Lista

Uma lista é uma estrutura que armazena elementos de forma alinhada, ou seja, com elementos dispostos um após o outro, como em uma lista de nomes, peças, valores, pessoas, compras, etc. Uma lista, como um vetor, pode ser implementada como uma sequência de registros com elementos disponíveis de forma consecutiva - Lista Estática Sequencial - ou não consecutiva - Lista Estática Encadeada. Uma lista pode ser ordenada ou não.

Lista Estática Sequencial

Uma lista estática sequencial é um arranjo de registros onde estão estabelecidos regras de precedência entre seus elementos ou é uma coleção **ordenada** de componentes do mesmo tipo. O sucessor de um elemento ocupa posição física subseqüente.

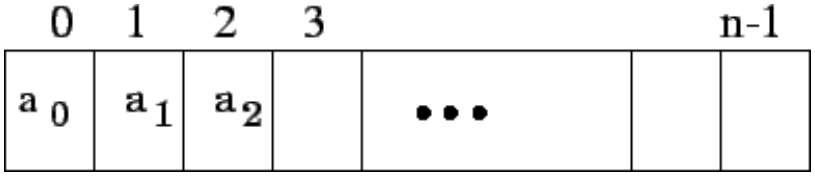
Ex: lista telefônica, lista de alunos.

A implementação de operações pode ser feita utilizando *array* e *record*, onde o vetor associa o elemento a(i) com o índice i (mapeamento sequencial).

Características de Lista Estática Sequencial

- elementos na lista estão ordenados;
- armazenados fisicamente em posições consecutivas;
- inserção de um elemento na posição a(i) causa o deslocamento a direita do elemento de a(i) ao último;
- eliminação do elemento a(i) requer o deslocamento à esquerda do a(i+1) ao último;

Desta forma, estamos armazenando o elemento a_i e a_{i+1} nas consecutivas posições *i* e *i+1* do array.



Assim as propriedades estruturadas da lista permitem responder a questões como:

- qual é o primeiro elemento da lista;
- qual é o último elemento da lista;
- quais elementos sucedem um determinado elemento;
- quantos elementos existem na lista;
- inserir um elemento na lista;
- eliminar um elemento da lista.

Consequência: As quatros primeiras operações são feitas em tempo constante. Mas, as operações de inserção e remoção requerem mais cuidados.

Vantagem:

- acesso direto indexado a qualquer elemento da lista;

- tempo constante para acessar o elemento *i* - dependerá somente do índice;

Desvantagem:

- movimentação quando eliminado/inserido elemento;
- tamanho máximo pré-estimado;

Quando usar:

- listas pequenas;
- inserção/remoção no fim da lista;
- tamanho máximo bem definido;

Definição da Estrutura de dados

A Lista Estática Sequencial pode ser definida da seguinte forma:

```
#define MAX 15
struct lista
{
      int elementos[MAX];
      int quantElem;
};
//Como a lista é estática, ela é alocada em tempo de compilação, dessa forma, se faz necessário o uso de um array (elementos[MAX], na struct), tendo em vista que a lista será sequencial. Além disso, o tamanho desse array já deve ser conhecido (MAX).
//O armazenamento da quantidade de elementos (quantElem, na struct) é ideal para facilitar algumas operações.
```

Supondo a seguinte declaração: `struct lista L1;`

O acesso aos campos da estrutura pode ser feito da maneira a seguir:

```
//Usando a struct                    Usando um ponteiro para a struct
L1.quantElem                         L1->quantElem
L1.elementos[0]                      L1->elementos[0]
```

Exercício

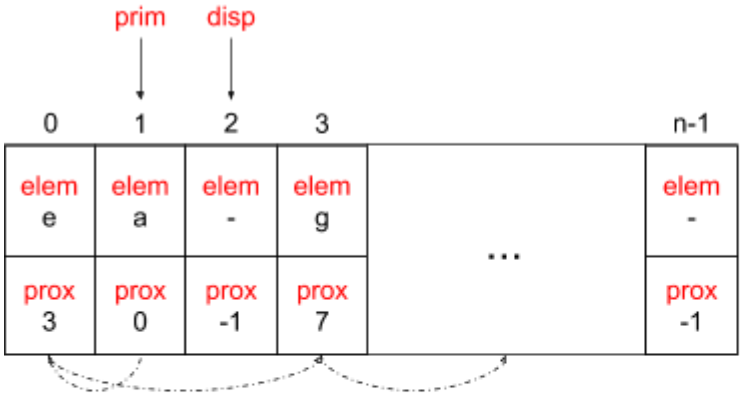
Implementar as seguintes operações em uma lista estática sequencial:

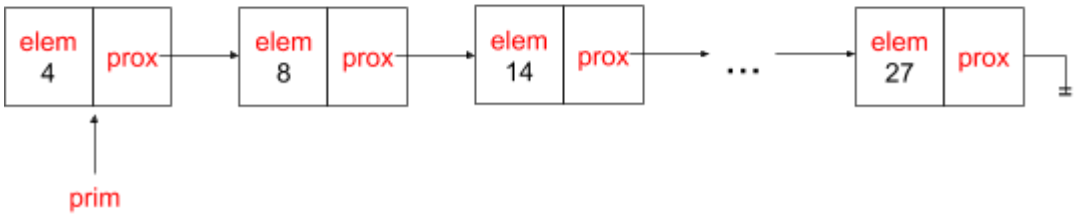
- Inserção de um elemento na posição *i*
- Remoção do i-ésino elemento
- Imprimir a lista de forma recursiva

Listas Encadeadas

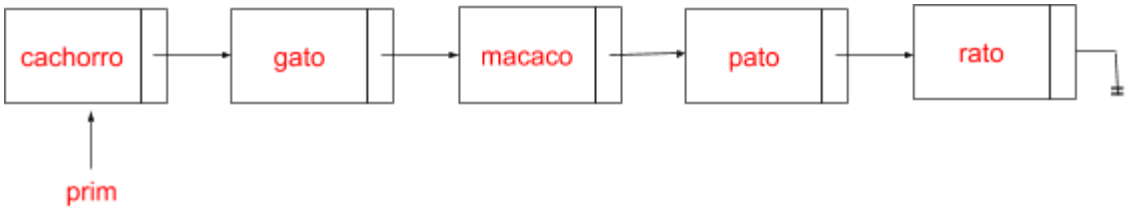
Em listas encadeadas, elementos consecutivos na lista não implica em elementos consecutivos na representação (a ordem é lógica).

Existem duas formas de se representar listas encadeadas, através de vetor, denominadas **listas encadeadas estáticas**, ou por ponteiros chamadas **listas encadeadas dinâmicas**.

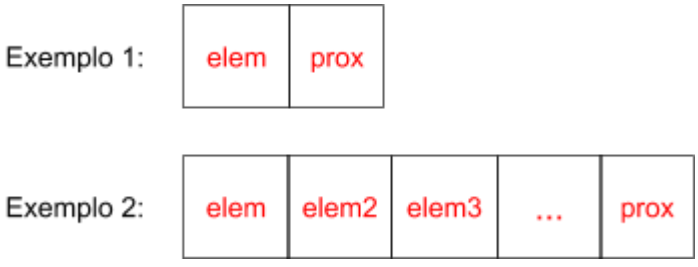




Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor (campo **prox**, na imagem acima).

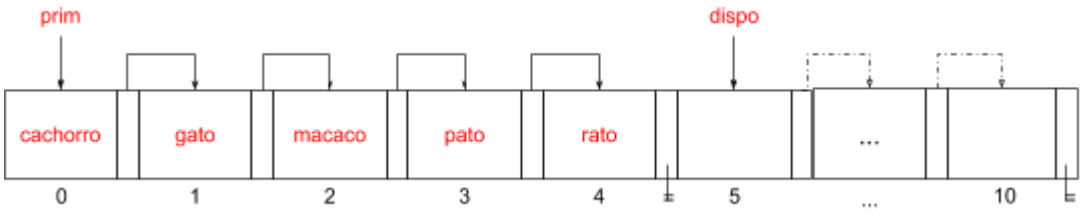


Cada registro pode ser composto por um ou vários campos para armazenar as informações (além do campo para guardar o próximo elemento):



Lista Estática Encadeada

Suponha a existência da seguinte lista estática encadeada (que chamaremos de L):

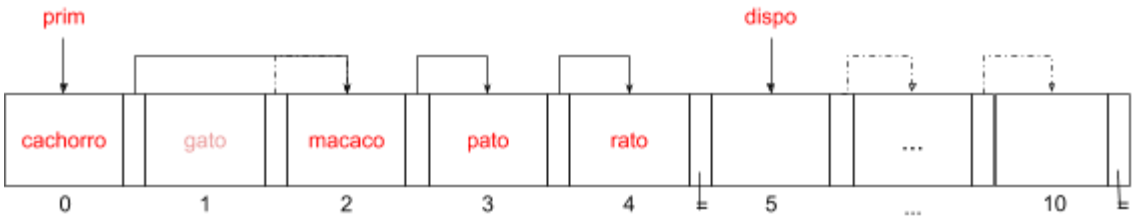


Prim = 0 **Dispo=5**

	elem	prox
0	cachorro	1
1	gato	2
2	macaco	3
3	pato	4
4	rato	-1
5		6
6		7
7		8
8		9
9		10
10		-1

Podemos perceber que **Prim** sempre armazena a posição do primeiro elemento da lista, da mesma maneira, **Dispo** armazena a posição disponível. O campo **elem** contém a informação a ser armazenada. **Prox** indica o índice do próximo elemento, o que garante o encadeamento na estrutura estática.

Eliminando o elemento "gato" teremos:



Prim = 0 **Dispo=5**

	elem	prox
0	cachorro	**2**
1	**gato**	**2**
2	macaco	3
3	pato	4
4	rato	-1
5		6
6		7
7		8
8		9
9		10
10		-1

O registro 1 tornou-se disponível para as próximas inserções.

Após sucessivas inserções e eliminações como descobrir quais registros estão disponíveis?

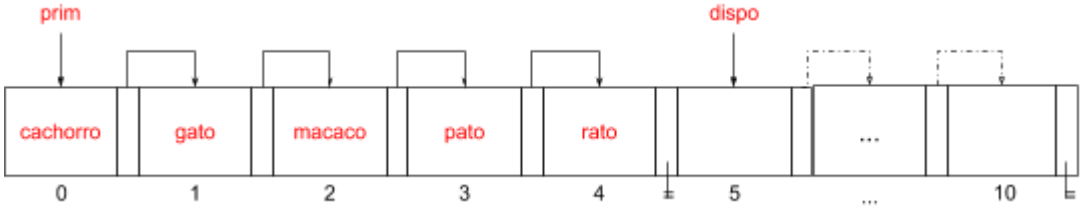
Juntá-los numa lista DISPO. Assim, os registros 5, 6, …, 10 estariam inicialmente na lista DISPO.

Como deverá ser a lista Dispo ? Sequencial?

Ela deve ser capaz de anexar os registros eliminados da lista principal L. Suponha que queremos inserir algum elemento.

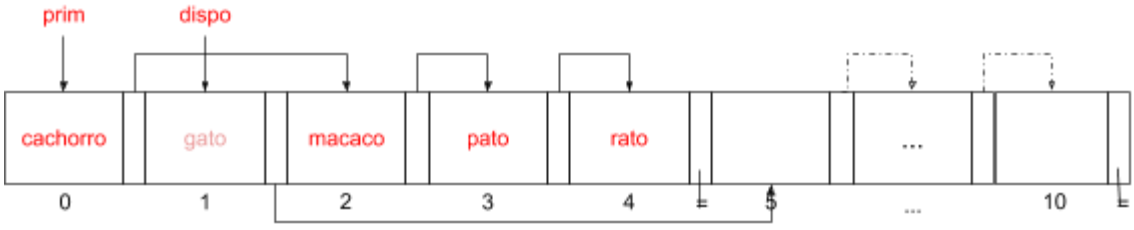
Isso implica que :

- a eliminação de um elemento da lista principal causa a inserção de um registro na lista Dispo;
- a inserção de um elemento na lista principal causa a utilização de um dos registros da Dispo;



No exemplo dado (lista acima), ao eliminar "gato" anexamos esse registro à dispo.

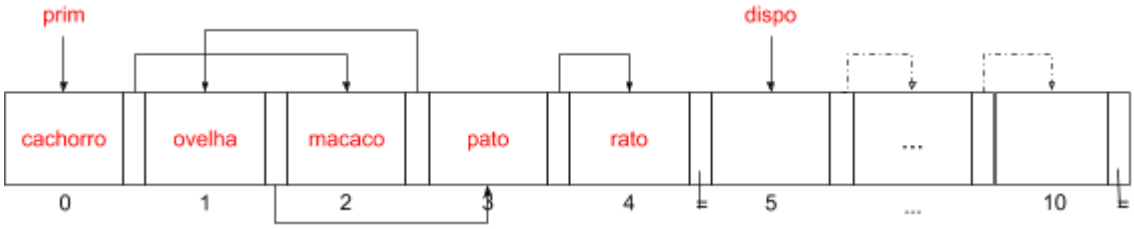
A princípio podemos utilizar qualquer posição (já que todas são vazias). A posição mais conveniente é a do primeiro elemento do Dispo - uma vez que requer o acesso a poucos ponteiros.



Prim = 0 **Dispo=1**

	elem	prox
0	cachorro	**2**
1	gato	5
2	macaco	3
3	pato	4
4	rato	-1
5		6
6		7
7		8
8		9
9		10
10		-1

Se a próxima operação é a inserção do elemento **ovelha** temos:



Prim = 0 **Dispo=5**

	elem	prox
0	cachorro	**2**
1	ovelha	3
2	macaco	1
3	pato	4
4	rato	-1
5		6
6		7
7		8
8		9
9		10
10		-1

Com várias inserções e eliminações, os registros da lista principal ficariam espalhados pelo vetor, intermediados por registros disponíveis.

Vantagens:

- não requer mais a movimentação de elementos na inserção e eliminação (como na lista sequencial);
- apenas os ponteiros são alterados (lembre que cada registro pode conter elementos muito complexos).

Desvantagens:

- necessário prever espaço máximo da lista;
- necessário gerenciar a Dispo;
- o acesso é não indexado, para acessar a(i) temos que percorrer a(1) ... a(i-1) pois o endereço de a(i) está disponível apenas em a(i-1);
- aumento do tempo de execução, o qual é gasto obtenção de espaço de memória;
- reserva de espaço para a Dispo;
- tamanho máximo pré-definido.

Quando usar:

- quando for possível fazer uma boa previsão do espaço utilizado (lista principal + Dispo) e quando o ganho dos movimentos sobre a perda do acesso direto a cada elemento for compensador.

Definição da Estrutura de dados

```
#define MAX 11
struct reg
{
      //Aqui estamos utilizando apenas um campo para inteiro para armazenar os dados (elem), porém poderia ser uma outra struct (depende da aplicação)
    int elem;
```

```
    int prox;
      //Este campo é necessário para gerar o encadeamento entre os
elementos da lista
};
struct lista
{
    int disp;
    int prim;
    struct reg A[MAX];
};
```

Lista Encadeada Estática Inicializada:

	Prim = -1	Dispo= 0
	elem	**prox**
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		-1

Para inicializar a lista encadeada estática, podemos utilizar a seguinte função:

```
void InicializaLista(struct lista *L)
{
    int i = 0;
    L->disp = 0;
    L->prim = -1;
    for(i=0 ; i<MAX-1 ; i++)  //Inicializando o prox de cada elemento
        L->A[i].prox=i+1;
    L->A[i].prox= -1;
}
```

OBS 1: nesta função, um ponteiro para a struct lista é utilizado na intenção de alterar os valores desta lista. Dessa forma, se faz necessário o uso de "`->`" para acessar os campos da struct.
OBS 2: Supondo a declaração da seguinte maneira: `struct lista L1;` na função main, a função deverá ser chama da seguinte maneira: `InicializaLista(&L1);` o endereço da lista é passado como parâmetro, pois a função `InicializaLista` utiliza um ponteiro (L).
OBS 3: Ao imprimir a lista após a inicialização (feita pela função `InicializaLista`), o campo `elem` terá valores inesperados, uma vez que este campo não foi alterado. Dessa forma, haverá "lixo" em seu conteúdo.

Exercícios

1) Considerando as estruturas de uma lista estática encadeada, implemente as seguintes funções que operam sobre uma lista. Construa também uma função main que faça uso das funções, usando um menu de escolhas.

a) FLVazia
parâmetros: TipoLista L;

pós-condição: Lista L vazia;
funcionalidade: cria uma lista vazia;
resultado: retorno de uma lista vazia criada;

b) Vazia
parâmetros: TipoLista L, int flag;
funcionalidade: testa se a lista está vazia ou não;
resultado: flag=TRUE se a lista está vazia, flag=FALSE caso contrário;

c) Retira
parâmetros: TipoLista L, TipoItem x, int pos int flag;
pré-condição: Lista L tem n>1 elementos e pos <= posição do último;
pós-condição: Lista L tem n – 1 elementos;
funcionalidade: retorna o item x que está na posição pos da lista mantendo a ordenação dos demais;
resultado: flag=TRUE se item removido com sucesso, flag=FALSE caso contrário;

d) Insere
parâmetros: TipoItem x, TipoLista L, int flag;
pré-condição: Lista L tem n>= 0 elementos e n <TamMax;
pós-condição: Lista L tem n+1 elementos;
funcionalidade: insere o elemento x ordenado na lista;
resultado: flag=TRUE se item inserido com sucesso, flag=FALSE caso contrário;

e) Imprime
parâmetros: TipoLista L;
funcionalidade: imprime os itens da lista;
resultado: impressão dos itens da lista na ordem de ocorrência;

2) Uma empresa precisa organizar os dados de seus funcionários. Estes dados são: nome, matrícula (numérico), departamento, salário. Construa um programa em C e estruturas apropriadas para organizar estes dados em uma lista estática. Use as funções já implementadas no item anterior como auxiliares, modificando-as conforme necessário.

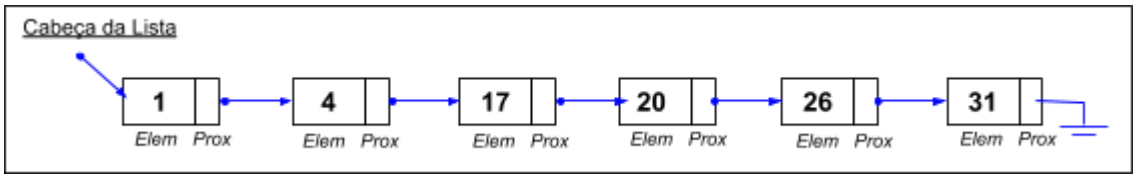
Pesquisar Algoritmos de Busca em Lista Estática:

- **não ordenadas**
- **ordenadas**

Lista Encadeada Dinâmica

Lista encadeada

Numa lista encadeada, para cada novo elemento inserido na estrutura, se é alocado um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não se pode garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto, não se tem acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, deve-se explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A figura a seguir ilustra o arranjo da memória de uma lista encadeada.



A estrutura consiste numa sequência encadeada de elementos, em geral chamados de *nós da lista*. A lista é representada por um ponteiro para o primeiro elemento (ou nó). Essa variável (também chamada de *cabeça da lista*) possibilita o acesso aos demais elementos contidos nela. Do primeiro elemento, pode-se alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para ***NULL***, sinalizando que não existe um próximo elemento.
Nesse caso, o primeiro elemento da lista, representado pelo atributo ***Elem***, é **1**, o último elemento é **31**, o predecessor de **31** é **26**, o sucessor de **1** é **4** e assim por diante até o último elemento. Lembrem-se, os elementos dessa lista não estão organizados na memória sequencialmente, igual a um arranjo. O campo ***Prox*** representa o ponteiro para o próximo elemento (endereço onde está armazenado o elemento seguinte da lista).
Na representação algorítmica é bastante simples ilustrar esses apontadores ou ponteiros, mas a linguagem de programação Java não aceita ponteiros – eles são representados por uma referência ao elemento (auto-referência).

Estruturas auto-referenciadas

Uma estrutura **auto-referenciada** contém um membro ponteiro que aponta para uma estrutura do mesmo tipo. Por exemplo, a declaração define um tipo **No**, com os atributos **elem** (conteúdo) e **prox** que faz a referência a uma estrutura do tipo No.

```
typedef struct no *No;
//estamos chamando de No, todo ponteiro para a struct no
struct no {
   int elem;
   No prox;    //auto-referência
};
```

Sem o uso do `typedef` podemos fazer da seguinte maneira:

```
struct no {
   int elem;
```

```
   struct no *prox;    //auto-referência
};
```

Alocação Dinâmica da memória

Alocação dinâmica consiste na capacidade de um programa obter mais espaço de memória em tempo de execução para armazenar novos *nós* e liberar espaço não mais necessário.
O limite para alocação de memória pode ser tão grande quanto à quantidade de memória física disponível no computador ou o espaço disponível em um sistema de memória virtual. Em geral esse espaço é menor por ter que ser compartilhado com outros aplicativos.

As funções **malloc** e **free** e o operador **sizeof** são essenciais para a alocação dinâmica de memória. A função **malloc** utiliza como argumento o número de bytes a serem alocados e retorna um ponteiro do tipo **void*** (*ponteiro para void*) para a memória alocada. Um ponteiro **void*** pode ser atribuído a uma variável de qualquer tipo de ponteiro. A função **malloc** é usada normalmente com o operador **sizeof**. Por exemplo, as instruções:

```
struct no *novoNo = (struct no*) malloc (sizeof(struct no));
```

ou:

```
No novoNo = (No) malloc (sizeof(struct no));
```

processa **sizeof(struct no)** para determinar o tamanho em bytes de uma estrutura do tipo **struct no,** aloca uma nova área na memória (por meio da função **malloc**) e armazena na variável **novoNo** um ponteiro para a memória alocada. Se não houver memória disponível, **malloc** retorna um ponteiro **NULL;**
A função free libera memória alocada – i.e., a memória é retornada ao sistema para que possa ser realocada no futuro.

```
free(novoNo);
```

TAD Lista

Para se implementar o Tipo Abstrato Lista tem-se que armazenar informações para controle dessa lista. Informações estas como início da lista (primeiro elemento) e o último elemento. Além dessas informações, são necessárias algumas operações, as quais são apresentadas a seguir:

1. Criar lista vazia
2. Inserir no início de uma lista
3. Acessar primeiro elemento
4. Acessar último elemento
5. Obter o número de elementos (tamanho) da lista
6. Inserir valor ***v*** na i-ésima posição
7. Eliminar elemento da i-ésima posição
8. Eliminar primeiro elemento
9. Eliminar valor ***v***
10. Inserir valor ***v*** como antecessor do elemento apontado por ***p***
11. Criar uma lista com registros numerados
12. Eliminar sucessor de ***p***
13. Imprimir o conteúdo da lista

```
struct Lista {
  No priElem;
  No ultElem;
};
```

ou:

```
struct Lista {
  struct no *priElem;
  struct no *ultElem;
};
```

Principais Operações para uma Lista Encadeada

Operação de Inicialização

O método que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro (referência) para o primeiro e último elemento, uma lista vazia é representada pela referência ***NULL***, pois não existem elementos na lista. O método tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é ***NULL***.

```
void inicializaLista(struct Lista *list){    //Inicialização da Lista
   list->priElem = NULL;
   list->ultElem = NULL;
}
//Como utilizamos um ponteiro para a strict Lista, o acesso aos seus
campos deve ser feito por meio do operador "->"
```

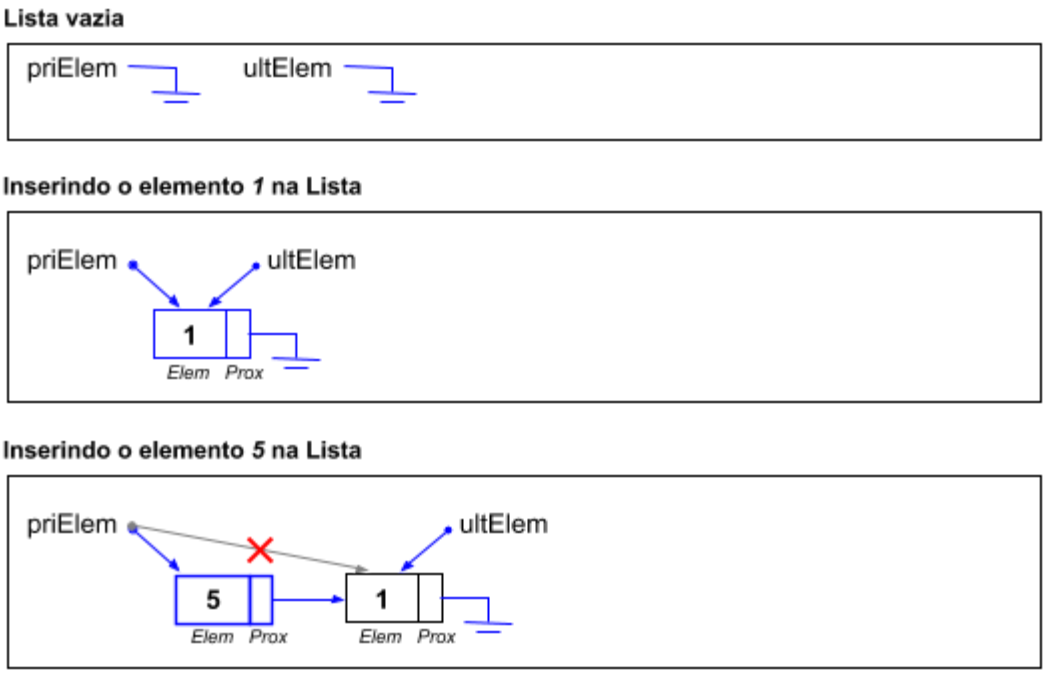
Operação de Inserção

Uma vez criada a lista vazia, pode-se inserir novos elementos nela. Para cada elemento inserido na lista, deve-se alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Deve-se notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
//Insere um novo elemento no início da lista
void insereNo(int elem, struct Lista *list){
     struct no *novoNo;
     novoNo = (struct no*) malloc (sizeof(struct no));
     novoNo->elem = elem;
     novoNo->prox = NULL;
     if (list->priElem == NULL){ // se a lista está vazia
        list->ultElem = novoNo;// configura o último elemento da lista
        }
     else novoNo->prox = list->priElem;
     // configurou o atributo Prox para o primeiro elemento
    list->priElem = novoNo;
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. Se a lista estiver vazia ele também configura esse novo nó como sendo o último da lista. Observe que não se pode deixar de atualizar a variável que representa o início da lista (`priElem`) a cada inserção de um novo elemento. A figura a seguir ilustra a operação de inserção de um novo elemento no início da lista.



Operação que percorre os elementos da lista

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, considere a criação de um método que imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.

```
//Imprime os valores dos elementos contidos na lista
void imprime(struct Lista *list){
   struct no *aux;  // variável auxiliar para percorrer a lista
   aux = list->priElem;
   while(aux != NULL) //enquanto não chegar no final da lista
   {
          printf("\n Elemento = %d", aux->elem);
          aux = aux->prox; //vai para o próximo
   }
}
```

Operação que verifica se lista está vazia

Pode ser útil implementar um método que verifique se uma lista está vazia ou não. A função retorna **`true`** se estiver vazia ou **`false`** caso contrário. Como se sabe, uma lista está vazia se o valor do atributo **`priElem`** é **`null`**. A implementação desse método é mostrada a seguir:

```
//Retorna true se vazia ou false se não vazia
short listaVazia (struct Lista *list){
   if (list->priElem == NULL)
      return 1;     //verdadeiro
   return 0;        //falso
}
```

Operação de busca

Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que se quer buscar e fornece como valor de retorno **a referência** (o ponteiro) do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é **`NULL`**.

```
//Busca um elemento na lista e retorna sua referência caso seja
 encontrado
struct no * buscaNo (int elem, struct Lista *list){
    struct no *p; // variável auxiliar para percorrer a lista

    for (p = list->priElem; p != NULL; p = p->prox)
      if (p->elem == elem)
         return p;

    return NULL; // caso não tenha achado o elemento
}
```

Operação para remoção de um elemento da lista

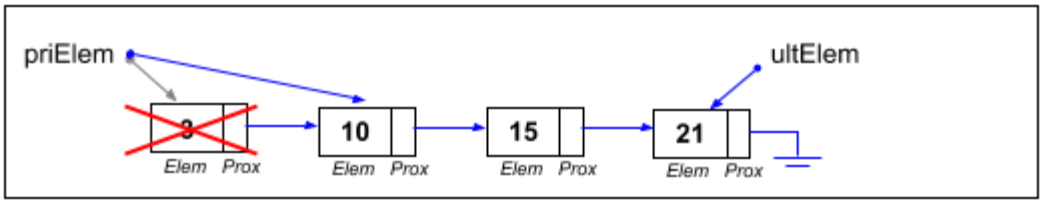
Para completar o conjunto de funções que manipulam uma lista, deve-se implementar uma função que permita remover um elemento. A função tem como parâmetros de entrada o valor do elemento que se quer remover. Se o elemento a ser removido for o primeiro da lista ou o último, esses atributos devem ser atualizados.
O método para retirar um elemento da lista é mais complexo. Se for descoberto que o elemento a ser retirado é o primeiro da lista, deve-se fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então se pode liberar o espaço alocado para o elemento que se quer retirar. Se o elemento a ser removido estiver no meio da lista, se deve fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então se pode liberar o elemento que se quer retirar. Deve-se notar que, no segundo caso, é preciso do

ponteiro para o elemento anterior para se poder arrumar o encadeamento da lista. As figuras a seguir ilustram as operações de remoção.

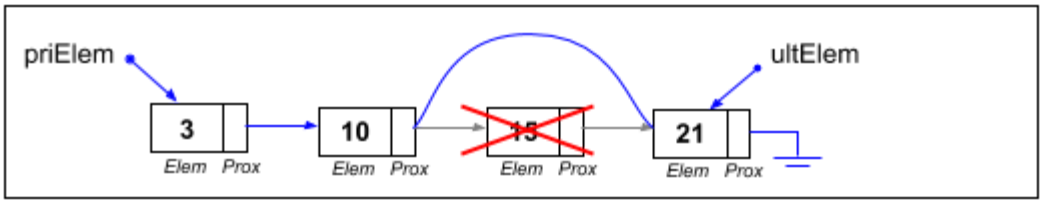
```
//Remove o elemento da lista. Se conseguir retorna true, caso
contrário, retorna false.
short removeNo(int elem, struct Lista *list){
      struct no *ant = NULL; // referência para elemento anterior
      struct no *p = list->priElem; // variável para percorrer a lista
      // procura elemento na lista, guardando o anterior
      while (p != NULL && p->elem != elem){
         ant = p;
         p = p->prox;
      }
      // verifica se achou elemento
      if (p == NULL)
        return 0; // não achou: retorna 0 (false)

      // retira o elemento
      if (ant == NULL) {
      // retira o primeiro elemento
         list->priElem = p->prox;
      }
      else if (p == list->ultElem){
         // retira o último elemento da lista
           list->ultElem = ant;
           list->ultElem->prox = NULL;
         }
      else {// retira elemento do meio da lista
           ant->prox = p->prox;
         }
      return 1; //achou: retorna 1 (true)
}
```

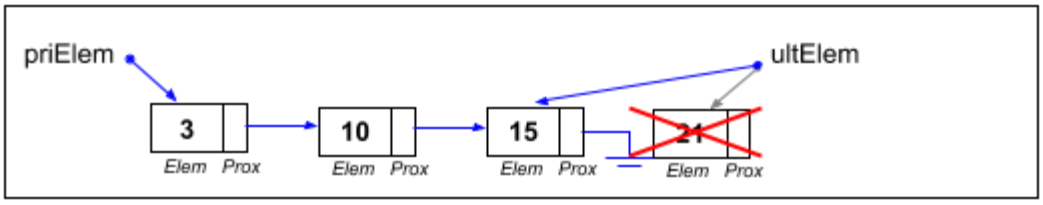
Remove o 1º elemento da Lista



Remove o 3º elemento da Lista (remoção no meio)



Remove o último elemento da Lista



Operação para acessar o primeiro e o último elemento da lista

Acessa o primeiro e o último elemento da lista, o que permite e também facilita o acesso e controle da mesma.

```
//Retorna o primeiro elemento (nó) da Lista
struct no getPriElem(struct Lista *list){
    return *(list->priElem);
}

//Retorna o último elemento (nó) da Lista
struct no getUltElem(struct Lista *list){
    return *(list->ultElem);
}
```

Ou, pode ser retornado um ponteiro para o primeiro elemento.

```
//Retorna o primeiro elemento (nó) da Lista
struct no * getPriElem(struct Lista *list){
    return (list->priElem);
}

//Retorna o último elemento (nó) da Lista
struct no * getUltElem(struct Lista *list){
    return (list->ultElem);
}
```

Operação para verificar o tamanho da lista (quantidade de elementos)

Há ainda a necessidade de se conhecer quantos elementos estão contidos na lista. Para isso é útil se implementar um método que retorne o tamanho atual da mesma.

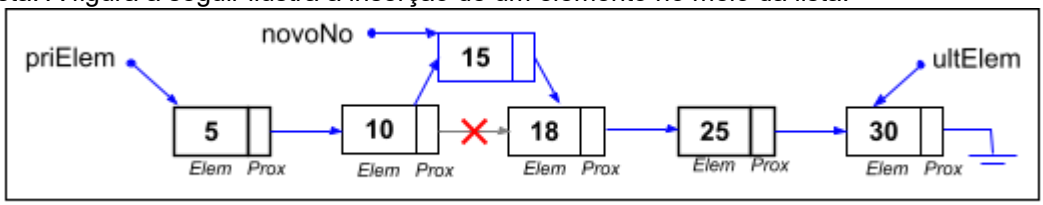
```
int tamanho(struct Lista *list){
    struct no *p = list->priElem;
    int tam=0;
    while (p!=NULL){
      tam++;
      p=p->prox;
    }
    return(tam);
}
```

Exercícios

1) Implemente as demais operações do **TAD Lista,** apresentadas anteriormente.

2) A função de inserção vista anteriormente armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista. Se for necessário manter os elementos na lista numa determinada ordem, tem-se que encontrar a posição correta para inserir o novo elemento. Essa função não é eficiente, pois se tem que percorrer a lista, elemento por elemento, para se achar a posição de inserção. Se a ordem de armazenamento dos elementos dentro da lista não for relevante, opta-se por fazer inserções no início, pois o custo computacional disso independe do número de elementos na lista.

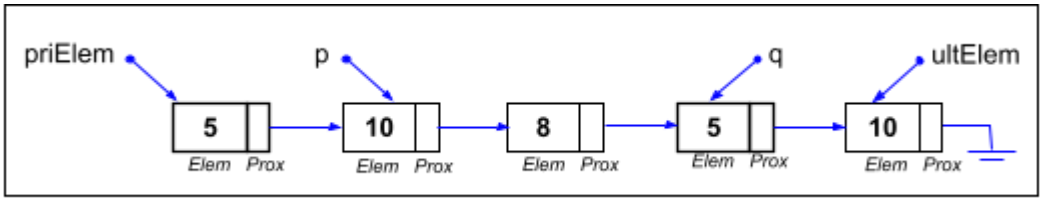
No entanto, se for desejado manter os elementos ordenados, cada novo elemento deve ser inserido na ordem correta. Para exemplificar, considere que se quer manter a lista de números inteiros em ordem crescente. A função de inserção, neste caso, tem a mesma assinatura da função de inserção mostrada, mas percorre os elementos da lista a fim de encontrar a posição correta para a inserção do novo. Com isto, tem-se que saber inserir um elemento no meio da lista. A figura a seguir ilustra a inserção de um elemento no meio da lista.



Faça a alteração necessária no método `void insereNo(int elem, struct Lista *list)` de forma que a inserção dos elementos (números inteiros) seja em ordem crescente.

3) Elaborar um TAD para uma Lista Encadeada Ordenada. Dada uma lista ordenada L1 encadeada alocada dinamicamente, escreva as operações:
- Verifica se L1 está ordenada ou não (a ordem pode ser crescente ou decrescente)
- Faça uma cópia da lista L1 em uma outra lista L2;
- Faça uma cópia da Lista L1 em L2, eliminando elementos repetidos;
- inverta L1 colocando o resultado em L2;
- inverta L1 colocando o resultado na própria L1;
- intercale L1 com a lista L2, gerando a lista L3, considerando L1, L2 e L3 ordenadas.

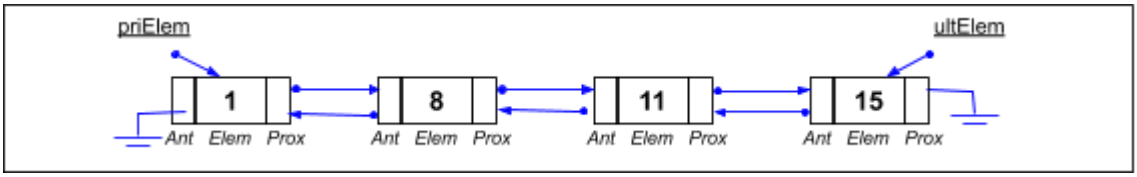
4) Explique o que acontece nas atribuições abaixo considerando a lista dada como exemplo (Dica: use desenhos)



a) p.setProx(q);
b) p.setProx(q.getProx());
c) p.setElem(q.getElem());
d) p = q;
e) p.setProx(null);
f) p = p.getProx();
g) p = p.getProx().getProx();

Lista duplamente encadeada

A estrutura de lista encadeada caracteriza-se por formar um encadeamento simples entre os elementos: cada elemento armazena uma referência para o próximo elemento da lista. Dessa forma, não se tem como percorrer eficientemente os elementos em ordem inversa, isto é, do final para o início da lista. O encadeamento simples também dificulta a retirada de um elemento da lista. Mesmo se tiver o ponteiro do elemento que se quer retirar, tem-se que percorrer a lista, elemento por elemento, para se encontrar o elemento anterior, pois, dado um determinado elemento, não há como acessar diretamente seu elemento anterior.
Para solucionar esses problemas, pode-se formar o que se chama de listas duplamente encadeadas. Nelas, cada elemento tem uma referência (“ponteiro”) para o próximo elemento e outra para o elemento anterior. Dessa forma, dado um elemento, pode-se acessar ambos os elementos adjacentes: o próximo e o anterior. Se tiver uma referência para o último elemento da lista, pode-se percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior, até alcançar o primeiro elemento da lista, que não tem elemento anterior (a referência do elemento anterior vale NULL).



Para exemplificar a implementação de listas duplamente encadeadas, considere o exemplo simples no qual se armazena valores inteiros na lista. O nó da lista pode ser representado pela classe abaixo e a lista pode ser representada por meio das referências para o primeiro e o último nó.

```
typedef struct NoD *NoDuplo;
struct NoD {
  int Elemento;
  NoDuplo Ant;
  NoDuplo Prox;
};
```

Ou:

```
struct NoD {
  int Elemento;
  struct NoD *Ant;
  struct NoD *Prox;
};
```

```
struct Lista {
  NoDuplo priElem;
  NoDuplo ultElem;
};
```

Ou:

```
struct Lista {
  struct NoD *priElem;
  struct NoD *ultElem;
};
```

O código abaixo realiza a inserção de um novo elemento no início da lista:

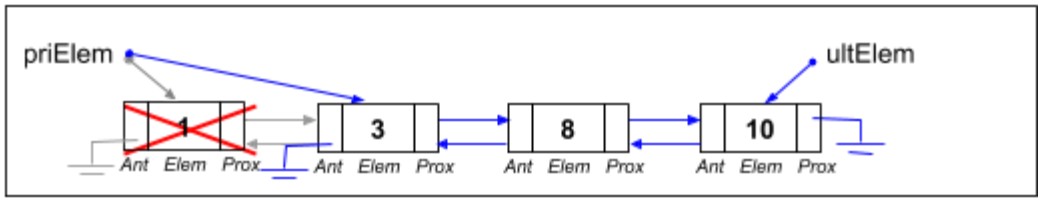
```
void insereNo(int elem, struct ListaDE *List) {
      struct NoD *novoNo;  // cria o novo elemento
      novoNo = (struct NoD*) malloc (sizeof(struct NoD));
      novoNo->Elemento = elem;
      novoNo->Prox = novoNo->Ant = NULL;
      if (List->priElem == NULL) { // se a lista está vazia
        List->ultElem = novoNo; //configura o último elemento da lista
      }
      else {      //configura o atributo Ant do priElem para
referenciar o novoNo, somente se o mesmo existir (lista não vazia)
        novoNo->Prox = List->priElem;
        List->priElem->Ant = novoNo;
      }
     List->priElem = novoNo; //o novo elemento passa a ser o primeiro
da lista
}
```

O seguinte código realiza a remoção de um elemento da lista:

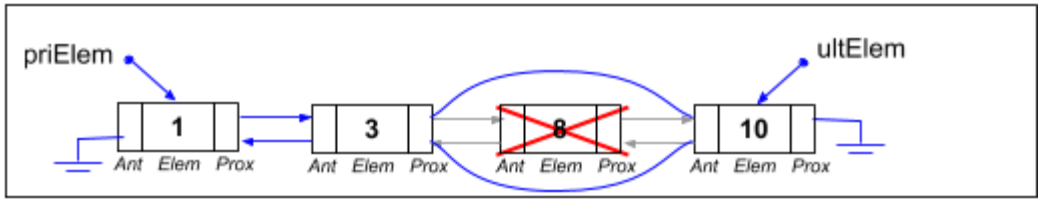
```
//Remove o elemento da lista. Se conseguir retorna true, caso
contrário, retorna false.
short removeNo(int elem, struct Lista *List){
     struct NoD *p = List->priElem; //variável para percorrer a lista
     //procura elemento na lista, guardando o anterior
     while (p != NULL && p->Elemento != elem){
        p = p->Prox;
     }
     if (p == NULL) //não achou o elemento: retorna 0/false
       return 0;
     //retirando o elemento encontrado
     if (p->Ant == NULL) { //retira o primeiro elemento
        List->priElem = p->Prox;
        List->priElem->Ant = NULL; //corrige a referência ao anterior
para null
     }
     else if (p == List->ultElem){ //retira o último elemento da lista
          List->ultElem = p->Ant;
          List->ultElem->Prox = NULL;
        }
     else {  //retira elemento do meio da lista
          p->Ant->Prox = p->Prox;
          p->Prox->Ant = p->Ant;
     }
     return 1;
   }
```

A seguir têm-se os exemplos para as diferentes possibilidades de remoção em uma lista encadeada.

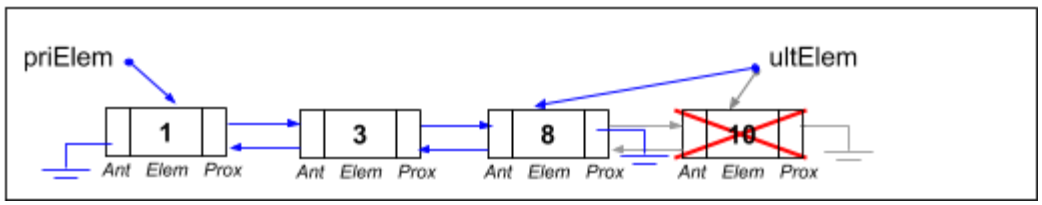
Remove o 1º elemento da Lista



Remove o 3º elemento da Lista (remoção no meio)



Remove o último elemento da Lista



O código a seguir realiza a impressão da lista:

```
//Imprime os valores dos elementos contidos na lista ao contrário
  void imprimeListaInverso(struct Lista *List){
    struct NoD *p;  // variável auxiliar para percorrer a lista

    for (p = List->ultElem; p != NULL; p = p->Ant)
      printf("\n Elemento = %d",p->Elemento);
  }
```

OBS: certifique-se se a lista foi inicializada antes de realizar as operações.