CAPÍTULO 11

ALOCAÇÃO DINÂMICA DE MEMÓRIA

11.1 Introdução

Uma variável de duração fixa tem memória reservada para si durante todo o tempo de execução do programa, enquanto uma variável de duração automática é alocada a cada vez que seu escopo é executado. Ambas as abordagens de alocação de memória, no entanto, assumem que o programador sabe, implicitamente, no instante da escrita do programa, que quantidade de memória será necessária para sua execução. Existem muitas situações, entretanto, em que a quantidade de memória necessária não pode ser determinada precisamente em tempo de programação. Isto é, freqüentemente, deseja-se ter a capacidade de alocar memória de acordo com a demanda apresentada durante a execução do programa. Um exemplo clássico de uma tal situação é o de um programa que permite múltiplos documentos abertos simultaneamente, como muitos processadores de texto. Um programa desses não estima, em princípio, quantos documentos o usuário irá abrir nem qual será o tamanho de cada documento aberto pelo usuário.

Outro exemplo no qual não é possível prever a quantidade de memória a ser utilizada é o de um programa para gerenciamento de lista de espera de uma companhia aérea. Neste caso, a quantidade de dados (i.e., registros de passageiros) pode variar bastante entre uma execução do programa e outra (ou, pelo menos, de uma temporada para outra). Por exemplo, num período de baixa temporada, pode não haver nenhum

passageiro em lista de espera, enquanto num período de alta temporada a quantidade de passageiros poderá atingir um número indeterminado.

Existem duas formas de abordagem para um programa como esse de lista de espera. A mais fácil, mas menos eficiente, consiste em prever um valor máximo de dados de entrada (por exemplo, 200 passageiros) e utilizar um array capaz de conter estes dados. No caso do programa de lista de espera, poder-se-ia ter as seguintes declarações:

```
#define  MAXIMO_DE_PASSAGEIROS  200

typedef  struct  {
            char  nome[30];
            char  identidade[12];
            char  numeroDoBilhete[10];
         } tPassageiro;

tPassageiro  listaDeEspera[MAXIMO_DE_PASSAGEIROS];
```

Existem dois problemas com esta abordagem. Primeiro, o programador deve estabelecer um máximo arbitrário e isto não é bom, porque, talvez, mais adiante, este número precise ser acrescido (implicando a devida recompilação do programa). O segundo e maior problema desta abordagem é que quanto maior for o máximo estipulado, maior será o desperdício de memória em execuções do programa que não utilizem todo este valor. Por exemplo, num período de baixa estação, onde houvesse, diga-se, no máximo, 5 passageiros em lista de espera, o programa do último exemplo estaria desperdiçando, em média, 195 vezes o espaço em memória necessário para conter um elemento do tipo `tPassageiro`. Além disso, durante esse período de subutilização de memória, haveria também desperdício de tempo para a alocação de memória que não seria efetivamente utilizada durante aquele período.

A melhor solução para problemas como este, onde a quantidade de dados pode variar drasticamente entre várias execuções de um programa, é a alocação de memória durante a execução do programa de acordo com a demanda. Este tipo de alocação é denominado **alocação dinâmica de memória** e contrasta com qualquer outro tipo de alocação

de memória visto até aqui, denominado **alocação estática de memória**, cujo espaço a ser alocado é conhecido em tempo de programação (i.e., antes mesmo de o programa ser executado).

Tipos de dados cujas variáveis são alocadas estaticamente são denominados **tipos de dados estáticos**, enquanto tipos de dados cujas variáveis são alocadas dinamicamente são denominados **tipos de dados dinâmicos**. Todos os tipos de dados vistos até aqui são estáticos. Um exemplo de tipo de dados dinâmico (i.e., cujos tamanhos das variáveis podem aumentar ou diminuir durante a execução do programa) são as listas encadeadas, que serão vistas na **Seção 11.6**.

11.2 Funções de Alocação Dinâmica de Memória

A alocação dinâmica de memória em C é feita por meio de ponteiros e das quatro funções de biblioteca resumidas na **Tabela 35**. Para utilizar estas funções, inclua o arquivo `<stdlib.h>`.

Função	Descrição Resumida
malloc()	Aloca um número especificado de bytes em memória e retorna um ponteiro para o início do bloco de memória alocado.
calloc()	Similar a **malloc()**, mas esta função inicia todos os bytes alocados com zeros. Ela também permite a alocação de memória para mais de um bloco numa mesma chamada.
realloc()	Modifica o tamanho de um bloco previamente alocado dinamicamente.
free()	Libera o espaço de um bloco de memória previamente alocado com **malloc()**, **calloc()** ou **realloc()**.

Tabela 35: Funções de alocação dinâmica de memória

Essas funções serão exploradas em profundidade a seguir, mas, antes de prosseguir, é importante introduzir o conceito de **bloco de memória** (ou, simplesmente, **bloco**), que é fartamente utilizado em menções a alocação dinâmica de memória. Neste contexto, um bloco é simplesmente um array

unidimensional de bytes, e este conceito é utilizado para enfatizar que todos os bytes que compõem um bloco ocupam posições contíguas em memória.

O tipo de dados utilizado para especificar tamanhos de blocos de memória, requeridos pelas funções de alocação de memória, é **size_t**, definido no arquivo `<stddef.h>`. Este tipo, que usualmente é definido como sendo **unsigned int**, **unsigned long** ou **unsigned long long**, também é o tipo do valor resultante da aplicação do operador **sizeof** (v. **Seção 1.6.5**).

11.2.1 A Função malloc()

A função **malloc()**, cujo protótipo é apresentado a seguir, recebe como argumento o tamanho, em bytes, do bloco a ser dinamicamente alocado e retorna o endereço inicial deste bloco.

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

Usualmente, o argumento `tamanho` envolve o uso do operador **sizeof**, cujo uso é recomendado, principalmente, por questões de portabilidade Por exemplo, supondo que `ptrPassageiro` seja um ponteiro para o tipo `tPassageiro` definido na **Seção 11.1**, então a chamada da função **malloc()** a seguir:

```
ptrPassageiro = malloc(sizeof(tPassageiro));
```

alocaria (se fosse possível) um bloco capaz de conter uma estrutura do tipo `tPassageiro` e retornaria o endereço inicial deste bloco.

Note que, de acordo com o protótipo acima, o tipo de retorno de **malloc()** é **void ***, que representa um tipo não apresentado até então. Este é o tipo dos ponteiros genéricos, que serão formalmente descritos na **Seção 11.4**. Um ponteiro genérico pode ser atribuído a qualquer ponteiro sem que seja necessário o uso de conversão explícita, conforme mostra o exemplo de uso de **malloc()** apresentado acima.

Quando a função **malloc()** não consegue alocar espaço em memória para o bloco solicitado, ela retorna **NULL**[1].

[1] Este assunto será discutido em profundidade mais adiante na Seção 11.3.

11.2.2 A Função calloc()

A função **calloc()** recebe dois argumentos: o primeiro é o número de blocos a ser alocados e o segundo é o tamanho de cada bloco. Seu protótipo é:

```
void *calloc(size_t nBlocos, size_t tamanho)
```

Quando possível, a função **calloc()** aloca o espaço necessário para conter os blocos requisitados e retorna o endereço inicial do primeiro bloco alocado. Todos os bits do espaço alocado são iniciados com zeros.

Apesar de não ser necessário, é instrutivo examinar como a função **calloc()** pode ser implementada utilizando **malloc()**.

```
void *MinhaCalloc(size_t nBlocos, size_t tamanho)
{
   unsigned long i, nBytes = nBlocos*tamanho;
   char          *ptr = malloc(nBytes);

   if (!ptr) /* Não foi possível alocar o espaço solicitado */
      return NULL;

      /* Aqui, o espaço já foi alocado */
      /* Resta apenas zerar os bytes   */
   for (i = 0; i < nBytes; ++i)
      ptr[i] = 0; /* Zera cada byte */

   return ptr; /* Trabalho completo */
}
```

Para entender a implementação da função `MinhaCalloc()` apresentada acima, note que esta função deve alocar um número de blocos determinado pelo parâmetro `nBlocos` e que o tamanho de cada bloco é dado pelo parâmetro `tamanho`. Ora, mas isto é equivalente a alocar um único bloco cujo tamanho é dado por:

```
nBlocos*tamanho
```

uma vez que não apenas os bytes de cada bloco são contíguos como também todos os blocos devem ser contíguos. A variável local `nBytes` é utilizada para

conter este valor e, embora não seja estritamente necessária, ela é utilizada como um fator de otimização da função, conforme será visto a seguir.

O ponteiro `ptr`, que representa o valor retornado por **malloc()** e que será posteriormente retornado pela função `MinhaCalloc()`, é definido com o tipo **char *** porque ele também será utilizado com o objetivo de zerar cada byte do bloco alocado[2]. O objetivo do laço **for** da função `MinhaCalloc()` é exatamente realizar a tarefa de zerar cada byte do bloco. Este laço também justifica o uso da variável `nBytes`; i.e., se esta variável não fosse utilizada, o produto `nBlocos*tamanho` teria que ser calculado a cada avaliação da condição de parada deste laço.

11.2.3 A Função realloc()

A função **realloc()** recebe dois argumentos. O primeiro deve ser um ponteiro para o início de um bloco de memória alocado utilizando **malloc()**, **calloc()** ou a própria função **realloc()** e o segundo especifica um novo tamanho desejado para o bloco. A função **realloc()** é tipicamente utilizada para redimensionar blocos previamente alocados dinamicamente e seu protótipo é:

```
void *realloc(void *ptr, size_t tamanho)
```

Se o novo tamanho, especificado pelo segundo argumento de **realloc()**, for menor do que o tamanho atual do bloco, a diferença será retirada do final do bloco. Isto é, a porção final do bloco, cujo tamanho é a diferença entre o tamanho original e o novo tamanho, será liberada. Ainda neste caso, o ponteiro retornado pela função apontará para o mesmo endereço que apontava anteriormente e a porção de memória que não foi liberada manterá seu conteúdo original. A situação na qual a função **realloc()** é chamada para reduzir o tamanho de um bloco é ilustrada na **Figura 28**.

[2] Lembre-se de que o tipo **char** ocupa exatamente um byte.

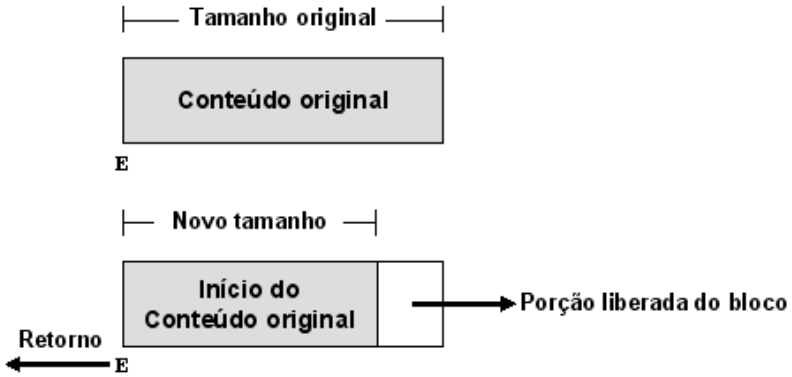

Figura 28: Função realloc(): O novo bloco é menor do que o bloco original

Se o novo tamanho for maior do que o tamanho atual do bloco, um novo bloco do tamanho especificado será alocado (se for possível), o conteúdo do bloco original será copiado para a porção inicial deste novo bloco, o bloco original será liberado e, finalmente, a função retornará um ponteiro para o novo bloco alocado. A porção final do novo bloco não será iniciada. Esta situação é ilustrada na **Figura 29**.

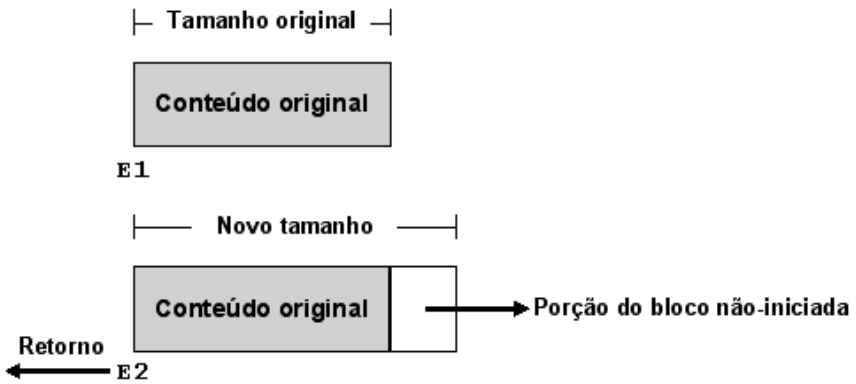

Figura 29: Função realloc(): O novo bloco é maior do que o bloco original

Se o novo tamanho for maior do que o tamanho atual do bloco e não for possível alocar um novo bloco, o bloco antigo manterá seu conteúdo e seu endereço originais e a função retornará **NULL**.

Se um ponteiro nulo for passado como primeiro argumento para a função **realloc()**, ela se comportará como **malloc()** para o tamanho especificado (segundo argumento). Se o novo tamanho for igual a zero e o ponteiro passado como primeiro argumento não for nulo, a chamada de **realloc()** se comportará como uma chamada da função **free()** (i.e., irá liberar o espaço em memória apontado pelo ponteiro). Estes duas últimas formas de utilização de **realloc()**, entretanto, são atípicas.

11.2.4 A Função free()

A função **free()** recebe como único argumento um ponteiro que aponta para uma posição de memória alocada utilizando **malloc()**, **calloc()** ou **realloc()** e libera este espaço em memória. Se o ponteiro passado para **free()** for nulo, a função retorna sem executar nada. O protótipo da função **free()** é:

```
void free(void *ptr)
```

Após uma chamada da função **free()**, você não deverá mais utilizar o ponteiro utilizado nesta chamada para acessar o espaço de memória liberado; caso contrário, seu programa poderá apresentar um comportamento indefinido.

É importante salientar que, apesar de um ponteiro ser considerado inválido após ser utilizado numa chamada da função **free()**, o sistema não é capaz de detectar esta situação. Portanto, sugere-se que sempre se atribua **NULL** a um ponteiro logo após ele ser utilizado numa chamada da função **free()**. Tendo em vista esta recomendação, alguns programadores definem a macro[3]:

```
#define FREE(x) do { free(x); x = NULL; } while(0)
```

e a utilizam em substituição à simples chamada da função **free()**.

A função **free()** deve ser chamada apenas com **NULL** ou um ponteiro que esteja correntemente apontando para o início de um bloco de memória alocado com alguma das funções de alocação descritas aqui. Este ponteiro

[3] O uso de do-while é um jargão utilizado para encapsular o corpo de uma macro numa única instrução (v. Seção 5.3.2).

não deve ter sido previamente liberado ou passado como parâmetro para **realloc()**. Caso contrário, o resultado será imprevisível[4].

11.3 O Heap

Heap é a partição da memória alocada para execução de um programa reservada para alocação dinâmica de memória[5]. A fragmentação de ***heap*** ocorre em conseqüência de várias alocações e liberações de blocos de tamanhos variados durante a execução do programa. Pode-se fazer uma analogia entre fragmentação de ***heap*** e a fragmentação que freqüentemente ocorre em meios de armazenamento, notadamente em discos rígidos.

De modo análogo à causa de fragmentação de ***heap***, a fragmentação de um disco rígido é causada pela criação e remoção freqüentes de arquivos de tamanhos diferentes. Entretanto, diferentemente do que ocorre em sistemas de arquivos, nos quais as partes que compõem um arquivo não precisam ser contíguas, os bytes que compõem um bloco devem ser contíguos. Assim, quando ocorre fragmentação de ***heap***, pode ser impossível alocar um bloco de memória contíguo, mesmo que a quantidade total de memória disponível no ***heap*** seja maior do que o tamanho deste bloco. Inclusive, alguns sistemas de gerenciamento de ***heap*** provêem esquemas de compactação que visam combater fragmentação, mas esta discussão está além do escopo deste texto.

Todas as funções de alocação de memória retornam um ponteiro nulo quando não é possível alocar a quantidade de memória requerida (devido, por exemplo, à fragmentação de ***heap***). Portanto, é considerada boa prática de programação testar o valor do ponteiro retornado com **NULL** antes de tentar utilizá-lo para acessar uma porção de memória que não se tem certeza se foi realmente alocada. Caso o ponteiro retornado tenha valor nulo, o programador deve tomar as devidas providências

---

[4] Esta recomendação, que também se aplica ao caso da função **realloc()**, deve-se ao fato de o sistema de gerenciamento de memória dinâmica ser capaz de reconhecer apenas ponteiros que apontem para o início de blocos alocados dinamicamente.

[5] Deve-se salientar que o padrão de C não especifica que deva existir heap ou mesmo pilha de execução, mas esta divisão de memória é comum em vários sistemas operacionais.

antes de prosseguir. Neste caso, talvez, o programa precise ser abortado ***graciosamente***[6] se o bloco de memória requisitado for crucial para o prosseguimento do processamento. Por exemplo:

```
ptrPassageiro = malloc(sizeof(tPassageiro));

if  (ptrPassageiro != NULL){
   /* O bloco foi alocado e pode ser acessado com segurança */
}
else {
   /* Aqui o programador deve tomar as providências cabíveis */
  /* quando não é possível alocar o espaço em memória necessário.
*/
   /* Talvez seja preciso abortar o programa, mas pode ser que */
   /* haja alternativa menos drástica, dependendo da situação. */
}
```

Note que o teste:

```
if  (ptrPassageiro != NULL)
```

pode ser escrito, de forma equivalente, como:

```
if  (ptrPassageiro)
```

Esta última forma é a preferida por muitos programadores de C. Outros programadores preferem, ainda, utilizar:

```
if    ((ptrPassageiro  =  malloc(sizeof(tPassageiro)))  !=
NULL) {
   ...
}
```

ou ainda:

```
if  (ptrPassageiro = malloc(sizeof(tPassageiro))) {
   ...
}
```

em vez de:

---

[6] Abortar um programa graciosamente significa causar voluntariamente sua interrupção, informando ao usuário qual a razão desta medida e desculpando-se pelo inconveniente. Provavelmente, o usuário não achará graça nenhuma nisso, mas este procedimento é melhor do que abortar o programa sem sequer despedir-se do usuário.

```
ptrPassageiro = malloc(sizeof(tPassageiro));

if  (ptrPassageiro != NULL){
   ...
}
```

Entretanto, em termos de estilo, não é recomendável misturar chamada de função com teste do valor retornado por ela numa única expressão.

A macro **NULL** é definida no arquivo de cabeçalho `<stddef.h>`. Portanto, para utilizá-la, inclua este arquivo no seu programa ou defina-a você mesmo (v. **Seção 3.2.4**).

11.4 Ponteiros Genéricos

Recorde-se que, em C, dois ponteiros são compatíveis apenas quando eles apontam para tipos de dados da mesma espécie. Em algumas bibliotecas muito antigas, **malloc()**, **calloc()** e **realloc()** retornam um ponteiro para o tipo **char**, em vez de um ponteiro para o tipo **void**. Portanto, usando-se estas funções obsoletas, deve-se fazer uma conversão explícita do tipo de retorno de uma chamada de uma destas funções para o tipo do ponteiro desejado. Se for o caso, a chamada de **malloc()** apresentada anteriormente:

```
ptrPassageiro = malloc(sizeof(tPassageiro));
```

deve ser substituída por:

```
ptrPassageiro = (tPassageiro *) malloc(sizeof(tPassageiro)
);
```

Em bibliotecas mais recentes, as funções de alocação dinâmica de memória retornam ponteiros de tipo genérico que não precisam de ***casting***.

O conceito de **ponteiro genérico** é o de um ponteiro compatível com ponteiros para quaisquer tipos de dados[7]. Sintaticamente, um

[7] NB: A compatibilidade entre ponteiros genéricos e outros tipos de ponteiros não inclui ponteiros para funções (v. Seção 10.5).

ponteiro genérico é um ponteiro para o tipo **void**. Por exemplo, o ponteiro `ponteiroGenerico` a seguir é um ponteiro genérico:

```
void   *ponteiroGenerico;
```

Ponteiros genéricos são normalmente utilizados em duas situações:

(1) como tipos de **retorno de funções** e
(2) como tipos de **argumentos de funções**.

No primeiro caso, ponteiros genéricos são utilizados para representar endereços retornados por funções que podem ser implicitamente convertidos para ponteiros de quaisquer tipos. Por exemplo, as funções de alocação de memória **malloc()**, **calloc()** e **realloc()** retornam ponteiros genéricos, e isto significa que os endereços retornados por estas funções podem ser atribuídos a ponteiros de quaisquer tipos sem a necessidade de conversão explícita, conforme visto anteriormente.

No segundo caso de uso de ponteiros genéricos, eles são utilizados para representar argumentos compatíveis com qualquer tipo de ponteiro. Um exemplo é a função **free()**, cujo protótipo foi apresentado na **Seção 11.2.4**. A definição dela permite a passagem de um ponteiro de qualquer tipo como parâmetro sem necessidade de conversão explícita.

11.5 Exemplo de Alocação Dinâmica de Memória

Voltando ao exemplo de gerenciamento de lista de espera descrito no início deste capítulo, um programa para tal finalidade poderia solicitar ao usuário o número máximo de elementos a ser alocados para o array `listaDeEspera[]`, conforme mostrado no trecho de programa a seguir:

```
typedef  struct  {
            char  nome[30];
            char  identidade[12];
            char  numeroDoBilhete[10];
         } tPassageiro;

int main(void)
{
   tPassageiro   *listaDeEspera;
```

```
   unsigned  int  numeroDePassageiros;

   ...

   printf( "Introduza o número máximo de passageiros "
           "na lista de espera: ");
   scanf("%d", &numeroDePassageiros);

    listaDeEspera = malloc(numeroDePassageiros*sizeof(tPas
sageiro));

   if (listaDeEspera) {
      ...      /* Processamento da lista  */
   }
}
```

No trecho de programa acima, `numeroDePassageiros` é uma variável que irá representar o tamanho do array alocado dinamicamente. Esta variável é lida com uma chamada de **scanf()** e, de posse do valor introduzido, o array é alocado com a chamada da função **malloc()**. Nesta última chamada, o espaço necessário para conter o array é especificado por:

```
numeroDePassageiros*sizeof(tPassageiro)
```

pois cada elemento ocupa um espaço, em bytes, dado por `sizeof(tPassageiro)`. Portanto, o espaço total a ser ocupado pelo array é dado pelo produto do número de elementos do array pelo espaço ocupado por cada elemento.

De acordo com o padrão C99, é possível conseguir um resultado semelhante ao obtido com o segmento de programa apresentado acima utilizando um array de tamanho variável (v. **Seção 7.7**). Neste caso, o fragmento de programa apresentado acima poderia ser reescrito como:

```
int main(void)
{
   unsigned  int  numeroDePassageiros;
  tPassageiro    listaDeEspera[numeroDePassageiros];

   ...

   printf( "Introduza o número máximo de passageiros "
           "na lista de espera: ");
```

```
    scanf("%d", &numeroDePassageiros);

    ...
}
```

Esta última solução não apresenta nenhum melhoramento com relação àquela apresentada antes. Ao contrário, agora, o array não pode ser redimensionado (v. **Seção 7.7**), enquanto, no caso anterior, o array pode ser redimensionado utilizando **realloc()**.

Observe que, como todos os elementos do array são alocados como um único bloco, é garantido que estes elementos estarão em posições contíguas de memória. Conseqüentemente, estes elementos podem ser acessados por meio de índices do mesmo modo que um array convencional (i.e., alocados estaticamente quando o programa é carregado em memória). Se estes elementos tivessem sido alocados individualmente por meio de chamadas sucessivas de **malloc()** ou qualquer outra função de alocação, a contigüidade deles em memória não estaria garantida. Por exemplo, suponha que `p1`, `p2` e `p3` sejam ponteiros para o tipo `tPassageiro`; então, após a execução das chamadas:

```
p1 = malloc(sizeof(tPassageiro));
p2 = malloc(sizeof(tPassageiro));
p3 = malloc(sizeof(tPassageiro));
```

não haveria garantia de que as posições em memória apontadas por `p1`, `p2` e `p3` sejam adjacentes. Ou seja, a posição apontada por `p2` pode não começar logo após o final do espaço em memória apontado por `p1` e a posição apontada por `p3` pode não começar logo após o final do espaço apontado por `p2`. Pode ser que, neste caso, a contigüidade em memória eventualmente aconteça, mas o programador não deve jamais confiar nisto.

A solução apresentada até aqui para o problema da lista de espera ainda não representa a solução ideal, apesar de já apresentar uma melhora com relação à solução inicial apresentada na **Seção 11.1**, na qual um array é alocado estaticamente. Ou seja, a última solução apresentada aqui ainda requer que seja especificado um número máximo de passageiros a cada execução do programa e antes do processamento da lista. A melhor solução

seria a alocação de espaço à medida que se precisasse dele (i.e., alocação de um elemento na lista a cada vez que surgisse um novo candidato à lista de espera). Esta solução será esboçada na **Seções 11.6** e **11.7**.

11.6 Estruturas Recursivas e Listas Encadeadas

Na **Seção 11.5** foi visto que uma solução para um problema no qual a quantidade de memória alocada não pode ser determinada a priori é a alocação dinâmica de memória. Ainda naquela seção, foi apresentado um exemplo no qual um array tinha seu tamanho especificado em tempo de execução do programa. Mas, mesmo neste caso, a solução não era ideal, pois aquela especificação de tamanho era apenas uma previsão de demanda para uma dada execução, de modo que a alocação prevista poderia não corresponder exatamente à demanda real. Assim, a melhor solução para o problema é alocar memória à medida que esta se faz realmente necessária.

No caso do problema específico da lista de espera, a melhor solução corresponde a alocar cada registro no instante em que for necessário inserir mais um passageiro na lista de espera. Acontece que, conforme foi visto na seção anterior, quando registros são alocados individualmente, não se tem a garantia de que erão alocados em posições adjacentes e, conseqüentemente, estes registros não podem ser acessados seqüencialmente como um array. A solução para este problema de acesso a elementos alocados individualmente está no uso de listas encadeadas.

Uma **lista encadeada** é uma estrutura de dados composta de elementos usualmente denominados **nós**. Cada nó numa lista encadeada pode ser conceitualmente dividido em duas partes:

(1) Uma parte contendo os dados propriamente ditos – esta parte será doravante identificada por `dado`.

(2) Um ponteiro para o próximo elemento da lista – esta parte será doravante identificada por `próximo`.

O diagrama da **Figura 30** apresenta esquematicamente uma lista simplesmente encadeada.

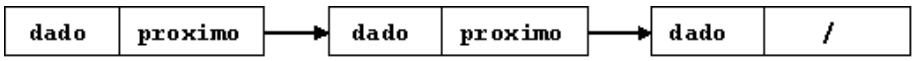

Figura 30: Diagrama de uma lista simplesmente encadeada com três nós

O tipo de lista esquematizado na **Figura 30** é denominado **lista simplesmente encadeada** porque os ponteiros apontam numa única direção. Existem listas, denominadas **duplamente encadeadas**, nas quais cada nó possui ponteiros que apontam em ambas as direções (i.e., para o nó anterior e para o próximo nó). Apenas listas simplesmente encadeadas serão abordadas aqui.

11.7 Implementação de Listas Encadeadas em C

Os nós de uma lista (simplesmente) encadeada podem ser implementados em C como estruturas contendo dois campos que correspondem às partes componentes dos nós. Por exemplo, as declarações a seguir poderiam ser utilizadas para definir o tipo dos nós de uma lista encadeada cujos nós podem conter informações sobre passageiros do problema de lista de espera descrito no início do capítulo[8]:

```
typedef  struct {
            char  nome[30];
            char  identidade[12];
            char  numeroDoBilhete[10];
         } tPassageiro;

typedef  struct  no {
            tPassageiro  dado;
            struct no   *proximo;
         } tNo;

tNo *listaDeEspera;
```

Observe que, na última declaração de tipo acima, foi utilizado um rótulo para a estrutura, de modo a tornar possível a auto-referência no

[8] A primeira declaração de tipo foi acrescenta apenas para facilidade de referência. Concentre-se na última declaração, que é a que, efetivamente, é utilizada na criação da lista encadeada.

segundo campo desta estrutura. Como o exemplo acima sugere, o campo `dado` dos nós de uma lista encadeada pode ser de qualquer tipo.

A forma de estruturação de dados apresentada acima facilita a modificação do tipo de informação contida em cada nó da lista. Para aproveitar a última declaração de tipo acima para a criação de uma nova lista com nós de outro tipo, a única coisa que precisa ser feita é a substituição do tipo `tPassageiro` pelo tipo desejado. Mas o identificador `dado` é genérico demais para ser útil em termos de legibilidade. No caso da lista de espera, provavelmente o nome `passageiro` tornaria o programa mais legível.

11.8 Operações Básicas sobre Listas Encadeadas

O uso de listas encadeadas num programa tipicamente requer, pelo menos, a implementação das seguintes operações:

- Criação da lista
- Inserção de um elemento na lista
- Remoção de um elemento da lista
- Localização de um elemento da lista

Nesta seção, serão apresentados exemplos de implementação de cada uma destas operações. Como ilustração do uso de ponteiros e alocação dinâmica de memória, será examinada a implementação de listas encadeadas como estruturas dinâmicas[9].

11.8.1 Criação de uma Lista Encadeada

Um exemplo de lista encadeada com três nós, na qual cada nó contém um ***string*** representando o nome de uma cor, é apresentado na **Figura 31** a seguir[10].

---

[9] Listas encadeadas podem também ser implementadas por meio de arrays, mas isso não é de interesse aqui.

[10] Todas as cores apresentadas aqui são strings que, por simplicidade, são apresentados sem aspas. Também, quando se fala em nó de uma certa cor, pretende-se dizer que o campo dado do nó contém o string que representa essa cor. Por exemplo, nó branco significa nó cujo conteúdo do campo dado é "branco".

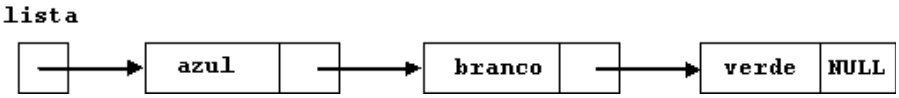

Figura 31: Exemplo de lista encadeada contendo strings

Na lista ilustrada no diagrama da **Figura 32**, cada nó possui dois campos: o primeiro campo é um ***string*** e representa o conteúdo efetivo do nó e o segundo campo, representado por setas, é um ponteiro para o próximo elemento na lista. O valor **NULL** no segundo campo do último nó indica que ele não aponta para nenhum outro nó. A variável `lista` é um ponteiro para o primeiro elemento da lista. Cada elemento da lista pode ser acessado começando-se pelo início da lista e seguindo-se os ponteiros para os nós seguintes.

O primeiro passo para a criação de uma lista encadeada é a definição do tipo de cada nó da lista. Conforme já foi visto, para assegurar o acesso a todos os elementos da estrutura, é necessário que cada elemento inclua uma indicação de onde o próximo elemento localiza-se em memória. Assim, cada elemento deve consistir em duas partes: (1) a informação que deve ser armazenada no elemento e (2) o endereço em memória do próximo elemento. Portanto, para implementação da lista encadeada ilustrada no diagrama da Figura **31**, pode-se utilizar a seguinte declaração de tipos:

```
typedef  struct  no {
            char       dado[10];
            struct no *proximo;
         } tNo, *tLista;
```

Para a criação da lista, serão utilizados três ponteiros do tipo `tLista`: o primeiro ponteiro representará exatamente o início da lista, enquanto o segundo e o terceiro facilitarão a inserção de novos nós na lista, como será visto em seguida. Estes ponteiros são definidos como:

```
tLista   lista, ponteiroAux1, ponteiroAux2;
```

Procede-se agora à criação da lista ilustrada no diagrama da **Figura 32**. O primeiro nó pode ser criado por meio das seguintes instruções[11]:

```
lista = malloc(sizeof(tNo));       /* 1 */
strcpy(lista->dado, "azul");       /* 2 */
```

Após a execução da instrução 1, tem-se a situação ilustrada na **Figura 32**, na qual o símbolo **?** significa que o conteúdo do campo que o contém é indeterminado naquele instante:

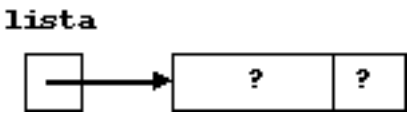

Figura 32: Criação do primeiro nó de uma lista encadeada

Recorde-se, da **Seção 8.4.4**, que a função de biblioteca **strcpy()** copia o conteúdo de seu segundo argumento (um *string*) no primeiro argumento (um array). Assim, após a execução da instrução 2, a situação passa a ser aquela apresentada na **Figura 33**:

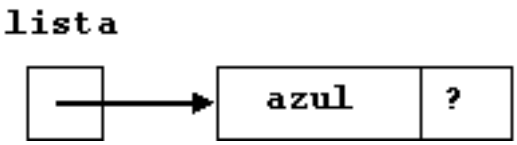

Figura 33: Preenchimento do primeiro nó de uma lista encadeada

O segundo nó da lista é criado e acrescentado à lista por meio das seguintes instruções:

[11] As instruções foram numeradas para facilitar a discussão que se segue.

```
ponteiroAux1 = malloc(sizeof(tNo));      /* 3 */
strcpy(ponteiroAux1->dado, "branco");    /* 4 */
lista->proximo = ponteiroAux1;           /* 5 */
```

Após a execução da instrução 3, tem-se a situação ilustrada na **Figura 33**. Note que, no instante representado pela **Figura 35**, o novo nó ainda não está encadeado na lista.

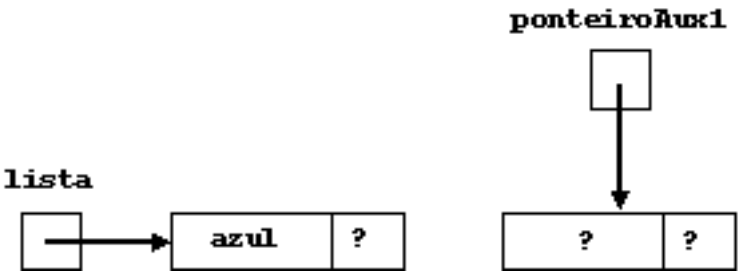

Figura 34: Criação do segundo nó de uma lista encadeada

A instrução 4 preenche o valor do primeiro campo do novo nó, enquanto a instrução 5 faz a devida anexação deste nó à lista. Assim, após a execução da instrução 5, a situação pode ser representada como na **Figura 35**.

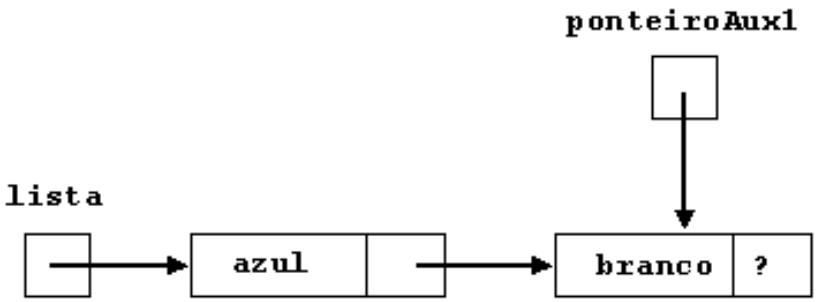

Figura 35: Preenchimento do segundo nó de uma lista encadeada

O último conjunto de instruções, responsável pela inserção do último elemento da lista, é:

```
ponteiroAux2 = malloc(sizeof(tNo));        /* 6 */
strcpy(ponteiroAux2->dado, "verde");       /* 7 */
ponteiroAux1->proximo = ponteiroAux2;      /* 8 */
ponteiroAux2->proximo = NULL;              /* 9 */
```

Após a execução da instrução 6, tem-se a situação ilustrada na **Figura 36**. Note, novamente, que, nesse ponto, o novo nó ainda não está anexado à lista.

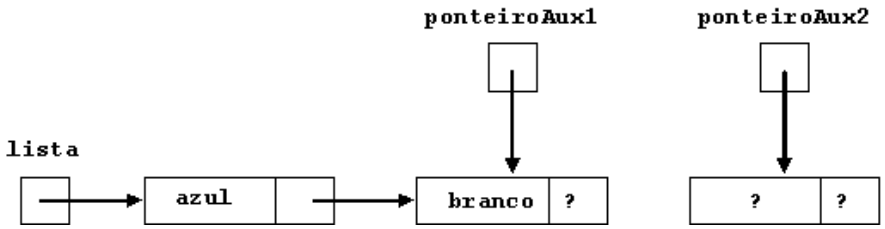

Figura 36: Criação do terceiro nó de uma lista encadeada

A instrução 7 preenche o campo de dado do nó apontado por `ponteiroAux2`, enquanto a instrução 8 faz a ligação do nó apontado por `ponteiroAux1` para o nó apontado por `ponteiroAux2`. A instrução 9 simplesmente indica que o último elemento não aponta para nenhum outro nó. A situação final da lista é ilustrada na **Figura 37.**

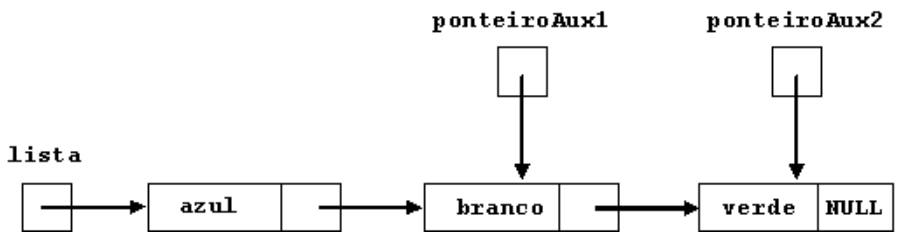

Figura 37: Preenchimento do terceiro nó de uma lista encadeada

Pode parecer que foram utilizados três ponteiros (`lista`, `ponteiroAux1` e `ponteiroAux2`) devido ao fato de a lista também possuir três elementos, mas isto não corresponde à realidade. Isto é, se fosse desejado acrescentar um quarto elemento à lista, não seria necessário um quarto ponteiro auxiliar. Neste caso, o que se teria que fazer seria o seguinte:

(1) Alocar o novo nó utilizando o ponteiro `ponteiroAux1`

(2) Atribuir os devidos valores aos campos do novo nó (ainda por meio do ponteiro `ponteiroAux1`)

(3) Fazer a ligação do nó apontado pelo ponteiro `ponteiroAux2` para o nó apontado por `ponteiroAux1`

Existe algo importante ausente no conjunto de instruções apresentado acima. Como foi visto na **Seção 11.2**, deve-se sempre testar se uma dada tentativa de alocação dinâmica de memória foi bem-sucedida antes de utilizar o espaço que se espera ter sido alocado. Isto não foi incluído no exemplo apresentado nesta seção apenas para não desviar a atenção dos pontos mais importantes de criação da lista.

11.8.2 Visita Seqüencial aos Elementos de uma Lista Encadeada

Suponha que você deseje imprimir o valor contido no campo `dado` de cada nó da lista encadeada da **Figura 31**. Para fazer isso, você precisaria **atravessar** toda a lista do início ao fim. A função a seguir realiza isto:

```
/****
 *
 * Função ImprimeListaEncadeada()
 *
 * Descrição: Imprime o conteúdo (string) de cada nó de
 *            uma lista simplesmente encadeada.
 *
 * Parâmetros:
 *      p (entrada): ponteiro para o início da lista encadeada
 *
 * Retorno: Nada.
 *
 ****/
```

```
void  ImprimeListaEncadeada(tLista  p)
{
     while  (p)  {
          printf("%s\n", p->dado); /* Imprime o conteúdo do nó
e... */
        p = p->proximo;  /* avança para o próximo nó  */
     }
}
```

Uma chamada `ImprimeListaEncadeada(lista)` imprimiria todos os valores armazenados nos campos `dado` dos nós da lista encadeada apontada por `lista`. A função `ImprimeListaEncadeada()` recebe como entrada um ponteiro para o início de uma lista encadeada. Esta lista pode estar vazia ou conter um ou mais elementos. No primeiro caso, o ponteiro, representado pelo parâmetro formal `p`, é **NULL** e o laço **while** não é executado. No segundo caso, entra-se no laço **while**, imprime-se o valor do campo `dado` do primeiro elemento e faz-se `p` apontar para o próximo nó. Se o ponteiro que aponta para este nó for **NULL**, a iteração pára; caso contrário, repete-se o processo até que `p` assuma eventualmente o valor **NULL**.

A função `ImprimeListaEncadeada()` representa um exemplo de **visita seqüencial** aos elementos de uma lista encadeada com o objetivo de realizar alguma operação sobre cada um deles. No caso atual, a operação executada é a impressão do conteúdo de cada nó, mas, para a execução de qualquer outra operação, a abordagem básica seria a mesma. Isto é, para executar qualquer outra operação sobre cada nó de uma lista encadeada, a única alteração a ser feita seria substituir a chamada de **printf()** na função `ImprimeListaEncadeada()` pela chamada de uma função que executasse a devida operação.

11.8.3 Inserção de Elementos numa Lista Encadeada

Elementos (nós) podem ser inseridos numa lista encadeada de várias maneiras. Pode-se, por exemplo, desejar que um nó seja inserido no início ou no final da lista ou pode-se querer que a inserção seja feita imediatamente antes ou imediatamente depois de um nó contendo um determinado valor. Como exemplo, a função apresentada em seguida é utilizada para inserir um nó imediatamente depois de um nó contendo um determinado valor

numa lista cujos nós são do tipo do exemplo apresentado na **Seção 11.8.1** (v. **Figura 31**)

Como ilustração do processo, suponha que se deseje inserir um nó cujo valor do campo `dado` seja `amarelo` imediatamente depois do nó contendo `branco` na lista esquematizada na **Figura 31**. Então, a situação final deveria ser aquela ilustrada na **Figura 38**.

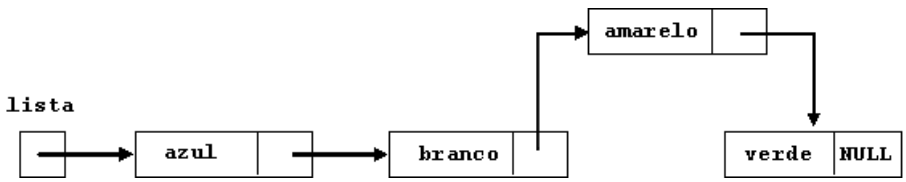

Figura 38: Inserção de um nó numa lista encadeada

As mudanças feitas na lista para a inserção do nó `amarelo` são estas: o nó `branco` passa a apontar para o novo nó (`amarelo`), que, por sua vez, passa a apontar para o nó `verde`. Note, entretanto, que não se pode fazer as novas atribuições de endereços nesta ordem. Isto é, se o endereço do nó `amarelo` for atribuído ao campo `proximo` do nó `branco` antes da atribuição do endereço do nó `verde` ao campo `proximo` do nó `amarelo`, o nó `verde` estará perdido para sempre. No diagrama da **Figura 38**, isto significa que se você desenhar primeiro a seta do nó `branco` para o nó `amarelo`, nunca será capaz de desenhar a seta do nó `amarelo` para o nó `verde`, simplesmente porque não sabe onde o nó `verde` se encontra.

A função `InsereDepois()`, apresentada a seguir, implementa as idéias expostas acima.

```
/****
 *
 * Função InsereDepois()
 *
  * Descrição: Insere um novo nó numa lista simplesmente
encadeada
 *   após um nó cujo conteúdo é especificado.
 *
 * Parâmetros:
```

```
 *   p (entrada): ponteiro para o início da lista encadeada
 *   stringNovo (entrada): conteúdo do novo nó
 *    stringNaLista (entrada): conteúdo do nó após o qual será
feita a inserção
 *
 * Retorno: Nada.
 *
 ****/

void   InsereDepois( tLista  p, char  *stringNovo,
                     char  *stringNaLista )
{
    tLista   posicaoDoNo, noNovo;

    posicaoDoNo = NULL;

    while  (p  &&  !posicaoDoNo){  /* Procura o nó */
       if   (!strcmp(p->dado, stringNaLista))
     posicaoDoNo = p;  /* Nó procurado foi encontrado */
       else
        p = p->proximo;    /* Passa para o próximo nó */
    }

     if  (posicaoDoNo){ /* posicaoDoNo foi encontrado e aponta
para */
                /* o nó após o qual a inserção será feita */
     noNovo = malloc(sizeof(tNo)); /* Aloca o novo nó */
       if (noNovo) {
              /* Preenche o conteúdo do novo nó */
           strcpy(noNovo->dado, stringNovo);

             /* Faz o novo nó apontar para o nó   */
            /* apontado antes pelo nó encontrado */
           noNovo->proximo = posicaoDoNo->proximo;

         /* Faz o nó encontrado apontar para o nó novo */
           posicaoDoNo->proximo = noNovo;
       }  /* if */
    } /* if */
}  /* InsereDepois */
```

11.8.4 Remoção de Elementos de uma Lista Encadeada

Do mesmo modo que a inserção, a remoção de nós de uma lista encadeada pode ser realizada de acordo com diversos critérios. A função a

ser apresentada a seguir remove um nó, cujo conteúdo é especificado, de uma lista do tipo apresentado na **Figura 31**.

Como ilustração, suponha que se deseje remover o nó cujo valor do campo `dado` seja `branco` na lista ilustrada na **Figura 31**. Então, a situação final deveria ser aquela apresentada na **Figura 39**.

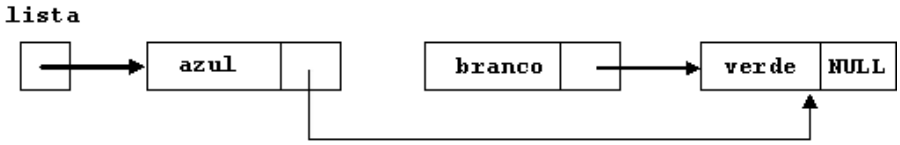

Figura 39: Remoção de um nó de uma lista encadeada

Conforme pode-se observar na**Figura 44**, a remoção do nó desejado envolve apenas o desvio do endereço do campo `proximo` do nó anterior àquele sendo removido para o nó que era anteriormente apontado pelo nó a ser removido. Na prática, deve-se também liberar o espaço reservado para o nó removido, mas, em princípio, o desvio representado pelo esquema acima é suficiente para a remoção do nó da lista encadeada, pois, com esse desvio, o referido nó jamais será acessado. A função `RemoveNo()` apresentada a seguir produz o efeito de remoção desejado.

```
/****
 *
 * Função RemoveNo()
 *
 * Descrição: Remove o primeiro nó cujo conteúdo é especificado
de uma lista encadeada.
 *
 * Parâmetros:
 *   p (entrada/saída): endereço do ponteiro que representa o
 *                           início da lista encadeada
 *         stringNaLista (entrada): conteúdo do nó que será
removido
 *
 * Retorno: Nada.
 *
 ****/
```

```
void   RemoveNo(tLista   *p, char   *stringNaLista)
{
   tLista  noAnterior = NULL, /* Não há nó anterior no início
*/
    posicaoDoNo= *p; /* Começa busca no início da lista */
  unsigned noFoiEncontrado = 0; /* O nó não foi ainda encontrado
*/

   while  (posicaoDoNo  &&  !noFoiEncontrado)  {  /* Procura o
nó */
      if   (!strcmp(posicaoDoNo->dado, stringNaLista))
       noFoiEncontrado = 1; /* Nó procurado foi encontrado */
      else {              /* O nó anterior passa a ser */
          noAnterior = posicaoDoNo; /* o atual e o atual passa
*/
        posicaoDoNo = posicaoDoNo->proximo;  /* a ser o próximo
*/
       }
   }

      /* Se o nó não foi encontrado, não há mais nada a fazer.
*/
     /* Se ele foi encontrado, o trecho a seguir faz a remoção.
*/

   if  (noFoiEncontrado) {
      if  (!noAnterior) /* O nó a remover é o primeiro da lista
*/
         *p = (*p)->proximo; /* Primeiro nó será antigo segundo
nó  */
     else   /* O nó a ser removido NÃO é o primeiro da lista
*/
         noAnterior->proximo = posicaoDoNo->proximo;/* Faz o
desvio */

     /* Neste ponto, o nó apontado por posicaoDoNo não mais
*/
    /* pertence à lista e a remoção está logicamente concluída.
*/
    /* Resta apenas liberar o espaço ocupado pelo nó removido.
*/

      /* Libera a alocação de memória do nó removido */
       free(posicaoDoNo);
    }
}
```

Observe que a remoção do primeiro nó da lista é um caso que deve ser tratado à parte pela função `RemoveNo()`. Note, ainda, que se houver na lista mais de um nó com o mesmo conteúdo especificado para remoção, apenas o primeiro nó com este conteúdo será removido com uma única chamada da função acima.

É importante salientar que o primeiro argumento da função `RemoveNo()` não é do tipo `tLista` como na função `InsereDepois()`; ele é um ponteiro para este tipo. A razão pela qual o primeiro argumento, que representa o início da lista encadeada, é um ponteiro para o tipo `tLista` é que existe a possibilidade de o primeiro elemento da lista ser removido e, neste caso, o ponteiro para o início da lista precisa ser alterado. No caso da função `InsereDepois()`, não havia possibilidade de este ponteiro ser modificado.

11.9 Algoritmos de Classificação II

Nesta seção serão apresentadas duas funções capazes de ordenar arrays cujos elementos são de tipos arbitrários. Uma destas funções, denominada **qsort()**, é bastante eficiente e já existe pronta para uso na biblioteca padrão de C. Apesar disso, o uso desta função não é tão trivial quanto aparenta ser e, por isso, ela é discutida em profundidade juntamente com a apresentação de exemplos que ilustram seu uso.

A segunda função de classificação a ser apresentada aqui é uma nova versão da função `BubbleSort()`, que foi introduzida na **Seção 10.8.1** e recebeu um pequeno melhoramento na **Seção 10.8.2**, onde foi implementada com a denominação `BubbleSort2()`. A nova versão apresenta um sensível melhoramento com relação às versões anteriores. De fato, a nova versão, denominada `BubbleSort3()`, será tão genérica quanto **qsort()**. Em uso prático, porém, deve-se dar preferência à função **qsort()**, pois ela implementa um método de ordenação muito mais eficiente do que o método da bolha.

11.9.1 A Função de Biblioteca qsort()

A função **qsort()** da biblioteca padrão de C implementa um método de ordenação, denominado ***quicksort***, que é bem mais eficiente do que o método da bolha apresentado na **Seção 10.8**. Além disso, diferentemente das funções `BubbleSort()` e `BubbleSort2()`, apresentadas na **Seção 10.8**, a função **qsort()** é capaz de ordenar arrays de elementos de quaisquer tipos[12].

A função **qsort()** faz parte do módulo de biblioteca stdlib e o protótipo dela é:

```
void qsort( void *array, size_t nElementos,          size_t
tamanhoDoElemento,
 int (*compara) (const void *, const void *))
```

De acordo com o protótipo apresentado, os parâmetros da função **qsort()** possuem os seguintes significados:

- `array` é um ponteiro para o início do array a ser classificado
- `nElementos` é o número de elementos no array a ser ordenado
- `tamanhoDoElemento` é o tamanho, em bytes, de cada elemento do array
- `compara` é um ponteiro para uma função que compara elementos do array

A função de comparação, cujo endereço deve ser passado como último argumento numa chamada da função **qsort()**, deve ter dois argumentos, cada um dos quais é um elemento do array a ser classificado. Assim, o protótipo de tal função deve ser:

```
int Comparacao(const void *elemento1, const void *elemento2)
```

[12] As funções BubbleSort() e BubbleSort2() são capazes de ordenar apenas arrays cujos elementos são do tipo int; no entanto, isto não constitui uma limitação do método da bolha, mas sim das implementações representadas por estas funções. Na Seção 11.9.2 será mostrado como o método da bolha pode ser implementado numa função capaz de superar esta deficiência.

Esta função compara os dois elementos do array a ser ordenado e retorna um valor do tipo **int** baseado no resultado da comparação, de acordo com o que é apresentado na **Tabela 36**.

SE...	A FUNÇÃO DE COMPARAÇÃO RETORNA...
`elemento1 < elemento2`	um valor menor do que 0
`elemento1 == elemento2`	0
`elemento1 > elemento2`	um valor maior do que 0

Tabela 36: Valores retornados pela função de comparação usada com qsort()

Os valores especificados na **Tabela 36** pressupõem que se deseja ordenar um array em ordem crescente. Se a ordem desejada for decrescente, basta trocar < por > e vice-versa na primeira e na última linhas da tabela.

Como exemplo de uso da função **qsort()**, considere o seguinte programa, que ordena um array de elementos do tipo **int** nas ordens crescente e decrescente:

```
#include <stdio.h>
#include <stdlib.h>

int ComparaIntsCrescente(const void *elem1, const void *elem2)
{
   return *(int *)elem1 - *(int *)elem2;
}
int  ComparaIntsDecrescente(const  void  *elem1,  const  void
*elem2)
{
   return *(int *)elem2 - *(int *)elem1;
}
int main(void)
{
   int  arrayDeInts[] = {12, 55, 21, 1, 6, 8, 17, 220, 5, 83};
   int  i, tamanhoAr;

   tamanhoAr = sizeof(arrayDeInts)/sizeof(arrayDeInts[0]);

   printf("Ordem original:\n");
   for (i=0; i < tamanhoAr; i++) {
     printf(" %d ", arrayDeInts[i]);
   }
```

```
            qsort(arrayDeInts,    tamanhoAr,    sizeof(int),
ComparaIntsCrescente);

   printf("\n\nOrdem crescente:\n");
   for (i=0; i < tamanhoAr; i++) {
     printf(" %d ", arrayDeInts[i]);
   }

            qsort(arrayDeInts,    tamanhoAr,    sizeof(int),
ComparaIntsDecrescente);

   printf("\n\nOrdem decrescente:\n");
   for (i=0; i < tamanhoAr; i++) {
     printf(" %d ", arrayDeInts[i]);
   }

   return 0;
}
```

Conforme você deve ter observado no programa acima, o uso da função **qsort()** em si é fácil. O difícil é definir a função de comparação de elementos utilizada por ela. Para entender como uma tal função de comparação é definida, note que, em primeiro lugar, ela deve comparar os conteúdos de dois elementos do array a ser ordenado. Acontece que, de acordo com o protótipo que deve ser seguido pela função de comparação, ela recebe dois ponteiros genéricos que apontam para os elementos a ser comparados e estes ponteiros não fornecem informação sobre como os conteúdos apontados devem ser interpretados. Por exemplo, a função de comparação `ComparaIntsCrescente()` utilizada pelo programa acima tem como cabeçalho:

```
int ComparaIntsCrescente(const void *elem1, const void
*elem2)
```

Os argumentos `elem1` e `elem2` desta função podem apontar para objetos de quaisquer tipos de dados. Portanto, é necessário informar ao compilador como o objeto apontado deve ser interpretado. Isto é feito utilizando conversão explícita para um ponteiro para o tipo ao qual se deseja que o conteúdo apontado seja interpretado. Por exemplo, a aplicação do operador de conversão explícita que aparece no corpo da função `ComparaIntsCrescente()`:

```
(int *)elem1
```

informa que o conteúdo apontado pelo parâmetro `elem1` é do tipo **int**.

Após identificar o tipo dos conteúdos apontados pelos parâmetros da função de comparação, o acesso a estes conteúdos é obtido por meio da aplicação do operador de indireção. Por exemplo, o conteúdo apontado pelo parâmetro `elem1` é obtido mediante a expressão:

```
*(int *)elem1
```

O restante da implementação da função de comparação utilizada por **qsort()** depende do tipo dos elementos do array na ser ordenado. No programa apresentado como exemplo, o tipo dos elementos do array é **int**. Assim, o retorno da função `ComparaIntsCrescente()` é equivalente à expressão:

o conteúdo do segundo elemento menos o conteúdo do primeiro elemento

É fácil verificar que o retorno especificado pela **Tabela 35** corresponde exatamente a esta expressão nos casos em que se comparam elementos de tipos numéricos.

Considere agora um segundo exemplo de uso da função **qsort()** e observe como a função de comparação é definida:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int Compara(const void *e1, const void *e2)
{
   return(strcmp((char *)e1,(char *)e2));
}

int main(void)
{
     char  cores[5][8]  =  {  "azul",  "verde",  "preto",  "roxo",
"branco" };
   int  i;

   qsort(cores, 5, sizeof(cores[0]), Compara);
```

```
   for (i = 0; i < 5; i++)
      printf("%s\n", cores[i]);

   return 0;
}
```

Neste último exemplo, os elementos da lista que será ordenada são ***strings***. Portanto, a primeira medida da função de comparação é interpretar seus argumentos como ***strings***. Isto é efetuado por meio da conversão `(char *)`. Em seguida, é chamada a função **strcmp()**, que compara ***strings*** e cujos valores retornados correspondem exatamente àquilo especificado pela **Tabela 35.**

11.9.2 Generalizando a Função BubbleSort() II

Uma limitação das funções `BubbleSort()`, apresentada na **Seção 10.8.1**, e `BubbleSort2()`, apresentada na **Seção 10.8.2**, é que elas são capazes de ordenar apenas arrays de elementos do tipo **int**. Esta seção mostra, utilizando ponteiros genéricos e alocação dinâmica de memória, como é possível generalizar a função `BubbleSort()` de modo que ela seja capaz de ordenar arrays com elementos de quaisquer tipos.

A função `BubbleSort3()` apresentada a seguir representa uma generalização da função `BubbleSort2()` apresentada na **Seção 10.8.2**. As alterações feitas naquela função que resultaram na presente função generalizada foram numeradas e aparecem em forma de comentário na função que será apresentada.

```
/****
 *
 * Função BubbleSort3()
 *
 * Descrição: Ordena um array de elementos de tipos arbitrários
usando o método da bolha.
 *
 * Parâmetros:
 *      array (entrada): ponteiro para o início do array
 *                       a ser ordenado
 *    nElementos (entrada): número de elementos do array
 *   tamElemento (entrada): tamanho de cada elemento do array
 *  compara (entrada): ponteiro para uma função de comparação
```

```
 *
 * Retorno: Nada.
 *
 ****/

void BubbleSort3( void *lista, size_t nElementos, /* Alteração
1 */
            size_t tamElemento,
            int (*compara) (const void *, const void *))
{
   unsigned  i, ordenada = FALSE;
   char     *aux;  /* Alteração 2 */

   aux = malloc(tamElemento); /* Alteração 3 */

   if (!aux)    /* Alteração 4 */
      return;

   while (!ordenada){
      ordenada = TRUE;  /* Supõe que a lista está ordenada */

      for (i = 0; i < nElementos - 1; i++){
         if (compara(lista + i*tamElemento,
          lista + (i+1)*tamElemento) > 0){/*Alteração 5 */
           ordenada = FALSE; /*Pelo menos um elemento fora de
ordem*/

        /* Troca elementos adjacentes */
        /* Alterações 6, 7 e 8 */
        emmove(aux, lista + i*tamElemento, tamElemento);
            memmove(lista + i*tamElemento,
            lista + (i + 1)*tamElemento, tamElemento);
       memmove(lista + (i + 1)*tamElemento, aux, tamElemento);
          }  /* final do if */
      } /* for */
   } /* while */
} /* BubbleSort3 */
```

Comentários sobre a função BubbleSort3():

- *Alteração 1*: a lista de parâmetros da função é semelhante àquela da função **qsort()** da biblioteca padrão (v. **Seção 11.9.1**)

- *Alteração 2*: a variável `aux` é definida como um ponteiro para **char**. Ela irá apontar para um bloco de memória do tamanho

de cada elemento da lista que auxilia a troca de valores entre elementos (blocos) adjacentes desta lista.

• *Alteração 3*: Faz-se alocação do bloco que será utilizado como auxiliar na troca de valores entre elementos da lista.

• *Alteração 4*: Quando não é possível alocar espaço para o bloco que auxilia na troca de valores, a função retorna imediatamente. É possível que haja algum outro modo de realizar trocas entre elementos sem este bloco, mas este problema não foi investigado aqui.

• *Alteração 5*: A função de comparação, representada pelo ponteiro para função `compara`, é chamada para comparar dois elementos adjacentes. É interessante notar que a função de comparação recebe os endereços destes elementos e que o i-ésimo elemento da lista encontra-se no endereço dado por:

```
lista + i*tamElemento
```

• *Alterações 6, 7 e 8*: Essas instruções fazem a troca de valores entre elementos adjacentes. A novidade aqui é o uso da função de biblioteca **memmove()** (`#include <string.h>`). Esta função copia n bytes de um bloco de memória para outro. O primeiro argumento desta função é um ponteiro para o bloco que receberá a cópia; o segundo argumento é um ponteiro para o bloco de onde os bytes serão copiados e o terceiro argumento representa o número de bytes a ser copiados. Maiores detalhes sobre o funcionamento da função **memmove()** são encontrados no **Volume II**.

11.10 Exercícios de Revisão

1. Cite cinco situações em programação nas quais alocação dinâmica de memória se faz necessária.

2. Defina: (a) variável estática e (b) variável dinâmica.

3. Descreva o funcionamento de cada uma das funções a seguir:

(a) malloc()
(b) calloc()
(c) free()

4. (a) O que acontece quando o tamanho do ***bloco novo*** é menor do que o tamanho do ***bloco antigo*** numa chamada de **realloc()**? (b) O que pode acontecer quando o tamanho do ***bloco novo*** é maior do que o tamanho do ***bloco antigo*** numa chamada de **realloc()**?

5. Como funciona a função **realloc()** quando o primeiro argumento é um ponteiro nulo?

6. Como funciona a função **realloc()** quando o segundo argumento é zero?

7. (a) O que é ***heap*** em alocação dinâmica de memória? (b) O que significa fragmentação de ***heap***?

8. (a) O que são ponteiros genéricos? (b) Em que situações ponteiros genéricos são úteis?

9. (a) Como deve ser testado um ponteiro retornado por uma função de alocação dinâmica de memória? (b) Por que é sempre recomendado testar o ponteiro retornado por uma função de alocação dinâmica de memória?

10. (a) O que é uma estrutura com auto-referência? (b) Qual é a utilidade de estruturas com auto-referência?

11. Na **Seção 11.5** um array dinâmico foi alocado utilizando o seguinte fragmento de programa:

```
printf("Introduza o número máximo de passageiros na lista: ");
scanf("%d", &numeroDePassageiros);

listaDeEspera = malloc(numeroDePassageiros*sizeof(tPassageiro)
);
```

No entanto, se o compilador utilizado for aderente ao padrão C99, poder-se-ia obter o mesmo resultado utilizando o trecho de programa:

```
tPassageiro listaDeEspera[numeroDePassageiros];
...
printf("Introduza o número máximo de passageiros na lista: ");
scanf("%d", &numeroDePassageiros);
```

Qual é a vantagem obtida ao usar o primeiro trecho de programa em relação ao segundo trecho?

12. (a) O que são listas encadeadas? (b) Por que listas encadeadas são consideradas estruturas de dados dinâmicas? (c) Qual é a vantagem do uso de listas encadeadas com relação a arrays? (d) Qual é a desvantagem do uso de listas encadeadas com relação a arrays?

13. Por que cada nó de uma lista encadeada precisa armazenar o endereço do próximo elemento da lista?

14. Por que a função `RemoveNo()` recebe o endereço do ponteiro que aponta para o início de uma lista encadeada como parâmetro enquanto a função `InsereDepois()` recebe como parâmetro simplesmente um ponteiro para o início da lista?

15. Escreva um conjunto de instruções para acrescentar um quarto nó na lista encadeada esquematizada na **Figura 31** (**Seção 11.8**). O conteúdo deste novo nó deve ser o string `"vermelho"`.

16. Explique por que os argumentos da função de comparação utilizada como quarto argumento da função **qsort()** são do tipo **const void ***.

17. (a) Explique o uso da função **memmove()** na definição da função `BubbleSort3()` apresentada na **Seção 11.9.2**. (b) Por que o uso da função **memmove()** se faz necessário nesta definição?

11.11 Exercícios de Programação

EP11.1) Escreva uma função em C que retorne o número de nós de uma lista simplesmente encadeada. O protótipo desta função deve ser:

```
unsigned Comprimento(tLista L)
```

onde `L` é um ponteiro para o primeiro nó de uma lista simplesmente encadeada e o tipo `tLista` é definido como na **Seção 11.7**. (**Sugestão**: Você não precisa saber o tipo do campo `dado` dos nós da lista encadeada para escrever esta função.)

EP11.2) Escreva uma função em C cujo protótipo seja dado por:

```
tLista Concatena(tLista L1, tLista L2)
```

que recebe dois ponteiros para listas encadeadas como entrada e retorna um ponteiro para uma terceira lista que é a concatenação das duas listas recebidas como argumentos. As duas listas de entrada devem permanecer imutáveis. Suponha que o tipo do campo `dado` dos nós da lista encadeada seja do tipo `tDado` previamente definido. (**Sugestão**: Utilize a função **memcpy()** apresentada no **Volume II** para copiar o conteúdo de um nó para outro.)

EP11.3) Construa um programa completo em C que crie a lista encadeada apresentada na **Seção 11.8.1**. Neste programa, inclua um teste após cada chamada da função **malloc()** que verifique se cada tentativa de alocação de memória foi bem-sucedida.

EP11.4) Modifique a função `InsereDepois()`, apresentada na **Seção 11.8.4**, numa função que retorne `1` se a inserção foi bem-sucedida (i.e., se o nó tiver sido realmente inserido na lista) e `0` em caso contrário.

EP11.5) Modifique a função `RemoveNo()` apresentada na **Seção 11.8.4** de modo que ela seja capaz de remover (com uma única chamada) todas as ocorrências de nós com o conteúdo especificado e não apenas o primeiro nó encontrado.

EP11.6) Escreva um programa que teste a função `BubbleSort3()` apresentada na **Seção 11.9.2**, chamando-a para ordenar em ordens crescente e decrescente um array de elementos do tipo **int**.

EP11.7) Muitas extensões da biblioteca padrão de C oferecem uma função, comumente denominada `strdup()`, que cria, utilizando alocação dinâmica de memória, uma cópia de um ***string*** recebido como argumento. Implemente esta função, cujo protótipo é dado por:

```
char *strdup(const char*);
```

EP11.8) Escreva um programa em C, denominado `penta`, que imprima a formação da seleção brasileira na final da copa do mundo de 2002. Este programa deve oferecer duas opções ao usuário introduzidas via linha de comando:

- `E` ou nenhuma opção: imprime os nomes dos jogadores em ordem de formação (i.e., do goleiro ao ponta-esquerda);
- `A`: imprime os nomes dos jogadores em ordem de alfabética crescente

O programa deverá possuir um array de ponteiros para ***strings*** constantes, onde estes ***strings*** representam os nomes dos jogadores e são arranjados seguindo a escalação da seleção em ordem de formação. O programa deverá também utilizar a função **qsort()** da biblioteca padrão de C para ordenar este array quando necessário. Exemplo de uso do programa:

```
$ penta
Escalacao original: Marcos, Cafu, Edmilson, Lucio,
Roque Junior, Roberto Carlos, Gilberto Silva,
Kleberson, Rivaldo, Ronaldinho, Ronaldo

$ penta A
Escalacao ordenada: Cafu, Edmilson, Gilberto Silva,
Kleberson, Lucio, Marcos, Rivaldo, Roberto Carlos,
Ronaldinho, Ronaldo, Roque Junior
```

Sugestão

Este problema não é tão trivial quanto parece. A dificuldade provavelmente aparecerá quando você tentar definir a função de comparação requerida pela função **qsort()**. Consulte a **Seção 11.9.1**, que mostra como uma função de comparação deve ser definida.

EP11.9)

(a) Escreva um módulo em C, denominado `Alunos`, que exporte o seguinte:

(i) O tipo `tAluno`, definido como uma estrutura capaz de conter as seguintes informações sobre um aluno: ***nome***, ***matrícula***, ***três notas*** (variando de 0.0 a 10.0) e a ***média*** dessas notas.

(ii) O tipo `tNo` definido como um nó de uma lista encadeada capaz de conter dados do tipo `tAluno` e o tipo `tTurma` definido como um ponteiro para o tipo `tNo`.

(iii) As macros `MAX_NOME` e `CARACTERES_EM_MATRICULA`, que representam, respectivamente, o número máximo de caracteres permitidos no nome de um aluno e o número (exato) de caracteres permitidos numa matrícula.

(iv) Uma função que leia, no meio de entrada padrão, os dados contidos numa variável do tipo `tAluno`, com exceção da média que será calculada. Esta função deve utilizar funções do módulo ***Interface*** (v. a seguir) para leitura e validação desses dados. A condição de validação da matrícula de um aluno é que ela contenha apenas dígitos na quantidade exata especificada pela constante `CARACTERES_EM_MATRICULA`. A condição de validação do nome de um aluno é que ele contenha apenas letras, com o número máximo de caracteres representado pela constante `MAX_NOME`. Depois de validado, o nome do aluno deve ser armazenado num espaço em memória de tamanho exatamente suficiente para contê-lo.

Sugestões

(1) Declare o campo `nome` da estrutura do tipo `tAluno` como sendo um ponteiro para **char** e não como um arranjo de elementos do tipo **char**. (2) Use a função `strdup()` do exercício **EP11.7** para criar uma cópia de cada ***string*** que representa o nome de um aluno sem desperdiçar espaço em memória e atribua o resultado ao campo `nome` de cada estrutura do tipo `tAluno`. (3) Outras sugestões são aquelas apresentadas para o exercício **EP10.3** do **Capítulo 10**.

(v) Uma função que receba um registro de aluno como entrada e calcule a média do aluno.

(vi) Uma função que receba uma lista encadeada do tipo `tTurma` representando uma turma e calcule a média da turma.

(vii) Uma função que receba uma lista encadeada do tipo `tTurma` representando uma turma e um ***string*** representando uma matrícula e retorne um ponteiro para o elemento da lista (estrutura) cuja matrícula seja igual à matrícula fornecida como parâmetro. Caso a matrícula não seja encontrada na lista, esta função deve retornar **NULL**.

(b) Escreva um módulo em C, denominado ***Leitura***, que exporte o seguinte:

(i) Uma função para leitura de notas de alunos.

(ii) Uma função para leitura de nomes e matrículas de alunos.

Sugestão

Este módulo deverá ser exatamente igual àquele do exercício **EP10.3**.

(c) Escreva um módulo em C, denominado ***Interface***, que exporte o seguinte:

(i) Uma função que faça a apresentação do programa.

(ii) Uma função que faça a despedida do programa.

(iii) Uma função que limpe a tela.

(iv) Uma função que apresente na tela um menu contendo as seguintes opções:

Acrescentar um aluno na turma

Remover um aluno da turma

Modificar um registro de aluno

Enumerar todos os alunos da turma

Imprimir a média da turma

Sair do programa

(v) Uma função que leia e valide opções de menu. Esta função deve retornar 0 se a opção for inválida ou o caractere correspondente se a opção for válida. Utilize para validação desta opção uma função local (**static**) que receba como entrada um caractere representando uma opção e um ***string*** representando a concatenação de todas as opções válidas (por exemplo, no menu apresentado acima, este ***string*** seria `"ARMEIS"`). Esta função local deve retornar 1 se o caractere representar uma opção válida e zero em caso contrário. Além disso, esta função ***não*** deve fazer distinção entre letras maiúsculas e minúsculas.

Sugestão

Este módulo difere do módulo correspondente do exercício **EP10.3** pelo fato de não oferecer a opção de enumeração da turma em nenhuma ordem específica. Portanto, para escrever este módulo, simplesmente remova as opções

de ordenação do módulo correspondente do exercício **EP10.3** e faça as devidas adaptações.

(d) Escreva um módulo em C, denominado `Operacoes`, que implemente as operações oferecidas ao usuário no menu descrito no item (c). (**Sugestão**: Novamente, este módulo difere do módulo correspondente do exercício **EP10.3** pelo fato de neste não haver opção de enumeração da turma em nenhuma ordem específica.)

(e) Escreva um programa que mantenha um conjunto de registros de alunos (turma) numa lista encadeada do tipo `tTurma`. Este programa deve utilizar os módulos descritos nos itens (a), (b), (c) e (d) para oferecer ao usuário as opções de operações apresentadas no menu descrito no item (c), executar cada operação escolhida pelo usuário e apresentar resultados ou mensagens de erro, conforme for o caso.

Sugestões

As diferenças básicas entre este programa e aquele do exercício **EP10.3** são: (1) não existe a opção de enumeração da turma em nenhuma ordem específica, (2) não existe limite quanto ao número de alunos na turma, (3) não há necessidade de manter atualizada uma variável informando o número de alunos na turma (simplesmente, use a função do exercício **EP11.1** para calcular o número de nós na lista), (4) Não há necessidade de movimentar registros sempre que há a inserção ou remoção de um registro.

EP11.10)

(a) Uma ***pilha*** é um tipo especial de lista na qual o acréscimo de um elemento, denominado ***empilhamento***, ocorre apenas no final da lista e a retirada de um elemento, denominado ***desempilhamento***, também ocorre sempre no final da lista. Em outras palavras, o último elemento acrescentado à pilha é sempre o primeiro a ser retirado. (Por causa disto, a pilha é chamada estruturada ***LIFO***, denominação originada do inglês: ***Last In, First Out***). Considerando uma implementação de pilhas por meio de listas encadeadas, escreva funções em C que implementem as seguintes operações sobre pilhas cujos conteúdos dos elementos são do tipo `tElemento` previamente definido.

(i) ***Verificação se uma pilha está vazia***. A função `PilhaEstaVazia()`, que executa esta operação, deve retornar 1 se a pilha recebida como argumento está vazia e zero em caso contrário.

(ii) ***Acréscimo de um elemento a uma pilha***. A função `Empilha()`, que executa esta operação, deve receber como argumentos um ponteiro para uma pilha e um ponteiro para o tipo `tElemento` e retornar 1 se o empilhamento for bem- sucedido ou zero em caso contrário.

Sugestão

Como o tipo do elemento a ser empilhado é um tipo arbitrário denominado `tElemento`, use a função **memcpy()** para copiar o conteúdo do elemento recebido como argumento para o conteúdo do nó armazenado na pilha.

(iii) ***Retirada de um elemento de uma pilha***. A função `Desempilha()`, que executa esta operação, deve receber como argumento um ponteiro para uma pilha e retornar um ponteiro para o elemento desempilhado se a operação for bem-sucedida ou um ponteiro nulo em caso contrário.

Sugestão

Como o tipo do elemento a ser empilhado é um tipo arbitrário denominado `tElemento`, use a função **memcpy()** para copiar o conteúdo do nó armazenado na pilha para o conteúdo do elemento recebido como argumento para.

(b) Escreva um programa em C que possua uma pilha cujo conteúdo (efetivo) dos elementos são ***strings*** alocados dinamicamente. Este programa deve oferecer as opções de empilhamento e desempilhamento de ***strings*** introduzidos pelo usuário. No caso de empilhamento, o programa deve receber um ***string*** como entrada e armazená-lo na pilha ocupando o espaço exato requerido para armazenamento do ***string***. No caso de desempilhamento, o programa deve apresentar ao usuário o ***string*** desempilhado.

Sugestões

(1) Use a função `strdup()` do exercício **EP11.7** para armazenar os ***strings***. (2) Aqui, o tipo `tElemento` é **char *****; i.e., apenas os ponteiros que representam os strings são armazenados na pilha. (3) Como os ***strings*** serão armazenados em espaços alocados dinamicamente pela função `strdup()`, use a função **free()** para liberar o espaço ocupado por um string quando ele for desempilhado.

EP11.11)

(a) Uma fila é um outro tipo especial de lista na qual o acréscimo de elementos ocorre apenas no final da lista, enquanto a retirada de elementos ocorre sempre no início da lista. Em outras palavras, o primeiro elemento acrescentado na fila é sempre o primeiro a ser retirado. (Por causa disto, a fila é chamada estrutura FIFO, denominação originada do inglês: ***First***

***I**n, **F**irst **O**ut*). Considerando uma implementação de filas por meio de listas encadeadas, escreva funções em C que implementem as seguintes operações sobre uma fila cujos conteúdos dos elementos são do tipo `tElemento` previamente definido.

(i) ***Verificação se uma fila está vazia***. A função `FilaEstaVazia()`, que executa esta operação, deve retornar 1 se a fila recebida como argumento estiver vazia ou zero em caso contrário.

(ii) ***Acréscimo de um elemento a uma fila***. A função `Acrescenta()`, que executa esta operação, deve receber como argumentos um ponteiro para uma fila e um ponteiro para o tipo `tElemento` e retornar 1 se o acréscimo for bem-sucedido ou zero em caso contrário.

(iii) ***Retirada de um elemento de uma fila***. A função `Remove()`, que executa esta operação, deve receber como argumento um ponteiro para uma fila e retornar um ponteiro para o elemento removido se a operação for bem-sucedida ou um ponteiro nulo em caso contrário.

(b) Escreva um programa em C que gerencie uma lista de espera implementada como uma fila cujo conteúdo (efetivo) dos elementos são estruturas do tipo `tPassageiro` definido na **Seção 11.1**. Este programa deve oferecer as opções de acréscimo e retirada de passageiros da lista de espera. No caso de acréscimo, o programa deve receber as informações sobre cada passageiro no meio de entrada padrão e armazená-las na fila. No caso de remoção, o programa deve apresentar ao usuário as informações contidas no nó removido da fila.