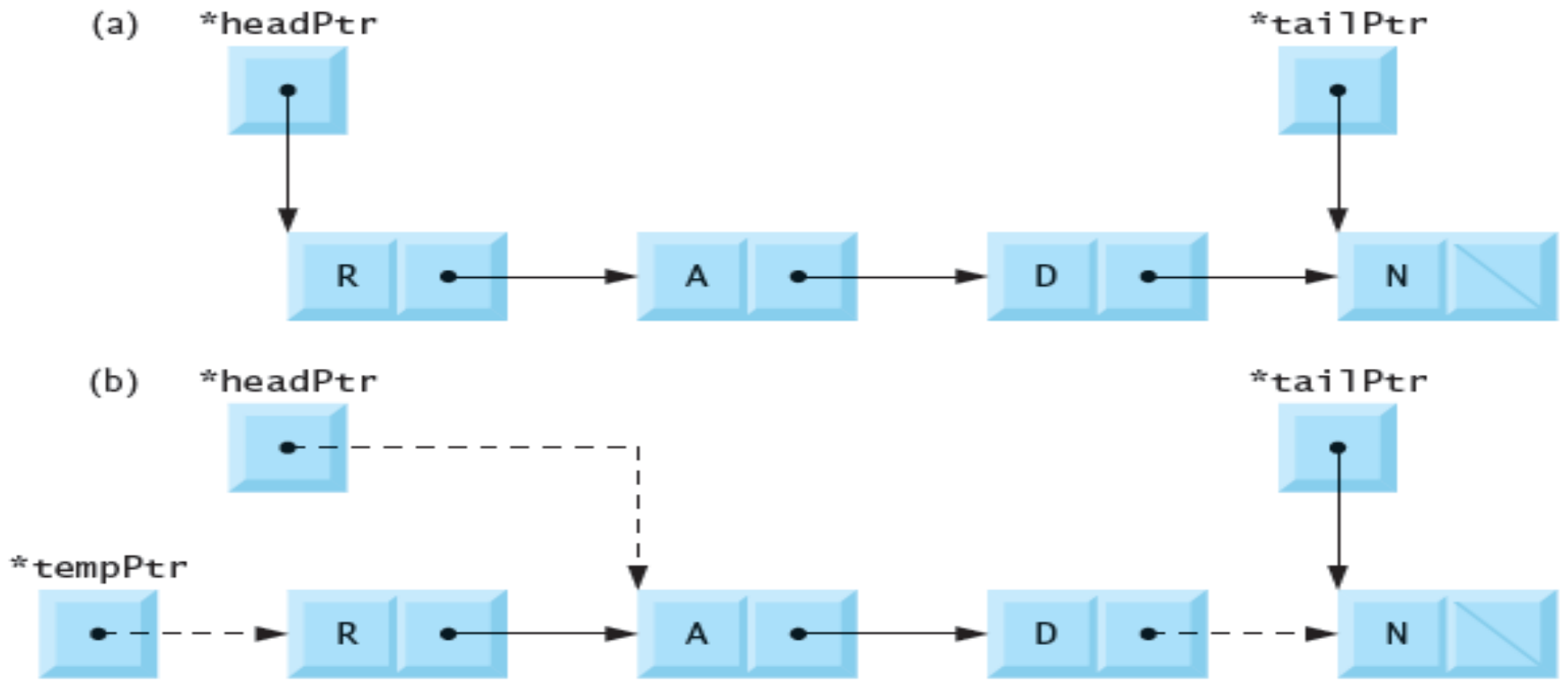

Figura 12.16 ■ Operação dequeue.

Pilhas

- A Figura 12.16 ilustra a função dequeue.
- A parte (a) mostra a fila após a operação enqueue anterior.
- A parte (b) mostra tempPtr apontando para o nó desenfileirado, e headPtr apontando para o novo primeiro nó da fila.
- A função free é usada para recuperar a memória apontada por tempPtr.

Árvores

- Listas encadeadas, pilhas e filas são estruturas de dados lineares.
- Uma árvore é uma estrutura de dados não linear, bidimensional, com propriedades especiais.
- Três nós contêm dois ou mais links.

Árvores

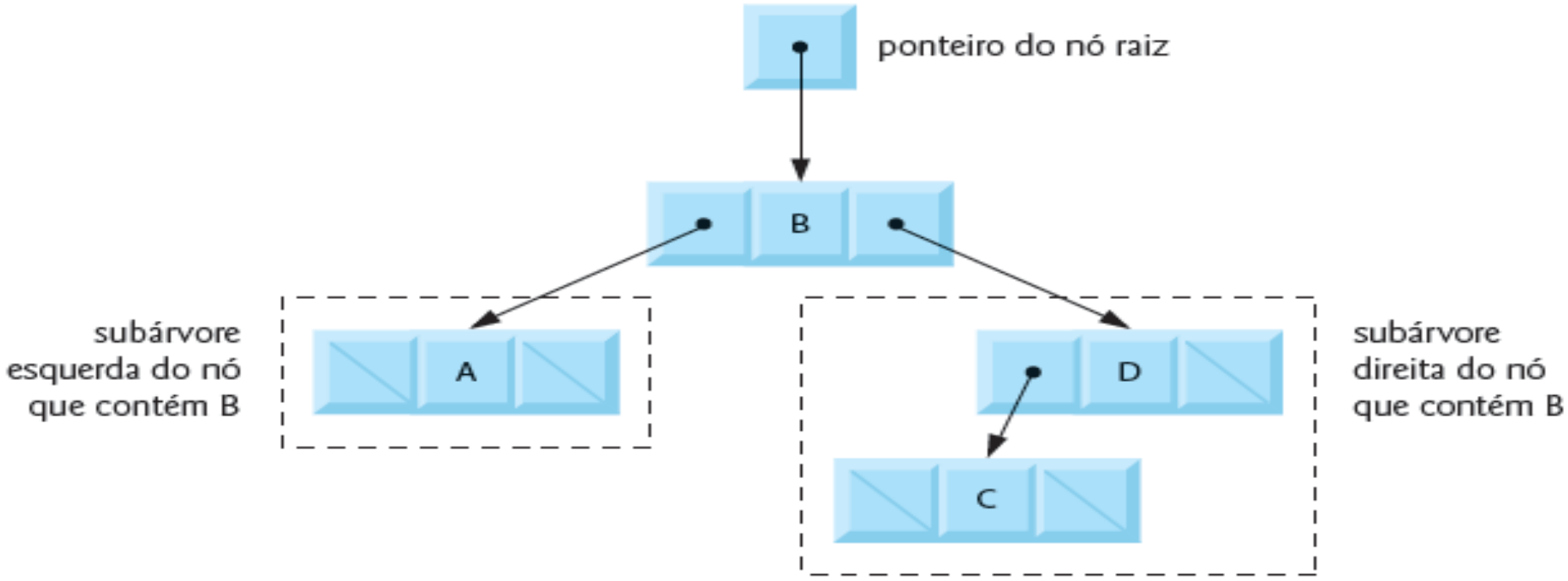

Figura 12.17 ■ Representação gráfica da árvore binária.

Árvores

- Esta seção discute as árvores binárias (Figura 12.17) — árvores em que todos os nós contêm dois links (dos quais um, ambos ou nenhum podem ser NULL).
- O nó raiz é o primeiro nó em uma árvore.
- Cada link no nó raiz refere-se a um filho.
- O filho esquerdo é o primeiro nó da subárvore esquerda, e o filho direito é o primeiro nó da subárvore direita.
- Os filhos de um mesmo nó são chamados de irmãos.
- Um nó sem filhos é chamado de nó folha.
- Os cientistas da computação normalmente desenham árvores do nó raiz para baixo — exatamente o oposto das árvores na natureza.

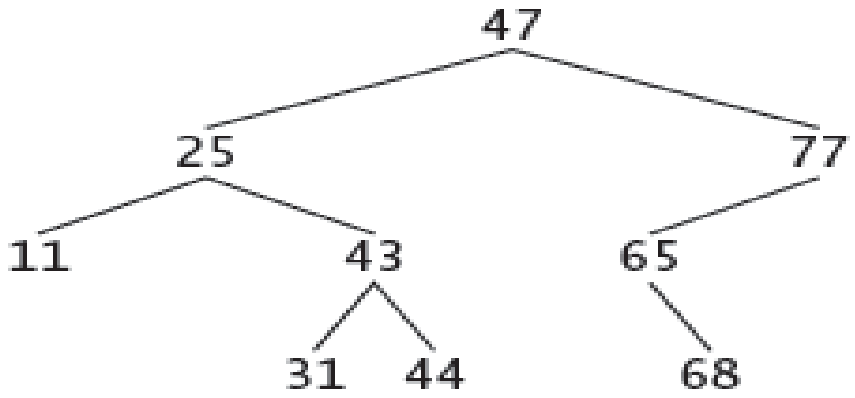

Figura 12.18 ■ Árvore binária de busca.

Árvores

- Nesta seção, é criada uma árvore binária especial, chamada de **árvore binária de busca**.
- Caracteristicamente, em uma árvore binária de busca (sem valores de nó duplicados), os valores em qualquer subárvore esquerda são menores do que o valor em seu **nó pai**, e os valores em qualquer subárvore direita são maiores que o valor em seu nó pai.
- A Figura l2.18 ilustra uma árvore binária de busca com 12 valores.
- A forma da árvore binária de busca que corresponde a um conjunto de dados pode variar, dependendo da ordem em que os valores são inseridos na árvore.

Erro comum de programação 12.8
Não definir como NULL os links nos nós folha de uma árvore pode provocar erros de tempo de execução.

Árvores

- A Figura 12.19 (saída mostrada na Figura 12.20) cria uma árvore binária de busca e a atravessa de três maneiras: em ordem, pré-ordem e pós-ordem.
- O programa gera 10 números aleatórios e insere cada um na árvore, exceto que os valores duplicados são descartados.

Árvores

```
1  /* Fig. 12.19: fig12_19.c
2     Cria uma árvore binária e a atravessa em
3     pré-ordem, em ordem e pós-ordem */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  /* estrutura autorreferenciada */
9  struct treeNode {
10    struct treeNode *leftPtr; /* ponteiro para subárvore esquerda */
11    int data; /* valor do nó */
12    struct treeNode *rightPtr; /* ponteiro para subárvore direita */
13 }; /* fim da estrutura treeNode */
14
15 typedef struct treeNode TreeNode; /* sinônimo para struct treeNode */
```

Figura 12.19 ■ Criação e travessia de uma árvore binária. (Parte 1 de 3.)

Árvores

```
16  typedef TreeNode *TreeNodePtr; /* sinônimo para TreeNode* */
17
18  /* protótipos */
19  void insertNode( TreeNodePtr *treePtr, int value );
20  void inOrder( TreeNodePtr treePtr );
21  void preOrder( TreeNodePtr treePtr );
22  void postOrder( TreeNodePtr treePtr );
23
24  /* função main inicia execução do programa */
25  int main( void )
26  {
27     int i; /* contador para loop de 1 a 10 */
28     int item; /* variável para manter valores aleatórios */
29     TreeNodePtr rootPtr = NULL; /* árvore inicialmente vazia */
30
31     srand( time( NULL ) );
32     printf( "Os números sendo colocados na árvore são:\n" );
33
34     /* insere valores aleatórios entre 0 e 14 na árvore */
35     for ( i = 1; i <= 10; i++ ) {
36        item = rand() % 15;
37        printf( "%3d", item );
38        insertNode( &rootPtr, item );
39     } /* fim laço for */
40
41     /* atravessa a árvore preOrder */
42     printf( "\n\nA travessia na pré-ordem é:\n" );
43     preOrder( rootPtr );
44
45     /* atravessa a árvore inOrder */
46     printf( "\n\nA travessia na ordem é:\n" );
47     inOrder( rootPtr );
48
```

```
49       /* atravessa a árvore postOrder */
50       printf( "\n\nA travessia na pós-ordem é:\n" );
51       postOrder( rootPtr );
52       return 0; /* indica conclusão bem-sucedida */
53    } /* fim do main */
54
55    /* insere nó na árvore */
56    void insertNode( TreeNodePtr *treePtr, int value )
57    {
58       /* se árvore estiver vazia */
59       if ( *treePtr == NULL ) {
60          *treePtr = malloc( sizeof( TreeNode ) );
61
62          /* se a memória foi alocada, então atribui dados */
63          if ( *treePtr != NULL ) {
64             ( *treePtr )->data = value;
65             ( *treePtr )->leftPtr = NULL;
66             ( *treePtr )->rightPtr = NULL;
67          } /* fim do if */
68          else {
69             printf( "%d não inserido. Não há memória disponível.\n", value );
70          } /* fim do else */
71       } /* fim do if */
72       else { /* árvore não está vazia */
73          /* dado a inserir é menor que dado no nó atual */
74          if ( value < ( *treePtr )->data ) {
75          insertNode( &( ( *treePtr )->leftPtr ), value );
76          } /* fim do if */
77
```

Figura 12.19 ■ Criação e travessia de uma árvore binária. (Parte 2 de 3.)

Árvores

```
78          /* dado a inserir é maior que dado no nó atual */
79          else if ( value > ( *treePtr )->data ) {
80             insertNode( &( ( *treePtr )->rightPtr ), value );
81          } /* fim do else if */
82          else { /* valor de dado duplicado é ignorado */
83             printf( "dup" );
84          } /* fim do else */
85       } /* fim do else */
86    } /* fim da função insertNode */
87
88    /* inicia travessia da árvore na ordem */
89    void inOrder( TreeNodePtr treePtr )
90    {
91       /* se árvore não está vazia, então atravessa */
92       if ( treePtr != NULL ) {
93          inOrder( treePtr->leftPtr );
94          printf( "%3d", treePtr->data );
95          inOrder( treePtr->rightPtr );
96       } /* fim do if */
97    } /* fim da função inOrder */
98
99    /* inicia travessia da árvore na pré-ordem */
100   void preOrder( TreeNodePtr treePtr )
101   {
```

```
102      /* se a árvore não está vazia, então atravessa */
103      if ( treePtr != NULL ) {
104         printf( "%3d", treePtr->data );
105         preOrder( treePtr->leftPtr );
106         preOrder( treePtr->rightPtr );
107      } /* fim do if */
108   } /* fim da função preOrder */
109
110   /* inicia travessia da árvore na pós-ordem */
111   void postOrder( TreeNodePtr treePtr )
112   {
113      /* se a árvore não está vazia, então atravessa */
114      if ( treePtr != NULL ) {
115         postOrder( treePtr->leftPtr );
116         postOrder( treePtr->rightPtr );
117         printf( "%3d", treePtr->data );
118      } /* fim do if */
119   } /* fim da função postOrder */
```

Figura 12.19 ■ Criação e travessia de uma árvore binária. (Parte 3 de 3.)

```
Os números colocados na árvore são:
 6 7 4 12 7dup 2 2dup 5 7dup 11

A travessia na pré-ordem é:
 6 4 2 5 7 12 11

A travessia na ordem é:
 2 4 5 6 7 11 12

A travessia na pós-ordem é:
 2 5 4 11 12 7 6
```

Figura 12.20 ■ Exemplo de saída do programa da Figura 12.19.

Árvores

- **As funções usadas na Figura 12.19 para criar uma árvore binária de busca e atravessar a árvore são recursivas.**
- **A função insertNode (linhas 56-86) recebe o endereço da árvore e um inteiro para serem armazenados na árvore como argumentos.**
- ***Um nó somente pode ser inserido como um nó folha em uma árvore binária de busca.***

Árvores

- As etapas de inserção de um nó em uma árvore binária de busca são as seguintes:
  - Se *treePtr é NULL (linha 59), crie um novo nó (linha 60). Chame malloc, atribua a memória alocada a *treePtr, atribua a (*treePtr)->data o inteiro a ser armazenado (linha 64), atribua a (*treePtr)->leftPtr e (*treePtr)->rightPtr o valor NULL (linhas 65-66) e retorne o controle a quem chamou main ou uma chamada anterior a insertNode).
  - Se o valor de *treePtr não for NULL e o valor a ser inserido for menor que (*treePtr)->data, a função insertNode é chamada com o endereço de (*treePtr)->leftPtr (linha 75). Se o valor a ser inserido for maior que (*treePtr)->data, a função insertNode é chamada com o endereço de (*treePtr)->rightPtr (linha 80). Caso contrário, as etapas recursivas continuam até que um ponteiro NULL seja encontrado, depois a Etapa 1 é executada para que se insira o novo nó.

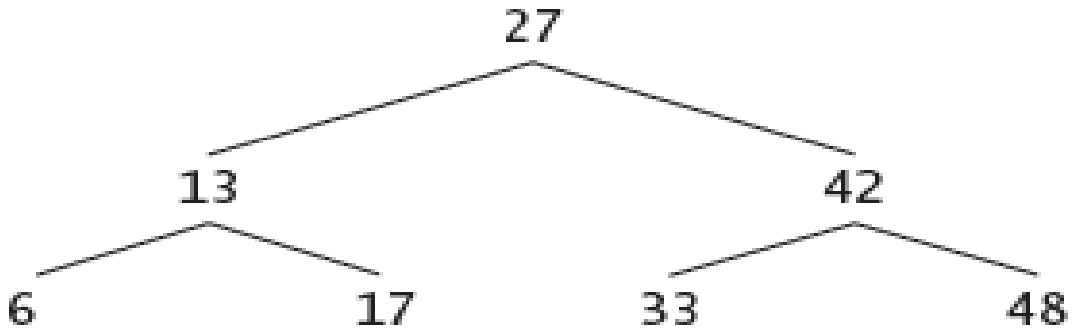

Figura 12.21 ■ Árvore binária de busca com sete nós.

Árvores

- As funções inOrder (linhas 89-97), preOrder (linhas 100-108) e postOrder (linhas 111-119) recebem uma árvore cada uma (ou seja, o ponteiro para o nó raiz da árvore) e atravessam a árvore.
- As etapas para uma travessia:
  - Atravessar a subárvore esquerda inOrder.
  - Processar o valor no nó.
  - Atravessar a subárvore direita inOrder.
- O valor em um nó não é processado até que os valores em sua subárvore esquerda sejam processados.
- A travessia inOrder da árvore na Figura 12.21 é:
  - 6 13 17 27 33 42 48

Árvores

- **A travessia inOrder de uma árvore binária de busca imprime os valores de nó em ordem crescente.**
- **Na verdade, o processo de criar uma árvore binária de busca classifica os dados — e, assim, esse processo é chamado classificação por árvore de busca binária.**

Árvores

- As etapas para uma travessia preOrder são:
  - Processar o valor no nó.
  - Atravessar a subárvore esquerda preOrder.
  - Atravessar a subárvore direita preOrder..
- O valor em cada nó é processado à medida que o nó é visitado.
- Após o valor em determinado nó ser processado, os valores na subárvore esquerda são processados, e depois os valores na subárvore direita são processados.

Árvores

- A travessia preOrder da árvore na Figura 12.21 é:
  - 27 13 6 17 42 33 48
- As etapas para uma travessia postOrder são:
  - Atravessar a subárvore esquerda postOrder.
  - Atravessar a subárvore direita postOrder.
  - Processar o valor no nó.
- O valor em cada nó não é impresso até que os valores de seus filhos sejam impressos.
- A travessia postOrder da árvore na Figura 12.21 é:
  - 6 17 13 33 48 42 27

Árvores

- A árvore binária de busca facilita a eliminação de duplicatas.
- Enquanto a árvore está sendo criada, uma tentativa de inserir um valor duplicado será reconhecida, porque uma duplicata seguirá as mesmas decisões 'ir para a esquerda' e 'ir para a direita' em cada comparação, assim como fez o valor original.
- Dessa maneira, a duplicata finalmente será comparada com um nó na árvore que contém o mesmo valor.
- O valor duplicado pode ser, simplesmente, descartado nesse ponto.

Árvores

- **Procurar, em uma árvore binária, por um valor que combine com um valor de chave também é rápido.**
- **Se a árvore estiver balanceada, cada nível conterá cerca do dobro de elementos do nível anterior.**
- **Assim, uma árvore binária de busca com *n* elementos teria um máximo de $\log_2 n$ níveis e, portanto, um máximo de $\log_2 n$ comparações teriam de ser feitas para encontrar uma correspondência, ou para determinar que não existe correspondência.**
- **Isso significa, por exemplo, que quando se procura uma árvore binária de busca (balanceada) com 1.000 elementos, não mais que 10 comparações precisam ser feitas, pois $2^{10} > 1.000$.**

Árvores

- **Ao pesquisar uma árvore binária de busca (balanceada) de 1.000.000 elementos, não mais que 20 comparações precisam ser feitas, pois $2^{20} > 1.000.000$.**
- **A travessia por ordem de nível de uma árvore binária visita os nós da árvore linha por linha, começando no nível de nó raiz.**
- **Em cada nível da árvore, os nós são visitados da esquerda para a direita.**

Lembre-se

“ Faça funcionar, faça certo, faça isso rápido. ”

Kent Beck