6

Listas lineares ligadas

Nos capítulos anteriores, as pilhas e as filas foram representadas usando-se o armazenamento seqüencial implementado por um vetor. Esse tipo de representação tem a vantagem de favorecer a compreensão e a correspondente implementação das pilhas e filas em uma linguagem de programação.

Entretanto, devido à necessidade de estipular antecipadamente a quantidade de elementos que um vetor pode conter, existe a possibilidade de haver um subdimensionamento ou superdimensionamento do tamanho do vetor. No caso do subdimensionamento, existirá ainda a possibilidade do '*overflow*'.

Para ilustrar a inconveniência de trabalhar sempre com vetores para representar pilhas e filas, suponha a implementação de três pilhas: s_1, s_2 e s_3.

```
#define MAXPILHA 100
struct stack {
      int topo;
      int item [MAXPILHA];
};
struct stack s1,s2,s3
```

Embora s_1, s_2 e s_3 tenham sido declaradas com o tamanho máximo de cem elementos, considere que durante a manipulação real dessas pilhas, s_1, s_2 e s_3 têm, respectivamente, 40, 20 e 150 elementos no máximo.

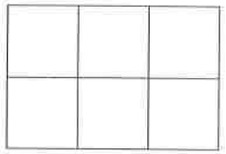
Percebemos que nas pilhas s_1 e s_2 não haverá a possibilidade de *overflow*, ao passo que na pilha s_3 isso já será possível. Notamos que existe um desperdício de 60 células de memória na pilha s_1 e 80 células de memória na pilha s_2. Essas 140 células de memória poderiam ser usadas pela pilha s_3, se houvesse o compartilhamento dessas células de memória pelas três pilhas.

A memória ainda é um recurso escasso e de preço elevado nos sistemas de computação. Portanto, existe a preocupação de encontrar um mecanismo que faça a alocação de memória somente no momento realmente necessário para armazenar alguma informação, e que exista também um processo de liberação dessa área de memória quando não se precisa mais dessa

informação, tornando possível seu uso para a realização do armazenamento de outras informações. Esse mecanismo pode estar disponível por meio do uso das listas ligadas.

Para ilustrar como o compartilhamento de células de memória pode ser útil no uso racional de espaço de memória, vamos considerar a figura a seguir, que representa, de forma abstrata, uma parte da memória de um computador à medida que ocorre o empilhamento e o desempilhamento de elementos nas três pilhas s_1, s_2 e s_3.

Momento 1
Observe que as células de memória disponíveis estão livres.



Momento 2
Considere a realização das seguintes operações *push* sobre as três pilhas s_1, s_2 e s_3: *Push(s_1, A), Push(s_2, B), Push(s_2, C), Push(s_1, D)* e *Push(s_3, E)*.

A escolha exata de qual célula será usada para armazenar os elementos desejados é irrelevante, pois em um computador quem faz esse gerenciamento é o próprio sistema operacional. Após a realização dessas cinco operações *push*, as células de memória poderiam estar como segue:

E	A	C
B	D	

Momento 3
Na situação atual, temos apenas uma célula de memória que poderia ser ocupada pela operação *push* para qualquer uma das pilhas. Vamos considerar que ocorre a seguinte operação: *Push(s_3,F)*. As células de memória ficariam como é exibido abaixo:

E	A	C
B	D	F

Observamos que não existe mais células de memória disponíveis para armazenar informações. Na prática, isso corresponde a atingir o limite de memória física disponível para as alocações ligadas.

Momento 4
Agora, vamos considerar a realização das seguintes operações: *Pop(s_2), Pop(s_3), Pop(s_1)* e *Push(s_3,G)*. As células de memória poderiam ficar exibidas como:

E	A	
B		G

Acompanhando a evolução da alocação de memória do exemplo anterior, percebemos que no decorrer da execução de uma aplicação a ordem das células disponíveis em um computador pode mudar de forma substancial. De fato, o mais importante não é manter necessariamente um controle físico sobre as células, e sim um controle lógico de como está ocorrendo a ocupação e a liberação de cada célula de memória. Todo esse gerenciamento das células de memória disponíveis, como já foi dito, é realizado pelo sistema operacional. É óbvio, portanto, que a quantidade de células de memória disponíveis para a alocação ligada pode variar de sistema para sistema.

Detalhando a estrutura das listas ligadas

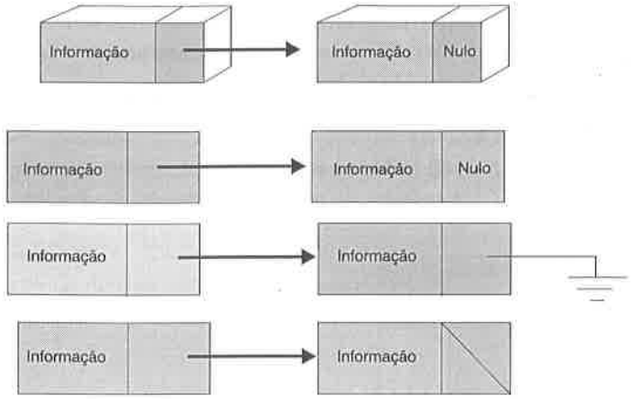
Com o uso de listas ligadas não é necessário usar áreas de memória contígua. Conforme já foi apresentado no Capítulo 1, as listas ligadas utilizam a chamada alocação encadeada, que pressupõe o uso de nós. Cada nó contém duas partes distintas, chamadas de campos; o primeiro campo contém a informação propriamente dita, e o segundo campo contém o endereço do próximo nó.

Se soubermos o endereço do primeiro nó, o segundo é atingido utilizando o endereço que estará no primeiro nó, e assim sucessivamente até chegarmos no último nó, quando o endereço será nulo. O endereço do primeiro nó é armazenado em uma variável externa sob a forma de um ponteiro.

Uma lista ligada sem nós é uma lista nula ou vazia. Neste caso, o ponteiro externo que aponta para o início da lista ligada é de valor nulo. Uma lista ligada pode ser iniciada realizando-se uma operação do tipo plist = NULL.

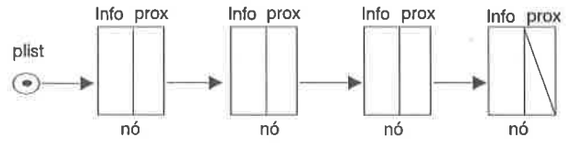
As listas ligadas podem ser representadas de forma abstrata de diversas maneiras. A seguir, estão algumas possibilidades:

Figura 6.1
Formas de representação das listas ligadas.



Todas as possibilidades anteriores são boas escolhas; neste livro, estaremos priorizando a quarta representação. Logo a seguir, podemos observar a representação de uma lista ligada qualquer. Observe a presença da variável externa *plist,* que contém o endereço do primeiro nó da lista ligada e o último nó cujo campo de endereço contém o valor nulo. Observe que, no exemplo da Figura 6.2, o campo *info* é onde colocamos a verdadeira informação de um nó, e o campo *next* é onde está localizado o endereço do próximo nó ou o valor nulo.

Figura 6.2
Uma lista ligada qualquer.



Operações com listas ligadas

Para compreendermos de forma gradativa como se realizam as operações sobre as listas ligadas, vamos discutir inicialmente a operação de inserção de um novo nó em uma lista ligada qualquer.

As duas ilustrações a seguir mostram os principais passos para a inserção de um nó em uma lista ligada já existente.

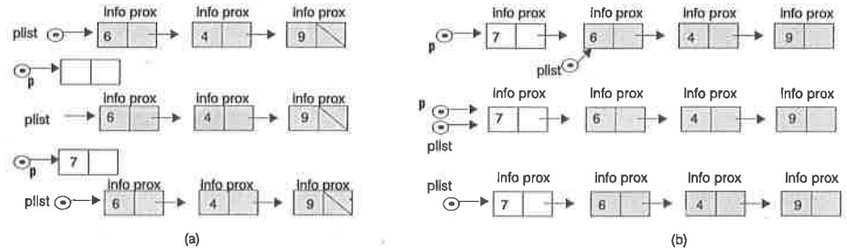
a. Conseguir um endereço p para um novo nó. Deixaremos para mais adiante a explicação de como se obtém um novo nó em uma linguagem de programação;

b. Em seguida, é necessário colocar a informação desejada no respectivo campo do nó apontado por *p*;

c. A seguir, o endereço do primeiro nó existente é colocado no campo de endereço do nó recém-criado;

d. Finalmente o conteúdo da variável externa lista é atualizado com o valor do ponteiro presente em *p*. E o conteúdo de *p* pode, então, ser descartado.

Figura 6.3
Passos para a inserção de um novo nó na lista ligada.



Os quatro passos citados anteriormente e ilustrados na Figura 6.3 podem ser reescritos na forma do seguinte algoritmo:

```
p ←getnodo ( )
info (p) ← x
prox (p) ← plist
plist ← p
```

Detalharemos mais adiante neste capítulo como a função *getnodo ( )* retorna um ponteiro para um nó específico.

A próxima ilustração mostra os passos principais para realizar a remoção de um certo nó de uma lista ligada qualquer.

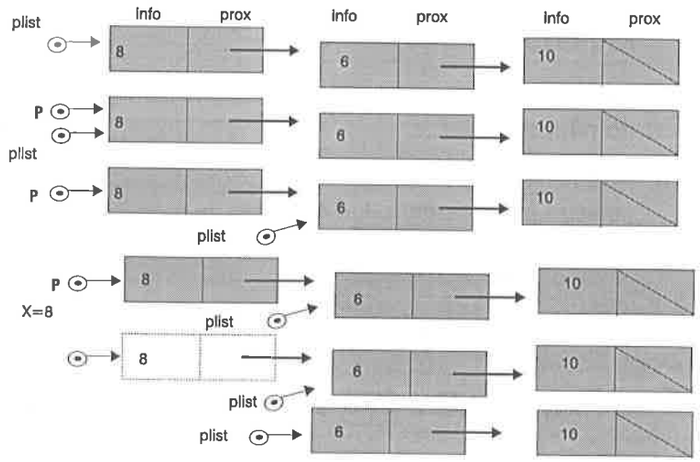
a. Primeiramente, o valor do endereço armazenado em *plist* é guardado na variável *p*;

b. A seguir, o conteúdo do campo endereço do nó a ser excluído é armazenado em *plist;*

c. O conteúdo do campo de informação dos nós a serem excluídos é armazenado em x para uma possível utilização;

d. Finalmente, o endereço dos nó a serem excluídos é fornecido como área de memória livre no sistema para outro uso futuro.

Figura 6.4
Passos para a remoção de um nó da lista ligada.



Os quatros passos anteriores e ilustrados na Figura 6.4 podem ser sintetizados por meio do seguinte algoritmo:

```
p ← plist
plist ← prox (p)
x ← info (p)
freenodo (p)
```

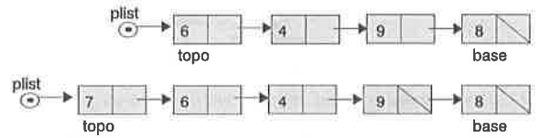
A função *freenodo ( )*, responsável pela liberação da área de memória ocupada por um nó excluído, também será explicada mais adiante.

Representação de pilhas com listas ligadas

Uma lista ligada pode ser usada para representar uma pilha. Como já vimos, a entrada e a saída de um elemento em uma pilha são feitas pelo topo. Então, podemos assumir que a inclusão de um elemento no início de uma lista ligada é semelhante à inclusão de um elemento em uma pilha. No caso de uma pilha, o primeiro nó da lista representaria o topo da pilha. A vantagem significativa em representar pilhas por meio das listas ligadas está na possibilidade de todas as pilhas — que possam eventualmente existir em um programa — cômpartilharem a mesma lista de nós disponíveis, desde que exista esses nós.

Na Figura 6.5, podemos observar a ilustração de uma pilha representada sob a forma de uma lista ligada. Observe que em um primeiro momento, a pilha tem quatro elementos, e o elemento do topo tem como informação o valor 6. Em seguida, é inserido mais um elemento na pilha, representado por uma lista ligada. Agora, o elemento do topo é aquele que possui como informação o valor 7. Observe:

Figura 6.5
Pilha representada por uma lista ligada.



Como o primeiro nó da lista representa seu topo, o algoritmo da operação *push* (s,x) poderia ficar da seguinte forma:

```
p ← getnodo ( )
info (p) ← x
prox (p) ← plist
plist ←p
```

Notamos que, no algoritmo sugerido anteriormente, não foi realizado o teste de *overflow* para a operação de *push (s,x),* pois estamos assumindo que *getnodo ( )* fará este tratamento da forma devida. Consideramos que *s* é o ponteiro para um nó inicial da fila (topo da pilha), ou seja, é como se fosse o *plist* das figuras 6.3 e 6.4.

Uma sugestão de algoritmo da operação *pop (s)* poderia ficar como:

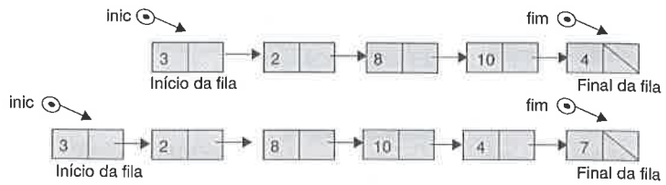
```
p←plist
plist←prox (p)
x←info(p)
freenodo(p)
```

Chamamos sua atenção também para a verificação de que *s* tem o mesmo significado de *plist* das figuras 6.3, 64 e 6.5, ou seja, *s* é o ponteiro para o primeiro nó da lista ligada.

Representação das filas como listas ligadas

Como já foi visto no Capítulo 1, as filas são estruturadas de formas que os elementos entram de um lado e saem por outro. As listas ligadas também podem ser usadas para representar as filas. Nesse caso, o primeiro nó é usado para representar o início da fila, e o último nó para representar seu final. Na Figura 6.6, podemos encontrar a representação de uma fila com cinco elementos e, depois, a mesma fila após a inclusão do sexto elemento com valor 7. Temos ainda a presença de dois ponteiros: *inic* e *fim*, que indicam, respectivamente, o primeiro e o último elemento da fila. Observe:

Figura 6.6
Fila representada por uma lista.



Para remover um elemento de uma fila representada sob a forma de listas ligadas, o algoritmo *remFila(q)* pode ser construído de forma muito semelhante ao algoritmo da operação *pop (s)*. Então, as operações *filaVazia(q)* e *x=remFila(q)* podem ser análogas à *pilhaVazia(s)* e *x=pop(s)*. Assim, basta substituir *s* por *q.inic*. Entretanto, deve-se prestar muita atenção no conteúdo de *q.fim* ao fazer a remoção do último elemento da fila. Seu conteúdo deve ficar nulo, porque, como a fila ficará vazia, tanto *q.inic* como *q.fim* devem ser nulos. A seguir, podemos encontrar uma sugestão de algoritmo para a operação de remoção, *remFila(q)*, que pode ser chamada como *x=remFila(q)*:

```
se (filaVazia(q)= "V")  então
|  imprima "Ocorreu underflow na fila!") //recomenda-se um tratamento adequado
                        senão           //para essa situação
|  p←q.inic
|  x←info(p)
|  q.inic←prox(p)
|  se (q.inic = NULO)    então
|  |            q.fim ← NULO
|  fim se
|  freenodo(p)
fim se
```

Na seqüência, encontramos uma sugestão de algoritmo para a realização da operação *insFila(q,x)*, responsável pela inserção de um elemento na fila representada pelas listas ligadas. Observe:

```
p←getnodo ( )
info (p)←x
prox(p)←NULO
se (q.fim=NULO)    então
|     q.inic ← p
|              senão
|     prox (q.fim)←p
fim se
q.fim←p
```


A representação de pilhas e de filas sob a forma de listas ligadas pode apresentar algumas desvantagens. Entre outras, podemos citar:

- Ocupa mais espaço de armazenamento do que o vetor. Isso ocorre porque cada nó possui, pelo menos, dois campos — um para a informação real e o outro para armazenar o endereço do próximo nó. Lembramos, entretanto, que o espaço de armazenamento não é o dobro, e que ainda é possível compactar posteriormente as informações.
- Cada operação de inclusão e de exclusão da fila corresponde a uma operação de inclusão e de exclusão na lista de nós disponíveis. Isso pode acarretar uma observação na queda do desempenho da aplicação, pois o sistema estará ocupado com essa tarefa de inclusão e exclusão.

Entretanto, mesmo considerando as desvantagens apresentadas, a representação das pilhas e das filas resulta em grande vantagem: a possibilidade de compartilhamento de nós livres, desde que não seja ultrapassado o número de nós disponíveis no sistema.

Listas ligadas como estruturas de dados

As listas ligadas não são importantes apenas porque possibilitam a representação de pilhas e filas. Elas também são isoladamente uma estrutura de dados muito importante, e esse é um bom motivo para estudá-las como tal.

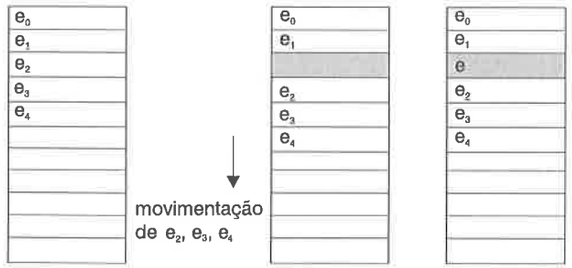
Por exemplo, um elemento presente na lista pode ser acessado percorrendo essa lista desde seu início. Como já vimos, um vetor permite o acesso direto ao *n-ésimo* item com uma única operação. Em uma lista, são exigidas *n* operações, além de ser necessário percorrer cada um dos primeiros *n-1* elementos antes de atingir o *n-ésimo* elemento desejado, se é que ele existe na lista.

Apesar dessa grande desvantagem em relação às listas, no momento da inserção ou da remoção de um elemento aparecerá a grande vantagem, porque não é necessária nenhuma movimentação dos *n* elementos para a realização da inserção ou da remoção do elemento. Basta haver uma atualização adequada dos endereços nos respectivos nós.

A Figura 6.7 ilustra o esforço computacional exigido para a movimentação de elementos durante o processo de inserção de um elemento *'e'* em um vetor.

Figura 6.7

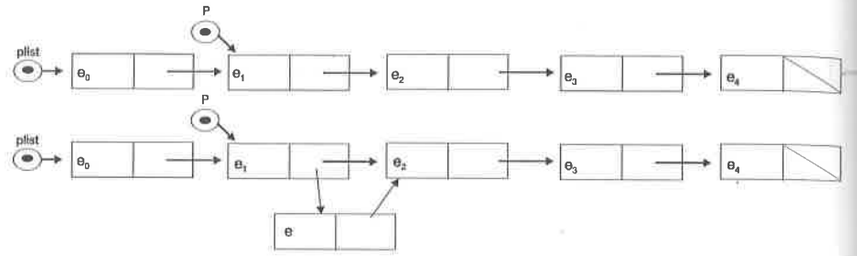
Esforço de movimentação para a inserção de um elemento em um vetor.



Se a inserção for realizada em uma lista ligada, podemos observar, pela ilustração da Figura 6.8, que não existe a necessidade de realizar-se nenhuma movimentação dos elementos já existentes na lista; basta concluir a devida atualização de endereços dos nós. Vale lembrar que esse trabalho de inclusão independe do tamanho da lista; ele será sempre o mesmo.

Figura 6.8

Esforço de movimentação para a inserção de um elemento.



A seguir, estão os algoritmos que fazem a inserção e a remoção de elementos em uma lista ligada. Perceba que *p* aponta para o nó após o qual será inserido o novo nó ou aponta para o nó a ser removido. Observe:

```
insDepois (p,x)
q← getnodo( )
info(q) ← x
prox (q) ←prox (p)
prox (p) ← q
```

```
remDepois (p,x)
q← prox (p)
x ← info (q)
prox (p) ← prox (q)
freenodo (q)
```

Analisando o algoritmo de inserção, podemos perceber que não é possível inserir um item antes de um nó, apenas depois de um determinado nó. Como, então, alcançar esse efeito? Para eliminar um nó também, não basta apenas ter um ponteiro para ele. É necessário alterar o campo *prox* do antecessor do nó removido para apontar para seu sucessor. Como alcançar esse efeito?

A questão da inserção ocorre porque em uma lista linear para identificarmos o antecessor de um nó, é necessário percorrer a lista de seu início. Inserir um novo elemento antes de *nodo(p)*, significa alterar o campo *prox* de seu antecessor para apontar para o nó recém-criado. Mas, baseando-se em *p* não há como identificar o seu antecessor. Ela pode ser resolvida, inserindo-se o novo elemento imediatamente depois de *nodo(p)* e, em seguida, realizar um intercâmbio entre o campo *info(p)* e o campo *info* do nó recém-criado. Também na remoção, não basta ter um ponteiro *p* para o nó a ser removido, pois, é necessário atualizar o campo *prox* de seu antecessor de forma que aponte para o seu sucessor. Para obtermos o efeito desejado, a recomendação é salvar o conteúdo do nó seguinte e, em seguida, eliminá-lo. Depois substituir o conteúdo do nó apontado por *p* pela informação gravada, obtendo, dessa forma, a eliminação do nó, a não ser que ele seja o último na lista. A implementação desses dois algoritmos pode ser um bom exercício para o leitor.

Implementação de listas ligadas na linguagem C

Na linguagem C, temos duas possibilidades a considerar para implementar listas ligadas:

- Implementação de listas como um vetor.
- Implementação de listas de forma dinâmica.

Na implementação de listas como um vetor, a declaração de um nó poderia ocorrer como é exibido a seguir:

```
#define NRNODOS 1000
struct tiponodo {
      int info;
      int prox;
};
struct tiponodo nodo [NRNODOS];
```

Para inicializar esse vetor, poderíamos utilizar a rotina exibida a seguir:

```
disponivel = 0 /* variável global para apontar lista */
for (k = 0; k < NRNODOS - 1 ; k++)
      nodo[ k ].prox = k + 1;
nodo [NRNODOS - 1 ].prox = -1;
```

A idéia de usar um vetor como um depósito de nós disponíveis serve para demonstrar que existe a possibilidade da estrutura de um vetor funcionar de forma análoga ao mecanismo que existe na maioria dos sistemas quando se deseja trabalhar com listas ligadas. No caso da representação do depósito de nós livres com vetores, toda vez que o campo *prox* possuir o valor -1, sinaliza-se que essa posição do vetor é apresentada de forma livre, ou seja, possuímos um nó livre.

As operações *getnodo ( )* e *freenodo ( )* seriam implementadas como é exibido a seguir:

```
getnodo ( )
{
      int p;
      if (disponivel == -1){
      printf("Estouro no número dos nós disponíveis\n");
            exit (1);
      }
p= disponivel;
disponivel = nodo[disponivel].prox;
return (p);
}

freenodo ( int p )
{
            nodo [p].prox = disponivel;
            disponivel = p;
            return;
}
```

A inserção em um vetor que representa uma lista ligada pode ser realizada da seguinte forma:

```
insDepois(int p, int x)
{
      int q;
      if (p== -1) {
            printf ("Inserção nula \n");
            return
      }
      q = getnodo ( );
      nodo [q].info = x;
      nodo [q].prox = nodo [ p].prox;
      nodo [p].prox = q;
      return;
}
```

Para remover um elemento do vetor que representa uma lista ligada, podemos usar a seguinte rotina:

```
remDepois ( int p, *px)
{
      int q;
      if ((p == -1) || (nodo[p].prox == -1)){
            printf("Remoção nula \n");
            return;
      }
      q = nodo [ p].prox;
```



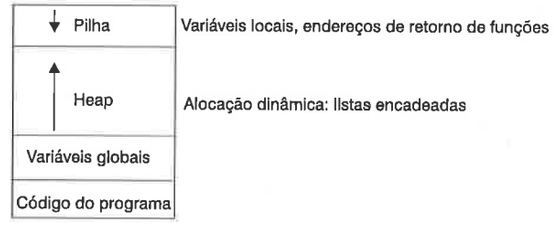
```
      *pq = nodo [q].info;
      nodo[p].prox = nodo [q].prox;
      freenodo (q);
      return;
}
```

As considerações sobre o uso do vetor para representar uma lista ligada passa novamente pela necessidade de estabelecer um conjunto de nós no início da execução da aplicação. Pode ser difícil prever exatamente o número de nós que serão necessários. Novamente, pode ocorrer o problema de subdimensionamento ou superdimensionamento. Além do mais, toda quantidade de nós declarada permanece alocada até o término do programa.

A solução é usar os nós dinâmicos, em vez dos estáticos (vetor). Dessa forma, se um nó é necessário, o espaço de memória fica reservado para ele. Quando o nó não é mais necessário, o espaço é liberado, ficando esse espaço disponível para um outro nó. Vale lembrar que, com nós dinâmicos, um limite predefinido do número de nós não é estabelecido. Desde que haja espaço de memória para o programa na sua totalidade, parte desse espaço pode ser reservado e usado como um nó.

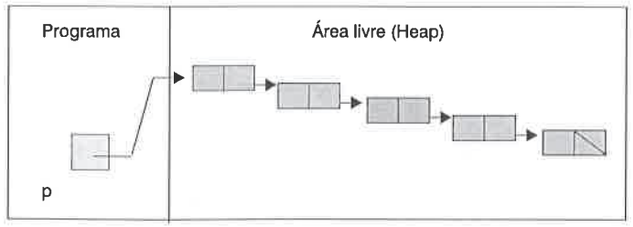
A apresentação do mapa conceitual da memória do computador durante a execução de um programa em C pode ajudar a compreender como são gerados os nós usados nas listas ligadas. Observe:

Figura 6.9
Mapa conceitual de um programa em C.



Se fosse possível fotografar a memória do computador no momento do uso das listas ligadas, teríamos uma imagem como a exibida a seguir:

Figura 6.10
Listas ligadas na memória do computador.



A alocação da área livre no *heap* na linguagem C (Figura 6.10) é realizada pela função *malloc ( )*. Se a chamada da função *malloc ( )* for bem-sucedida, será retornado um ponteiro para o exato primeiro *byte* (endereço) da região de memória alocada do *heap*. Entretanto, se não existir memória disponível para satisfazer essa necessidade de alocação, ocorrerá uma falha nesse processo de alocação, e *malloc ( )* retornará um valor nulo. Por exemplo, o fragmento de código abaixo mostra como se aloca 100 *bytes* de memória:

```
char *p;
p= malloc (100); /* é obtido 100 bytes */
```

Após a atribuição, o ponteiro *p* estará apontando para o primeiro *byte* do conjunto de 100 *bytes* de memória livre alocada do *heap*.

Para alocar um espaço de memória para a alocação de 10 inteiros, podemos usar o *malloc ( )* com o *sizeof* para assegurar a portatibilidade do código, pois o número de *bytes* de um tipo de dados pode variar conforme o sistema utilizado.

```
int *p;
p = malloc (10*sizeof(int));
```

Já a liberação da memória alocada anteriormente por *malloc ( )* ao *heap* para uma possível reutilização é realizada pela função *free ( )*.

Ambas as funções estão na biblioteca C <*stdlib.h*>. É importante que *free( )* seja chamada somente com um ponteiro válido, alocado anteriormente; caso contrário, poderá haver comprometimento na organização do *heap*, podendo, inclusive, causar um 'travamento' do programa.

A seguir, estão alguns exemplos do uso de *malloc ( )*. Observe:

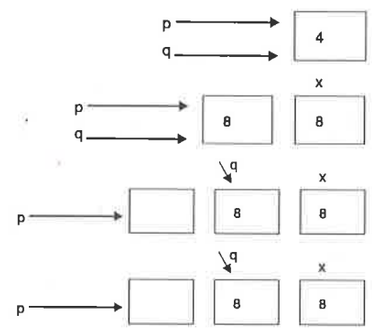
```
a.  char * p
b.  p = (char *) malloc (30);
c.  int *p;
    p = (int * ) malloc (40* sizeof(int));
d.  int *p;
```



```
if ((p = (int * ) malloc (200) = = NULL){
     printf("Falta de memória!\n");
     exit (1);
}
```

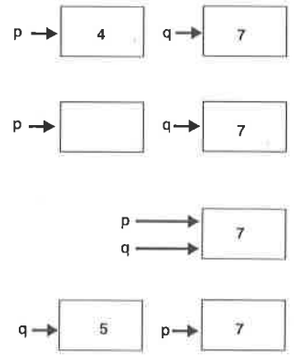
A macro *NULL* está definida em <*stdlib.h*>. Agora, vamos apresentar mais exemplos de alocação dinâmica com ilustrações dos ponteiros. Do lado direito, apresentamos várias operações usando a alocação dinâmica, e do lado direito colocamos desenhos que ilustram o relacionamento dos ponteiros com as variáveis. Observe a presença das chamadas variáveis anônimas, das quais sabemos apenas o endereço, ou seja, elas não têm nome. Acompanhe todo o processo com cuidado e, caso precise, consulte mais detalhes no Apêndice A.

```
int *p, *q
int x;
p= (int *) malloc (sizeof (int));
*p=4;
q=p;
printf("%d %d\n),*p,*q);
x=8;
*q=x;
printf("%d %d\n",*p,*q);
p=(int*) malloc (sizeof (int));

*p=6;
printf("%d %d\n",*p,*q);
```



A seguir, estão alguns exemplos de *malloc ( )* e *free ( )*. Observe:

```
p=(int*)malloc (sizeof(int));
*p=4;

q=(int*) malloc(sizeof(int));
*q=7;

free(p);
p=q;
q=(int*) malloc(sizeof(int));
*q=5;
printf("%d %d\n", *p , *q);
```



Na utilização de *malloc ( )* e *free ( )*, comete-se muitas vezes erros terríveis, porque eles podem causar a perda total de listas ligadas. Vejamos alguns exemplos desses enganos:

```
p=(int *malloc(sizeof(int));
*p=4;
p=(int*)malloc(sizeof(int));
*p=8;
```

Analisando as quatro instruções anteriores, podemos observar que a primeira retorna um endereço de área dinâmica para *p*. Em seguida, é armazenado o valor 4 neste endereço. Na terceira instrução, é realizado novamente um novo alocamento dinâmico, sem antes ter sido aplicado o *free (p)*. Esse equívoco faz com que a primeira variável dinâmica esteja irremediavelmente perdida. Seu espaço de memória não pode sequer ser liberado por *free ( )* e, dessa forma, ser possivelmente reaproveitado em outras alocações dinâmicas.

Declarando um nó de forma dinâmica

Lembramos que uma lista ligada é um conjunto de nós. Cada nó tem dois campos: o primeiro é usado para conter a informação útil do nó, enquanto o segundo é usado para armazenar um ponteiro (endereço) do próximo nó da lista. O primeiro nó da lista ligada é identificado por um ponteiro externo. Para declarar uma estrutura de lista ligada na linguagem C, é necessário que um dos membros da estrutura seja uma variável ponteiro para uma estrutura do mesmo tipo. Então, o tipo de um ponteiro de um nó pode ser definido como:

```
struct nodo {
      int info;
      struct nodo *prox;
};
typedef struct nodo *NODOPTR
```

Um ponteiro *p* pode ser declarado da seguinte forma:

```
NODOPTR p;
```

A operação *getnodo ( )* pode ficar da forma como é exibida abaixo:

```
NODOPTR getnodo ( )
{
      NODOPTR p;
      p = (NODOPTR) malloc (sizeof (struct nodo));
      return (p);
}
```


A execução do comando *p = getnodo ( )* deve colocar o endereço de um nó disponível em *p*. O comando *free (p)*, ao ser executado, deve liberar o nó com o endereço *p* para a lista dos nós livres. Observe:

```
freenodo ( NODOPTR p )
{
      free ( p );
}
```

Como podemos perceber, as funções *getnodo* e *freenodo* são simples; portanto, podem ser usadas diretamente no programa como é exibido a seguir:

```
p = (NODOPTR) malloc (sizeof (struct nodo));

free ( p );
```

Evidenciamos que não é mais responsabilidade do programador controlar o gerenciamento da quantidade de nós disponíveis. É o sistema quem faz automaticamente a alocação e a liberação de nós. Não há também a necessidade de prever a ocorrência do estouro no momento da alocação de novos nós, porque essa condição será detectada na execução de *malloc ( )*, e o sistema retornará, nesse caso, o valor *NULL*.

A seguir, estão as definições das funções *insDepois (p, x)* e *remDepois (p, px)*, que fazem, respectivamente, a inserção e a remoção de um elemento na lista ligada. Observe que as duas operações fazem a inserção ou a remoção depois de um certo nó:

```
insDepois (NODOPTR p, int x)
{
      NODOPTR q;
      if (p = = NULL){
      printf("Inserção nula\n");
      exit (1);
      }
      q = getnodo ( );
      q → info = x;
      q → prox = p → prox;
      p → prox = q;
}

remDepois( NODOPTR p, int *px)
{
      NODOPTR q;
      if (( p == NULL) || (p → prox == NULL)){
            printf("Remoção nula \n");
            exit (1);
      }
      q = p → prox;
      *px = q → info;
      p → prox = q → prox;
      freenodo(q);
}
```

As filas, assim como as listas ligadas, podem usar os vetores ou a alocação dinâmica. Para facilitar a comparação entre as duas possibilidades, colocou-se a seguir uma implementação do lado da outra. Assumiu-se que as estruturas *nodo* e *NODOPTR* foram previamente declaradas. Observe:

Implementação com vetor

```
struct queue
   int inic, fim;
};
struct queue q;

filaVazia(struct queue *pq)
{
   if (pq →inic == -1)
      Return 1; // verdadeiro
   else
      return 0; // falso
}
insFila(struct queue *pq, int x)
{
   int p;
   p=getnodo ();
   nodo[p].info = x;
   nodo[p].prox= -1;
   if (pq→ fim == -1)
      pq →inic =p;
   else
      nodo[pq→fim].prox=p;
   pq→fim=p;
}

remFila(struct queue *pq)
{
   int p,x;
      int x;
   if (filaVazia(pq)) {
      printf("Ocorreu underflow
      na fila\n");
      exit(1);
   }
   p=pq→inic;
   x=nodo[p].info;
   pq→inic = nodo[p].prox;
   if (pq→inic == -1)
      pq→fim = -1;
   freenodo (p);
   return (x);
}
```

Implementação de forma dinâmica

```
struct queue {
    NODOPTR inic, fim;
};
struct queue q;

filaVazia(struct queue *pq)
{
    if (pq→inic == NULL)
        Return 1; verdadeiro
    Else
        Return 0; // falso
}
insFila(struct queue *pq, int x)
{
    NODOPTR p;
    p=getnodo ();
    P→info =x;
    p→prox =NULL;
    if (pq→fim ==NULL)
        pq→inic=p;
    Else
        (pq→fim)→prox =p;
    pq→fim=p;
}

remFila(struct queue *pq)
{
    NODOPTR p;

    if (filaVazia(pq) {
        printf("Ocorreu underflow na
        fila\n");
        exit(1);
}
    p=pq→inic;
    x=p→info;
    pq→inic=p→prox;
    if (pq→inic ==NULL)
        pq→fim = NULL;
    freenodo (p);
    Return
}
```


Sugestões de programas com operações sobre listas ligadas

A seguir, apresentamos várias sugestões de implementação de alguns programas em C que fazem a manipulação de listas ligadas. Para todos os programas, será pressuposto o uso da biblioteca *'sugest.h'*, detalhada a seguir e que contém todos os protótipos e definições das funções usadas nos programas de exemplos.

```
/*********************************************************/
/* Protótipos das funções para os programas sugeridos */
/*********************************************************/
/* Estrutura do nó */
struct nodo {
      int num;
      int info;
      struct nodo *prox;
};
typedef struct nodo *NODOPTR;
void insDepois(NODOPTR p, int x);
void remDepois(NODOPTR p, int *px, int *py);
void freenodo(NODOPTR p);
NODOPTR getnodo(void);
void insFim(NODOPTR *plist, int x);
NODOPTR localiza(NODOPTR plist, int x);
void remTudo(NODOPTR *plist, int x);

/*********************************************************/
/* Definição das funções para os programas sugeridos */
/*********************************************************/
/* Insere um nó, depois de outro */
void insDepois(NODOPTR p, int x)
{
      NODOPTR q;
      if (p==NULL) {
            printf ("Inserção nula\n");
            exit (1);
      }
      q=getnodo();
      q->num=x;
      q ->info =x*10;
      q ->prox = p->prox;
      p->prox=q;
}

/* Aquisição de um novo nó */
NODOPTR getnodo()
{
      NODOPTR p;
      p=(NODOPTR)malloc (sizeof(struct nodo));
```

```
        return (p);
}

/*Exclui um nó, depois de um outro */
void remDepois(NODOPTR p, int *px,int *py)
{
        NODOPTR q;
        if ((p==NULL || (p->prox ==NULL))) {
                printf ("Remoção nula\n");
                exit(1);
        }
        q=p->prox;
        *px=q->num;
        *py=q->info;
        p->prox=q->prox;
        freenodo(q);
}
/*Libera um nó */
void freenodo(NODOPTR p)
{
        free(p);
}

/*Insere um nó no final da lista ligada */
void insFim(NODOPTR *plist, int x)
{
        NODOPTR p,q;
        p=getnodo();
        p->num=x;
        p->info=x*10;
        p->prox=NULL;
        if (*plist==NULL)
                *plist=p;
        else{
                /*procura o ultimo no */
                for (q=*plist;q->prox !=NULL;q=q->prox)
                        ;
                q->prox=p;
        }
}

/* Localiza um nó com certa informação */
NODOPTR localiza(NODOPTR plist, int x)
{
        NODOPTR p;
        for (p=plist;p!=NULL;p=p->prox)
                if (p->info ==x)
                        return (p);
        /* elemento nao esta' na lista */
        return (NULL);
}
```



```
/*Remove todos nós com certa informação */
void remTudo(NODOPTR *plist, int x)
{
        NODOPTR p,q;
        int y;
        q=NULL;
        p=*plist;
        while (p!=NULL)
                if (p->info ==x) {
                        p= p->prox;
                        if (q==NULL) {
        /* remove o primeiro no da lista*/
        freenodo (*plist);
        *plist=p;
                }
                else
                        remDepois(q,&y);
                }
                else {
                /* avança para o próximo no da lista */
                q = p;
                p = p->prox;
        }
}
```

O programa a seguir cria seis nós e, em seguida, realiza a exclusão desse conjunto de nós. Para tanto, são utilizadas as funções *insDepois ( )* e *remDepois ( )* presentes na biblioteca *'sugest.h'*. Procure realizar a execução do programa em um computador para verificar o comportamento de funcionamento das listas ligadas.

```
/*************************************************************************/
/* Ilustração do funcionamento de listas singularmente ligadas    */
/* Seis nós são criados e, em seguida, excluídos                  */
/*************************************************************************/
#include "sugest.h"

typedef struct nodo *NODOPTR;

NODOPTR plist=NULL;

void cria_1o_no(int x);
void exibe();
void elimina(void);

void main (void)
{
        int i,nr=1;
        clrscr();
```

```
        cria_1o_no(nr);
        for (i=1;i<=5;i++)
             {
                  nr=nr+1;
                  insDepois(plist,nr);
             }
        exibe();
        printf("\nDigite <enter> para continuar . . .");
        getch();
        elimina();
        printf("\nDigite <enter> para continuar . . .");
        exibe();
        printf("\nDigite <enter> para finaliza . . .");
        getch();
}
void cria_1o_no(int x)
{
        NODOPTR q;
        q=getnodo();
        q->num=x;
        q->info=x*10;
        q->prox=NULL;
        plist=q;
}

void exibe ()
{
        NODOPTR nodo;
        nodo=plist;
        printf("\nExibe nos");
        printf("\n*********");
        printf("\nNodo nr. : %i Nodo inf. : %i",nodo->num,nodo->info);
        printf("<primeiro no>");
        nodo=nodo->prox;
        while (nodo!=NULL)
             {
             printf ("\nNodo nr. : %i Nodo inf. : %i",nodo->num,nodo->info);
             nodo=nodo->prox;
             }
        printf("<ultimo no>");
}

void elimina(void)
{
        NODOPTR nodo;
        int px=0;
        int py=0;
        nodo=plist;
        printf("\nExclusao de todos nos");
        printf("\n**********************");
        while (nodo!=NULL)
```



```
             {
                  remDepois(plist, &px,&py);
                  printf("\nNodo nr. %i excluido com informacao : %i",px,py);
                  nodo=plist->prox;
             }

        printf("\nNodo nr. %i excluido com informacao : %i",plist->num,plist->info);
        freenodo(plist);
        getch();
}
```

O programa a seguir mostra como é possível, utilizando a função *insFim ( )*, fazer a inserção de novos nós no final de uma lista ligada. Observe mais uma vez a presença da utilização da biblioteca *'sugest.h'*:

```
/*********************************************************************/
/* Ilustração do funcionamento de listas singularmente ligadas       */
/* Os nós são criados no final de uma lista ligada                   */
/*********************************************************************/

#include "sugest.h"

typedef struct nodo *NODOPTR;

NODOPTR plist=NULL;
/* Protótipo da função exibe */
void exibe();

void main (void)
{
        int i,nr=0;
        clrscr();
        for (i=1;i<=5;i++)
             {
                  nr=nr+1;
                  insFim(&plist,nr);
             }
        exibe();
        printf("\nDigite <enter> para finalizar . . .");
        getch();
}

void exibe ()
{
        NODOPTR nodo;
        nodo=plist;
        clrscr();
        printf("\nExibe nos apos inclusao");
        printf("\n**********************");
```

```
        printf("\nNodo nr. : %i Nodo inf. : %i",nodo->num,nodo->info);
        printf("<primeiro no>");
        nodo=nodo->prox;
        while (nodo!=NULL)
             {
                  printf ("\nNodo nr. : %i Nodo inf. : %i",nodo->num,
nodo->info);
                  nodo=nodo->prox;
             }
        printf("<ultimo no>");
}
```

Na necessidade de pesquisar a ocorrência de um determinado elemento nos nós presentes em uma lista ligada, podemos utilizar a função *localiza ()*, também presente em *'sugest.h'*, cuja sugestão de codificação é apresentada a seguir:

```
/*********************************************************************/
/* Ilustração do funcionamento de listas singularmente ligadas       */
/* A rotina localizada realiza a pesquisa da ocorrência de um        */
/* valor específico na lista ligada                                  */
/*********************************************************************/

#include "sugest.h"

typedef struct nodo *NODOPTR;

NODOPTR plist=NULL;

void exibe();

void main (void)
{
     int i,nr=0;
     NODOPTR ptr;
     clrscr();
     for (i=1;i<=5;i++)
          {
               nr=nr+1;
               insFim(&plist,nr);
          }
     exibe();
     /* Procura um elemento na lista */
     printf("\n\nQual informacao desejada : ");
     scanf("%i",&nr);
     ptr=localiza(plist,nr);
     if (ptr==NULL)
          printf("\nInformacao nao encontrada!");
     else
          printf("\nInformacao presente!");
     getch();
```



```
}

void exibe ()
{
     NODOPTR nodo;
     nodo=plist;
     clrscr();
     printf("\nExibe nos");
     printf("\n*********");
     printf("\nNodo nr. : %i  Nodo inf.  : %i",nodo->num,nodo->info);
     printf("<primeiro no>");
     nodo=nodo->prox;
     while (nodo!=NULL)
          {
            printf ("\nNodo nr. : %i  Nodo inf.  : %i",nodo->num,nodo->info);
            nodo=nodo->prox;
          }
     printf("<ultimo no>");
}
```

Em determinadas situações, pode-se querer determinar não só se um elemento está ou não em uma lista ligada, mas também remover todos os nós que contenham uma determinada informação. O programa sugerido abaixo, que faz uso da função *remTudo ()*, realiza essa tarefa. Observe:

```
/*********************************************************************/
/* Ilustração do funcionamento de listas singularmente ligadas       */
/* Elimina todos os nós cujo campo info contém o valor de x          */
/*********************************************************************/

#include "sugest.h"

typedef struct nodo *NODOPTR;

NODOPTR plist=NULL;

void exibe();

void main (void)
{
     int i,nr=0;
     NODOPTR ptr;
     clrscr();
     for (i=1;i<=5;i++)
          {
               nr=nr+1;
               insFim(&plist,nr);
          }
     exibe();
```

```
        /* Elimina elementos com certas informações */
        printf("\n\nQual informacao desejada : ");
        scanf("%i",&nr);
        remTudo(&plist, nr);
        exibe();
        printf("\nDigite <enter> para finalizar . . .");
        getch();
}

void exibe ()
{
        NODOPTR nodo;
        nodo=plist;
        clrscr();
        printf("\nExibe nos");
        printf("\n*********");
        printf("\nNodo nr. : %i  Nodo inf.  : %i",nodo->num,nodo->info);
        printf("<primeiro no>");
        nodo=nodo->prox;
        while (nodo!=NULL)
                {
                  printf ("\nNodo nr. : %i  Nodo inf.  : %i",nodo->num,nodo->info);
                nodo=nodo->prox;
                }
      printf("<ultimo no>");
}
```

Classificando as listas de acordo com suas operações efetuadas

Considerando-se as operações efetuadas sobre as listas, podemos classificá-las como :

a. Listas ordenadas;

b. Lista ordenada como fila de prioridade ascendente;

c. Lista ordenada como fila de prioridade descendente.

Uma lista ordenada $L_{ordenada}$:{e_1,e_2,..., e_n} é uma lista linear de elementos, tal que, se n > 1, podemos observar as seguintes relações de ordem nesses elementos:

- $e_1 \leq e_k$, para qualquer $1 < k \leq n$;
- $e_k \leq e_n$, para qualquer $1 \leq k < n$;
- $e_{k-1} \leq e_k \leq e_{k+1}$, para qualquer $1 < k < n$.

As relações anteriormente apresentadas garantem que nenhum elemento presente em $L_{ordenada}$ seja inferior a e_1 ou superior a e_n. Se pegarmos qualquer elemento no meio da lista, nenhum elemento à esquerda irá superá-lo e nenhum elemento à direita será inferior a ele. Em uma lista ordenada, são possíveis as operações de inserção, remoção e localização de um elemento.

As listas ordenadas com filas de prioridade são estruturas de dados na qual a ordem em que se deseja manter sobre seus elementos é o que determina a definição das operações básicas. Uma fila de prioridade pode ser de dois tipos:

- Ascendente: Os elementos são inseridos de forma arbitrária, porém é permitida apenas a remoção do menor elemento existente.
- Descendente: Os elementos são inseridos também de forma arbitrária, porém é permitida apenas a remoção do maior elemento existente.

Podemos identificar imediatamente duas operações sobre a fila de prioridade: a inserção e a remoção. A operação *insFp(qfpa,x)*, ou *insFp(qfpd,x)*, é usada para inserir um elemento em uma fila de prioridade ascendente *(qfpa)* ou descendente *(qfpd)*. Para remover e retornar o menor elemento em uma fila de prioridades ascendente, contudo, podemos aplicar a operação *remMinfp(qfpa)*. Se a fila é de prioridade descendente, podemos considerar a aplicação da operação *remMaxfp(qfpd)* para remover e retornar o maior elemento em uma fila de prioridades descendentes.

Podemos notar que a execução sucessiva de *remMinfp(qfpa)* removerá sempre o menor elemento existente em uma fila de prioridade ascendente. Mesmo depois de uma série de inclusões com *insFp(qfpa)*, se o último elemento inserido for menor em relação ao conjunto de elementos que já estavam na fila de prioridades ascendentes, ele será removido na aplicação da operação *remMinfd(qfpa)*. Essa observação explica a razão de especificarmos uma fila de prioridade como ascendente ou descendente.

Em relação à implementação de uma fila de prioridades, podemos fazer algumas considerações na hora de manter a ordem intrínseca em uma fila de prioridade. Se houver uma freqüência maior de inserções do que remoções, podemos optar em não realizar nenhuma classificação nos elementos no momento da inserção. Isso poderia representar ganhos de processamento, pois não seria necessário movimentar os elementos existentes como forma de manter a ordem previamente existente. Faríamos, dessa forma, a devida varredura no conjunto de elementos no momento da remoção para identificar e remover o menor ou o maior elemento da fila de prioridades.

Se a implementação da fila de prioridades for realizada por meio de um vetor, podemos colocar um certo valor em uma de suas posições. Por exemplo, -1 para indicar a remoção lógica de um elemento da fila de prioridade. Depois, de vez em quando, poderíamos realizar uma operação de compactação, ou seja, proceder a reorganização dos elementos com sua devida

movimentação para as posições certas. Notamos que esse processo de compactação pode exigir um razoável esforço computacional. Além disso, não seriam permitidas inclusões ou remoções na fila de prioridades durante sua realização.

De forma geral, a recomendação é que não se mantenha a fila de prioridades como um vetor sem classificação; em vez disso, deve-se mantê-la como um vetor classificado e na forma circular (é a mesma idéia da fila circular). Perceba que a responsabilidade de manter o vetor classificado passa a ser da operação de inserção; porém, como o vetor estará classificado, a identificação da posição de inserção poderá ser feita por meio de uma busca binária, que reduz de forma significativa a identificação do ponto da nova inserção. Não fugiremos da necessidade de executar uma operação que movimente os itens de forma adequada, mas a remoção de um elemento se tornará bem mais simples, porque exige apenas o aumento de *pq.inic* (fila ascendente) ou a diminuição de *pq.fim* (fila descendente). Como forma de ilustrar a implementação circular sugerida, a seguir podemos verificar uma sugestão da estrutura de uma fila de prioridades. Recomendamos que o leitor procure, como prática, desenvolver as operações sugeridas anteriormente.

```
#define MAXPQ
struct pfila {
      int elementos[MAXPQ]
      int inic, fim;
}
struct pfila pq;
```

Podemos perceber que o uso das filas de prioridade pode resultar em soluções muito interessantes para problemas rotineiros, porém será sempre necessário considerar o esforço computacional necessário para manter a ordem dos elementos nos casos de novas inserções e remoções. Essas tarefas podem gerar implementações ineficientes, e esse é um dos motivos que devemos considerar em relação ao uso de tais estruturas.

Polinômios: um exemplo de uso das filas ordenadas

Um bom exemplo da utilização das filas ordenadas é a representação de polinômios. A premissa de ordenação existente nas filas ordenadas é o fator ideal para implementar um tipo de dados que permita a criação de variáveis capazes de armazenar polinômios na forma:

$$P(X) = C_nX^n + C_{n-1}X^{n-1} + ... + C_KX^K + ... + C_3X^3 + C_2X^2 + C_1X^1 + C_0$$

Observando a estrutura anterior, percebemos que um polinômio pode ser representado por um conjunto de pares que associam para um dado X o correspondente coeficiente e potência.

Por exemplo, $P(X) = 4X^5 - 11X^3 + 3X^2 - 4$ seria representado pelos pares {(4,5), (-11,3),(3,2),(-4,0)}. Para cada par, o primeiro elemento representa o coeficiente de X, e o segundo elemento representa o valor da potência de X.

Uma lista ordenada crescente pode representar, de forma adequada, um polinômio, se consideramos que não existem repetições da mesma potência X na lista. Essa consideração faz com que seja possível usar a própria potência de X como a chave de ordenação da lista.

Ao usar a potência de X para ordenar a lista, o exemplo apresentado anteriormente seria escrito da seguinte forma: P:{(-4,0), (3,2),(-11,3),(4,5)}. Observe, na seqüência, a ordenação crescente dos pares presentes em P, levando em consideração o valor da potência de X.

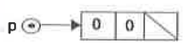
Definição dos nós do polinômio

As considerações anteriores nos permite definir a seguinte representação dos nós utilizados para armazenar os termos de um polinômio:

```
struct polin {
      float coeficiente;
      int expoente;
      struct polin *prox;
};
typedef struct polin *NODOPTR;
```

Criando um polinômio nulo

Assumiremos que todo polinômio terá pelo menos o termo $0.X^0$. Esse pressuposto facilitará a inserção de novos termos. Observe:




```
create (NODOPTR *plist)
{
      NODOPTR p;
      p = getnodo( );
      p→coeficiente=0;
      p→expoente=0;
      p→prox=NULL;
      *plist=p;
}
```

A seguir, encontramos a definição da função *getnodo ( )* para ser usada na obtenção de nós novos que representam os termos de um polinômio. Observe:

```
NODOPTR getnodo()
{
      NODOPTR p;
      p=(NODOPTR)malloc (sizeof(struct polin));
      return (p);
}
```

Inserção de um termo no polinômio

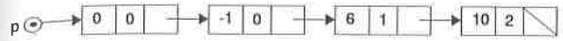
A rotina *insTermo (NODOPTR p, float c, int e)* será responsável pela inserção de um novo termo com um coeficiente e uma potência quaisquer. Entretanto, se durante a inserção for observada a preexistência de um termo com a mesma potência, haverá a soma dos coeficientes. Isso garante que em um polinômio não exista dois termos com o mesmo valor de potência, facilitando a tarefa de desenvolvimento da representação de um polinômio usando a lista ordenada.

A consideração de que todo polinômio tem pelo menos o termo de expoente zero garante uma facilidade na criação da rotina *insTermo( )* porque, ao incluir um novo elemento na lista, não é mais necessário pensarmos no tratamento que seria preciso para incluir um elemento no início da lista, pois lá estará pelo menos o termo de expoente zero.

```
void instermo (NODOPTR p, float c, int e)
{
      NODOPTR n;
      if (e == p→expoente)
            p→coeficiente = p→coeficiente + c;
      else {
            while (( p→prox != NULL) && (e > p→prox→expoente) )
                  {
                        p = p→prox;
                  }
            if ((p→prox != NULL ) && (e == p→prox→expoente))
            p→prox→coeficiente = p→prox→coeficiente + c;
            else {
                  n = getnodo();
                  n→coeficiente = c;
                  n→expoente = e;
                  n→prox = p→prox;
                  p→prox = n;
            }
      }
}
```


Simulação de rotinas de implementação de polinômios

É interessante, nesse momento, apresentar um trecho de código que pode criar um polinômio. Para melhorar o entendimento e certificar-se de que as rotinas *create( )* e *insTermo( )* fazem o que foi proposto, recomendamos que você faça uma simulação das duas rotinas usando o seguinte exemplo: $P(X) = 10X^2 + 6X - 1$.




```
. . .
NODOPTR p;
p=NULL;
clrscr( );
create (&p);                  /* cria polinômio nulo */
instermo ( p, 10, 2);         /* insere o termo 10X² */
instermo ( p, 6, 1);          /* insere o termo 6X¹  */
instermo ( P, -1, 0);         /* insere o termo -1X⁰ */
instermo ( P, -2, 2);         /* insere o termo -2X² */
mostra(p,'p');
printf("\nP ( 2 ) = %.2f", cal(p,2));
printf("\nDigite <enter> para finalizar . . .");
getch( );
. . .
```

Considerando $P(X) = 10X^2 + 6X - 1$, caso seja realizada a inserção do termo $-2X^2$, obtemos $P(X) = 8X^2 + 6X - 1$. A explicação para esse fato é que assumiu-se que não pode existir mais de um termo com a mesma potência de X. Então, a rotina de inserção soma automaticamente os termos de mesma potência; daí, $10X^2 - 2X^2 = 8X^2$ ficando $P(X) = -8X^2 + 6X - 1$.

Exibir o conteúdo do polinômio

A criação da rotina de exibição do conteúdo do polinômio pode ser facilitada, se considerarmos que os termos serão mostrados em ordem crescente e a potenciação pelo sinal '^' ou '↑'. Dessa forma, a^b ou a↑b significará a^b.

```
void mostra (NODOPTR p, char n)
{

      float coeficiente;
      clrscr ();
      printf("%c ( X ) =  ",n);
      while (p!= NULL)
       {
            if (p→coeficiente != 0 ) {
```

```
                if (p->coeficiente < 0 )
                     printf(" - ");
                else
                     printf(" + ");
                coeficiente=abs(p→coeficiente);
                printf("%.2f",coeficiente);
                if (p→expoente!=0)
                     printf("*X^%i",p→expoente);
           }
           p=p→prox;
      }
}
```

Observações sobre a função *mostra( )*

Apenas os termos com o coeficiente diferente de zero serão impressos. Se o expoente for nulo, o termo também não será impresso. Como a rotina 'mostra' tem a responsabilidade de exibir o sinal '+' ou '-',utilizou-se a função *abs ( )* para sempre exibir o valor absoluto dos números.

Supondo que $P(X) = 10X^2 + 6X - 1$, a execução da rotina 'mostra' produziria:

P (X) = -1.0 + 6.0*x^1 + 10.0*x^2

Rotina para resolver o polinômio para um valor específico

A função *calc( )* calcula o valor de um polinômio para um valor x qualquer, ou seja, realiza-se um somatório de cada termo presente no polinômio, considerando um certo valor para X. Observe:

```
float calc (NODOPTR p, float x)
{
     float s = 0;
     while (p!= NULL)
     {
          s = s + p→coef*pow(x, p→expoente);
          p = p→prox;
     }
     return (s);
}
```

A função *pow ( )* fica na biblioteca <*math.h*> da linguagem C. Caso não exista essa biblioteca, pode-se obter o mesmo efeito utilizando as funções *exp ( )* e *log ( )*.

Considerações finais sobre a implementação dos polinômios

É apresentada, a seguir, uma sugestão para usar as rotinas desenvolvidas para a escrita de um programa que cria, exibe e avalia o polinômio $P(X) = 8X^5 - 10X^3 - 2$, considerando a soma de $-22\ X^5$.

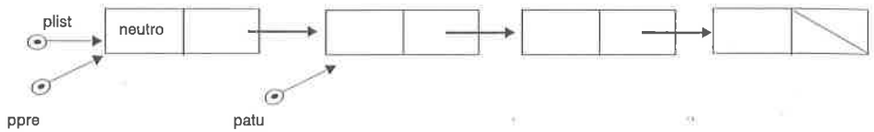
```
/* Programa : polinômio */
. . .
main ( )
{
     NODOPTR P;
     P = NULL;
     create (&p);
     instermo (P,8, 5);
     instermo (P, -10, 3);
     instermo (P, -2, 0);
     instermo (P,-22,5);
mostra( P , 'P');
printf("\nP ( 2 ) = %.2f", cal(P,2));
printf("\nDigite <enter> para finalizar . . .");
getch( );

}
```

Recomendamos que o leitor, a partir dos trechos apresentados e discutidos, faça a montagem completa do programa para o cálculo de polinômios. De fato, para que uma implementação de polinômios pudesse ser realmente completa, deveríamos considerar o desenvolvimento de rotinas como adição, subtração, multiplicação, derivada etc. Entretanto, o objetivo desse exercício foi alcançado, porque conseguimos ilustrar o uso de listas ordenadas na representação dos polinômios. Um aperfeiçoamento dessa implementação, deixamos para vocês, leitores corajosos.

EXERCÍCIOS DE APROFUNDAMENTO

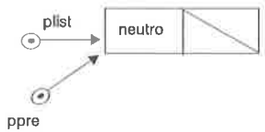
1. Suponha que temos uma lista ligada conforme é exibido a seguir. Qual problema o uso da seguinte instrução pode causar? Dado: plist = plist→prox;

2. Suponha que exista um nó neutro no início de uma lista ligada. Esse nó não tem nenhum dado útil. Ele não é o primeiro nó e trata-se de um nó vazio. Escreva um trecho de algoritmo que exclua o primeiro nó (o nó depois do nó neutro) da lista ligada.



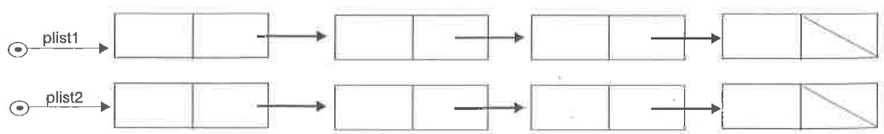
Baseando-se ainda na lista ligada anterior, escreva as instruções em pseudocódigo para excluir um nó do meio de uma lista ligada com um nó neutro. Compare esta resposta com a resposta da pergunta anterior. Elas são as mesmas? O que você pode concluir com isso? O uso de um nó neutro simplifica a operação em uma lista ligada? De que forma?

3. A figura abaixo mostra uma lista ligada vazia com um nó neutro. Escreva o código para adicionar um nó nesta lista ligada vazia.

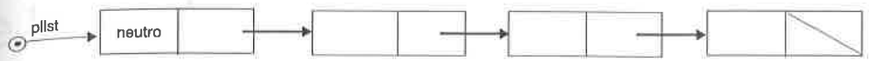


4. Suponha que exista uma lista ligada conforme é mostrado a seguir. O que acontece se executarmos a seguinte instrução para estas duas listas?

plist1 = plist2



5. É possível simplificar a maioria dos algoritmos apresentados no livro para a manipulação de listas ligadas utilizando um nó neutro no início da lista, conforme é mostrado a seguir. Escreva um algoritmo *InsereNó* usando uma lista ligada com um nó neutro. Consulte os algoritmos que foram apresentados no livro para avaliar o aparecimento de alguma facilidade na construção desse algoritmo.

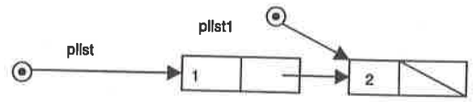


6. Analise o fragmento de programa exibido a seguir e aponte e comente os erros de compilação e os erros de execução que podem ser por ele produzidos.

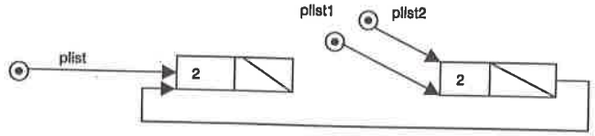
```
int a = 1;
int *p1, *p2, *p3;
p1 = (int *) malloc (sizeof(int));
*p1 = a;
*p2 = *p1;
printf("%d", *p2);
p2 = (int *) malloc (sizeof(int));
printf("%d", *p2);
*p2 = a*2;
*p3 = a;
free(p1);
printf("%d", *p1);
p1 = p2;
p2 = (int *) NULL;
printf("%d", p2);
printf("%d", p1);
```

7. A estrutura LISTA LINEAR pode ser definida como uma coleção ordenada de componentes. Existem, pelo menos, duas formas para realizar a representação de uma lista linear na linguagem C: LISTA SEQÜENCIAL ou LISTA ENCADEADA. Discorra sobre essas formas de representação, comentando as diferenças, as vantagens e as desvantagens de cada uma, principalmente no que tange às operações de inserção, remoção e acesso aos seus componentes. Para complementar sua resposta, apresente pequenos exemplos em C de ambas as estruturas de representação.

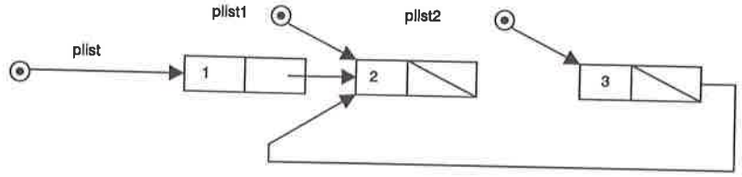
8. Escreva as instruções em C necessárias para criar as listas ligadas desenhadas na ilustração a seguir. Para cada diagrama, escreva a parte do programa que faria a respectiva exibição do campo *info* e *prox*.


plist
plist1
1
2

(a)


plist
plist1
plist2
2
2

(b)


plist
plist1
plist2
1
2
3

(c)