capítulo 8

Outros tipos de lista

> *A unidade é a variedade, e a variedade na unidade é a lei suprema do universo.*
>
> Isaac Newton

Podemos fazer pequenas variações nas listas ligadas e, com essas modificações, deixá-las mais adequadas para inúmeras aplicações.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer outros tipos de listas ligadas;
- diferenciar o uso e a adequação dos tipos de listas para a implementação de aplicações computacionais;
- implementar e manipular os diferentes tipos de lista.

Para começar

Uma sequência de itens pode ser arranjada de diversas formas.

Em nosso dia a dia, muitas vezes nos deparamos com várias listas: de compras, de coisas a fazer, de presentes, de amigos… Essas listas são, quase sempre, sequências de itens semelhantes.

Em sistemas computacionais, além de armazenar conteúdos de diversos tipos, podemos escolher diferentes tipos de lista conforme nossas necessidades e as necessidades de nossos sistemas.

Por serem de vários tipos, as listas podem servir a diversos tipos de aplicação e variar em diferentes aspectos, como:

- **tipo de informação** – as listas podem armazenar diferentes tipos de informação – nomes de amigos, itens de compras e todos os outros elementos que existem no mundo real e desejamos representar nos sistemas computacionais;
- **ordem dos elementos** – os elementos das listas podem ser armazenados segundo diferentes políticas – do mais recente para o mais antigo ou vice-versa, em ordem alfabética, em ordem crescente ou decrescente de código etc.;
- **homogeneidade da informação** – todos os elementos de uma lista podem ser do mesmo tipo (por exemplo, um registro que armazena informações sobre clientes) ou ter tipos diferentes.

Assim, além das listas que vimos no Capítulo 7 (algumas vezes, vamos chamá-las de *listas ligadas simples*), podemos ter listas duplamente ligadas, listas com cabeça fixa, listas circulares, listas multidimensionais e listas transversais, além das combinações entre esses tipos.

Vamos aproveitar nosso conhecimento sobre listas ligadas para facilitar o aprendizado sobre esses novos tipos de lista. Depois, vamos usar esse conhecimento para aprender outras estruturas de dados, como *filas* e *pilhas*.

Conhecendo a teoria para programar

Como vimos no Capítulo 7, podemos armazenar quaisquer informações sobre clientes, fornecedores, produtos, amigos etc. em uma lista ligada, sem precisar definir *a priori* o número máximo de registros dessa lista.

Para acessar os registros (ou nós) da lista, armazenamos em cada nó o endereço do seguinte. Para isso, acrescentamos ao registro de dados um campo ponteiro e, para acessar o primeiro registro, mantemos apenas uma variável do tipo ponteiro na função principal. Esse ponteiro armazena o endereço do primeiro nó da lista.

A seguir, veremos as principais variações na estrutura ou na manipulação das listas e analisaremos quando usar cada uma delas.

Listas duplamente ligadas

Em cada nó de uma lista duplamente ligada (ou *lista duplamente encadeada*), armazenamos o endereço do nó seguinte e também o endereço do nó anterior.

Isso permite:

- percorrer a lista nos dois sentidos (do começo até o final, e do final até o começo);
- inserir e remover elementos em ordem (alfabética, crescente ou decrescente) com mais facilidade, pois não precisaremos mais guardar o endereço do nó anterior enquanto percorremos a lista para fazer as ligações de ponteiros corretamente.

Vamos modificar o nosso nó, chamando o novo ponteiro de *ant*, para lembrarmos que ele armazena o endereço do nó anterior (Código 8.1).

Código 8.1

```
typedef struct tipo_no no;

struct tipo_no {

     int info;

     struct tipo_no *prox, *ant;

     };
```

Na lista duplamente ligada a seguir, o nó 3 é o primeiro nó da lista (*inicio*). Ele não possui anterior, portanto, seu ponteiro *ant* é NULL. Seu ponteiro *prox* contém o endereço do nó 8. O nó 8 possui nós antecessor e sucessor. Seu campo *ant* contém o endereço do nó 3, e o campo *prox* contém o endereço do nó 12. O nó 12 possui apenas nó anterior (o nó 8). Como não tem sucessor, seu campo *prox* é NULL.

A inicialização da lista duplamente ligada é feita da mesma forma que a de uma lista simples: inicializando o ponteiro para o primeiro elemento com o valor NULL.

Inserção de um nó em uma lista em ordem crescente

Para inserir na lista um novo nó com valor *n*, em ordem crescente, vamos primeiro alocar espaço para o novo elemento e, depois, ligá-lo à lista. Há várias formas de fazer isso, mas vamos dividir o algoritmo no caso mais simples (lista vazia) e no caso geral (lista não vazia). Caso a lista esteja vazia, o novo nó passará a ser o único elemento da lista. Senão, vamos inseri-lo como primeiro elemento. O nó que estava no início passará a ser o segundo elemento, e assim por diante, mas não será necessário deslocar os nós fisicamente. O Código 8.2 mostra essa função.

Código 8.2

```
void insere_em_ordem (int n, no **inicio)
{
no* aux = (no*) malloc (sizeof(no));    // aloca espaço para o novo nó
if (aux)                    // conseguiu alocar espaço
  {  aux->info = n;
      if (!(*inicio) || (n<(*inicio)->info))      // lista vazia  ou n é menor que o primeiro elemento
        {    (aux)->prox = (*inicio);          // preenche os ponteiros do novo nó
             (aux)->ant = NULL;
              if (*inicio)  (*inicio)->ant  = aux;   // liga o primeiro nó ao novo nó
             (*inicio) = aux;                 // o primeiro nó passa a ser aux
        }
         else             // procura o local de inserção com o ponteiro auxiliar p
              {     no *p = (*inicio);
                    while  ((p->prox!=NULL) && (n>p->info)) p=p->prox;
                    if (p->prox == NULL)      // insere no final da lista
                          {    p->prox = aux;
                               aux->prox=NULL;
                               aux->ant=p;
                          }
                          else                // insere no meio da lista, após o ponteiro p
                          {
                               aux-ant=p;
                               aux->prox=p->prox;
                               p->prox->ant =aux;
                               p->prox=aux;
                           }
               }
  }
  else printf ("Heap overflow\n");
}
```

Impressão de todos os nós da lista

Algoritmos que não alteram a lista, como os de percorrer a lista e encontrar elementos, podem permanecer os mesmos das listas simples. Veja o Código 8.3.

Código 8.3

```
void imprime (no *inicio)
{
      while (inicio)      // enquanto houver lista
      {
        printf("%d ",inicio->info);      // imprime o valor do nó
        inicio=inicio->prox;             // anda para o próximo nó
      }
}
```

Listas com cabeça fixa

No início de boa parte dos algoritmos de manipulação de listas ligadas, verificamos se a lista está vazia. Se estiver vazia, realizamos um procedimento para cuidar dessa exceção (dando uma mensagem para o usuário ou retornando à função principal, por exemplo). Depois, tratamos o caso geral, que é o da lista não vazia.

Para deixar o código mais simples e compacto, podemos usar listas com cabeça fixa. A única diferença delas é que o primeiro nó é fixo (e denominado *cabeça da lista*) e não armazena nenhum elemento.

Para inicializarmos uma lista com cabeça fixa, alocamos um nó e preenchemos seu campo *prox* com NULL. Os campos de informação não precisam ser preenchidos, pois só fazem parte da lista física (e não da lista lógica). Também podemos usá-los para armazenar informações de gerenciamento, como o número de nós da lista.

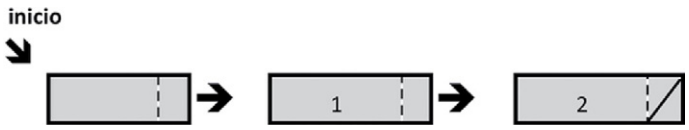
Dica

É possível misturar os vários tipos de lista conforme nossa necessidade.

Assim, podemos usar listas duplamente ligadas com cabeça fixa. Hibridizações como essa apresentam características dos tipos de lista que as compõem.

Na inicialização de uma lista duplamente ligada com cabeça fixa, além do ponteiro *prox*, o ponteiro *ant* também deve ser inicializado com *NULL*.

A seguir, podemos visualizar uma lista com cabeça fixa com dois elementos: um com valor 1 e outro com valor 2.



Inserção de um nó no início da lista

Novamente, para inserir um novo nó com valor *n* no início da lista, vamos primeiro alocar espaço para o novo elemento e, depois, ligá-lo a ela. Vamos chamar o novo nó de *aux*. Preenchemos o campo de informação e, depois, o campo *prox*. O nó que ficará após o nó *aux* é o primeiro da lista, ou seja, *inicio->prox* (lembre-se de que o nó *inicio* é apenas a cabeça da lista, não fazendo parte da lista lógica). Depois, fazemos com que o novo nó (*aux*) passe a ser o primeiro elemento da lista. Para isso, ligamos o campo *inicio->prox* a ele.

Código 8.4

```
void insere_inicio (int n, no *inicio)
{
no* aux = (no*) malloc (sizeof(no));   // aloca espaço para o novo nó
if (aux)                  // conseguiu alocar espaço
  {  aux->info = n;
     aux->prox = inicio->prox;
     inicio->prox = aux;
  }
  else printf ("Heap overflow\n");
}
```

Compare essa função, implementada no Código 8.4, com a função que insere um nó no final de uma lista ligada simples (ver Capítulo 7). Ficou mais simples, não é?

Dica

Como nenhuma função vai alterar o primeiro nó da lista, dado que esse nó é fixo, não precisamos passar o início da lista por referência. Podemos percorrê-la utilizando esse ponteiro (*inicio*). Como sabemos, quando executamos uma função, são feitas cópias de seus parâmetros, e vamos alterar apenas essa cópia.

Inserção de um nó no final da lista

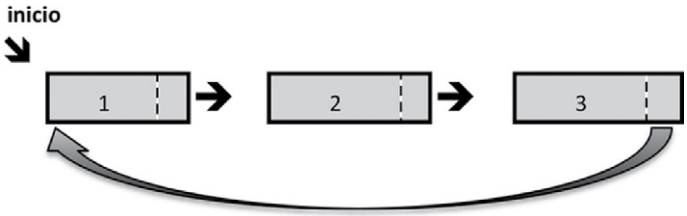
Agora, vamos ver como faríamos para inserir um novo nó no final da lista. Para isso, percorremos a lista até encontrar o último nó. Então, ligamos seu campo *prox* ao novo nó. Como em qualquer lista com cabeça fixa, não precisamos testar se ela está vazia, o que simplifica o algoritmo (Código 8.5).

Código 8.5

```
void insere_fim (int n, no *inicio)
{
no* aux = (no*) malloc (sizeof(no));   // aloca espaço para o novo nó
if (aux)                 // conseguiu alocar espaço
  {  aux->info = n;
     aux->prox = NULL; // como o novo nó será inserido no final da lista, ele não terá ninguém à frente
     while (inicio->prox != NULL)
         inicio = inicio->prox;
     inicio->prox = aux;
  }
  else printf ("Heap overflow\n");
}
```

Listas circulares

O que difere as listas circulares das outras é que nelas o último nó aponta para o primeiro, ou seja, o campo *prox* do último nó contém o endereço do primeiro nó. Observe a figura a seguir:



As listas circulares são usadas para manipular eventos cíclicos, tais como:

- se os nós armazenarem endereços de servidores, podemos fazer um programa que envie requisições de clientes para determinado servidor, depois para o ***prox***imo, depois para o ***prox***imo, e assim por diante. Dessa forma, conseguimos balancear a carga dos servidores, isto é, dividi-la, sem sobrecarregar nenhum servidor;

- se os nós representarem processos (programas sendo executados por um sistema operacional), o escalonador, que é a parte do sistema operacional que escolhe qual processo vai executar em determinado *core*, pode usar uma fila circular para executar vários processos, cada um por um pequeno intervalo de tempo. Se todos os processos forem executados rapidamente (em até 0.2 segundos, considerado o tempo médio de reação de um ser humano), o usuário terá a impressão de que os processos estão sendo executados ao mesmo tempo, em paralelo.

A manipulação de listas circulares é muito parecida com a das listas simples. A única diferença é a forma de reconhecer que a lista chegou ao fim. Você consegue imaginar como se pode fazer isso? Observe o último nó da lista circular.

Isso mesmo! O campo *prox* do último nó armazena o endereço do *inicio* em vez de armazenar *NULL*. A seguir veremos alguns exemplos.

Impressão de todos os nós da lista

Para imprimir todos os nós da lista, primeiro testamos se ela não está vazia (poderíamos não precisar testar isso, se usássemos uma lista circular com cabeça fixa). Se a lista não estiver vazia, devemos percorrê-la usando um ponteiro auxiliar, começando no início da lista e imprimindo todos os elementos até que ele chegue novamente ao início, já que a lista é circular (Código 8.6).

Código 8.6

```
void imprime (no *inicio)
{
      if (inicio)                          // se a lista não for vazia
            { no *aux = inicio;            // aux começa no inicio da lista
              do{
                  printf("%d ",aux->info);     // imprime o valor do nó
                  aux=aux->prox;               //  anda para o próximo nó
                  } while (aux != inicio);          // até aux dar a volta na lista
             }
}
```

Remoção do primeiro nó da lista

A remoção do primeiro nó de uma lista circular é um pouco mais complicada, porque precisamos encontrar o último nó e ligar seu campo *prox* ao segundo nó. Só então poderemos ligar o último nó ao segundo e fazer com que ele se torne o primeiro nó da lista. Mas, antes disso, testamos se a lista possui apenas um nó; nesse caso, temos que atribuir o valor *NULL* a ela, a fim de que as outras funções possam saber que a lista está vazia. O Código 8.7 mostra a função.

Código 8.7

```
int remove_inicio (no **inicio)
{
      if (!(*inicio)) return (-1);   // retoma a flag -1 para informar que a lista estava vazia
      no *aux=(*inicio);             // guarda o primeiro nó em aux
      int n=(*inicio)->info;         // guarda o valor do nó para retomar
      if ((*inicio)->prox == (*inicio)        // se houver apenas um nó na lista
            {    (*inicio) = NULL;            // a lista passa a ser vazia
                 free(aux);                   // libera o espaço de memória
                 return(n);                   // retoma o valor do nó removido
            }
            else {                            // lista com mais de um elemento
                 while (aux->prox !=(*inicio))
                       aux = aux->prox; // achamos o último nó para ligarmos ao primeiro
                 aux->prox = (*inicio)->prox;  // ligamos o último ao segundo nó
                 free(*inicio);
                 (*inicio)=aux->prox;   // anda com inicio para o segundo nó
                 return(n);
                 }
}
```

Listas multidimensionais

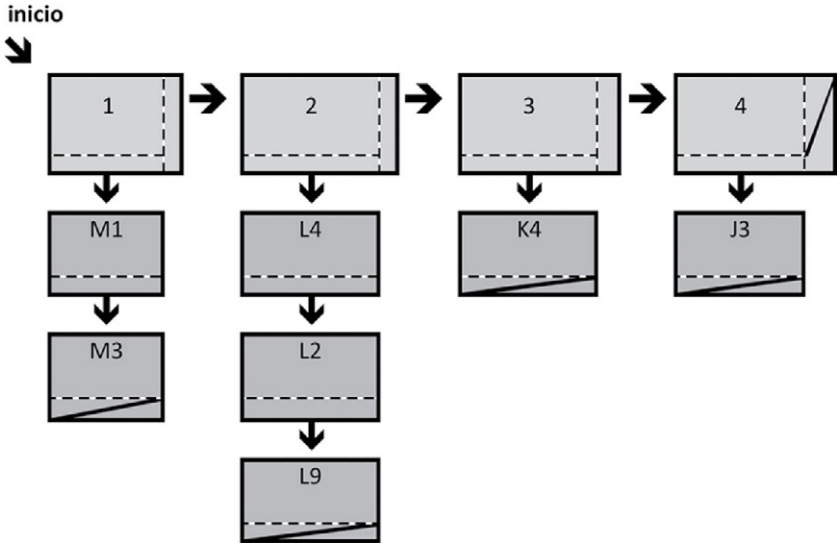
Também é possível (e, às vezes, muito útil) construir e utilizar listas multidimensionais, adequadas para organizar as informações e reduzir o tempo de busca de um registro.

Listas multidimensionais nada mais são do que listas dentro de listas. Para cada dimensão, somamos um ponteiro à nossa estrutura de nó.

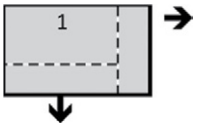
Na a seguir vemos uma lista bidimensional. A lista horizontal, composta pelos nós com valores 1, 2, 3 e 4, poderia ser uma lista de categorias, como nomes de cantores ou bandas em um sistema para gerenciar as músicas que você possui, por exemplo. Claro que o campo *info* dela poderia ser trocado para o tipo *string*, armazenando um nome.

A lista vertical é uma lista de elementos de dados (registros), como os nomes das músicas que você possui, por exemplo. Seu campo de informação está representado como uma *string*. Assim, a lista composta pelos nós M1 e M3 seriam as músicas do tipo 1 (da banda 1), os nós L4, L2 e L9 seriam as músicas do tipo 2, e assim por diante.

Para inserir, remover ou consultar uma música, procuraremos primeiro pelo identificador do cantor/banda, seguindo a lista horizontal, e depois buscaremos a música na lista vertical.



Mas vamos primeiro construir nossa lista, observando os nós da lista horizontal, onde estão as categorias (ou tipos):

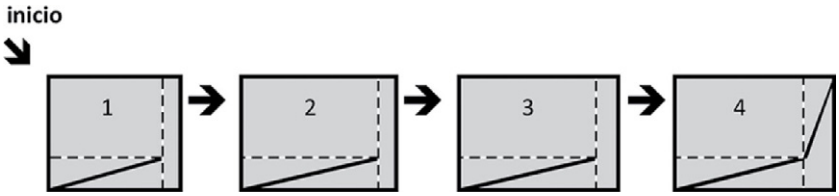


Cada nó é composto por um campo de informação e dois ponteiros. Um dos ponteiros (da lista horizontal) aponta para outro nó, que também armazena um tipo. O outro ponteiro (da lista vertical) aponta para outro tipo de nó, que armazena o dado propriamente dito (a música). Assim, esse nó, mostrado no Código 8.8, tem uma diferença em relação aos nós anteriores: possui um ponteiro para outro tipo de nó, a que chamaremos de ***tipo_no2***:

Código 8.8

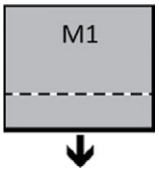
```
typedef struct tipo_no no;

struct tipo_no {

     int info;

      struct tipo_no *prox;

      struct tipo_no2 *ini;

      };
```

Para construir a lista horizontal, podemos utilizar os algoritmos anteriores, somente tomando o cuidado de inicializar o ponteiro *ini* (início da lista vertical) com *NULL*. Observe a lista de tipos:



Essa lista é muito parecida com nossas listas anteriores, não é?

A lista vertical também é muito parecida com nossas listas anteriores — ou igual a elas! —, mas, como ela armazena conteúdo diferente, vamos criar outro tipo de nó. Observe a estrutura de cada nó dessa lista:



Esse nó contém um campo de informação (agora *string*), que vamos chamar de *nome*, e um ponteiro para outro nó do mesmo tipo (***tipo_no2***) (Código 8.9).

Código 8.9

```
typedef struct tipo_no2 no2;

struct tipo_no2 {

     char nome[20];

      struct tipo_no2 *p;

      };
```

Dica

As listas de tipos funcionam como uma "lista de cabeças". Assim, cada lista de elementos (vertical) é uma lista com cabeça fixa.

Como dissemos, podemos usar os mesmos algoritmos de manipulação de listas, apenas tomando os seguintes cuidados:

- inicializar o ponteiro para a lista de elementos (*ini*) com *NULL*;
- identificar quando o ponteiro contém um endereço do nó de tipos (*no*) ou do nó de elementos (*no2*). No segundo caso, o nome do ponteiro que armazena o próximo nó foi chamado de *p* (isso foi feito para você identificar os dois tipos de nó, mas, se você quiser facilitar sua programação, pode chamar os dois de *prox*).

Vamos dar alguns exemplos.

Encontrar um elemento da lista de tipos

Para verificar se um tipo de música (por exemplo, de determinada banda) já está cadastrado, podemos usar o algoritmo de busca do Capítulo 7, sem nenhuma alteração, como visto no Código 8.10.

Código 8.10

```
no* busca (int n, no *inicio)
{
      while (inicio)
      {
        if (inicio->info == n) return (inicio);   //encontrou o elemento
        inicio=inicio->prox;
      }
      return (NULL);    // acabou a lista sem encontrar o elemento.
}
```

Inserção de um tipo no final da lista

Caso tenhamos buscado um tipo de música ainda não cadastrada, poderemos cadastrá-lo no final da lista usando o algoritmo do Capítulo 7, com uma pequena alteração.

Suponhamos que desejemos verificar se o tipo *n* está cadastrado. Se não estiver, vamos cadastrá-lo no final da lista. O comando seria este:

```
if (busca(n,inicio) == NULL) insere_fim(n,&inicio);
```

A mudança no algoritmo de inserção no final consiste em inicializar o ponteiro para a lista de elementos (*ini*) com *NULL*. A alteração (Código 8.11) está inserida em destaque no código original.

Código 8.11

```
void insere_fim (int n, no **inicio)
{
no* aux = (no*) malloc (sizeof(no));   // aloca espaço para o novo nó
if (aux)                  // conseguiu alocar espaço
  {  aux->info = n;
     aux->prox = NULL; // como o novo nó será inserido no final da lista, ele não terá ninguém à frente
     aux->ini = NULL; // a lista de elementos começa vazia até que algum elemento seja inserido
     if (!(*inicio))  //lista vazia
          (*inicio) = aux;
        else              // lista não vazia
          {
               no *p = (*inicio);      // usaremos o ponteiro p para encontrar o último nó
               while (p->prox != NULL)
                      p = p->prox;
               p->prox = aux;
            }
  }
  else printf ("Heap overflow\n");
}
```

Para a lista de elementos, também podemos utilizar os algoritmos anteriores, tomando os mesmos cuidados e lembrando-nos de que se trata de listas com cabeça fixa. Vamos ver alguns exemplos.

Inserção de um elemento no início da lista

Para inserir um novo elemento (uma nova música, por exemplo), podemos, fazer uma busca pelo tipo dele, usando a função *busca*, que retorna o endereço do registro com o valor pedido (*n*). No caso da nossa lista, esse endereço é o da cabeça da lista, que armazenamos na variável *aux*. Em seguida, inserimos a nova música (*nome*) na lista desejada.

Código 8.12

```
no *aux;
aux=busca(n,inicio;
insere_inicio(nome, aux);
```

Nessa chamada, mostrada no Código 8.12, consideramos que o tipo *n* já está cadastrado (se não estiver, podemos usar o algoritmo de inserção de um tipo no final da lista de tipos que vimos anteriormente).

A função de inserção no início da lista possui a estrutura já vista. As alterações feitas abaixo referem-se ao campo de informação (agora uma *string* chamada *nome*), o tipo do nó da lista vertical (*no2*) e os nomes dos ponteiros (o da cabeça da lista para o início da lista de elementos chama-se *ini*, e o ponteiro de próximo chama-se *p*). Vejamos no Código 8.13 como ficou.

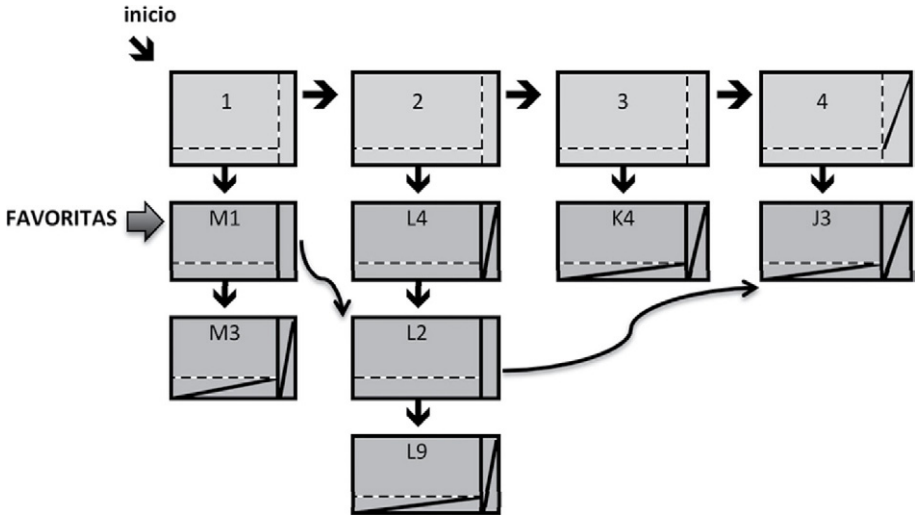
Código 8.13

```
void insere_inicio (char *nome, no2 *ini)
{
no2* aux = (no2*) malloc (sizeof(no2));   // aloca espaço para o novo nó
if (aux)                  // conseguiu alocar espaço
  {  strcpy(nome, aux->nome);
     aux->p = ini->p;
     ini->p = aux;
  }
  else printf ("Heap overflow\n");
}
```

Listas transversais

Em todos os tipos de lista, podemos inserir outras listas, chamadas de *transversais* ou *costuradas*. Assim como nas listas multidimensionais, cada lista transversal exige um ponteiro adicional na estrutura do nó.

No exemplo anterior, podemos criar uma lista transversal, por exemplo com as músicas favoritas, composta pelas músicas M1, L2 e J3.



Essa lista terá um ponteiro chamado *FAVORITAS* declarado e inicializado com *NULL*, assim como já havíamos feito com o ponteiro *inicio*. Como esse ponteiro conterá o endereço de um elemento, ele será do tipo *no2 **. A lista de músicas favoritas será ligada por meio do ponteiro *next*, que armazenará o endereço da próxima música favorita. Vejamos no Código 8.14 a alteração no registro:

Código 8.14

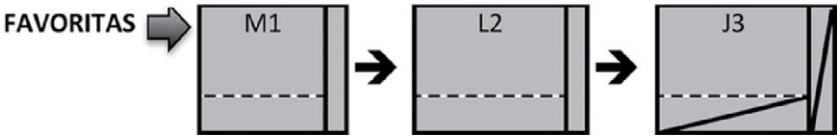
```
typedef struct tipo_no2 no2;

struct tipo_no2 {

     char nome[20];

      struct tipo_no2 *p;

      struct tipo_no2 *next;

      };
```

No Código 8.15 vemos a declaração e a inicialização do ponteiro *FAVORITAS*:

Código 8.15

```
no2 * FAVORITAS = NULL;
```

Como a lista de músicas favoritas nada mais é do que uma lista simples, podemos continuar usando nossos algoritmos para inserir nós, consultar e imprimir todas as músicas favoritas, por exemplo. Observe:



Vamos programar

Como vimos, a programação em listas multidimensionais e transversais é muito similar à programação nas listas lineares. Assim, mostraremos os algoritmos para listas lineares (Códigos 8.1 a 8.7).

Java

Código 8.1

```
public class No {

  public int codigo;
  public No prox;
  public No ant;

}
```

Código 8.2

```
public static void insere_em_ordem(int n, LinkedList<Integer> list){
    int i=0;
    if(list.isEmpty()){ // Se a lista é vazia, insere na primeira posição
      list.add(n);
    } else {

      while(i < list.size()){ // Percorre até o tamanho máximo da lista
        if (i >= (list.size())){
          list.addLast(n); // Insere na ultima posição
          break;
        }
        if (list.get(i).intValue() > n){
          list.add(i, n); // Insere na posicao correta
          break;
        }
        i++;
      }
    }
  }
```

Código 8.3

```
System.out.println("LinkedList: " + list); // Imprime a lista
```

Código 8.4

```
List.addFirst(Elemento); // Insere no Inicio
```

Código 8.5

```
List.addLast(Elemento); // Insere na Ultima posição
```

Código 8.6

```
List.removeFirst(); // Remove o primeiro elemento
```

Código 8.7

```
class No :
  def __init__(self, info=None, prox=None, ant=None):
    self.info = info
    self.prox = prox
    self.ant = ant
  #toString()
  def __str__(self):
    return str('Info:%s\n' %(self.info))
```

Python

Código 8.1

```
class No :
  def __init__(self, info=None, prox=None, ant=None):
    self.info = info
    self.prox = prox
    self.ant = ant
  #toString()
  def __str__(self):
    return str('Info:%s\n' %(self.info))
```

Código 8.2

```
class ListaDupla:
  def __init__(self):
    self.inicio = None

  def insere_em_ordem (self,n):
    aux = No()  #aloca espaCo para o novo nO
    aux.info = n
    aux.prox=None
    if self.inicio==None or n<self.inicio.info: #lista vazia ou n e menor que o primeiro elemento
      aux.prox=self.inicio  #preenche os ponteiros do novo no
      aux.ant=None
      self.inicio=aux  #o primeiro no passa a ser aux
    else:  #procura o local de insercao com o ponteiro auxiliar p
      p=self.inicio
      while p.prox<>None and n>p.prox.info:
        p=p.prox
      if p.prox == None:  #insere no final da lista
        aux.prox=None
        aux.ant=p
        p.prox=aux
      else:  #insere no meio da lista, apos o ponteiro p
        aux.ant=p
        aux.prox=p.prox
        p.prox.ant=aux
        p.prox=aux
```

Código 8.3

```
  def imprime(self):
   print "Elementos da lista: "
   aux = self.inicio
   while(aux):  #enquanto houver lista
     print aux.info  #imprime o valor do no
     aux=aux.prox    #anda para o proximo no
```

Código 8.4

```
  def insere_inicio(self, num):
   aux = No()  #aloca espaco para o novo no
   aux.info = num
   if self.inicio == None:
     self.inicio = aux
   else:
     aux.prox=self.inicio.prox
     self.inicio.prox = aux
```

Código 8.5

```
def insere_fim(self,n):
  aux=No()  #aloca espaco para o novo no
  aux.info=n
  aux.prox=None  #como o novo no sera inserido no final da lista, ele não tera ninguem a frente
  if self.inicio == None:
    self.inicio = aux
  else:
    p = self.inicio
    while(p.prox):
      p=p.prox
    p.prox=aux
```

Código 8.6

```
class ListaCircular:
  def __init__(self):
    self.inicio = None

  def imprime(self):
    print "Elementos da lista: "
    if self.inicio<>None:  #se a lista nao for vazia
      aux=self.inicio  #aux comeca no inicio da lista
      print aux.info  #imprime o valor do no
        aux=aux.prox  #anda para o proximo no
      while(aux<>self.inicio):  #ate aux dar a volta na lista
        print aux.info  #imprime o valor do no
        aux=aux.prox  #anda para o proximo no
```

Código 8.7

```
def remove_inicio(self):
  if self.inicio==None:  #retorna a flag -1 para informar que a lista estava vazia
    return -1
  aux=self.inicio  #guarda o primeiro no em aux
  n=self.inicio.info  #guarda o valor do no para retomar
  if self.inicio.prox==self.inicio:  #se houver apenas um no na lista
    self.inicio=None  #a lista passa a ser vazia
  else:  #lista com mais de um elemento
    while aux.prox <> self.inicio:  #achamos o último nó para ligarmos ao primeiro
      aux=aux.prox
    aux.prox=self.inicio.prox #ligamos o ultimo ao segundo no
    self.inicio=aux.prox  #anda com inicio para o segundo no
  return n  #retorna o valor do no removido
```

Para fixar

Implemente as funções a seguir para listas duplamente ligadas com elementos ordenados em ordem crescente:

1. função que remove o elemento de valor *n* da lista;
2. função que imprime todos os elementos da lista em ordem decrescente;
3. função que recebe um parâmetro inteiro *n* e retorna a quantidade de elementos com valores iguais ou maiores que *n*.

Para saber mais

Assim como para as listas ligadas simples, você pode ter mais exemplos de outros tipos de listas no Capítulo 4 do livro *Algoritmos e estruturas de dados* (Wirth, 1999).

Navegar é preciso

Vários algoritmos para a manipulação de registros em listas duplamente ligadas e ordenadas podem ser encontrados em:

http://www.mat.uc.pt/~amca/ED0506/FinalListas.pdf. Por meio do link http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html, você poderá obter outros exemplos de algoritmos para manipulação de listas simples e com cabeça fixa, por exemplo.

Exercícios

Complete o exemplo de uso das listas transversais e implemente:

1. uma função que busque um elemento (uma música) e a insira na lista de músicas favoritas;
2. uma função que imprima quais músicas favoritas são de determinado tipo (cantor ou banda).

Referência bibliográfica

WIRTH, N. *Algoritmos e estruturas de dados*. São Paulo: LTC, 1999.

O QUE VEM DEPOIS

As listas ligadas são estruturas úteis e muito usadas. Existem duas outras estruturas de dados que poderíamos considerar casos particulares das listas ligadas, mas, por terem políticas próprias de inserção e remoção de nós, além de muitas aplicações, são consideradas novas estruturas de dados – *filas* e *pilhas*.