LISTAS LINEARES

Uma das formas mais simples de interligar os elementos de um conjunto é através de uma lista. Lista é uma estrutura em que as operações inserir, retirar e localizar são definidas.

Na representação de um conjunto de dados pode-se usar *vetores*, contudo é necessário definir o número máximo de elementos desse conjunto e consequentemente o espaço da memória fica alocado durante todo o processamento. Nessa representação, o problema é o uso do espaço que pode ser menor do que foi declarado ou 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. Este tipo de estrutura é chamado de *dinâmica* e usam-se *ponteiros* numa implementação computacional.

Listas são estruturas muito flexíveis, porque podem crescer ou diminuir de tamanho durante a execução de um programa, de acordo com a demanda. Itens podem ser acessados, inseridos ou retirados de uma lista.

Na representação de uma lista de maneira a preservar a relação de ordem linear ou total utiliza-se a estrutura conhecida por lista linear.

Uma lista linear é formada por elementos denominados *nós*. Em cada nó pode-se ter dados primitivos (inteiro*, real,* caractere) ou dados compostos (registro, vetor).

Uma lista linear é formada por elementos denominados nós. Esquematicamente, uma lista linear formada por n nós pode ser representada pelos nós: x_1, x_2, ..., , x_n. Desta forma, mantendo a relação de ordem tem-se que x_1 é o primeiro elemento da lista e x_n é o último elemento da lista, disposto, por exemplo, de forma que $x_1 \leq x_2 \leq ... \leq x_n$.

Uma lista linear é dita *vazia* se não contém nó algum.

A seguir tem-se algumas operações que podem ser realizadas em lista:

- criar a lista;
- verificar se a lista contém ou não um determinado nó;
- encontrar a posição de um nó;
- ler a informação contida em um nó;
- incluir um nó na lista;
- excluir um nó da lista;
- determinar a quantidade de nós da lista;
- dividir uma lista em duas ou mais listas;
- trocar as posições dos nós;
- concatenar duas ou mais listas;
- etc.

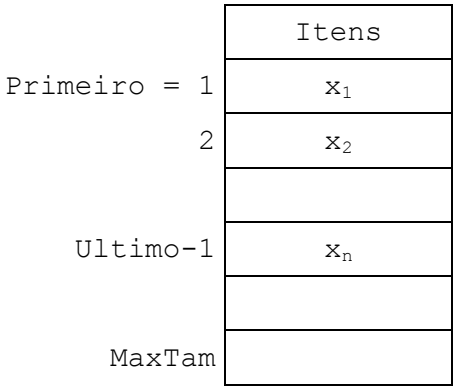
Uma lista pode ser representada por diversas formas, mas as duas mais usuais são:

- por contiguidade (alocação estática de memória);
- por encadeamento (alocação dinâmica de memória).

A escolha da representação depende das *operações* a serem realizadas sobre a lista e a *frequência* com que ocorrem.

REPRESENTAÇÃO DE LISTAS POR CONTIGUIDADE

Nesta forma de representação os nós são colocados em posições consecutivas da memória, explorando sua sequencialidade. Neste caso, a lista pode ser percorrida em qualquer direção.



O tipo de dado que representa tal lista é o vetor, onde os elementos do vetor correspondem aos nós da lista. Assim, todas as operações a serem realizadas com listas são operações sobre vetores.

A inserção de um novo item pode ser realizada após o último item com custo constante. A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção. Da mesma forma, retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.

Os itens são armazenados em um vetor de tamanho suficiente para armazenar a lista. O campo `Ultimo` contém um apontador para a posição seguinte a do último elemento da lista. A constante `MaxTam` define o tamanho máximo permitido para a lista.

Operações com listas por contiguidade

1) Criação da lista.
2) Ordenação.
3) Busca do valor do nó na k-ésima posição da lista.
4) Inclusão de um nó na k-ésima posição da lista.
5) Remoção de um nó na k-ésima posição da lista.
6) Procura por um determinado elemento na lista.
7) Busca recursiva por um determinado elemento na lista (ordenada).
8) Inclusão de um elemento no início da lista.
9) Inclusão de um elemento no fim da lista.

10) Inclusão de um elemento na lista ordenada.
11) Exclusão de um elemento da lista.
12) Exclusão de um elemento da lista ordenada.
13) Concatenação de duas listas.
14) Concatenação de duas listas gerando uma terceira lista ordenada.

Observações:

- A representação de listas por contiguidade apresenta dificuldades nas operações de inclusão e exclusão, já que tais operações envolvem o deslocamento de nós.

- Quando as inclusões/exclusões são realizadas no final da lista, o esforço computacional é menor do que as mesmas operações realizadas no início da lista. Para minimizar tal problema e aproveitar a sequencialidade das posições de um vetor, a sugestão é manter a lista centralizada no vetor, reduzindo assim o número de deslocamentos a serem realizados sem comprometer as demais posições.

Exercício

Faça um programa a seguir, utilizando o conceito de lista por contiguidade, que resolva o problema descrito.

Um pronto-socorro deseja informatizar o seu sistema de atendimento aos pacientes. As recepcionistas utilizarão um terminal para fazer a ficha de cada paciente que chega. Juntamente com os dados do paciente, a recepcionista preenche um campo informando o estado do paciente (regular, ruim, péssimo). Ao terminar de atender a um paciente, o médico consultará o sistema para chamar o próximo paciente e, naturalmente, o sistema deverá priorizar os pacientes que estiverem em pior estado de saúde.

PONTEIROS E ESTRUTURAS DINÂMICAS

Até agora todos os tipos de variáveis apresentados são tipos estáticos, ou seja, seus espaços são alocados no início do programa e possuem tamanho fixo.

A linguagem C permite alocar dinamicamente (em tempo de execução), blocos de memória usando ponteiros. Dada a intima relação entre ponteiros e vetores, isto significa que podemos declarar dinamicamente vetores de tamanho variável. Isto é desejável caso queiramos poupar memória, isto é não reservar mais memória que o necessário para o armazenamento de dados.

No padrão C ANSI existem 4 funções para alocações dinâmica pertencentes a biblioteca `stdlib.h`. São elas `malloc()`, `calloc()`, `realloc()` e `free()`. Sendo que as mais utilizadas são as funções `malloc()` e `free()`. As funções `malloc()` e `calloc()` são responsáveis por alocar memória, a `realloc()` por realocar a memória e por ultimo a função `free()` fica responsável por liberar a memória alocada.

Portanto, para a alocação de memória usamos a função `malloc()` (*memory allocation*). A função `malloc()` reserva, sequencialmente, um certo numero de blocos de memória e retorna, para um ponteiro, o endereço do primeiro bloco reservado.

Sintaxe: A sintaxe geral usada para a alocação dinâmica é a seguinte:

```
pont = (tipo *) malloc (tam);
```

em que
pont é o nome do ponteiro que recebe o endereço do espaço de memória alocado.
tipo é o tipo do endereço apontado (tipo do ponteiro).
tam é o tamanho do espaço alocado: numero de bytes.

A sintaxe seguinte, porem, é mais clara:

```
pont = (tipo*) malloc(num*sizeof(tipo));
```

em que
num é o numero de elementos que queremos poder armazenar no espaço alocado.

Exemplo: Declarar um vetor chamado `vet`, tipo `int`, com `num` elementos:

```
int *vet; // declaração do ponteiro
vet =(int*)malloc(num*2);
```

ou ainda

```
int *vet; // declaração do ponteiro
vet = (int*) malloc(num * sizeof(int));
```

Caso não seja possível alocar o espaço requisitado a função *malloc()* retorna a constante simbólica `NULL`.

Sintaxe: Para liberar (desalocar) o espaço de memória se usa a função `free()`, cuja sintaxe é a seguinte:

```
free (pont);
```

em que
`pont` é o nome do ponteiro que contem o endereço do inicio do espaço de memória reservado.

Assim, a linguagem C permite a criação de novas variáveis durante a execução de um programa que são as variáveis dinâmicas.

Ao se trabalhar com variáveis dinâmicas, pode-se:

- Alocar/desalocar espaço para as variáveis em tempo de execução;
- Definir ligações entre essas variáveis.

Exemplo:

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main () {
    int qte, i;
    int *vet;
    printf ("\nQuantidade de valores do vetor: ");
    scanf ("%d",&qte);
    if ((vet = (int *)malloc(qte*sizeof(int))) == NULL) {
        printf ("Memoria insuficiente");
        exit (1);
    }
    printf ("\nDigite os valores do vetor:");
    for (i = 0; i<qte; i++) {
        printf ("\nDigite o %d. elemento: ",i+1);
        scanf ("%d",&vet[i]);
    }
    printf ("\nImprimindo os valores do vetor:");
    for (i = 0; i<qte; i++)
        printf ("\nvet[%d] = %d",i+1,vet[i]);
    free (vet);
    getch ();
}
```

REPRESENTAÇÃO DE LISTAS POR ENCADEAMENTO

Além do esforço computacional exigido na manipulação de listas por contiguidade, outra desvantagem quando se trabalha com vetores é que o tamanho de um vetor deve ser definido no início do programa e, nem sempre o processamento utiliza todas as posições reservadas, podendo ocorrer inclusive erro por falta de memória.

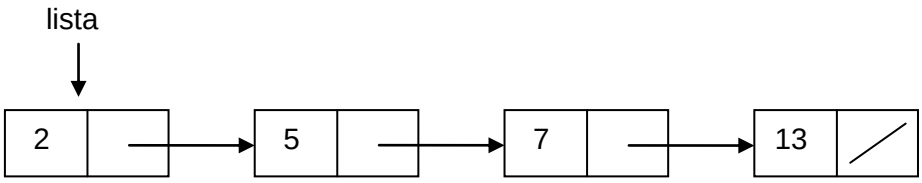
Outro aspecto é que pelo fato do vetor ter tamanho previamente definido, consequentemente o crescimento da lista está limitado pelo número máximo de elementos definido para o vetor que a comporta.

A linguagem C admite o crescimento dinâmico do tamanho de uma lista. Esse tipo de representação também facilita as operações de inclusão e exclusão de nós, já que elimina os deslocamentos. Esse tipo de representação é denominado *LISTA POR ENCADEAMENTO* ou *LISTA ENCADEADA.*

Nesta representação, os nós são representados por dois campos: o campo que contém a informação propriamente dita e o campo que permite sua ligação com o nó posterior (endereço do nó posterior). O último nó da lista tem endereço nulo (`NULL`).

A desvantagem dessa representação é que qualquer elemento da lista só pode ser acessado pelo início da lista. Assim, para se obter o valor do sétimo nó da lista é preciso percorrer os seis nós anteriores.

Esquematicamente, uma lista encadeada é representada por:



A barra (/) do último nó representa que o endereço deste campo é nulo, não aponta para lugar algum. O início da lista é reconhecido por um ponteiro que aponta para o primeiro nó.

Declaração de uma lista simplesmente encadeada

```
typedef struct reg *no;
struct reg {
          int info;
          struct reg *prox;
};

no lista;
```

Rotinas para manipulação de listas encadeadas

1) Criar uma lista vazia

```
// Cria uma lista vazia
void cria_lista (no *lista) {
    *lista=NULL;
}
```

2) Inserir um elemento no início da lista

```
// Inclui um elemento no início da lista
void inclui_elemento (no *lista, int info){
    no p = (no) malloc(sizeof(struct reg));
    p->info=info;
    p->prox=*lista;
    *lista=p;
}
```

3) Mostrar os elementos de uma lista

```
// Mostra os elementos da lista
void mostra_lista (no lista) {
    no p = lista;
    printf ("\nElementos da lista: ");
    while (p) {
       printf ("%d ",p->info);
       p = p->prox;
    }
}
```

4) Contar o número de elementos da lista

```
// Conta o numero de elementos da lista
int conta_nos (no lista){
    no p = lista;
    int n=0;
    while (p){
        n++;
      p = p->prox;
    }
    return n;
}
```

5) Inserir um elemento no final da lista

```
// Inclui um elemento no final da lista
void inclui_final (no *lista, int info){
    no p = (no) malloc(sizeof(struct reg));
    p->info=info;
    p->prox=NULL;
    if (*lista==NULL)
         *lista = p;
    else {
        no q = *lista;
```

```
            while (q->prox)
                 q = q->prox;
            q->prox = p;
         }
   }
```

6) Remover o primeiro elemento da lista

```
   // Remove elemento do inicio da lista
   int remove_inicio (no *lista) {
       if (!*lista)
          return 0;
       no p = *lista;
       *lista = p->prox;
       free (p);
       return 1;
   }
```

7) Criar uma lista com dois nós

8) Inserir elemento no início da lista

9) Verificar se um determinado elemento pertence à lista

10) Retornar o último elemento da lista

11) Inserir um elemento numa lista ordenada

12) Contar o número de vezes que um determinado elemento ocorre na lista

13) Copiar uma lista

14) Remover o último elemento da lista

15) Remover um determinado elemento da lista

16) Remover um determinado elemento da lista ordenada

17) Remover todos os elementos de uma lista

18) Remover todas as ocorrências de um determinado elemento na lista

19) Remover todas as ocorrências de um determinado elemento na lista ordenada

20) Concatenar duas listas

Programa que utiliza as rotinas apresentadas:

```
int main () {
  int info;
  char resp;
  cria_lista (&lista);
  do {
    printf ("\nDigite um numero inteiro: ");
```

```
    scanf ("%d",&info);
    inclui_final (&lista,info);
    printf ("\nContinua (S/N)?");
    do {
      resp = toupper(getchar());
    } while (resp!='N' && resp!='S');
  } while (resp!='N');
  mostra_lista (lista);
  printf ("\nQte de elementos da lista: %d",conta_nos(lista));
  remove_inicio (&lista);
  printf ("\n\nRemocao do 1º elemento da lista realizada!\n");
  mostra_lista (lista);
  printf ("\Qte de elementos da lista: %d",conta_nos(lista));
  getch();
  return 0;
}
```

Exercício

Faça um programa a seguir, utilizando o conceito de lista encadeada, que resolva o problema descrito.

Um pronto-socorro deseja informatizar o seu sistema de atendimento aos pacientes. As recepcionistas utilizarão um terminal para fazer a ficha de cada paciente que chega. Juntamente com os dados do paciente, a recepcionista preenche um campo informando o estado do paciente (regular, ruim, péssimo). Ao terminar de atender a um paciente, o médico consultará o sistema para chamar o próximo paciente e, naturalmente, o sistema deverá priorizar os pacientes que estiverem em pior estado de saúde.

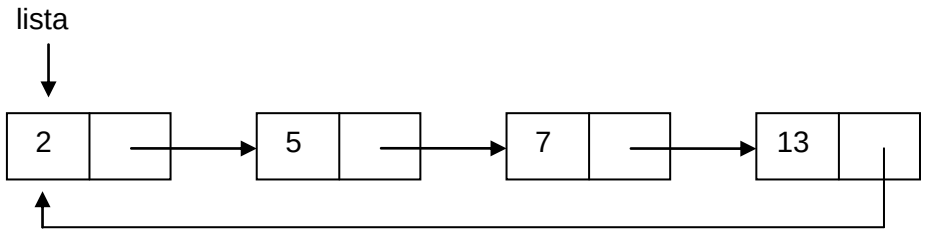
LISTA CIRCULAR SIMPLESMENTE ENCADEADA

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, as arestas que delimitam uma face podem ser agrupadas por uma estrutura circular. Para esses casos, podemos usar *listas circulares*.

Numa lista circular, o último elemento tem como próximo o primeiro elemento da lista, formando um ciclo. A rigor, neste caso, não faz sentido falarmos em primeiro ou último

Na lista circular não há `NULL` identificando o fim.

Esquematicamente, representa-se uma lista circular simplesmente encadeada por:



Para percorrer os elementos de uma lista circular, visita-se todos os elementos a partir do ponteiro do elemento inicial até alcançar novamente esse mesmo elemento. Deve-se salientar que o caso em que a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um elemento inicial vale `NULL`).

Declaração de uma lista circular simplesmente encadeada

```
typedef struct reg *no;
struct reg  {
           int info;
           struct reg *prox;
};

no lista;
```

Rotinas para manipulação de lista circular simplesmente encadeada

1) Criar uma lista circular vazia

```
// Cria uma lista circular vazia
void cria_lista (no *lista) {
  *lista = NULL;
}
```

2) Criar uma lista circular com apenas um nó

```
// Inicia lista circular com um elemento
void inclui_elemento (no *lista, int info) {
  no p = (no) malloc(sizeof(struct reg));
  p->info = info;
```

```
  p->prox = p;
  *lista = p;
}
```

3) Mostrar os elementos de uma lista circular

```
// Mostra os elementos da lista circular
void mostra_lista (no lista) {
  if (lista == NULL) {
      printf ("\nLista vazia");
      return;
  }
  no p = lista;
  printf ("\nElementos da lista: ");
  do {
    printf ("%d ",p->info);
    p = p->prox;
  } while (p != lista);
 }
```

4) Contar o número de elementos da lista circular

```
// Conta o numero de elementos da lista circular
int conta_nos (no lista) {
  if (lista == NULL)
     return 0;
  no p = lista;
  int n=0;
  do {
    n++;
    p = p->prox;
  } while (p != lista);
  return n;
 }
```

5) Inserir um elemento no início da lista circular

```
// Inclui um elemento no inicio da lista circular
void inclui_inicio (no *lista, int info) {
  no p = (no) malloc(sizeof(struct reg));
  p->info = info;
  if (!*lista) {
       p->prox = p;
       *lista = p;
  }
  else {
      no q = *lista;
      while (q->prox != *lista)
        q = q->prox;
      q->prox = p;
      p->prox = *lista;
      *lista = p;
  }
}
```

6) Inserir um elemento no final da lista circular

```
// Inclui um elemento no final da lista circular
void inclui_final (no *lista, int info) {
  no p = (no) malloc(sizeof(struct reg));
  p->info = info;
  if (!*lista) {
       p->prox = p;
       *lista = p;
  }
  else {
      no q = *lista;
      while (q->prox != *lista)
        q = q->prox;
      q->prox = p;
      p->prox = *lista;
  }
}
```

7) Remover o primeiro elemento da lista circular

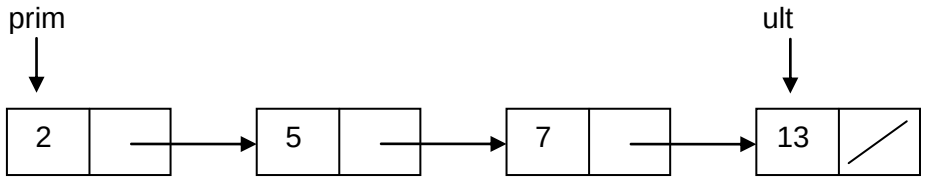
```
// Remove elemento do inicio da lista circular
int remove_inicio (no *lista) {
  if (!*lista)
    return 0;
  if ((*lista)->prox == *lista) {
      free (*lista);
      *lista = NULL;
  }
  else {
      no q = *lista;
      while (q->prox != *lista)
        q = q->prox;
      no p = *lista;
      *lista = p->prox;
      q->prox = *lista;
      free (p);
  }
  return 1;
}
```

LISTA SIMPLESMENTE ENCADEADA COM DESCRITOR

Nas estruturas de listas encadeadas estudadas até o momento, em qualquer operação que exija acesso ao último nó da lista haverá necessidade de percorrer todos os nós a partir do primeiro.

Com o objetivo de facilitar o acesso ao último nó da lista, utiliza-se uma segunda variável de referência (ponteiro) do nó para indicar o último nó.

Esquematicamente, para uma lista simplesmente encadeada tem-se:

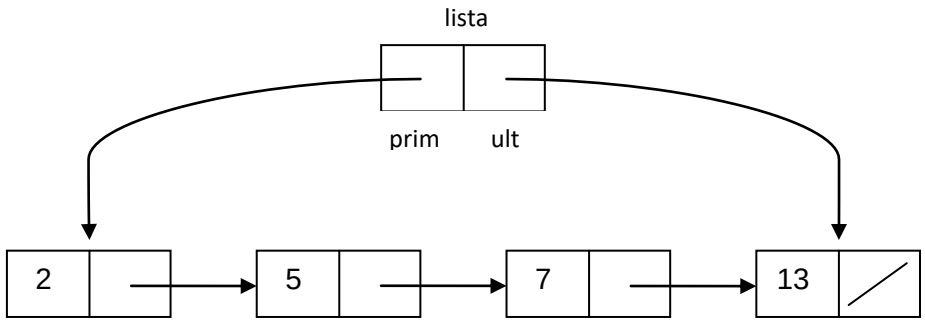



```
typedef struct reg *no;

struct reg {
           int info;
           struct reg *prox;
};

no prim, ult;
```

Pode-se reunir em um registro (`struct`) as variáveis `prim` e `ult`:

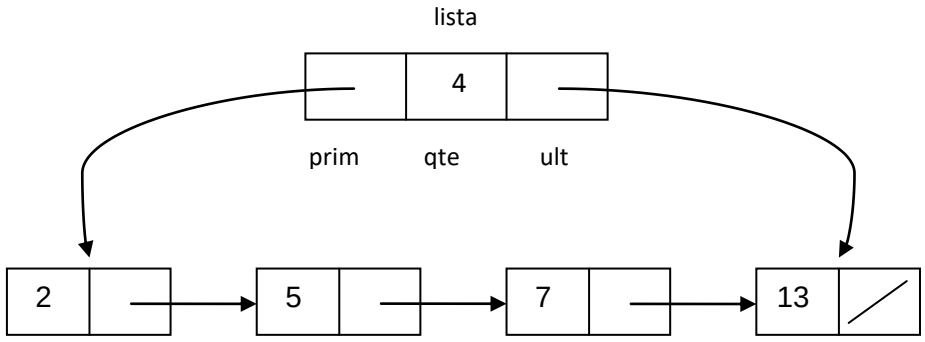


O elemento que reúne os dois ponteiros é denominado DESCRITOR (*descriptor*), LÍDER (*leader*) ou CABEÇA (*header*).

Tal descritor pode conter outras informações sobre a lista, por exemplo: quantidade de nós, finalidade da lista, descrição dos nós, etc.

No exemplo a seguir, a lista possui um descritor que contém 3 campos:

- `prim` – contém o endereço do primeiro nó;
- `qtd` – a quantidade de nós da lista;
- `ult` – o endereço do último nó.



Declaração de uma lista simplesmente encadeada com descritor

```
struct reg {
         int info;
         struct reg *prox;
};

typedef struct reg *no;

typedef struct {
               no prim, ult;
               int qte;
        } Descritor;

Descritor lista;
```

Acesso ao 1º nó da lista:

Acesso a informação do 1º nó da lista:

Obs: Observe que a variável *lista* não é mais uma variável do tipo ponteiro, mas uma variável estática do tipo registro (*struct*).

Rotinas para manipulação de listas encadeadas com descritor:

1) Criar uma lista vazia

```
// Cria uma lista simplesmente encadeada com descritor- LSECD
void inicia_LSECD (Descritor *lista) {
   (*lista).prim = (*lista).ult = NULL;
   (*lista).qte = 0;
}
```

2) Criar uma lista com apenas um nó

3) Criar uma lista com dois nós

4) Inserir elemento no início da lista

```
// Insere um elemento no início da LSECD
void insere_inicio_LSECD (Descritor *lista, int info) {
   no p;
   p = (no)malloc(sizeof(struct reg));
   p->info = info;
   p->prox = (*lista).prim;
   (*lista).prim = p;
   (*lista).qte++;
   if ((*lista).qte == 1)
      (*lista).ult = p;
}
```

5) Mostrar os elementos de uma lista

```
// Mostra os elementos da LSECD
void mostra_LSECD (Descritor lista) {
  if (lista.qte == 0) {
      printf ("\nLista vazia");
      return;
  }
  no p = lista.prim;
  printf ("\nElementos da lista: ");
  do {
    printf ("%d ",p->info);
    p = p->prox;
  } while (p != NULL);
  printf ("\nTotal de elementos: %d\n",lista.qte);
}
```

6) Contar o número de elementos da lista

7) Inserir um elemento no final da lista

```
// Insere um elemento no final da LSECD
void insere_final_LSECD (Descritor *lista, int info) {
   no p, q;
   p = (no)malloc(sizeof(struct reg));
   p->info = info;
```

```
   p->prox = NULL;
   if ((*lista).qte == 0)
      (*lista).prim = p;
   else
       (*lista).ult->prox = p;
   (*lista).ult = p;
   (*lista).qte++;
}
```

8) Verificar se um determinado elemento pertence à lista

```
// Verificar se um elemento pertence a LSECD
int verifica_LSECD (Descritor lista, int info) {
   if (lista.qte == 0)
     return 0;
   no p = lista.prim;
   do {
       if (p->info == info)
         return 1;
       p = p->prox;
   }  while (p != NULL);
   return 0;
}
```

9) Retornar o último elemento da lista

10) Inserir um elemento numa lista ordenada

11) Contar o número de vezes que um determinado elemento ocorre na lista

12) Copiar uma lista

13) Remover o primeiro elemento da lista

```
// remove um elemento do início da LSECD
int remove_inicio_LSECD (Descritor *lista) {
   if ((*lista).qte == 0)
      return 0;
   else  {
      no p = (*lista).prim;
      (*lista).prim = p->prox;
      if ((*lista).qte == 1)
         (*lista).ult = NULL;
      (*lista).qte--;
      free(p);
      return 1;
   }
}
```

14) Remover o último elemento da lista

```
// remove um elemento do final da LSECD
int remove_final_LSECD (Descritor *lista) {
   if ((*lista).qte == 0)
```

```
        return 0;
     no p, q;
     p = (*lista).prim;
     while (p->prox) {
        q = p;
        p = p->prox;
     }
     (*lista).qte--;
     if ((*lista).qte == 0) {
         (*lista).prim = NULL;
         (*lista).ult = NULL;
      }
      else {
          (*lista).ult = q;
           q->prox = NULL;
      }
      free(p);
      return 1;
  }
```

15) Remover um determinado elemento da lista

16) Remover um determinado elemento da lista ordenada

17) Remover todos os elementos de uma lista

18) Remover todas as ocorrências de um determinado elemento na lista

19) Remover todas as ocorrências de um determinado elemento na lista ordenada

20) Concatenar duas listas

Programa que utiliza as rotinas apresentadas:

```
int main () {
   int info;
   char resp;
   inicia_LSECD (&lista);
   mostra_LSECD (lista);
   do {
      printf ("\nDigite um numero inteiro: ");
      scanf ("%d",&info);
      insere_final_LSECD (&lista,info);
      mostra_LSECD (lista);
      printf ("\nContinua (S/N)?");
      do {
          resp = toupper(getchar());
      } while (resp!='N' && resp!='S');
   } while (resp!='N');
   mostra_LSECD (lista);
   remove_inicio_LSECD (&lista);
   printf   ("\n\nRemocao   do   primerio   elemento   da   lista
   realizada!\n");
   mostra_LSECD (lista);
```

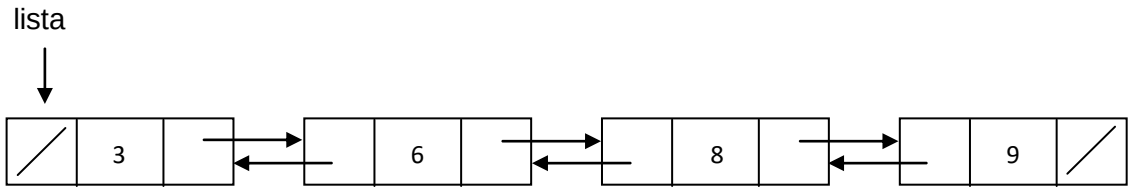
```
   remove_final_LSECD (&lista);
   printf ("\n\nRemocao do ultimo elemento da lista realizada!\n");
   mostra_LSECD (lista);
   do {
       printf ("\n\n\nDigite um numero inteiro: ");
       scanf ("%d",&info);
       if (verifica_LSECD (lista,info))
           printf ("\n%d pertence a lista\n",info);
       else
           printf ("\n%d nao pertence a lista\n",info);
      printf ("\nContinua (S/N)?");
      do {
          resp = toupper(getchar());
      } while (resp!='N' && resp!='S');
   } while (resp!='N');
   getch();
   return 0;
}
```

LISTA LINEAR DUPLAMENTE ENCADEADA

Nas listas com descritores, vistas anteriormente, a operação de remoção do último nó apresenta a necessidade de percorrer os nós sequencialmente, a partir do primeiro elemento até chegar no penúltimo nó.

Para evitar esse processo, é possível estruturar uma lista de forma que possa ser percorrida em ambos os sentidos (do início até o fim e do fim até o início). Uma organização implementando esse tipo de percurso é denominada *LISTA LINEAR DUPLAMENTE ENCADEADA* (*LLDE*).

Esquematicamente, pode-se representar uma lista linear duplamente encadeada como:



Cada nó de uma LLDE possui menos <u>três campos</u>. Um campo do tipo ponteiro indicando o próximo nó, outro ponteiro indicando o nó anterior e o terceiro campo contendo o dado.

<u>Declaração de uma lista linear duplamente encadeada</u>

```
typedef struct reg *no;

struct reg  {
            int info;
            no ant, prox;
      };

no lista;
```

Rotinas para manipulação de lista linear duplamente encadeada:

1) Criar uma lista vazia
2) Criar uma lista com apenas um nó
3) Criar uma lista com dois nós
4) Inserir elemento no início da lista
5) Mostrar os elementos de uma lista
6) Contar o número de elementos da lista
7) Inserir um elemento no final da lista
8) Verificar se um determinado elemento pertence à lista
9) Retornar o último elemento da lista
10) Inserir um elemento numa lista ordenada
11) Contar o número de vezes que um determinado elemento ocorre na lista
12) Copiar uma lista
13) Remover o primeiro elemento da lista

14) Remover o último elemento da lista
15) Remover um determinado elemento da lista
16) Remover um determinado elemento da lista ordenada
17) Remover todos os elementos de uma lista
18) Remover todas as ocorrências de um determinado elemento na lista
19) Remover todas as ocorrências de um determinado elemento na lista ordenada
20) Concatenar duas listas

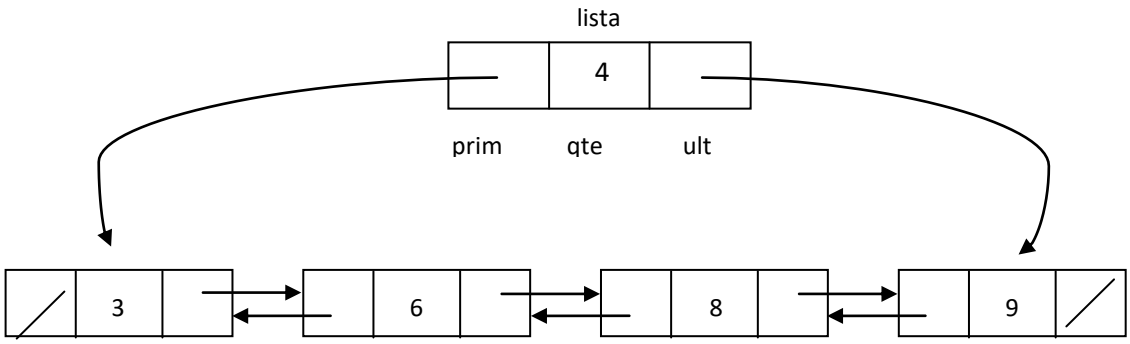
Trecho da inclusão de um nó entre dois nós (o nó antes do ponto de inclusão é indicado pelo ponteiro q):

```
p = (no)malloc(sizeof(struct reg));
p->ant = q;
p->info = info
p->prox = q->prox;
q->prox = p;
p->prox->ant = p;
```

Trecho para excluir um nó entre dois outros (o nó a ser excluído é indicado pelo ponteiro p):

```
p->ant->prox = p->prox;
p->prox->ant = p->ant;
info = p->info;
free(p);
```

LISTA LINEAR DUPLAMENTE ENCADEADA COM DESCRITOR



Declaração de uma lista linear duplamente encadeada com descritor

```
typedef struct reg *no;

struct reg  {
           int info;
           no ant, prox;
};

typedef struct {
               no prim, ult;
               int qte;
        } Descritor;

Descritor lista;
```