# Listas:

a última das 3 estruturas lineares (Pilhas, Filas e Listas)

... árvores e grafos são não lineares!

28/9/ , 30/9/ e 5/10/2010

Representação/Implementação:

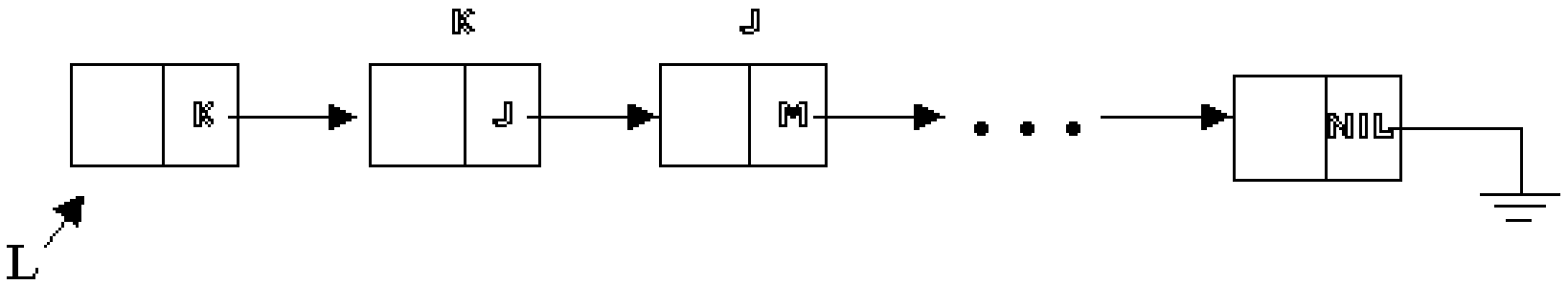
Encadeada dinâmica

Exercícios

# Lista Simplesmente Encadeada Dinâmica

**Visualização de uma lista encadeada**

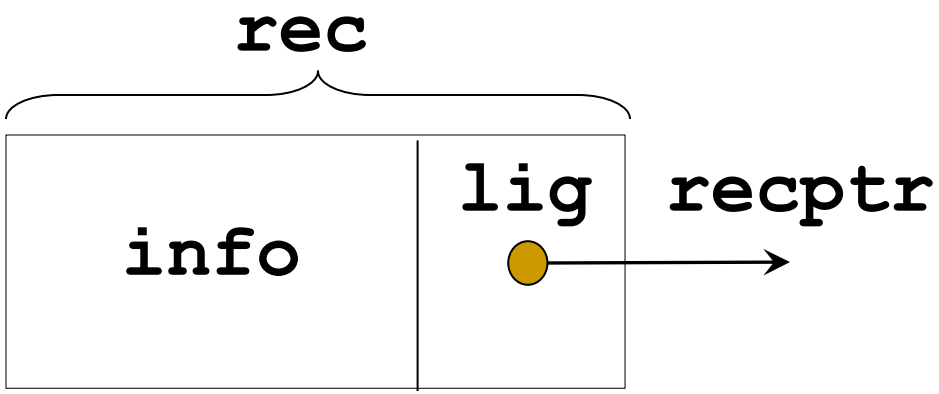
k, j e m são posições de memória não consecutivas e L é o ponteiro para o início da lista



# Lista Dinâmica

**/* declaração em Lista.c */**

```
 struct rec {
      elem info;
      struct rec *lig;
};
```




**/* declarações em Lista.h */**

```
typedef char elem;
typedef struct rec *recptr;
```

# Lista Dinâmica

- **Declaração**

```
recptr p;  {ponteiro p/ qqr elemento da lista}
recptr L;  {ponteiro p/ primeiro elemento da lista }
```

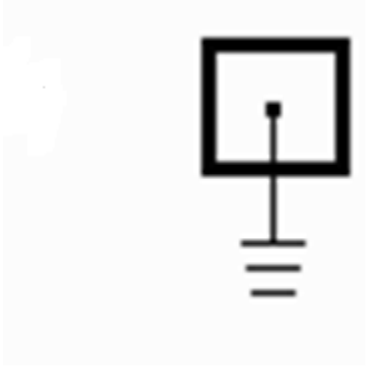
# Lista Dinâmica

Vamos exercitar Operações sobre Listas Dinâmicas sem pensar nos erros resultantes das operações. Faremos esta tarefa quando criarmos o TAD LISTA ORDENADA.

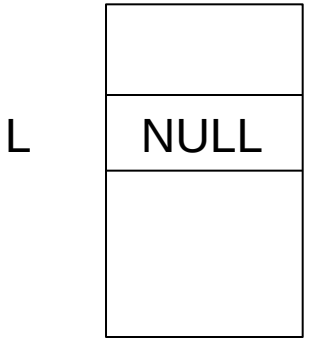
1. Criar lista vazia
2. Inserir primeiro elemento (`Insere_Prim`)
3. Inserir no início de uma lista (`Insere_Inicio`)
4. Acessar primeiro elemento
5. Contar o número de elementos (tamanho da lista)
6. Localizar registro de chave x em lista ordenada
7. Inserção do valor *v* depois do elemento apontado por k (`Insere_Depois`)
   Inserir registro com valor v de lista ordenada
8. Eliminar primeiro elemento (`Remove_Prim`)
9. Eliminar elemento apontado por j, que segue k (`Elimina_Depois`)
10. Eliminar registro com valor v de lista ordenada
11. Imprimir a lista

# Implementação das Operações

## 1) **Criação da lista vazia**



```
void Criar(recptr L){
L= NULL;
}
```



# Porque Criar não funciona?

A passagem de parâmetro por valor não modifica a **variável que aponta para o início da Lista.** Temos 2 opções para arrumá-la:

```
// Em Lista.h
typedef struct rec *recptr;

// Em Lista.c
// retornar a lista
// modificada
recptr Criar(void){
return NULL;
}
...
int main(void){
recptr L1 = Criar();
```

(a)

```
// Em Lista.h
typedef struct rec *recptr

// Em Lista.c
//usar ponteiro para
// Lista
void Criar(recptr *L){
*L = NULL;
}

...
int main(void){
recptr L1;
Criar(&L1);
```

(b)

## Temos que arrumar, todas as funções que podem alterar a variável Lista L – seguindo a opção (a)

```
recptr Criar(void);
recptr Insere_Prim(elem valor);
recptr Insere_Inicio(recptr L, elem valor);
void Insere_Depois(recptr k, elem v);
elem Primeiro(recptr L);
int Tamanho(recptr L);
int Tamanho_rec(recptr L);
recptr Busca_ord(recptr L, elem x);
```

# Continuação para a opção (a)

```
recptr Busca_rec(recptr L, elem x);
recptr Insere_ord(recptr L, elem x, int *ret);
recptr Insere_rec(recptr L, elem x, int *ret);
recptr Remove_Prim(recptr L);
void Elimina_Depois(recptr k);
recptr Remove_ord(recptr L, elem v, int *ret);
void Imprime(recptr L);
recptr Esvaziar(recptr L);
```

# Agora para a opção (b)

```
void Criar(recptr* L);
void Insere_Prim(recptr *L, elem valor);
void Insere_Inicio(recptr *L, elem valor);
void Insere_Depois(recptr k, elem v);
elem Primeiro(recptr L);
int Tamanho(recptr L);
int Tamanho_rec(recptr L);
recptr Busca_ord(recptr L, elem x);
```

# Continuação para a opção (b)

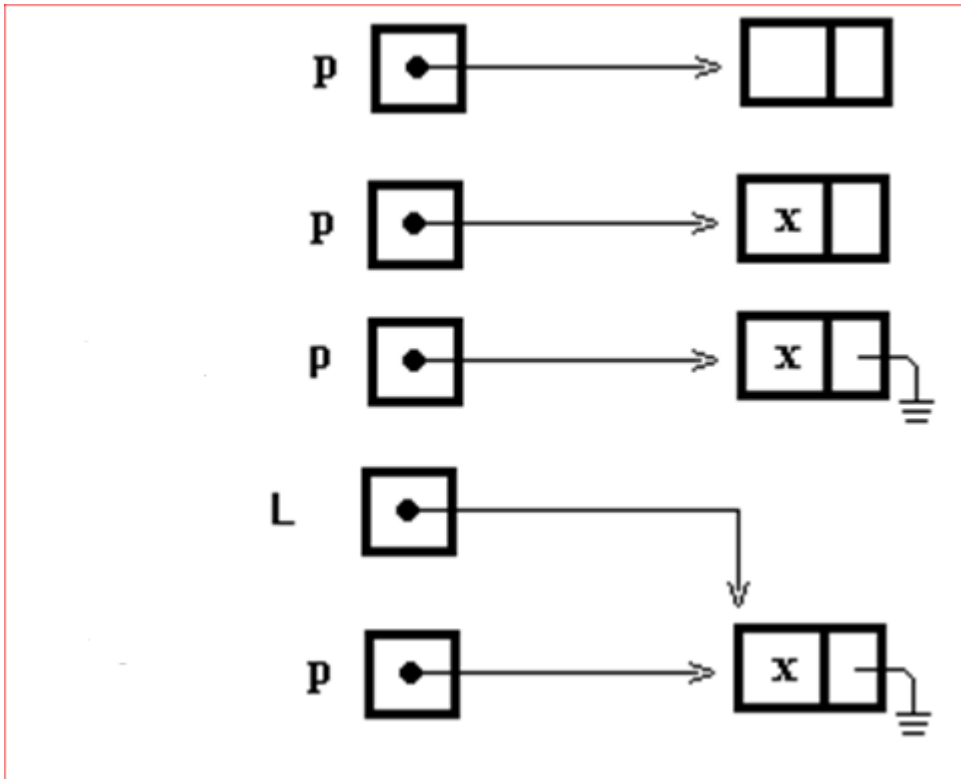
```
recptr Busca_rec(recptr L, elem x);
int Insere_ord(recptr *L, elem x);
int Insere_rec(recptr *L, elem x);
void Remove_Prim(recptr *L);
void Elimina_Depois(recptr k);
int Remove_ord(recptr *L, elem v);
void Imprime(recptr L);
void Esvaziar(recptr *L);
```

# Escolham uma das opções e alterem as funções a seguir com relação à passagem de parâmetro ou retorno

Na wiki da disciplina, temos dois TADs de Listas Ordenadas, seguindo as 2 opções acima.

# Implementação das Operações

2) **Inserção do primeiro elemento**

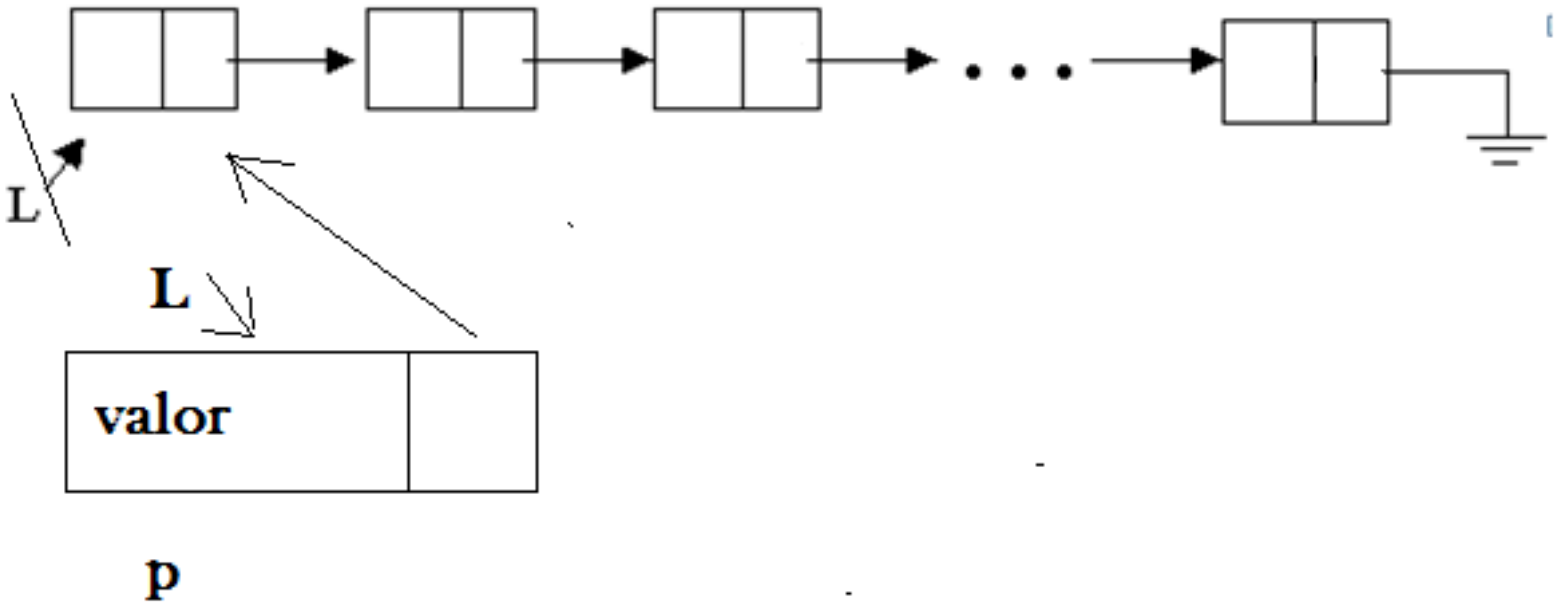


# Inserção do primeiro elemento

```
void Insere_Prim(recptr L, elem valor){
 recptr p =(recptr) malloc(sizeof(struct rec));
 p->info= valor;
 p->lig= NULL;
 L= p;
}
```

Deixem para relatar os erros de Memória Insuficiente quando estiverem fazendo o TAD, assim uniformizam todos os retornos de erros.

# 3) Inserção no início de uma lista



# Implementação das Operações

## 3) **Inserção no início de uma lista**

```
void Insere_Inicio(recptr L, elem valor){
recptr p =(recptr) malloc(sizeof(struct rec));
p->info= valor;
p->lig= L;
L= p;
}
```

Deixem para relatar os erros de Memória Insuficiente quando estiverem fazendo o TAD, assim uniformizam todos os retornos de erros.

# Implementação das Operações

4) **Acesso ao primeiro elemento da lista**

```
elem Primeiro(recptr L){
return (L->info);
}
```

Deixem para relatar os erros de Ponteiro Nulo quando estiverem fazendo o TAD, assim uniformizam todos os retornos de erros.

# Implementação das Operações

5) **Quantos elementos tem a lista ?**

```
int Tamanho(recptr L){
recptr p;
int aux;
p= L;   aux = 0;
        while (p != NULL){
              aux++; p= p->lig;
        }
return(aux);
}
```

# Implementação das Operações
## versão recursiva

```
int Tamanho_rec(recptr L){
if (L == NULL)
        return(0);
else
        return(1+ Tamanho_rec(L->lig));
}
```

# Implementação das Operações

## 6) Localizar chave x na lista ordenada

```
//Retorna (busca) o endereço de x numa Lista Ordenada.
   Se x não está, retorna NULL
recptr Busca_ord (recptr L, elem x){
recptr atual;

if (L==NULL) return(NULL)   // lista vazia, retorna NULL
else {
      atual= L;   //inicializa "atual" com o início de L
      while(atual != NULL) && (x > atual->info)){
            atual= atual->lig; //continua procurando

      };  //fim do while
```

```
//saiu do while porque achou x ou achou chave maior que x
   ou chegou no fim da lista

   if (atual != NULL) //então achou chave maior ou igual
              if (atual->info == x) return (atual);
                                    //retorna posição da chave
              else return (NULL);   //achou chave maior
   else return (NULL);  //não encontrou x; atual = NULL
}
}
```

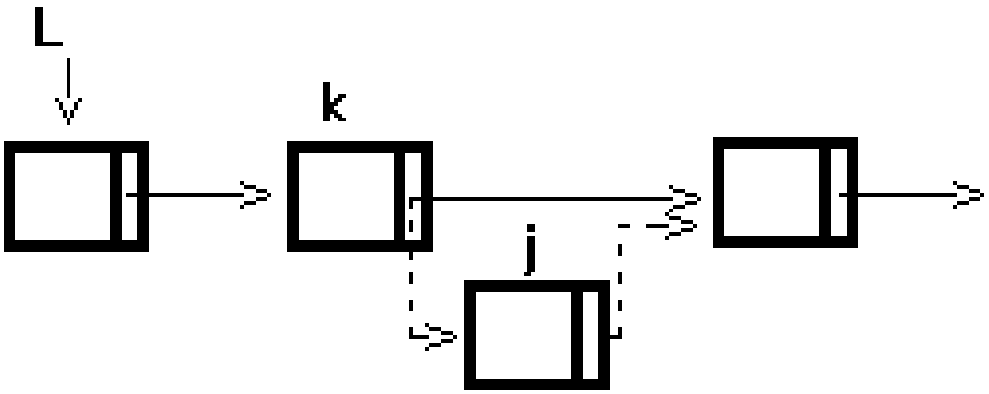
# Implementação das Operações

## 6) Localizar chave x na lista ordenada (Recursiva)

```
//Retorna endereço da chave, se encontrar, ou NULL,
   caso contrário
recptr Busca_rec(recptr L, elem x)
{
if (L==NULL) return (NULL);   // termina recursão
else if (x == L->info) return (L);     // achou
     else if (x > L->info) return(Busca_rec(L->lig, x)); // continua
          else return (NULL); // achou chave maior; retorna NULL
}
```

# Implementação das Operações

7) **Inserção de elemento (valor *v*) como sucessor do elemento no endereço k**



# Inserção do valor *v* depois do elemento apontado por k

```
void Insere_Depois(recptr k, elem v) {
recptr j;

j=(recptr) malloc(sizeof(struct rec));
j->info = v;
j->lig = k->lig;
k->lig = j;
}
```

**Obs**.: funciona para inserção após último elemento?

Deixem para relatar os erros de Ponteiro Nulo quando estiverem fazendo o TAD, assim uniformizam todos os retornos de erros.

# Inserção em Lista Ordenada

- Exercício: identificar os possíveis casos para a inserção
- Quebrar o processo de inserção em rotinas menores (casos)

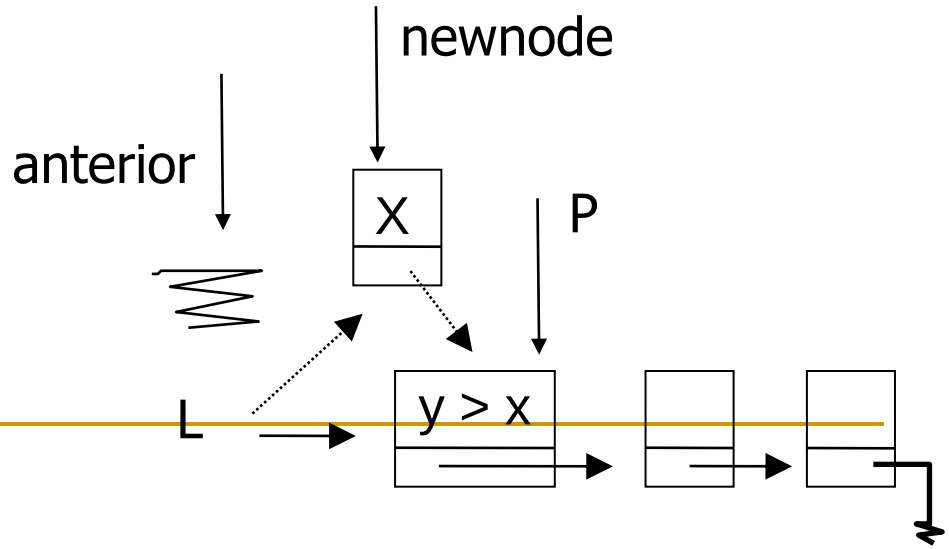
# Inserção – caso 1

- insere(X,L)
  - caso 1: X < 1º da lista

  - Se X < 1º da lista

    L ← newnode

    next(newnode) ← P



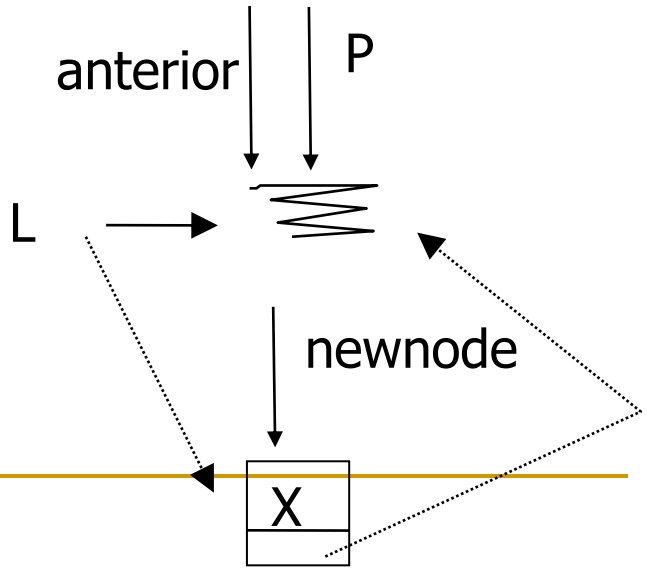
# Inserção – caso 2

- insere(X,L)
  - caso 2: a lista é vazia

  - Se a lista é vazia

    L ← newnode

    next(newnode) ← P



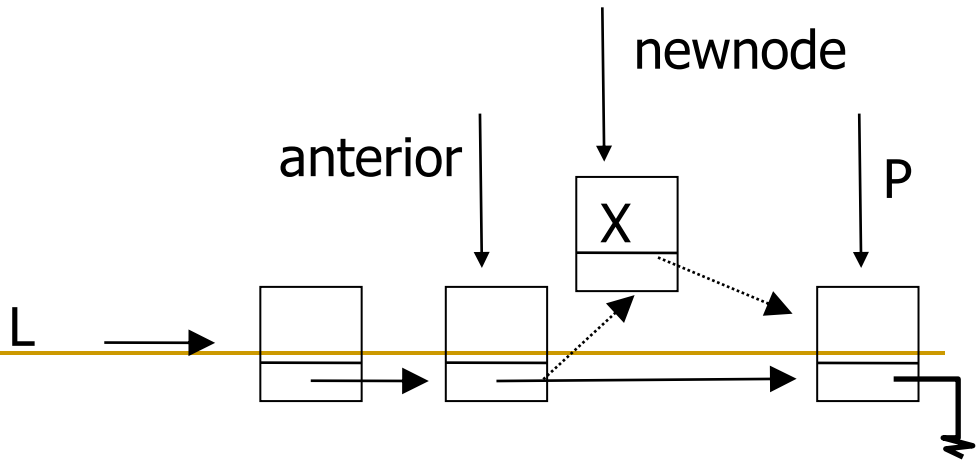
# Inserção – caso 3

- insere(X,L)
  - caso 3: X > 1º da lista

  - Se X não está na lista

    next(anterior) ← newnode

    next(newnode) ← P



# Implementação: Inserção do valor *v* na lista ordenada L

```
/*Insere item de forma a manter a lista ordenada, sem
   duplicados. Retorna true se inseriu; false, se não foi
   possível inserir*/

int Insere_ord(recptr L, elem x){
recptr pa, p;


// Busca ponto de inserção
     pa= NULL; p= L;
     While ((p != NULL) && (p->info < x)){
               pa= p; p= p->lig;
     }


/*Fim do while: p == NULL ou p->info == x ou p->info > x */
```

*(continua....)*

***OBS: Quando não dá para inserir???***

# Inserção na lista ordenada

```
if (p != null)
   if (p->info == x)
      return 0; //x já está na lista; retorna false
   else //achou maior; inserir entre pa e p
       if (pa != null){ //insere no meio
            Insere_Depois (pa, x); return 1;
       }
        else {//insere no início
            Insere_Inicio(L, x); return 1;
        }
 else //p=null; percorreu tudo ou lista estava vazia
   if (pa != null) {//percorreu tudo e insere no final
      Insere_Depois (pa, x); return 1;
    }
     else {// a lista estava vazia!
        Insere_Prim(L, x); return 1;
      }
}
```

# Inserção na lista ordenada - Recursivo

```
int Insere_rec(recptr L, elem x){
recptr p;
 if (L == NULL){ //condição de parada; inserir primeiro elemento
    Insere_Prim(L,x); return 1;
 }
  else //comparar com o primeiro elemento da lista
    if (x < L->info){   // insere no início
     Insere_Inicio(L,x); return 1;
    }
    else
      if (x == L->info) //não inserir novamente
        return 0;
      else //insere na lista depois do primeiro elemento
        return Insere_rec(L->lig, x);
}
```

# Implementação das Operações

8) **Remoção do primeiro elemento**

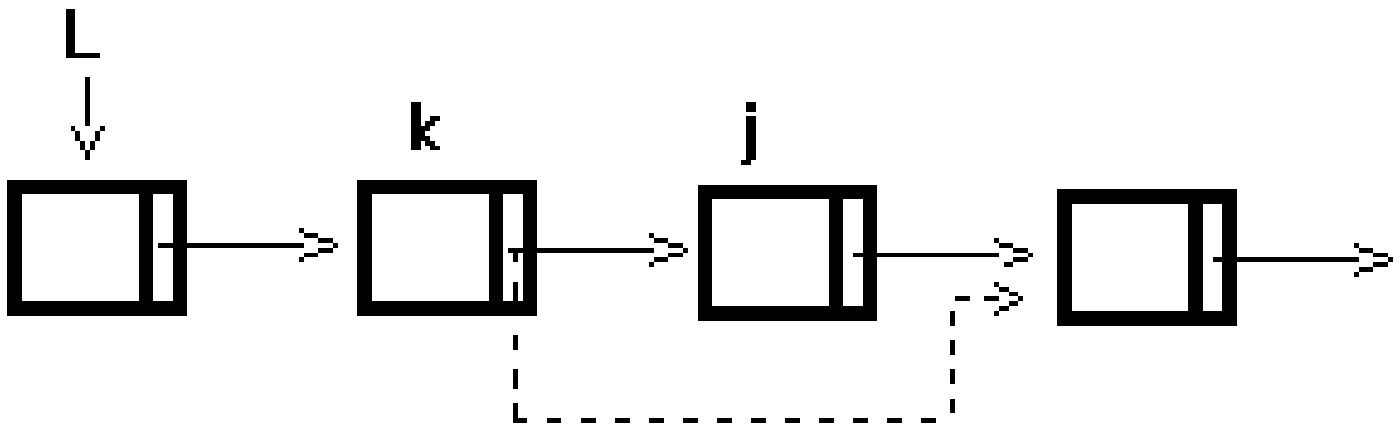
```
void Remove_Prim(recptr L){
recptr p;
p= L;
L= L->lig;
free(p);
}
```

Deixem para relatar os erros de Ponteiro Nulo quando estiverem fazendo o TAD, assim uniformizam todos os retornos de erros.

**Obs:** funciona no caso de remoção em lista com um único elemento?

# Implementação das Operações

9) **Eliminar elemento apontado por j, sucessor do elemento no endereço k**



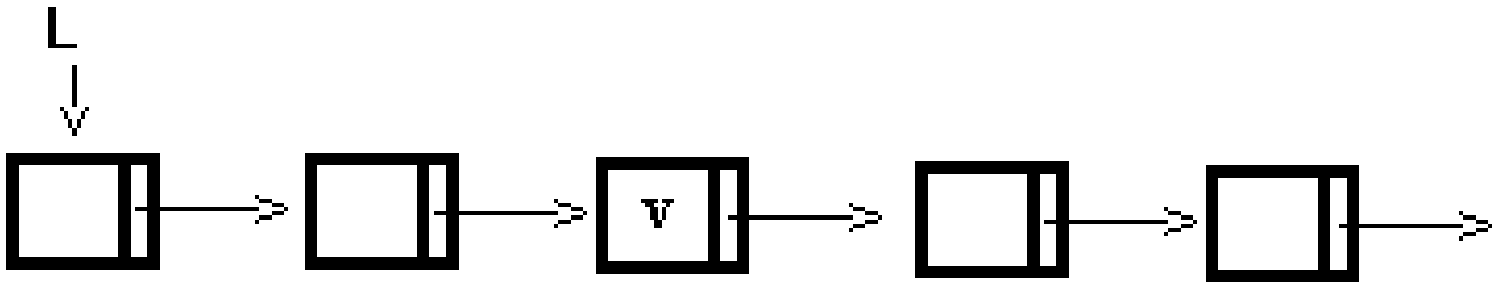
# Remoção do elemento apontado por j, que segue k

```
void Elimina_Depois(recptr k){
recptr j;
j= k->lig;
k->lig= j->lig;
free(j);
}
```

Deixem para relatar os erros de Ponteiro Nulo quando estiverem fazendo o TAD, assim uniformizam todos os retornos de erros.

# Implementação das Operações

10) **Eliminar valor *v* de uma lista ordenada L**

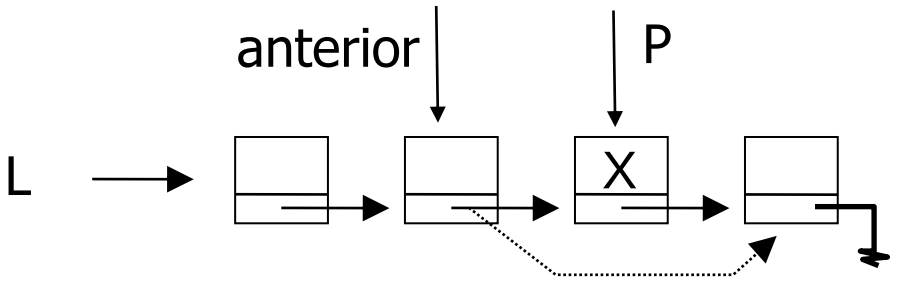


# Remoção em Lista Ordenada

- Operação de remoção de um elemento X
  - 4 casos
    - X é maior do que o primeiro elemento da lista
      - X está na lista
      - X não está na lista
    - X é igual ao primeiro elemento da lista
    - X é menor do que o primeiro elemento da lista
      - X não está na lista
    - Lista vazia

# Remoção: caso 1

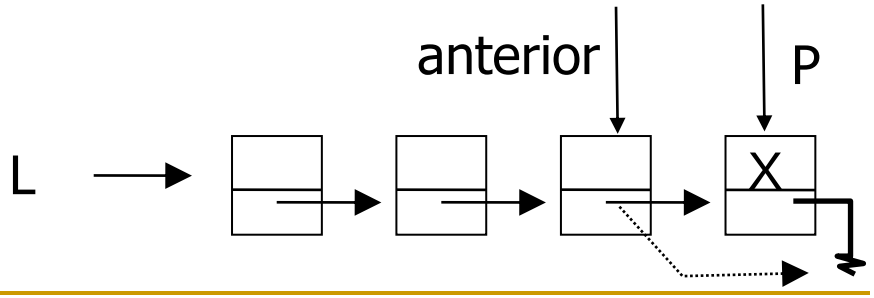
- remove(X,L,Achou)
  - X > 1º da lista




  - Se X está na lista

    next(anterior) ← next(P)

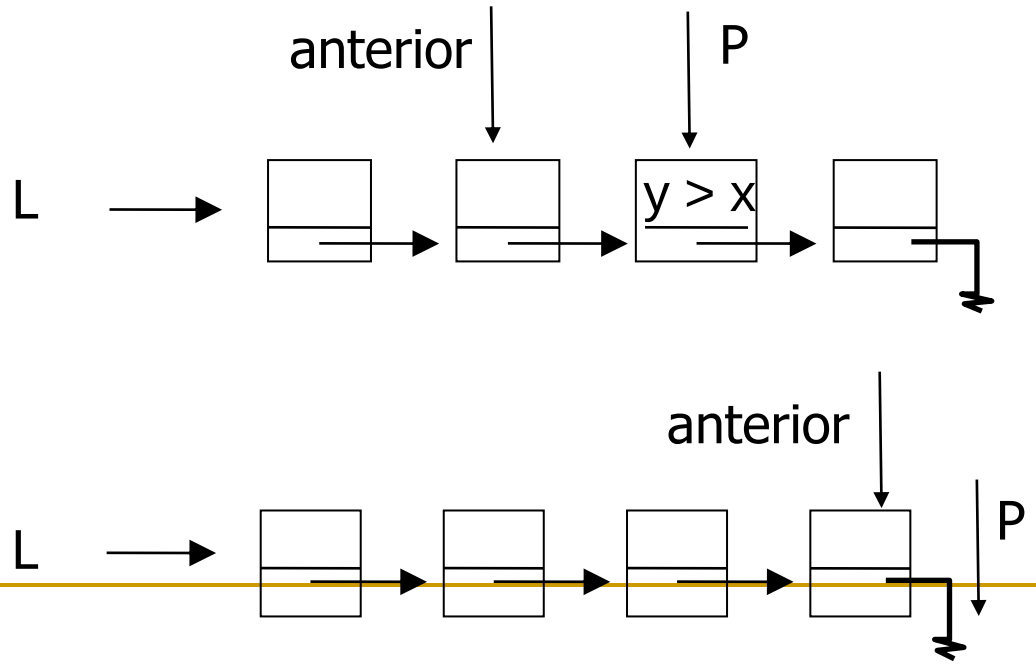
    freenode(P)



# Remoção: caso 1

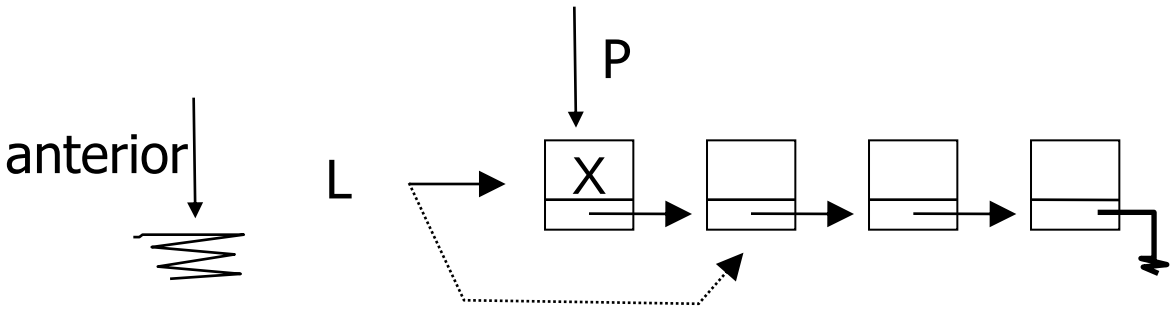
- remove(X,L,Achou)
  - Se X não está na lista

    Achou ← false (não remove)



# Remoção: caso 2

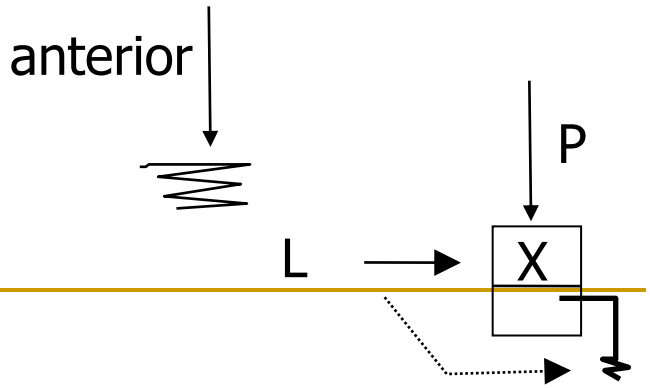
- remove(X,L,Achou)
  - X = 1º da lista




  - Se X é o 1º da lista

    L ← next(L)

    freenode(P)

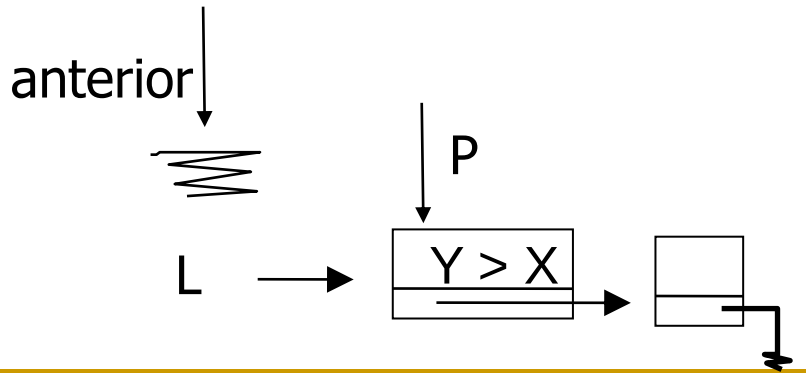


# Remoção: caso 3

- remove(X,L,Achou)
  - X < 1º da lista

  - Se X < 1º da lista

    Achou ← false

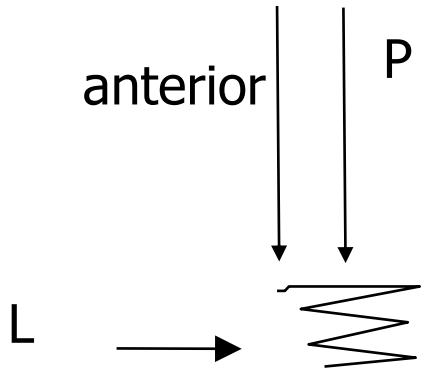


# Remoção: caso 4

- remove(X,L,Achou)
  - caso 4: a lista é vazia

  - Se a lista é vazia

    Achou ← false



# Remoção

- Algoritmo
  - anterior começa em NULL, P começa em L
  - avança anterior e P até que P chegue a X, a Y > X, ou ainda ao final da lista (anterior corre atrás de P)
  - com base na posição de anterior e P, identificar o caso e executar as operações necessárias

# Eliminar um valor v de uma lista ordenada L

```
/*Remove item de uma lista ordenada, sem duplicados.
   Retorna true se removeu; false, se não foi possível
   remover*/

int Remove_ord(recptr L, elem v){
 recptr p, pa;

// achar ponto de remoção
     pa= NULL; p= L;
     While ((p!= NULL) && (p->info < v)){
           pa= p; p= p->lig;
     }

/*Fim while: p == NULL ou p->info = v ou p->info>v */
```

*(continua....)*

# Eliminar valor *v* da lista L – cont.

```
    if (p == NULL)
      return 0; // lista não contém v; retorna false
    else
      if (p->info > v)
       return 0; // lista não contém v; retorna false
      else
         if (pa == NULL) {
            Remove_prim(L);return 1;
         }
         else {
            Elimina_depois(pa);return 1;
         }
}
```

# Implementação das Operações

## 11) **Impressão de uma lista**

```
void Imprime(recptr L){
   recptr p;
   p= L;
   while (p != NULL){
        imp_elem(p->info);
        p= p->lig;
    }
}
```

OBS: O tipo elem deve estar declarado no arquivo elemento.h numa implementação em C, junto com funções específicas de manipulação de tipos básicos:

igual, menor_que, imp_elem

# 11) **Impressão de uma lista (rec)**

```
/* Função imprime recursiva */
void imprime_rec (recptr L){
if (L == NULL)
return;

imp_elem(L->info);

imprime_rec(L->lig);
}
```

Como imprimir em ordem INVERSA???
Trocando a ordem de imp_elem e imprime_rec

# Funções no arquivo elemento.h

/* Compara se os dois elementos p e q são iguais. Deve ser implementada pelo usuário. */

int igual(elem p, elem q)

{ Compara se elemento p é menor do que elemento q. Deve ser implementada pelo usuário. }

int menor_que (elem p, elem q)

{ Imprime um elemento da lista. Deve ser implementada pelo usuário. Esta função deve ser usada dentro da função que imprime toda a lista}

void imp_elem (elem p);

# Implementação das Operações

- Elaborar os seguintes TADs, usando alocação dinâmica as operações que não foram citadas aqui – estão em vermelho
  - Lista Encadeada Ordenada
  - Lista Encadeada Não-ordenada

- Crie um arquivo header para alocar o tipo de elemento

- Verifique as pré-condições de cada operação para devolver os tipos de erros
  - Não imprima mensagens dentro das rotinas

- REAPROVEITEM código, reusando operações feitas no TAD, como feito na INSERÇÃO e REMOÇÃO

- **Criar(L):** criar uma lista vazia.
- **Esvaziar(L):** tornar uma lista L já existente vazia.
- **Destruir(L):** destruir uma lista L.

- **Acessar(L,i, x):** consulta o i-ésimo elemento de L.
- **Inserir(L,i,x):** Insere o elemento x depois do i-ésimo elemento da lista; altera os índices dos próximos elementos.
- **Remover(L,i,x):** Elimina o i-ésimo elemento da lista; altera os índices dos próximos elementos.

- **Insere_Ord(L, x):** Retorna true se inseriu; false, se não foi possível inserir
- **Remove_Ord(L, x):** Retorna true se removeu; false, se não foi possível remover
- **Busca_ord(L, x):** Retorna o endereço de x numa Lista Ordenada. Se x não está, retorna NULL

- **Tamanho(L):** retorna o tamanho da lista L.
- **Lista_vazia(L)** e **Lista_cheia(L):** checa se L está vazia e checa se L está cheia.
- **Primeiro(L)** e **Fim(L):** retornam o primeiro e a posição após o último elemento da lista L.
- **Localizar(L,x):** pesquisa a ocorrência de um item com um dado valor; retorna a posição de x na lista ou NULL se não achou

# Exercícios

- 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 (L1, L2 e L3 ordenadas)
  - Ordenar em ordem asc ou desc
  - Faça a união/concatenação de duas listas L1 e L2, resultando em uma lista L3

- Para estas operações seria interessante tê-las num TAD Lista?
  - Diferente de Pilhas e Filas, operações de ordenação, intercalação, união são bastante usuais e assim poderiam ser disponibilizadas no TAD Lista

# Exercícios

- Escreva um programa que gera uma lista L2, a partir de uma lista L1 dada, em que cada registro de L2 contém dois campos de informação
  - *elem* contém um elemento de L1, e *count* contém o número de ocorrências deste elemento em L1
- Escreva um programa que elimine de uma lista L dada todas as ocorrências de um determinado elemento (L ordenada)
- Assumindo que os elementos de uma lista L são inteiros positivos,
  - escreva um programa que informe os elementos que ocorrem mais e menos em L (forneça os elementos e o número de ocorrências correspondente)