Listas

Conteúdo:

1. Introdução

Na representação de um conjunto de dados pode-se usar vetores (simples ou contendo estruturas). Ao usar vetores é necessário escolher o número máximo de elementos desse conjunto e por conseqüência o espaço da memória fica alocado durante todo o processamento. Só que o uso do espaço pode ser menor do que foi declarado, bem como pode haver necessidade de trabalhar com mais elementos desse conjunto (no momento da execução do programa).

Uma solução é usar uma estrutura que possa ser dimensionada durante o processamento, ou seja à medida que é necessário armazenar novos elementos, a memória é requerida e usada. Esse tipo de estrutura é chamado de dinâmica e usam-se ponteiros numa implementação computacional.

Nesta apostila e na próxima será visto que há várias formas de representação dessas estruturas. De uma forma geral, essas estruturas são chamadas de listas encadeadas. A escolha por uma representação dependerá da aplicação. Algumas delas são:

- Lista Linear Simplesmente Encadeada
- Lista Circular Simplesmente Encadeada
- Lista Linear Simplesmente Encadeada com Descritor
- Lista Linear Duplamente Encadeada
- Lista Circular Duplamente Encadeada
- Lista Linear Duplamente Encadeada com Descritor

Em quaisquer representações deve-se ter em mente que a cada novo elemento do conjunto a ser armazenado deverá ser reservado um espaço na memória para a sua ocupação. Como a cada momento pode-se fazer essa alocação de memória não se pode garantir que os elementos armazenados ocupem um espaço de memória contíguo, como acontecia com os vetores. Dessa forma, tem de sempre referenciar a lista e seu próximo elemento. Caso contrário perde-se o acesso à lista.

2. Lista Linear Simplesmente Encadeada

Esta lista é a mais simples e mais geral que se pode usar. Uma representação típica é a mostrada na Figura1 abaixo.

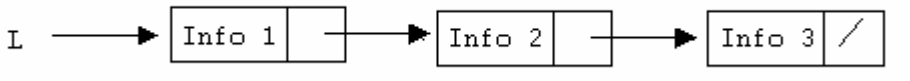

Figura1. Um exemplo de uma lista linear simplesmente encadeada.

Observa-se na Figura1 a existência de um ponteiro (*L*) para o primeiro nó da lista que contém um valor (*Info 1*) e um ponteiro para o próximo nó. O último nó aponta para NULL, ou seja, não aponta para nenhum lugar.

2.1. Definição da estrutura

```
struct lista{
            int info;
            struct lista* prox;};
typedef struct lista* def_lista;
```

2.2. Inicialização da lista

```
def_lista inicializa (void)
{
      return NULL;
}
```

A rotina inicializa( ) devolve uma lista do tipo *def_lista* (que é um ponteiro para a estrutura da lista) com o valor NULL. Inicialmente ela não aponta para nenhum lugar, já que ainda não se têm elementos armazenados.

2.3. Inserção de elementos no início da lista

Cada elemento a ser colocado na lista deve ser armazenado primeiramente em uma estrutura que representa um nó da lista. Esse nó possui dois campos: um para guardar o elemento e outro para guardar um ponteiro, que inicialmente deverá apontar para NULL. Então:

```
def_lista no = (def_lista) malloc(sizeof(struct lista));
no->info = nro;
no->prox = NULL;
```

Pode-se definir uma rotina para ficar encarregada desta tarefa:

```
def_lista cria_no (int nro){
      def_lista no = (def_lista) malloc(sizeof(struct lista));
      no->info = nro;
      no->prox = NULL;
      return no;
}
```

Criado o nó com o elemento, deve-se pensar na inserção. A inserção pode gerar dois casos: (1) nó é o primeiro elemento a ser inserido, (2) nó é mais um elemento a ser inserido.

No caso de ser o primeiro elemento da lista (*L*), basta fazer *L* apontar para o *nó* criado.

```
Lista = no;
```

Veja Figura 2.

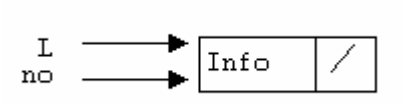

Figura2. Inserção do primeiro elemento na lista

Quando for necessário fazer várias inserções no início de uma lista deve-se criar o *nó* com o novo elemento, fazer o campo *prox* do *nó* recém criado apontar para onde *L* está apontado, ou seja para o primeiro elemento da lista, e depois fazer o ponteiro da lista (*L*) apontar para o *nó* recém criado. Veja a Figura 3, na qual é inserido o elemento *Info4* numa lista que já possui outros elementos.

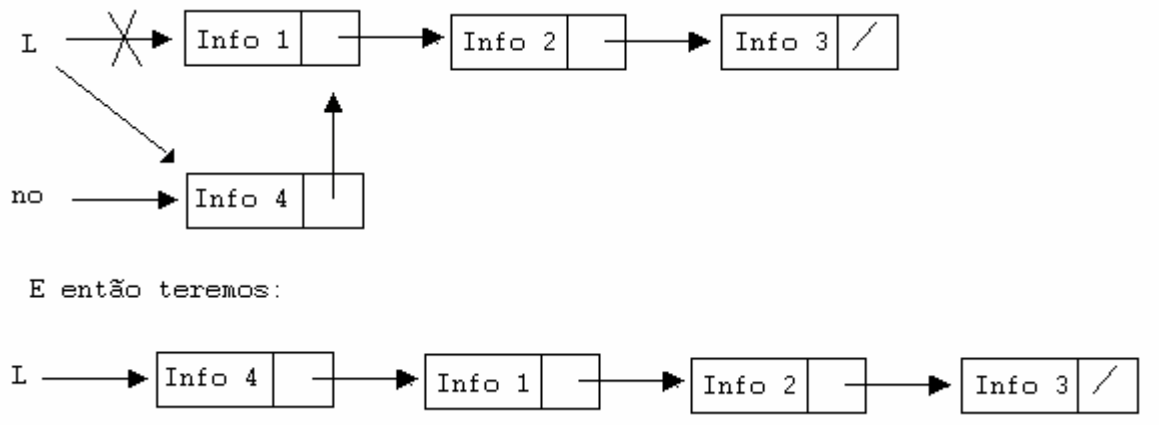

Figura3. Inserção de elementos numa lista

OBS: A ordem das operações é importante para não perder a lista. Se apontar primeiro *L* para *nó*, perde-se toda a lista já armazenada, já que o campo *prox* de *no* não saberá para onde apontar.

A implementação em C:

```
void insere_inicio (def_lista* Lista, int nro)
{     def_lista no = cria_no(nro);

      if (*Lista != NULL) no->prox = *Lista;
      *Lista = no;
}
```

2.4. Inserção de elementos no final da lista

Ao inserir um elemento no final da lista e ele é primeiro, a inserção se dá como já visto. Quando existem mais elementos na lista, tem de percorrer a lista até o final e aí inserir o elemento. Veja como fazer isto olhando a rotina abaixo:

```
void insere_final (def_lista* Lista, int nro)
{  def_lista no, aux;

   no = cria_no (nro);

   if (*Lista == NULL) *Lista = no;
   else{
            aux = *Lista;
            while (aux->prox != NULL) aux = aux->prox;
            aux->prox = no;}
}
```

2.5. Percorrendo a lista

Em situações de busca por um valor, ou para impressão de uma lista, deve-se percorrer a lista a partir do primeiro elemento até onde for desejado (até achar o elemento ou chegar no final). Para isso, veja as duas rotinas implementadas abaixo:

```
Boolean busca (def_lista Lista, int nro)
{     def_lista aux;

      for(aux = Lista; aux != NULL; aux = aux->prox){
            if (aux->info == nro) return true;}
      return false;
}


void imprime_lista (def_lista Lista)
{     def_lista aux;

      printf("\nA lista : \n");
      for(aux = Lista; aux != NULL; aux = aux->prox)
            printf("%d\t", aux->info);
      printf("\n");
}
```

Percebam alguns detalhes:

- o tipo Boolean não existe em C. Esse tipo de dado pode ser criado usando a estrutura enumeração que permite a variável só receber os valores de um conjunto definido. Os valores dos elementos desse conjunto começam em 0. Assim pode-se definir:

```
typedef enum bool {false, true} Boolean;
```

onde false valerá 0 (zero) e true valerá 1 (um). Se quiser isso mais claramente pode-se escrever:

```
typedef enum bool {false=0, true=1} Boolean;
ou
typedef enum bool {true=1, false=0} Boolean;
```

- a rotina de busca percorre toda a lista porque a lista não é ordenada, se fosse poderia parar a busca assim que chegar a uma posição que deveria existir o número procurado. Veja abaixo:

```
Boolean busca_lista_ordenada(def_lista Lista, int nro)
{  def_lista aux = Lista;

   while (aux != NULL && aux->info < nro)
          aux = aux->prox;
   if (aux->info == nro) return true;
   return false;
}
```

2.6. Remover um elemento da lista

Para remover um elemento (X) da lista, primeiro é necessário localizá-lo e sinalizá-lo. Para localizar um elemento basta fazer uma busca na lista, como feito acima, e deixar esse elemento (X) sinalizado, bem como o elemento que está antes dele. Isso deve ser feito para que o nó que está antes de X aponte para o elemento que está depois de X. Veja figura 4.

Depois do elemento estar localizado e sinalizado, faz-se o ajuste dos ponteiros e libera a memória ocupada pelo nó que possui o elemento removido.

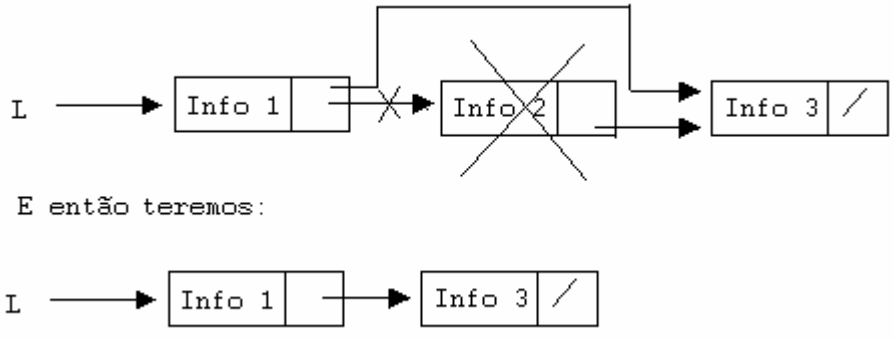

Figura4. Removendo um elemento da lista

A implementação é:

```
Boolean remove_lista (def_lista* Lista, int nro)
{  def_lista aux = *Lista;
   def_lista ant = NULL;

   /* Tenta localizar e sinalizar o elemento */
   while (aux != NULL && aux->info != nro){
            ant = aux;
            aux = aux->prox;}

   if (aux == NULL) return false; /* Não encontrou ou está vazia */
   else {
         if (ant==NULL) *Lista = aux->prox; /* primeiro elemento */
         else  ant->prox = aux->prox;
         free(aux);
         return true;}
}
```

2.7. Destruir uma lista

A cada inserção de elementos aloca-se memória. Ao terminar de usar a lista deve-se também liberar a memória utilizada. Veja com fazer isso:

```
void libera (def_lista Lista)
{     def_lista t, aux = Lista;
      while (aux != NULL) {
            t = aux->prox;
            free(aux);
            aux = t;}
}
```

2.8. Trabalhando com uma lista de forma recursiva

Aqui serão mostradas as rotinas de inserção no final, inserção ordenada, remoção, busca, impressão e destruição de uma lista de forma recursiva. As rotinas de inserção no início da lista, verificação de lista vazia, inicialização de uma lista, criação de um nó não precisam ser recursivas, já que são muito simples.

```
void insere_final_recurs (def_lista* Lista, int nro)
{
   if ((*Lista) == NULL) (*Lista) = cria_no(nro);
   else insere_final_recurs(&((*Lista)->prox),nro);
}


def_lista remove_lista_recurs (def_lista Lista, int nro)
{  def_lista aux;

   if (Lista==NULL) return Lista;

   if (Lista ->info == nro){
       aux = Lista;
       Lista = Lista ->prox;
       free(aux);}
   else Lista ->prox = remove_lista_recurs(Lista ->prox,nro);
   return Lista;
}
```

```
Boolean busca_recurs (def_lista *Lista, int nro)
{
   if (*Lista == NULL) return false;
   if ((*Lista)->info == nro) return true;
   if (*Lista != NULL) return (busca_recurs(&((*Lista)->prox),nro));
 }

void libera_recurs (def_lista Lista)
{
   if (Lista!=NULL){
      libera_recurs(Lista ->prox);
      free(Lista);}
}

void imprime_lista_recurs (def_lista* Lista)
{
   if (*Lista ==NULL){printf("\n\n"); return;}
   printf("%d\t",(*Lista)->info);
   imprime_lista_recurs(&((*Lista)->prox));
}
```

3. Lista Circular Simplesmente Encadeada

Na lista circular, como o próprio nome sugere, acontece a formação de um círculo. É como se ligasse o final da lista com o início da mesma. Veja a Figura5 abaixo.

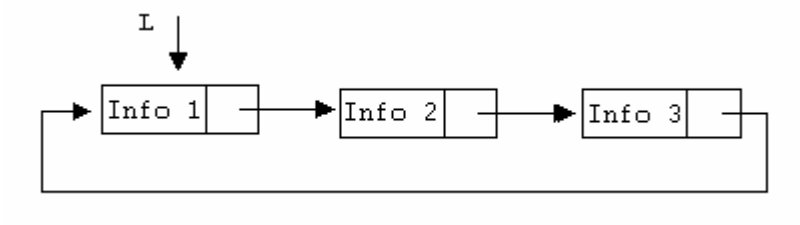

Figura5. Um exemplo de uma lista circular simplesmente encadeada

Com essa configuração deve-se definir para onde o ponteiro da lista está apontando: se para o início da lista (ou seja, para o primeiro elemento da lista) ou para o final da lista (ou seja, para o último elemento da lista). Precisa-se disto para saber onde inserir e remover os elementos.

1º.Caso: Se *L* aponta para o início da lista e a inserção do elemento é no início da lista (Figura 6).
O processo exigirá que se ache o último elemento para que ele aponte para o elemento novo e a lista permaneça circular.

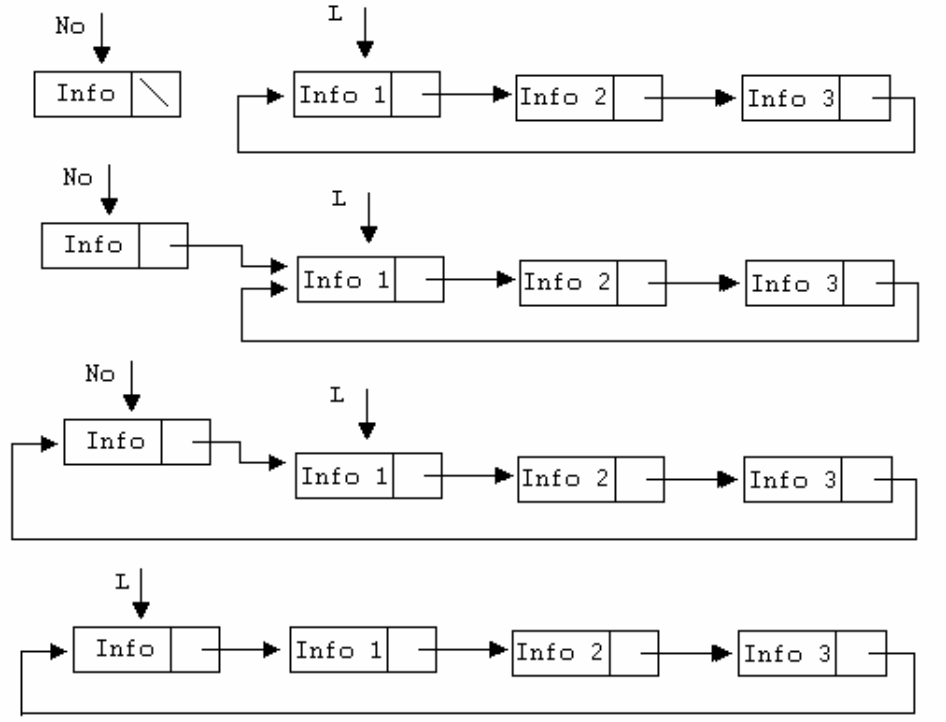

Figura 6. Inserindo um elemento no início da lista com L apontando para o início

Assim, o problema aqui está na necessidade de percorrer toda a lista para descobrir o ponteiro do campo *prox* do último elemento para que ele aponte para o nó recém criado.

```
Algoritmo:
      Criar o nó
      Percorrer a lista e parar no último elemento com o ponteiro aux;
      Fazer nó apontar para aux->prox
      Fazer aux->prox apontar para nó
      Fazer L apontar para nó
```

2º.Caso: Se *L* aponta para o início da lista e a inserção do elemento é no final da lista (Figura 7).

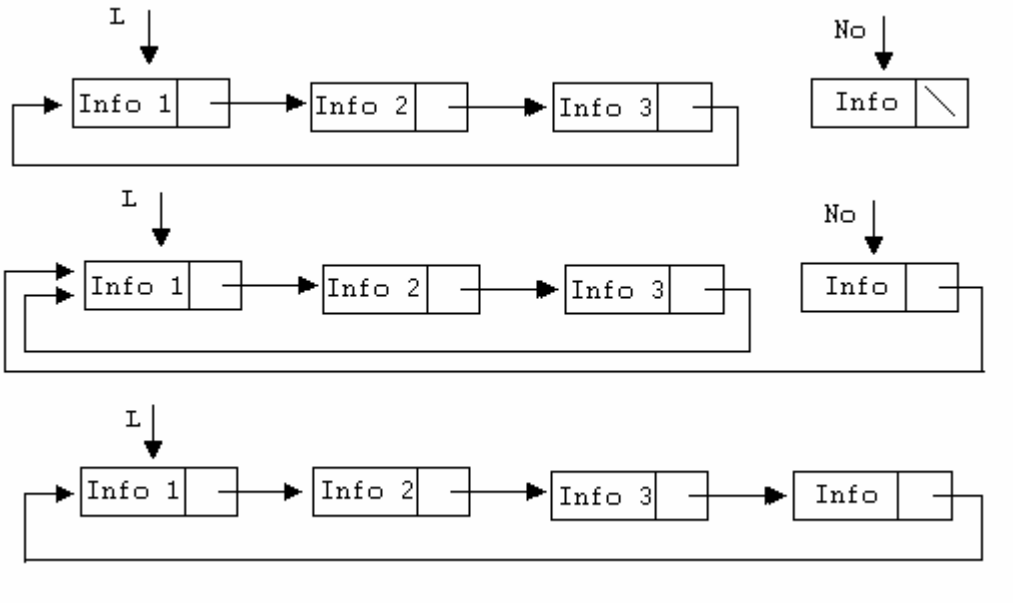

Figura 7. Inserindo um elemento no final da lista com L apontando para o início

Ao inserir no final da lista, tem de percorrer a lista, mas isso já era necessário numa lista linear simplesmente encadeada.

```
Algoritmo:
      Criar o nó
      Fazer nó apontar para L
      Percorrer a lista e parar no último elemento com o ponteiro aux;
      Fazer aux->prox apontar para nó
```

3º.Caso: Se *L* aponta para o final da lista e a inserção do elemento é no início da lista (Figura 8).

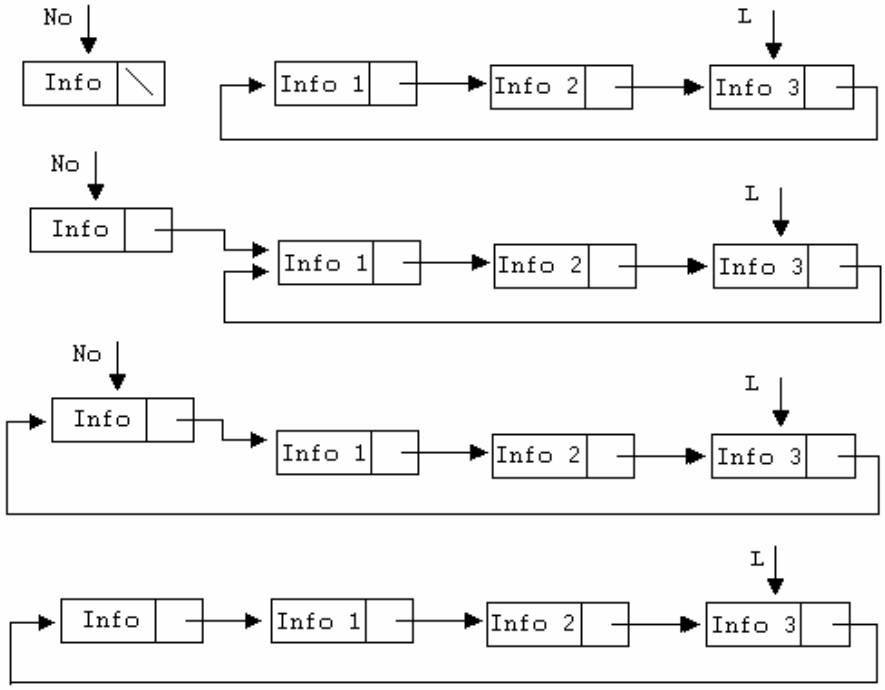

Figura 8. Inserindo um elemento no início da lista com L apontando para o fim

A inserção de um elemento no início de uma lista onde *L* aponta para o fim é simples, bem como se a inserção for no final da lista. O que muda é se *L* vai ser atualizado ou não.

```
Algoritmo:
      Criar o nó
      Fazer nó apontar para L->prox
      Fazer L->prox apontar para nó
```

4º.Caso: Se *L* aponta para o final da lista e a inserção é no final (Figura 9)

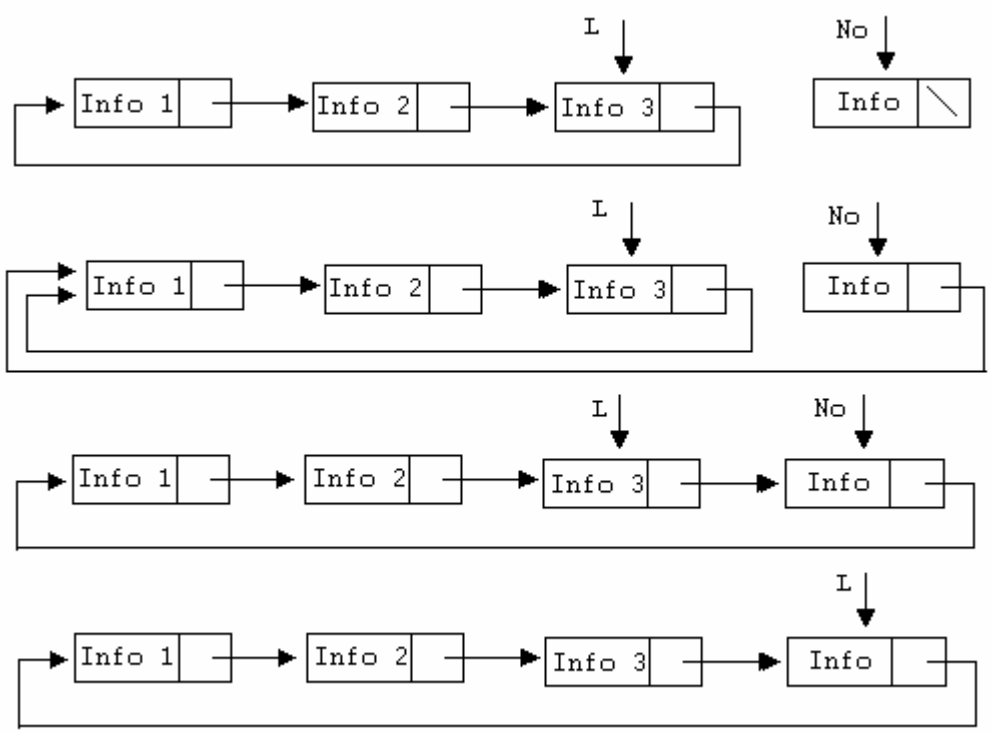

Figura 9. Inserindo um elemento no final da lista com L apontando para o fim

```
Algoritmo:
      Criar o nó
      Fazer nó apontar para L->prox
      Fazer L->prox apontar para nó
      Fazer L apontar para nó
```

Dessa forma, pode-se constatar que uma forma eficiente de trabalhar com uma lista circular é deixar o ponteiro da lista apontando sempre para o último elemento da lista. Porque aí tem:

Ponteiro para o último elemento = L Ponteiro para o primeiro elemento = L -> prox

Veja, a seguir, as rotinas para inserção (no início, no final e ordenada), remoção (do início, do final,de um número numa lista qualquer e numa lista ordenada), busca (numa lista qualquer e numa lista ordenada), impressão e liberação para esse tipo de lista.

(1) Inserção no início da lista

```
void insere_inicio (def_lista* Lista, int numero)
{     def_lista no = cria_no(numero);

   if (*Lista == NULL){
      *Lista = no;
      no->prox = *Lista;}
   else{
      no->prox = (*Lista)->prox;
      (*Lista)->prox = no;}
}
```

(2) Inserção no final da lista

```
void insere_final(def_lista* Lista, int numero)
{     def_lista no = cria_no(numero);

   if (*Lista == NULL){
      *Lista = no;
      no->prox = *Lista;}
   else{
      no->prox = (*Lista)->prox;
      (*Lista)->prox = no;
      *Lista=no;}
}
```

(3) Remoção de um elemento do início da lista

```
Boolean remove_inicio(def_lista* Lista, int* numero)
{ def_lista aux;

    if (*Lista == NULL) return false;
    else{
         aux = (*Lista)->prox;
         *numero = aux->info;
         if (aux == *Lista) *Lista=NULL;
         else (*Lista)->prox = aux->prox;
         free(aux);
         return true;}
}
```

(4) Remoção de um elemento do final da lista

```
Boolean remove_final(def_lista* Lista, int* numero)
{ def_lista aux,post;

    if (*Lista == NULL) return false;
    else{
         aux = *Lista;
         *numero = aux->info;
         if (aux->prox == *Lista) *Lista=NULL;
         else{
           post=(*Lista)->prox;
           while(post->prox != aux) post=post->prox;
           post->prox = aux->prox;
           *Lista=post;}
```

```
         free(aux);
         return true;}
}
```

(5) Remoção de um elemento qualquer da lista

```
Boolean remove_lista(def_lista* Lista, int numero)
{     def_lista aux, ant;

   if(*Lista == NULL) return false; /* Lista vazia */
   aux = (*Lista)->prox; ant = *Lista;
   while (aux!=(*Lista) && aux->info!=numero){
            ant=aux; aux=aux->prox;}
   if (aux==(*Lista)->prox) return false;
   else {
      ant->prox = aux->prox;
      if (aux==*Lista) *Lista = ant;
      free(aux);
      return true;}
}
```

(6) Busca numa lista

```
Boolean busca_lista (def_lista Lista, int nro)
{     def_lista aux = Lista;

      do{   if (aux->info==nro) return 1;
            aux = aux->prox;
   }while (aux!=Lista);
   return 0;
}
```

(7) Impressão de uma lista

```
void imprime_lista(def_lista Lista)
{ def_lista aux;

    aux = Lista->prox;
    do{
                 printf("%d\t", aux->info);
        aux = aux->prox;
    }while (aux != Lista->prox);
    printf("\n\n");
}
```

(8) Liberando uma lista

```
void libera (def_lista Lista)
{     def_lista aux = Lista->prox;
      def_lista t;

      while (aux != Lista) {
      t = aux->prox;
      free(aux);
      aux=t;}
}
```

4. Lista Linear Simplesmente Encadeada com Descritor

Uma outra forma de construir uma lista é através do conceito de Lista Linear Simplesmente Encadeada com Descritor. Um descritor é uma estrutura (normalmente uma estrutura simples) que pode conter informações que serão úteis na implementação. Por exemplo: quem é o primeiro nó, quantos elementos existem na lista, quantas operações de inserção foram feitas, quantas de remoção foram feitas, etc.

Veja a construção um descritor:

```
typedef struct no {     // definição da lista propriamente dita
            int dado;
            struct no* prox;
            } No;

typedef struct descritor {    // definição do descritor
            int quant;        // guarda o nro de nós da lista
            No* L             // ponteiro para a lista
            } def_lista;

def_lista Lista;
```

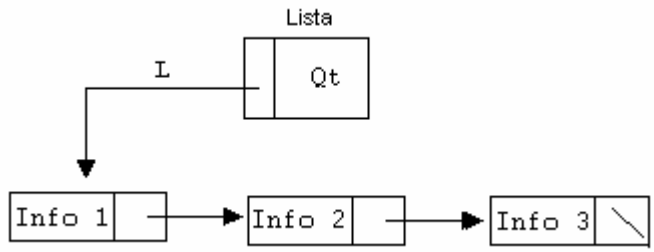

Figura10. Uma Lista Linear Simplesmente Encadeada com Descritor

Perceba que a definição da lista agora é uma estrutura que contém ponteiros para nós da lista.

Se quiser que tudo seja alocado de forma dinâmica basta criar a definição do descritor sendo um ponteiro:

```
typedef struct descritor{     int quant;
                              No* L;} *def_lista;

def_lista Lista; // esse descritor é um ponteiro para um registro
```

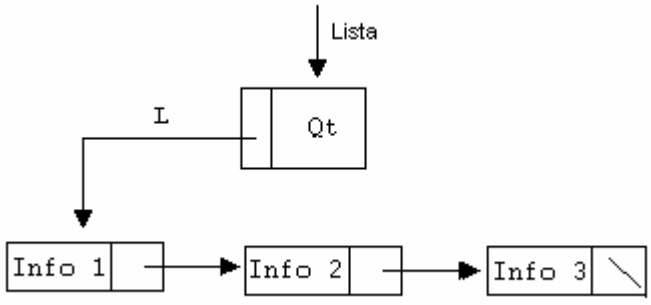

Figura10. Uma Lista Linear Simplesmente Encadeada com Descritor (com ponteiro)

Perceba que a definição da lista agora é um ponteiro para uma estrutura que contém ponteiros para nós da lista. Assim antes de usar o descritor deverá ser alocado um espaço para ele.

Usando o descritor como uma estrutura simples

(1) Criando um nó da lista

```
No* cria_no(int nro){
      No* aux = (No*)malloc(sizeof(struct no));
     aux->info=nro;
     aux->prox=NULL;
     return aux;
}
```

(2) Inserindo um elemento no início da lista

```
void insere_inicio(def_lista* Lista, int numero){
      No* aux = cria_no(numero);
      if (!vazia(*Lista)) aux->prox = Lista->L;
      Lista->L = aux;
      (Lista->quant)++;
}
```

(3) Inserindo um elemento no final da lista

```
void insere_final(def_lista* Lista, int numero){
     No* aux = cria_no(numero);
     if (vazia(*Lista)) Lista->L = aux;
     else{
        No* q = Lista->L;
        for(; q->prox != NULL; q=q->prox);
        q->prox = aux;}
     (Lista->quant)++;
}
```

(4) Removendo um elemento do início da lista

```
Boolean remove_inicio(def_lista* Lista, int* numero){
    No* q;

    if (vazia(*Lista)) return false;
    else{
       q = (*Lista).L;
       *numero = q->info;
       if ((*Lista).L->prox == NULL) (*Lista).L = NULL;
       else (*Lista).L = q->prox;
       free(q);
       (Lista->quant)--;
       return true;}
}
```

(5) Removendo um elemento do final da lista

```
Boolean remove_final(def_lista* Lista, int* numero){
    No *q, *ant;
    if (vazia(*Lista)) return false;
    else{
            q = Lista->L;
            if (Lista->L->prox == NULL){
                Lista->L = NULL;
                Lista->quant = 0;
                *numero = q->info;}
            else{ ant=NULL;
                  for(;q->prox!=NULL; q=q->prox) ant=q;
                  *numero = q->info;
                  ant->prox = NULL;}
            free(q);
            (Lista->quant)--;
            return true;}
}
```

(6) Removendo um elemento da lista

```
Boolean remove_lista(def_lista* Lista, int numero){
    No *aux, *ant;
    if (vazia(*Lista)) return false;
    else{
       for (aux = Lista->L, ant = NULL;
            aux != NULL && aux->info != numero;
            aux=aux->prox) ant=aux;
       if (aux == NULL) return false;
       if (ant == NULL) Lista->L = aux->prox;
       else       ant->prox = aux->prox;
       free(aux);
       (Lista->quant)--;
       return true;}
}
```

(7) Buscando um elemento da lista

```
Boolean busca_lista(def_lista Lista, int numero){
    No *aux;
    if (vazia(Lista)) return false;
    else{
          for  (aux = Lista.L;
                aux != NULL && aux->info != numero;
                aux=aux->prox);
         if (aux == NULL) return false;
         return true;}
}
```

(8) Imprimindo uma lista

```
void imprime(def_lista Lista){
    No* aux;
    printf("\nA lista tem %d elementos. Sao eles:\n",Lista.qt);
    aux = Lista.L;
    while (aux != NULL){
          printf("%d\t", aux->info);
          aux=aux->prox;}
    printf("\n\n");
}
```

Exercícios

1. Escreva uma rotina que retorne o número de elementos de uma lista circular simplesmente encadeadas.

2. Escreva uma rotina que receba uma lista e um número X e retorne o número de nós da lista que possuem o número X. Considere a lista sendo do tipo linear simplesmente encadeada.

3. Escreva uma rotina que receba uma lista e um número X e retorne o número de nós da lista que possuem valores maiores do que o número X. Considere a lista sendo do tipo circular simplesmente encadeada.

4. Escreva uma rotina que receba uma lista e um número X e retorne o número de nós da lista que possuem valores menores do que o número X. Considere a lista sendo do tipo linear simplesmente encadeada com descritor.