Listas Ligadas (Encadeadas)

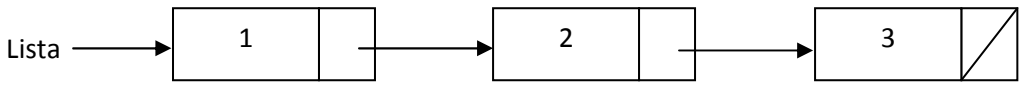
Matrizes são estruturas de dados muito úteis fornecidas nas linguagens de programação. No entanto, elas têm pelo menos duas limitações:

1- Seu tamanho tem que ser conhecido no momento da compilação.
2- Os dados em uma matriz estão separados na memória do computador pela mesma distância, o que significa que inserir um item dentro da matriz exige que se movam outros dados nessa matriz.

Tal limitação pode ser superada usando-se estruturas ligadas. Uma estrutura ligada é uma coleção de nós, que armazenam dados e ligações com outros nós. Desse modo, os nós podem estar em qualquer lugar na memória, e a passagem de um nó para outro da estrutura ligada é realizada armazenando os endereços de outros nós na estrutura ligada.

Listas Simplesmente Encadeadas

Uma lista simplesmente encadeada é uma sequência de nós, onde cada nó contém uma informação de algum tipo de dado e o endereço do nó seguinte. Vamos supor que a informação seja uma variável do tipo int. Veja abaixo um exemplo de uma lista simplesmente encadeada.



Onde:

lista é um ponteiro para o primeiro nó da lista que contém o valor "1" e um ponteiro para o próximo nó. Já o último nó da lista aponta para NULL, indicando fim da lista.

A estrutura de cada nó de uma lista pode ser definida assim:

```
typedef struct {
            int info;
            struct lista *proximo;
         }lista;
```

Criação de Novos Elementos (nós) para a lista

Antes de inserirmos um elemento na lista, precisamos criá-lo da seguinte maneira:

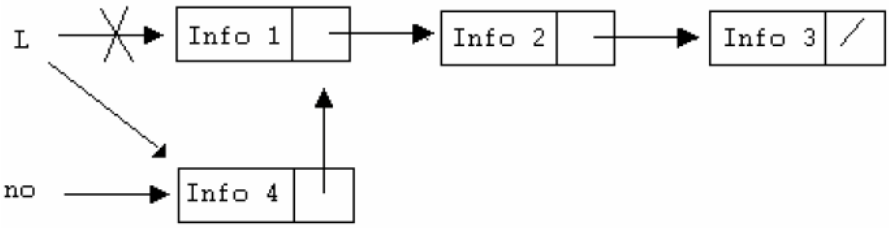
```
lista *criarElemento(int info){
        lista *no = (lista*) malloc(sizeof (lista));
        if (no == NULL){
          return NULL;
        }
        no->info = info;
        no->proximo = NULL;

        return no;
}
```

Inserção de Nós na Lista

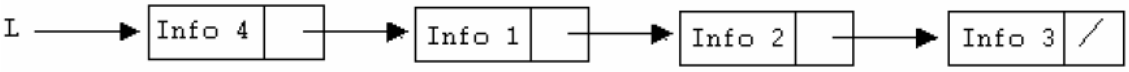
Após a criação do nó, devemos inseri-lo na lista. Temos três casos para inserção de elementos na lista:

O nó é inserido **no início** da lista
O nó é inserido **no meio** da lista
O nó é inserido **no fim** da lista

Quando o nó é inserido **no início** da lista deve-se fazer o campo "próximo" deste nó apontar para o primeiro elemento da lista, e em seguida fazer a lista apontar para este nó recém-criado.



E então teremos:



Segue abaixo um exemplo em linguagem C

```
void inserirInicio(lista **listaTotal, lista *elemento){

       if(*listaTotal == NULL){
          *listaTotal = elemento;
       }
       else{
            elemento->proximo =(struct lista *) *listaTotal;
             *listaTotal = elemento;
      }
}
```

Quando o nó é inserido **no fim** da lista, deve-se fazer o campo "próximo" do último nó da lista apontar para o elemento que se deseja inserir. Para tanto, devemos percorrer a lista até encontrar o último elemento. Lembrando que o último elemento sempre aponta para NULL.

Abaixo temos um exemplo em linguagem C.

```
void inserirFim(lista **listaTotal, lista *elemento){
	lista *elementoAuxiliar;

	elementoAuxiliar = *listaTotal;
	/* Percorre a lista ateh encontrar o ultimo elemento */
	while (elementoAuxiliar->proximo != NULL){
		elementoAuxiliar = (lista *) elementoAuxiliar->proximo;
	}

	elementoAuxiliar->proximo = (struct lista *) elemento;
}
```

Quando o nó é inserido **no meio** da lista, deve-se informar a partir de qual nó o novo elemento será inserido. Ou seja, vamos utilizar o campo "info" para fazer uma pesquisa na lista, e após encontrá-lo, inserir o novo elemento.

Abaixo temos um exemplo em linguagem C.

```
void inserirMeio(lista **listaTotal, lista *elemento, int info){
	lista *elementoAuxiliar;
	int encontrou=0;

	elementoAuxiliar = *listaTotal;
	/* Percorre a lista ateh encontrar o no procurado */
	while(encontrou == 0 && elementoAuxiliar->proximo != NULL){
		/* Verifica se o no atual eh o no procurado */
		if (elementoAuxiliar->info == info){
			/* O novo no sera inserido apos o no atual */
			elemento->proximo = elementoAuxiliar->proximo;
			elementoAuxiliar->proximo=(struct lista *) elemento;
			encontrou = 1;
		}
		else{
			elementoAuxiliar=(lista *) elementoAuxiliar->proximo;
		}
	}
}
```

Percorrendo a Lista

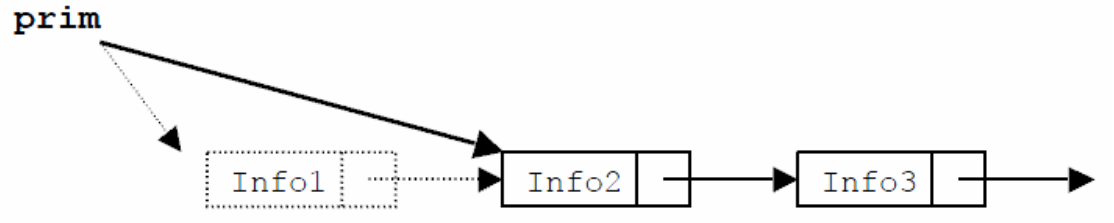
Em situações de busca por um valor, ou para impressão de uma lista, deve-se percorrer a lista a partir do primeiro elemento até onde for desejado, ou seja, até achar o nó (elemento) procurado ou chegar ao final da lista.

Veja abaixo um exemplo para encontrar o nó procurado.

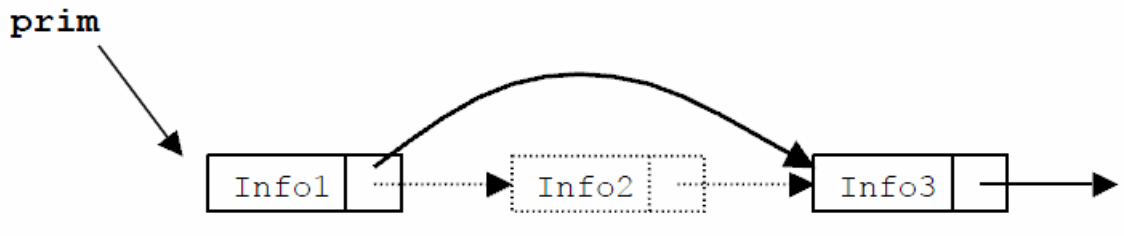
```
lista *buscarElemento(lista **listaTotal, int valorProcurado){
      lista *elementoAuxiliar;
      int encontrou=0;

      elementoAuxiliar = *listaTotal;
      /* Percorre a lista ateh encontrar o no procurado, ou ateh o fim
da lista */
      while(encontrou == 0 && elementoAuxiliar != NULL){
            /* Verifica se o no atual contem o valorProcurado */
            if (elementoAuxiliar->info == valorProcurado){
                  /* encontrou o no procurado */
                  encontrou = 1;
            }
            else{
                  elementoAuxiliar=(lista *)elementoAuxiliar->proximo;
            }
      }
      /* Se encontrou o no procurado, retorna o no */
      if (encontrou == 1){
            return elementoAuxiliar;
      }
      /* Se nao encontrou, retorna NULL */
      else{
            return NULL;
      }
}
```

Remover um Elemento da Lista

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. Veja abaixo as figuras que ilustram a remoção.



Remoção do primeiro elemento (nó) da lista



Remoção de um elemento (nó) no meio da lista

Segue abaixo um exemplo em linguagem C de uma função para retirar um elemento (nó) da lista. Inicialmente, busca-se pelo elemento (nó) que se deseja retirar, guardando uma referência para o elemento anterior.

```
int removerElemento(lista **listaTotal, int valorProcurado){
      lista *elementoAuxiliar, *elementoAnterior;
      int encontrou=0;

      elementoAuxiliar = *listaTotal;
      elementoAnterior = *listaTotal;

      /* Percorre a lista ateh encontrar o elemento (nó) procurado */
      while (encontrou == 0 && elementoAuxiliar != NULL){
            if (elementoAuxiliar->info == valorProcurado){
                  /* Encontrou o elemento (nó) procurado */
                  encontrou = 1;
            }
            else{
                  /*nao encontrou o elemento(nó). Avança ao proximo */
                  elementoAnterior = elementoAuxiliar;
                  elementoAuxiliar=(lista *) elementoAuxiliar->proximo;
            }
      }
      if (encontrou == 1){
            /* Verifica se o elemento (nó) encontrado eh o
               primeiro da lista */
            if (elementoAnterior == elementoAuxiliar
                        && elementoAuxiliar == *listaTotal){
                  *listaTotal = (lista *) elementoAuxiliar->proximo;
            }
      /*Caso o elemento(nó) encontrado nao seja o primeiro da lista*/
            else{
                  elementoAnterior->proximo=elementoAuxiliar->proximo;
            }
            /* Libera a memoria alocada */
            free(elementoAuxiliar);
            elementoAuxiliar = NULL;
            return 1;
      }
      /*Retorna zero indicando que o elemento(no) nao foi encontrado*/
      return 0;
}
```

Destruir a Lista

A memória é alocada para cada elemento novo que é inserido na lista. Conseqüentemente, ao terminar de usar a lista, deve-se liberar a memória utilizada.

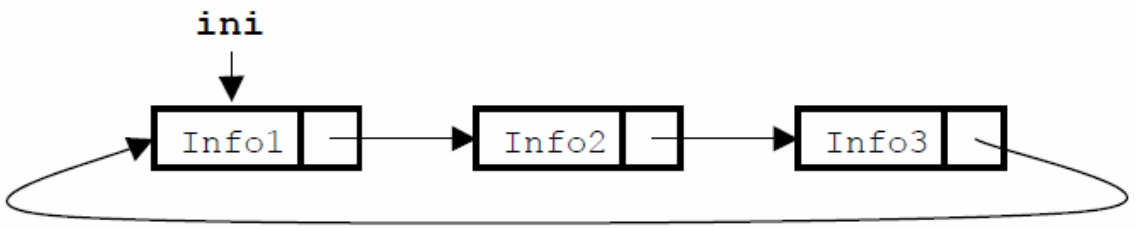
Exemplo em linguagem C para destruir a lista.

```
void desturirLista(lista **listaTotal){
      lista *elementoAuxiliar1, *elementoAuxiliar2;

      elementoAuxiliar1 = *listaTotal;
      while (elementoAuxiliar1 != NULL){
            elementoAuxiliar2 = (lista *)elementoAuxiliar1->proximo;
            free(elementoAuxiliar1);
            elementoAuxiliar1 = elementoAuxiliar2;
      }
      *listaTotal = NULL;
}
```

Listas Circulares

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, os dias da semana ou os meses do ano. Após o último, vem sempre o primeiro novamente, reiniciando um novo ciclo. Para esses casos, podemos utilizar 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 elemento. A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista. A figura abaixo ilustra a representação de uma lista circular.

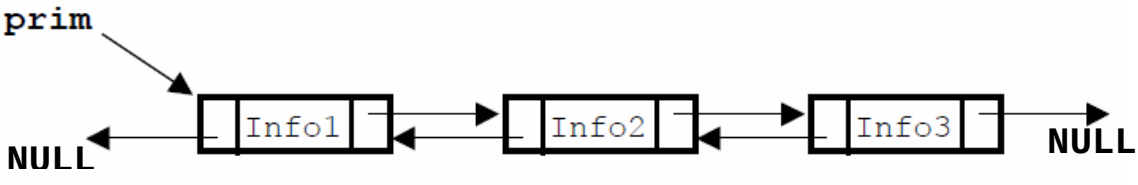


Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançarmos novamente esse mesmo elemento. O código abaixo exemplifica essa forma de percorrer os elementos.

```
void percorreListaCircular(lista **listaTotal){
      lista *elementoAuxiliar;

      elementoAuxiliar = *listaTotal;
      /* Verifica se a lista estah vazia */
      if (elementoAuxiliar == NULL){
            return;
      }

      do{
            printf("%d\n", elementoAuxiliar->info);
            elementoAuxiliar = (lista *)elementoAuxiliar->proximo;
      }while (elementoAuxiliar != *listaTotal);
}
```

Listas Duplamente Encadeadas

Em uma lista duplamente encadeada cada elemento (nó) tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Desta forma, dado um elemento, podemos acessar ambos os elementos adjacentes: o próximo e o anterior. Se tivermos um ponteiro para o último elemento da lista, podemos percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior, até alcançar o primeiro elemento da lista, que não tem elemento anterior, ou seja, aponta para NULL.



O elemento (nó) de uma lista duplamente encadeada pode ser representado pela estrutura abaixo.

```
typedef struct {
            int info;
            struct lista *proximo;
            struct lista *anterior;
         }lista;
```

Função main (principal) para testar a lista ligada

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

int main(int argc, char *argv[])
{
      lista *listaTotal=NULL, *elemento1=NULL, *elemento2=NULL;
      lista *elemento3, *elemento4, *elemento5, *elementoBusca;

      elemento1 = criarElemento(1);

      inserirInicio(&listaTotal, elemento1);

      elemento2 = criarElemento(2);

      inserirInicio(&listaTotal, elemento2);

      elemento3 = criarElemento(3);

      inserirFim(&listaTotal, elemento3);

      elemento4 = criarElemento(4);

      inserirFim(&listaTotal, elemento4);

      elemento5 = criarElemento(5);

      inserirMeio(&listaTotal, elemento5, 3);

      elementoBusca = buscarElemento(&listaTotal, 10);

      /* Exemplo para remover o primeiro elemento da lista */
      /* removerElemento(&listaTotal, 2); */

      /* Exemplo para remover o ultimo elemento da lista */
      /* removerElemento(&listaTotal, 4); */

      /* Exemplo para remover um elemento do meio da lista */
      removerElemento(&listaTotal, 10);

      desturirLista(&listaTotal);

      getch();
    return 0;
}
```