Tipo Abstrato de Dados: Listas

Rafael Viana de Carvalho

Tipo Abstrato de dados: Listas

- Estrutura de dados que preserva a relação de ordem linear entre os dados, porém com operações mais dinâmicas.
  - Uma lista é composta por nós que podem conter dados primitivos ou compostos e que podem ser inseridos/removidos independetemente de sua posição
- Definição
  - Listas são estruturas flexíveis que admitem as operações de **inserção, remoção e recuperação de itens**
  - É uma sequencia de zero ou mais itens $x_1, x_2, \ldots, x_n$ na qual x_i é de um determinado tipo e n representa o tamanho da lista
  - Sua principal propriedade estrutural diz respeito as posições relativas dos itens
    - Se n ≥ 1, x_1 é o primeiro item e x_n é o último
    - Em geral, x_i precede x_{i+1} para $i = 1,2,\ldots,n-1$ e x_i sucede x_{i+1} para $i = 2,3,\ldots,n$

Tipo Abstrato de dados: Listas

- Diversos tipos de aplicações requerem uma lista
  - Lista telefônica
  - Lista de tarefas
  - Gerência de memória
  - Simulação
  - Compiladores, etc.
- Operações
  - Criar lista linear vazia
  - Inserir um novo item imediatamente após o último
  - Retirar o último item
  - Localizar um item x para examiná-lo ou alterá-lo
  - Ordenar os itens da lista em ordem ascendente ou descendente
  - Pesquisar a ocorrência de um item com um valor

Tipo Abstrato de dados: Listas

- Listas sequenciais (lineares)
  - Explora a sequencialidade da memória do computador
  - Podem ser representadas por um vetor (array)
- Listas encadeadas
  - Sequência de elementos encadeadas por ponteiros
  - Elementos devem conter os dados e uma referência para o proximo elemento (nó) da lista

Listas Sequenciais (Lineares)

- Uma das formas mais simples de interligar os elementos de um conjunto.
  - Os itens da lista são armazenados em posições contíguas de memória
- Estrutura em que as operações inserir, retirar e localizar são definidas.
- 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.
  - A lista pode ser percorrida em qualquer direção
  - A inserção de um novo item pode ser realizada após o último item com custo constante

Listas Lineares

- Duas listas podem ser concatenadas para formar uma lista única, ou uma pode ser partida em duas ou mais listas.
- Adequadas quando não é possível prever a demanda por memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível.
- São úteis em aplicações tais como manipulação simbólica, gerência de memória, simulações e compiladores.

Listas sequenciais

- Operações associadas:
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Elementos podem ser inseridos/removidos independente da posição

Começo da Lista

Listas sequenciais

- Operações associadas:
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Elementos podem ser inseridos/removidos independente da posição

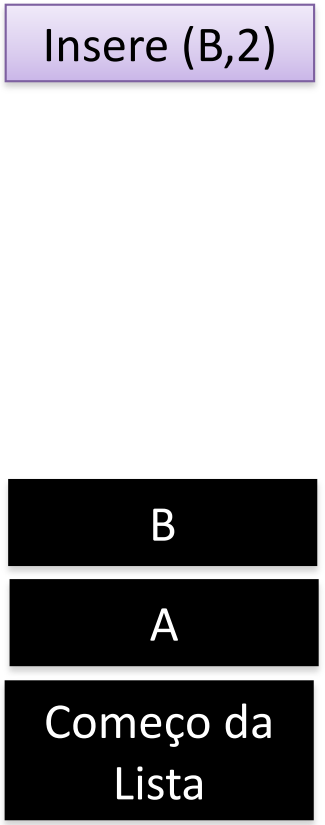
Insere (A,1)

A

Começo da Lista

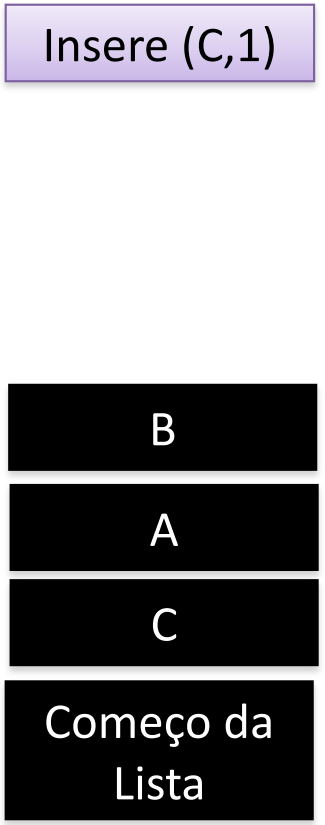
Listas sequenciais

- Operações associadas:
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Elementos podem ser inseridos/removidos independente da posição



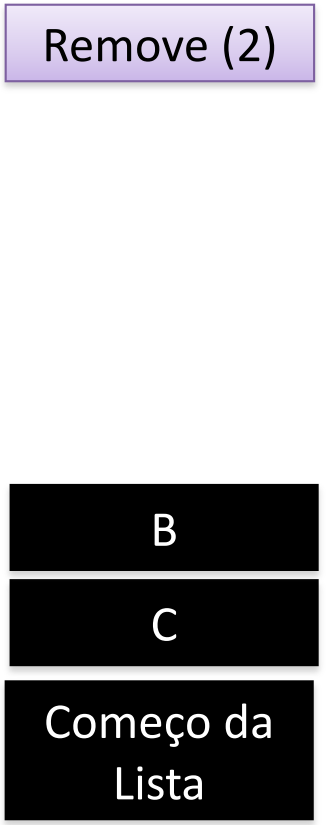
Listas sequenciais

- Operações associadas:
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Elementos podem ser inseridos/removidos independente da posição



Listas sequenciais

- Operações associadas:
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Elementos podem ser inseridos/removidos independente da posição

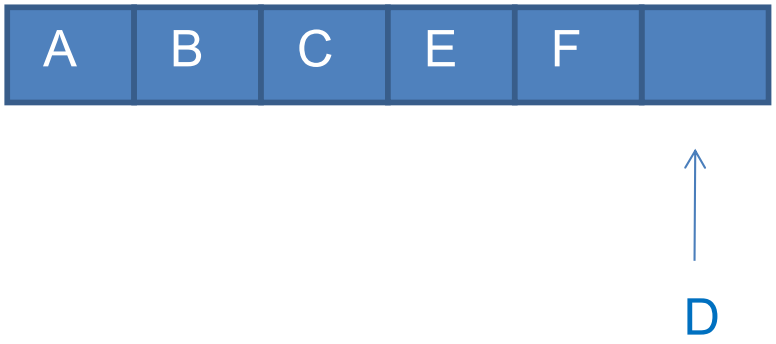


Listas sequenciais

- Operações associadas:
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Elementos podem ser inseridos/removidos independente da posição

Inserção nas Listas Lineares

- Exemplo
  - Inserindo no final da lista

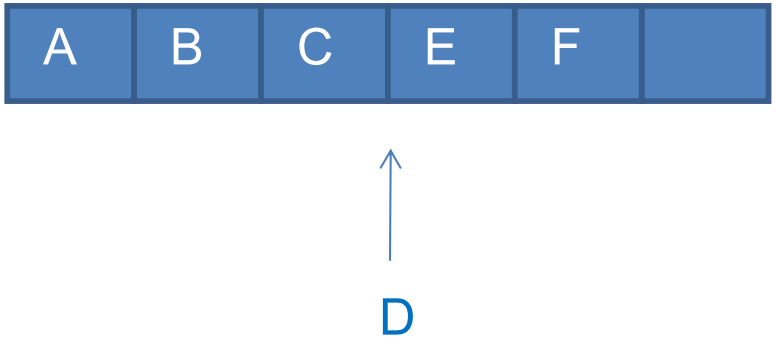


Inserção nas Listas Lineares

- Exemplo
  - Inserindo no final da lista

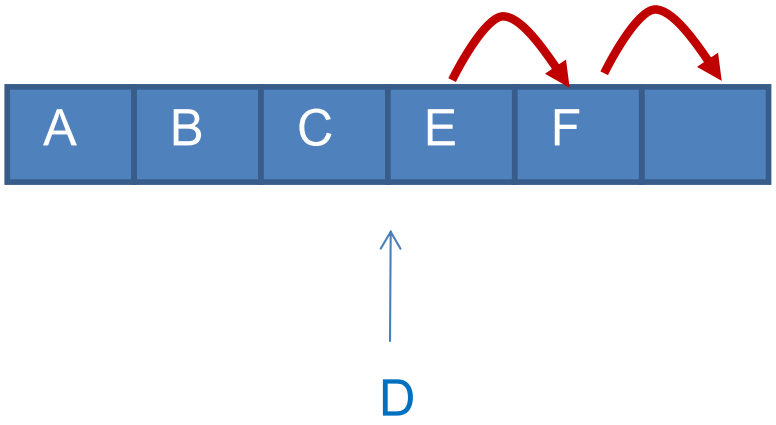
Inserção nas Listas Lineares

- Exemplo
  - Inserção no meio da lista



Inserção nas Listas Lineares

- Exemplo
  - Inserção no meio da lista

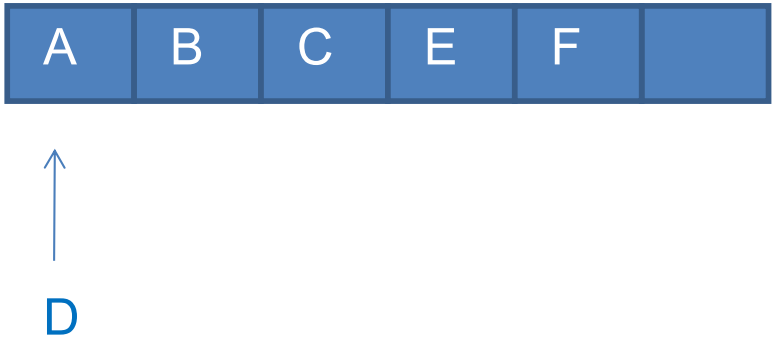


Inserção nas Listas Lineares

- Exemplo
  - Inserção no meio da lista

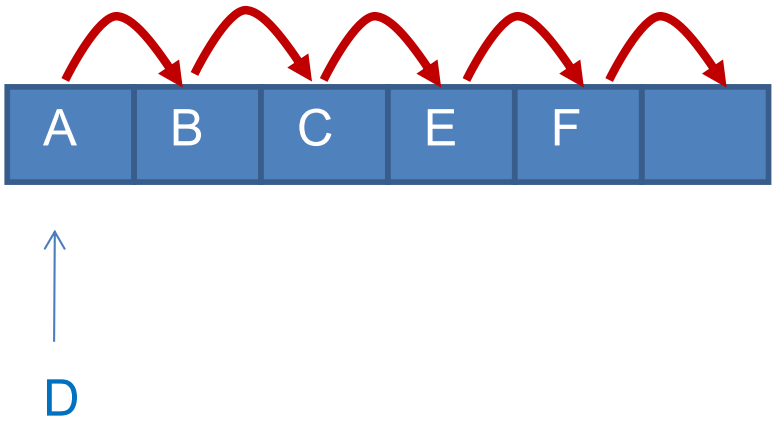
Inserção nas Listas Lineares

- Exemplo
  - Inserção no inicio da lista



Inserção nas Listas Lineares

- Exemplo
  - Inserção no inicio da lista



Inserção nas Listas Lineares

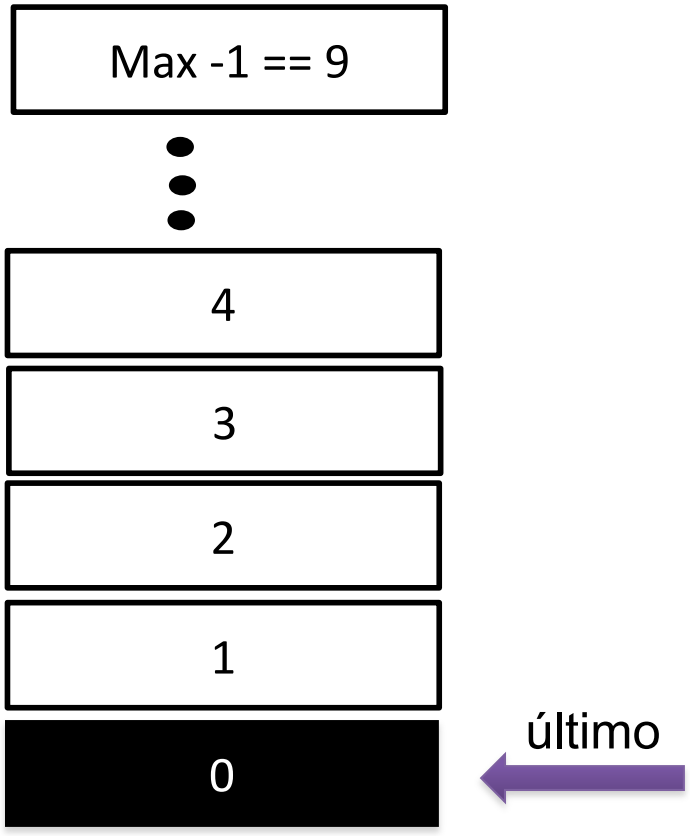
- Exemplo
  - Inserção no inicio da lista

Implementando uma lista sequencial

- Itens são armazenados em posições contínuas do array
- Guarda-se o índice do último elemento inserido

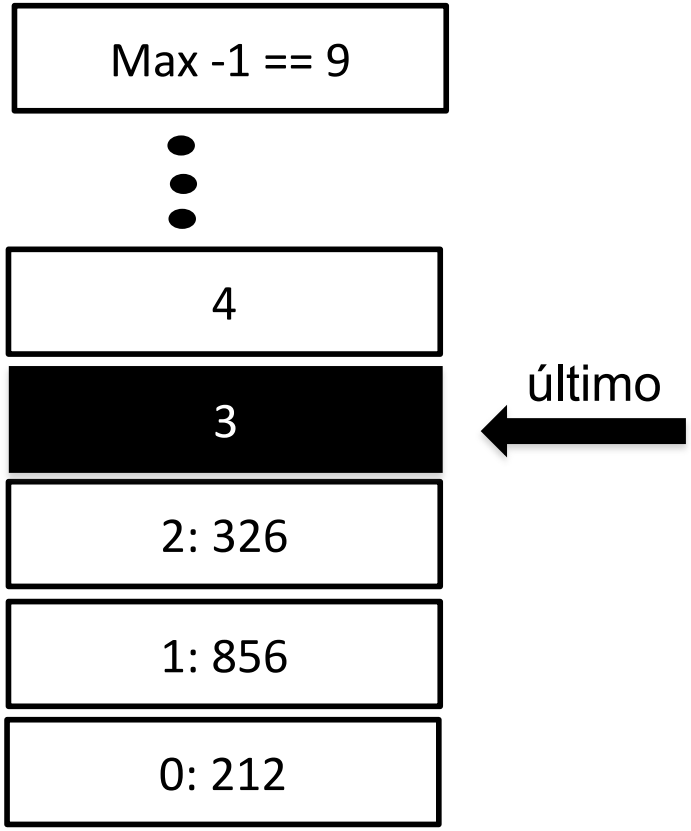
```
#define MAX = 10

struct lista {
        int elementos[MAX];
        int ultimo;
}
```

Criando uma lista sequencial



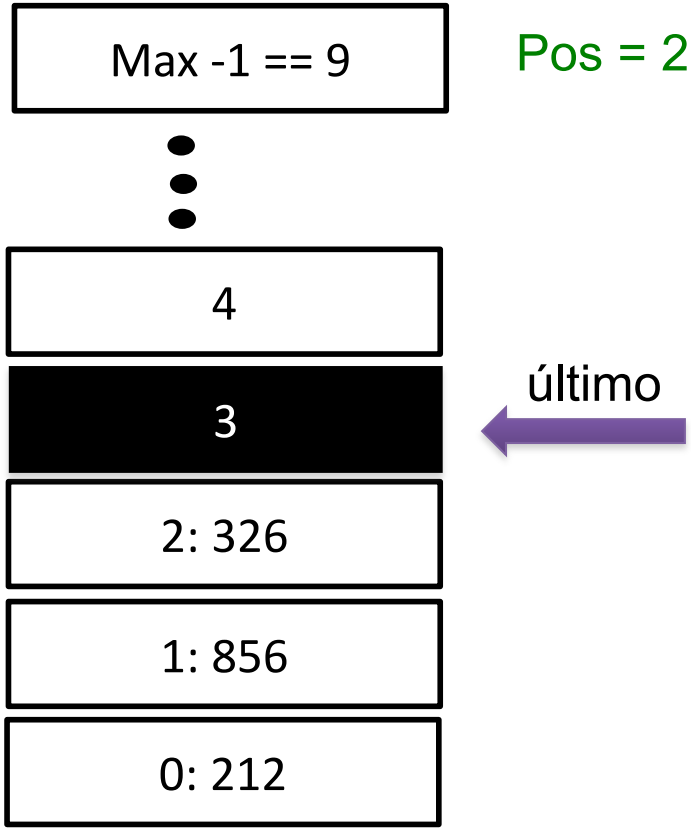

```
struct lista {
        int elementos[MAX];
        int ultimo;
}
struct lista * cria(void) {
        struct lista *ls;
        ls = malloc(sizeof(struct lista));
        if(!ls) {
                perror(NULL);
                exit(1);
        }
        /* IMPORTANTE: */
        ls->ultimo = 0;
}
```

Inserindo um elemento na posição X



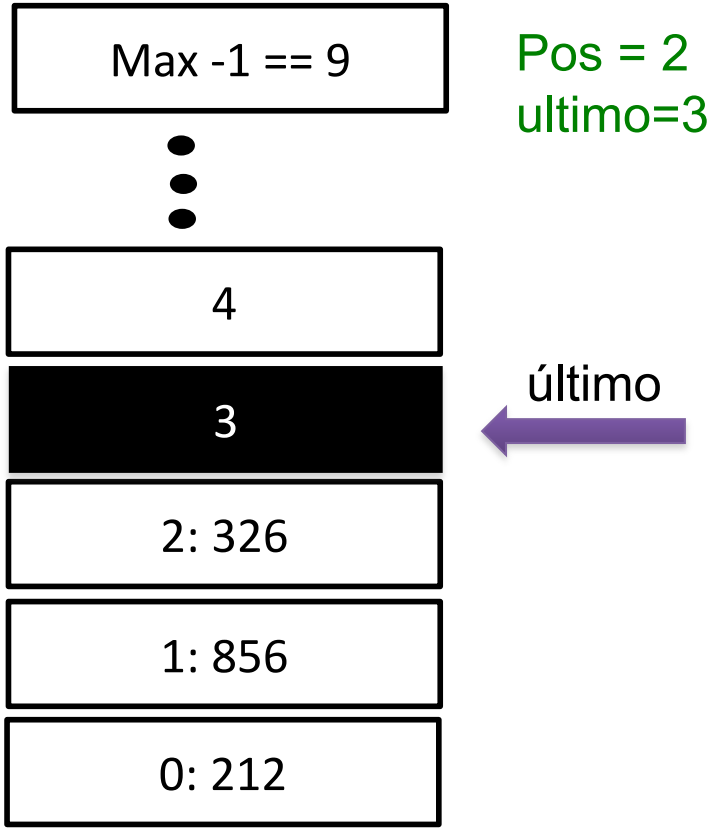

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
...
}
```

Inserindo um elemento na posição X



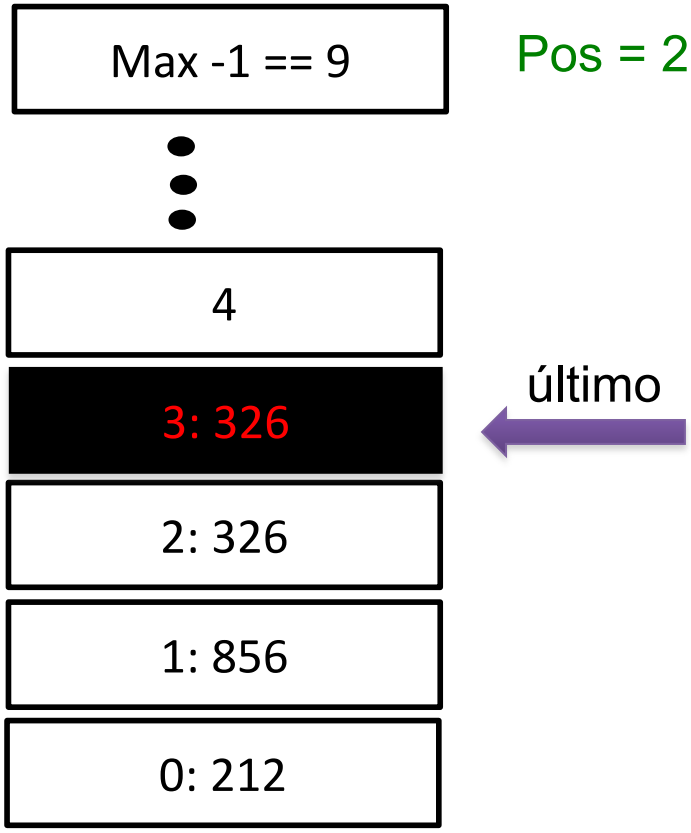

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
...
}
```

Inserindo um elemento na posição X



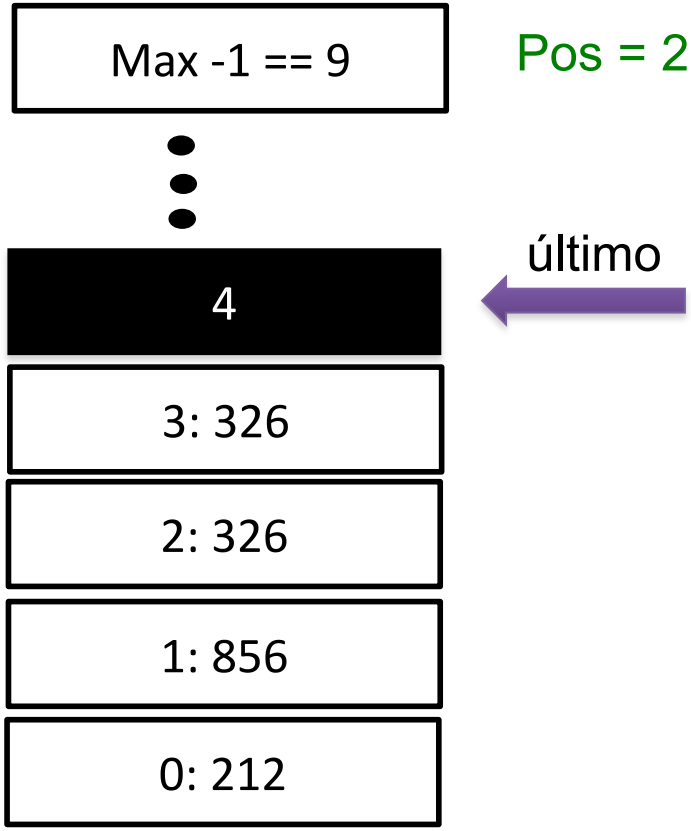

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
...
}
```

Inserindo um elemento na posição X



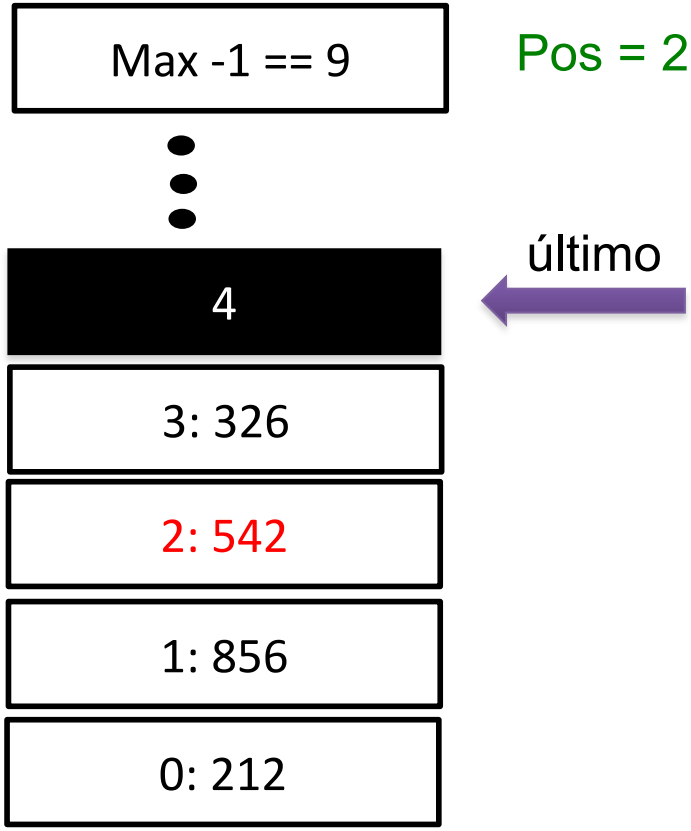

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
...
}
```

Inserindo um elemento na posição X



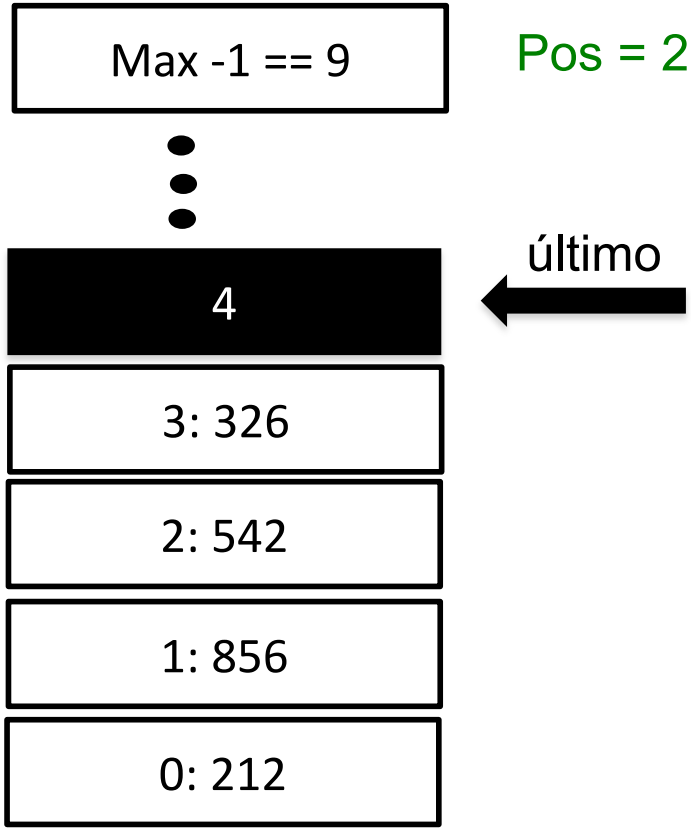

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
 ..
}
```

Inserindo um elemento na posição X



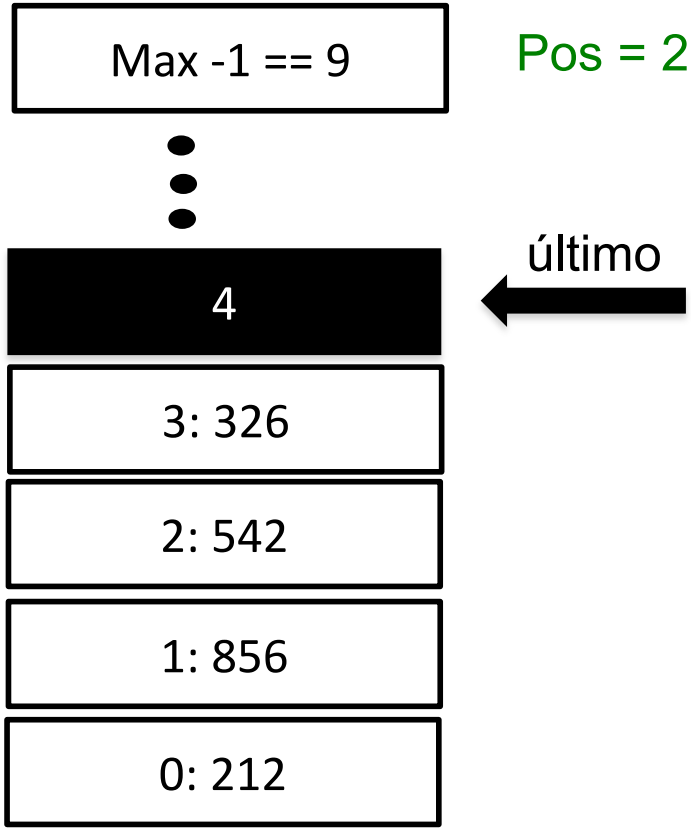

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;

void main () {
...
        insere (ls, 542, 2);
...
}
```

Inserindo um elemento na posição X



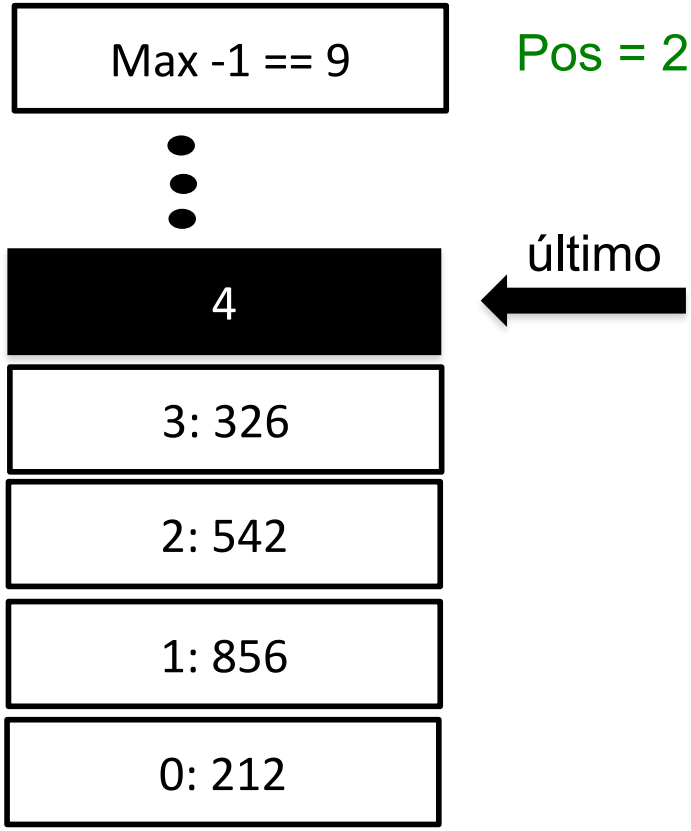

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
...
}
```

Inserindo um elemento na posição X



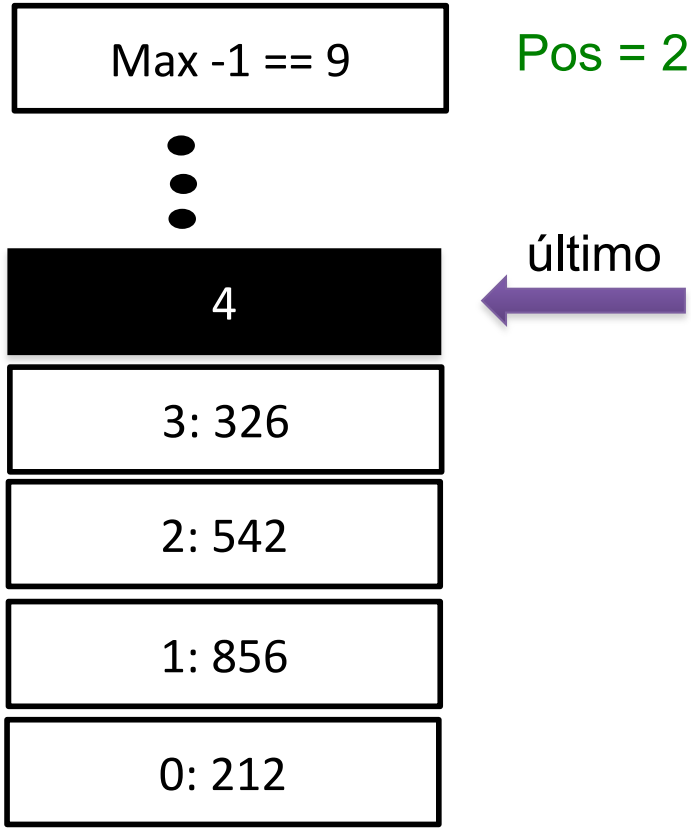

```
void insere( struct lista *ls, int A, int pos) {
        if (ls->ultimo == MAX) {
                printf (“lista cheia”);
                exit (1);
        }
        for (int i = ls->ultimo; i > pos; i--) {
                ls->elementos[i] = ls->elementos[i-1]
        }
        ls->ultimo += 1;
        ls->elementos[pos] = A;
}
void main () {
...
        insere (ls, 542, 2);
...
}
```

Removendo um elemento na posição X



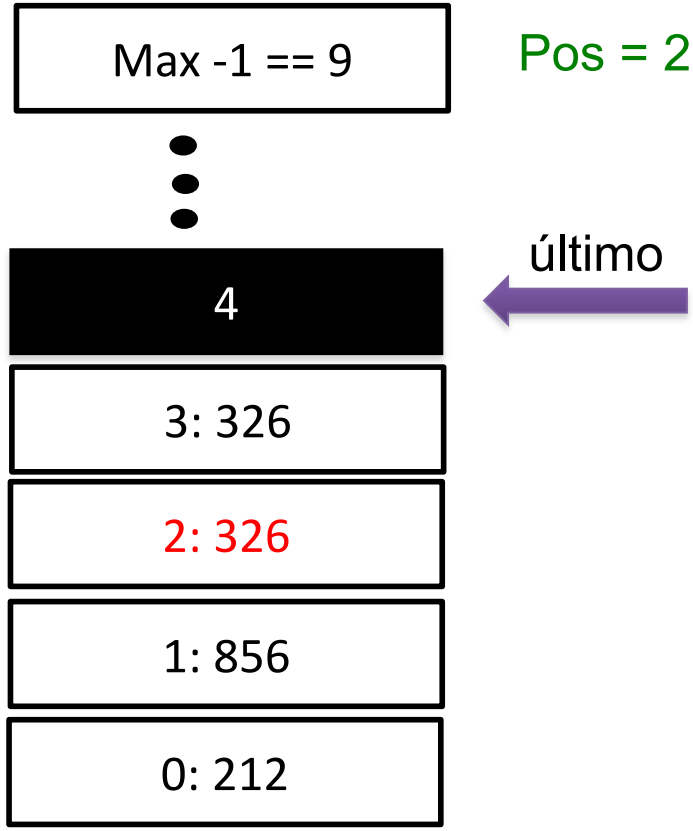

```
void remove( struct lista *ls, int pos) {
        if (ls->ultimo == 0) {
                printf (“lista vazia”);
                exit (1);
        }
        for (int i = pos; i < ls->ultimo; i++) {
                ls->elementos[i] = ls->elementos[i+1];

        }
        ls->ultimo -= 1;
}
void main () {
...
        remove (ls, 2);
...
}
```

Removendo um elemento na posição X



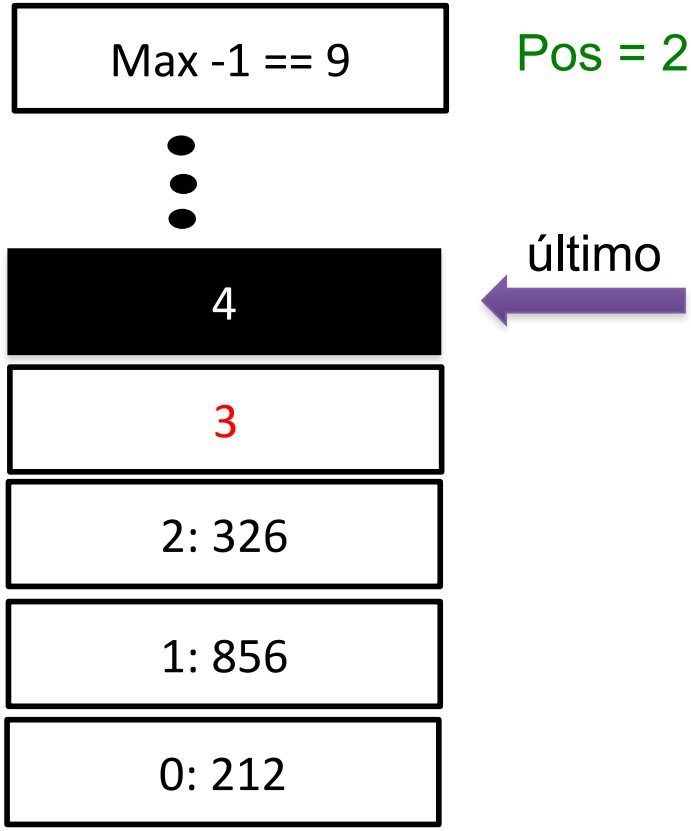

```
void remove( struct lista *ls, int pos) {
        if (ls->ultimo == 0) {
                printf (“lista vazia”);
                exit (1);
        }
        for (int i = pos; i < ls->ultimo; i++) {
                ls->elementos[i] = ls->elementos[i+1];

        }
        ls->ultimo -= 1;
}
void main () {
...
        remove (ls, 2);
...
}
```

Removendo um elemento na posição X



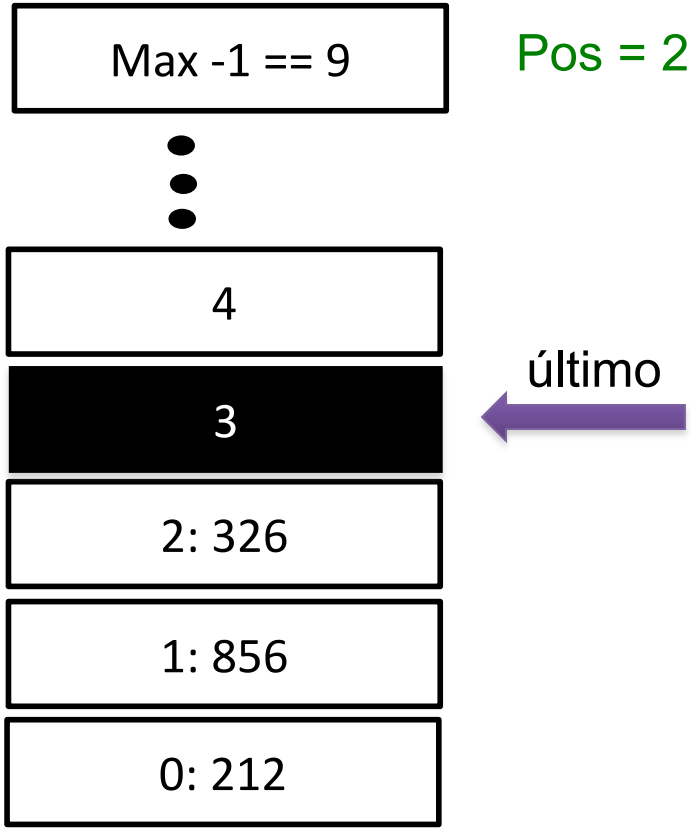

```
void remove( struct lista *ls, int pos) {
        if (ls->ultimo == 0) {
                printf (“lista vazia”);
                exit (1);
        }
        for (int i = pos; i < ls->ultimo; i++) {
                ls->elementos[i] = ls->elementos[i+1];

        }
        ls->ultimo -= 1;
}
void main () {
...
        remove (ls, 2);
...
}
```

Removendo um elemento na posição X



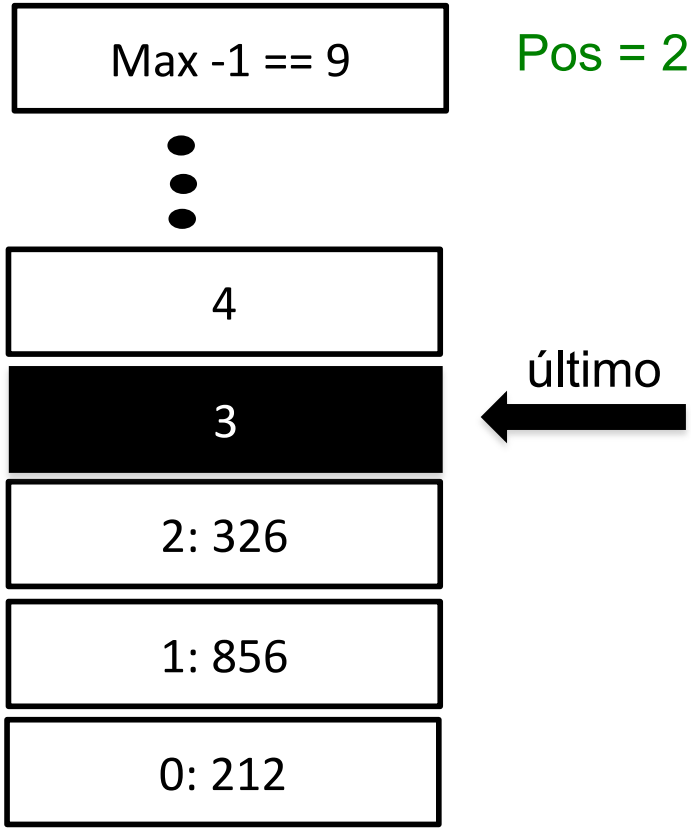

```
void remove( struct lista *ls, int pos) {
        if (ls->ultimo == 0) {
                printf (“lista vazia”);
                exit (1);
        }
        for (int i = pos; i < ls->ultimo; i++) {
                ls->elementos[i] = ls->elementos[i+1];

        }
        ls->ultimo -= 1;
}
void main () {
...
        remove (ls, 2);
...
}
```

Removendo um elemento na posição X



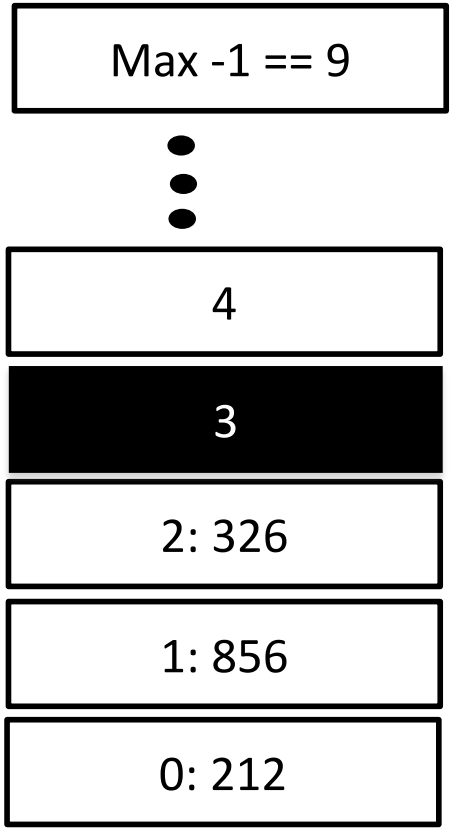

```
void remove( struct lista *ls, int pos) {
        if (ls->ultimo == 0) {
                printf (“lista vazia”);
                exit (1);
        }
        for (int i = pos; i < ls->ultimo; i++) {
                ls->elementos[i] = ls->elementos[i+1];


        ls->ultimo -= 1;
}
void main () {
...
        remove (ls, 2);
...
}
```

Removendo um elemento na posição X




```
void remove( struct lista *ls, int pos) {
        if (ls->ultimo == 0) {
                printf (“lista vazia”);
                exit (1);
        }
        for (int i = pos; i < ls->ultimo; i++) {
                ls->elementos[i] = ls->elementos[i+1];

        }
        ls->ultimo -= 1;
}
void main () {
...
        remove (ls, 2);
...
}
```

Destruindo uma lista




```
struct fila {
int elementos[MAX];
 int ultimo;
}
void destroi(struct lista *ls) {
free(ls);
}
```

Lista sequencial

- Vantagem
  - Economia de memória, pois os apontadores estão implícitos (o que não ocorre em alocação dinâmica)
- Desvantagens
  - O deslocamento causado pela inserção ou remoção no meio da lista
  - Necessidade de memória adicional irá exigir realocação

Exercício

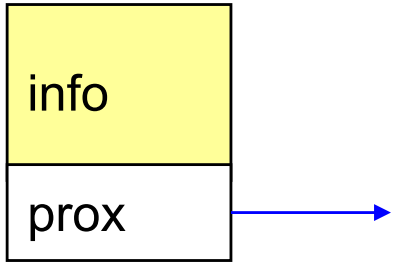
- Crie funções que implementem as seguintes operações:
  - Verificar se a lista L está ordenada (crescente ou decrescente)
  - Fazer uma cópia da Lista L1 em outra L2
  - Fazer uma cópia da Lista L1 em L2, eliminando repetidos
  - Inverter L1, colocando o resultado em L2
  - Inverter a própria L1
  - Intercalar L1 com L2, gerando L3 ordenada (considere L1 e L2 ordenadas)
  - Eliminar de L1 todas as ocorrências de um dado item (L1 está ordenada)
- L1 = {3, 1, 5, 2, 1, 8, 1, 5}
- L1(dec){8, 5, 5, 3, 2, 1, 1, 1}
- L1(cre){1, 1, 1, 2, 3, 5, 5, 8}

Listas Encadeadas

- Tamanho da lista não é pré-definido
  - Não é necessário informar o número de elementos em tempo de compilação
- Elementos não estão contíguos na memória
  - Cada item é encadeado com o seguinte mediante uma variável do tipo Ponteiro.
  - Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista
  - Cada elemento guarda quem é o próximo
- Há uma célula cabeça (inicio) para simplificar as operações sobre a lista

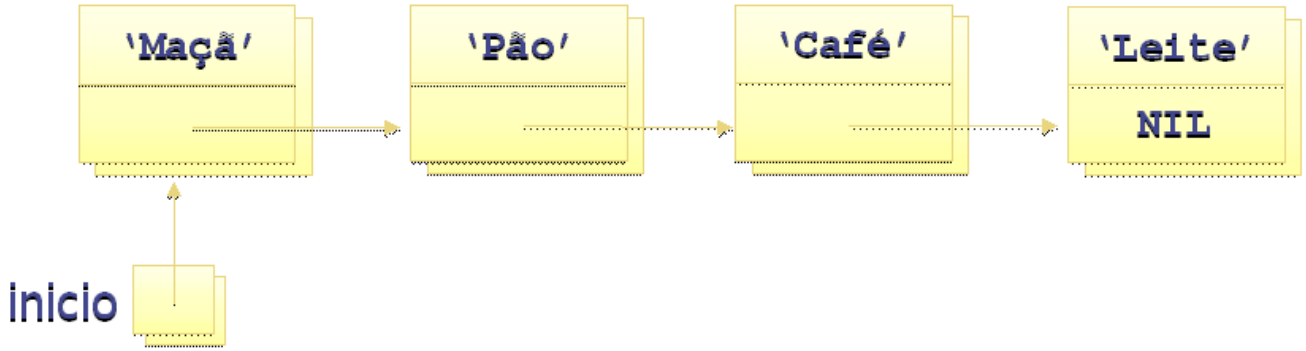
Sobre os Elementos da Lista

- Elemento: guarda as informações sobre cada elemento
- Para isso define-se cada elemento como uma estrutura que possui:
  - campos de informações
  - ponteiro para o próximo elemento



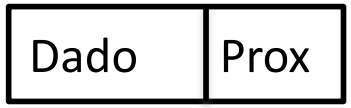
Sobre a Lista

- Uma lista **pode** ter uma célula cabeça (inicio)
- Uma forma bastante comum é manter uma variável ponteiro para o primeiro elemento da lista ligada
  - Convenciona-se que essa variável ponteiro deve ter valor NULL quando a lista estiver vazia
    - essa deve ser a iniciação da lista e também a forma de se verificar se ela se encontra vazia
  - Com listas ligadas, uma posição passa ser um ponteiro que aponta um determinado nó da lista



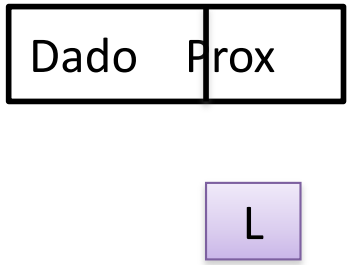
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada registro é:



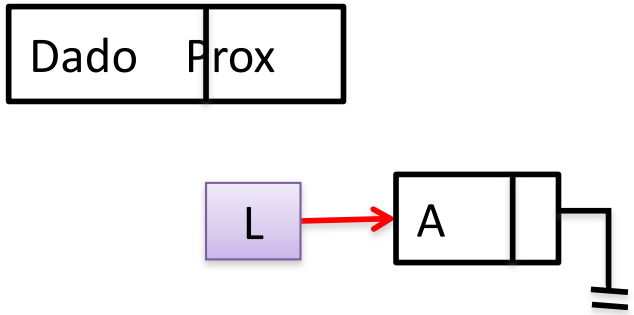
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



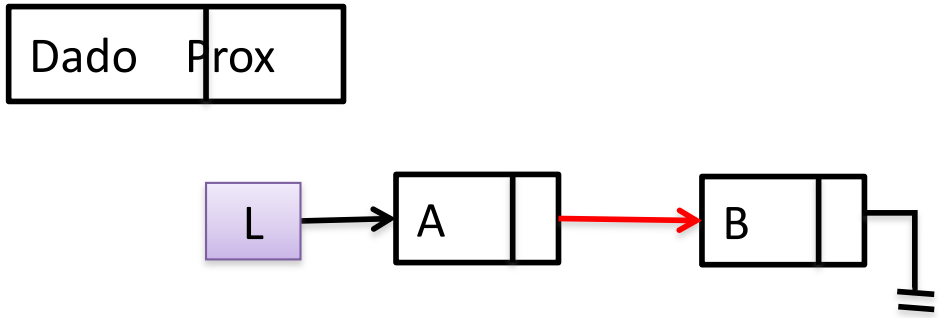
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



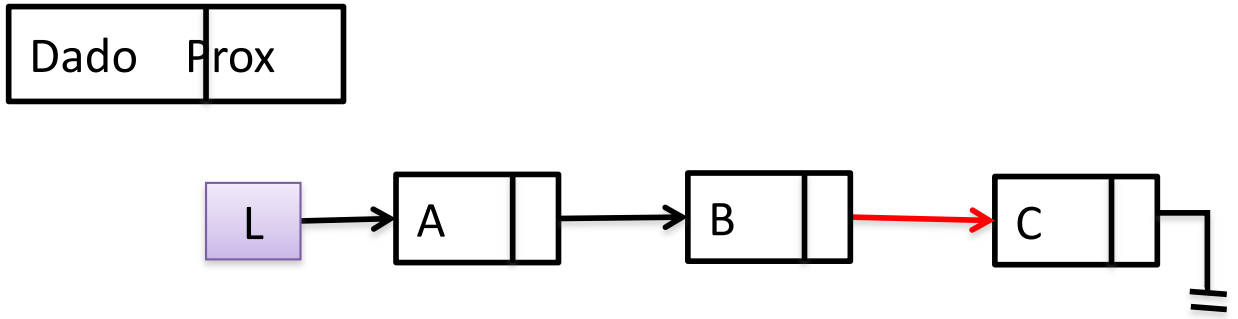
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



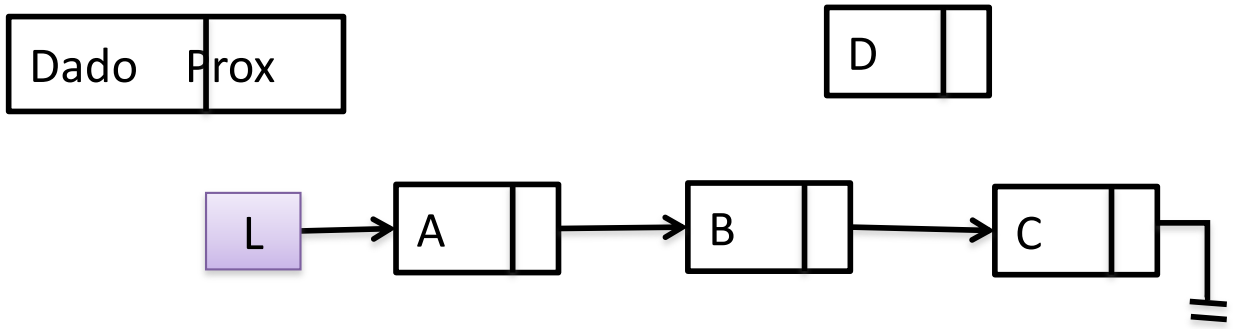
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



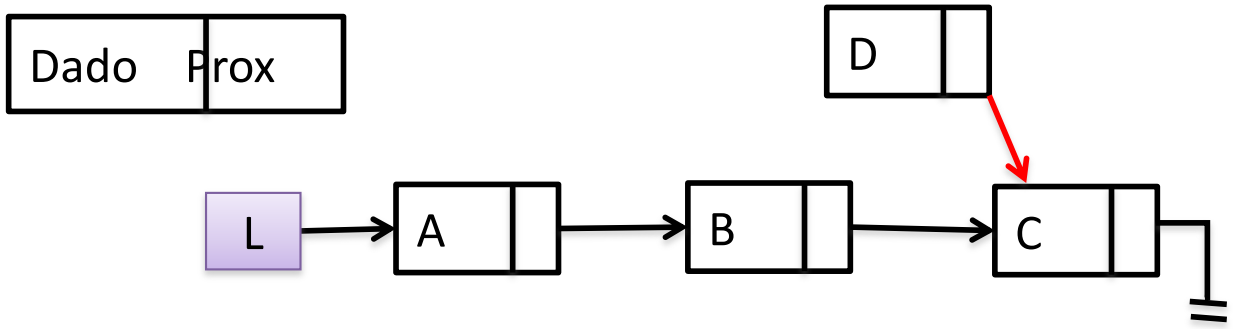
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



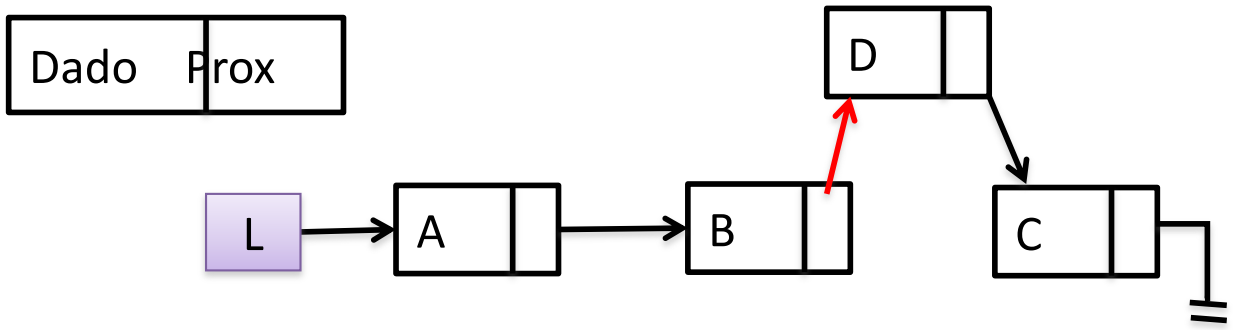
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



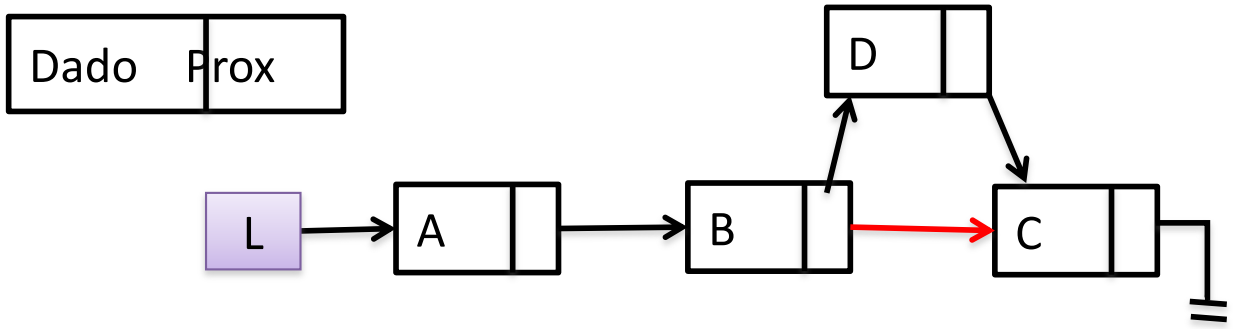
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



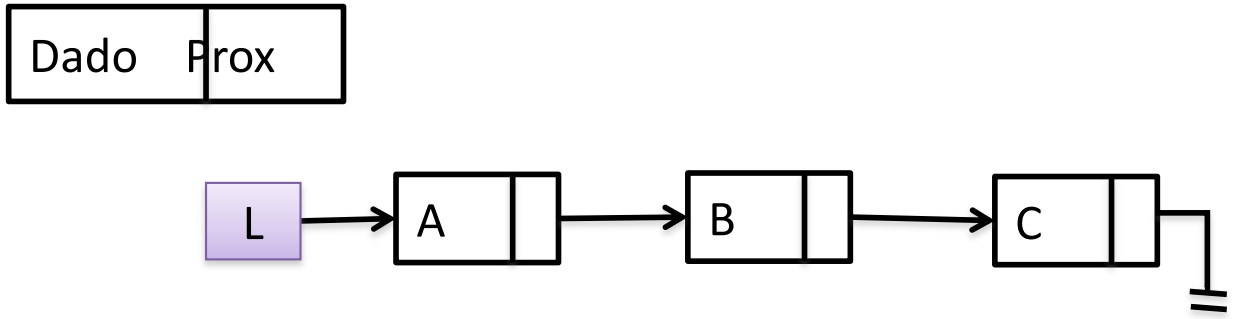
Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



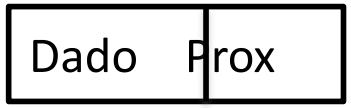
Implementando listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



Listas encadeadas

- Os elementos da lista são registros com um dos componentes destinado a guardar o endereço do registro sucessor.
  - Criar uma lista
  - Insere um elemento na posição X
  - Remove um elemento da posição Y
  - Destruir uma lista
- Cada refistro é:



Criando uma lista encadeada

- Para se criar uma lista ligada, é necessário criar um nó que possua um ponteiro para outro nó

```
// Estrutura de um nó da lista
struct No {
      int dado;
      struct No *proximo;
}
```

- Em seguida define-se a função para criar um nó

```
// Função para criar um novo nó
struct No* criaNo (int n) {
        struct No* novoNo = (struct No*)malloc(sizeof(struct No));
        if (novoNo == NULL) {
                printf("Erro: Falha na alocação de memória para o Nó.\n");
                return 0;
        }
        novoNo->dado = n;
        novoNo->proximo = NULL;
        return novoNo;
}
```

Inserindo um elemento na lista

- A estrutura de lista permite a inserção de elementos em qualquer posição
  - O custo computacional para inserir elementos em uma dada posição é maior que inserção nas extremidades
    - Busca

Inserindo um elemento na lista

- A estrutura de lista permite a inserção de elementos em qualquer posição
  - Inserção no Inicio

```
// Função para inserir um nó no início da lista
struct No* insereInicio (struct No* lista, int n) {
        struct No* novoNo = criaNo (n);
        if (novoNo == NULL) {
                return lista; // Se a alocação falhar, a lista permanece inalterada.
        }
        novoNo->proximo = lista;
        return novoNo;
}
```

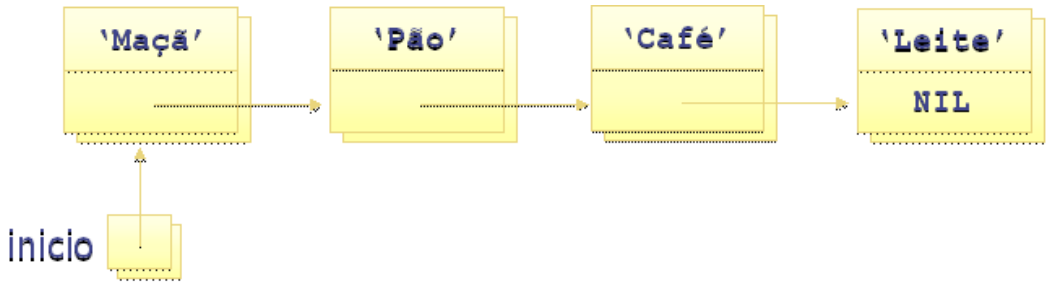
Inserindo um elemento na lista

- A estrutura de lista permite a inserção de elementos em qualquer posição
  - Inserção no Fim

```
// Função para inserir um nó no fim da lista
struct No* insereFim(struct No* lista, int n) {
        struct No* novoNo = criaNo (n);
        if (novoNo == NULL) {
                return lista; // Se a alocação falhar, a lista permanece inalterada.
        }
        if (lista == NULL) {
                return novoNo;
        }
        struct No* atual = lista;  //ponteiro atual usado para percorrer a lista
        while (atual->proximo != NULL) {
                atual = atual->proximo;
        }
        atual->proximo = novoNo;
        return lista;
}
```

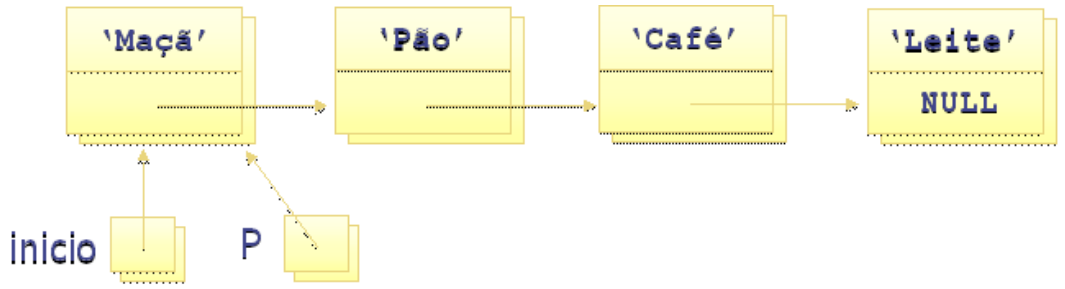
Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



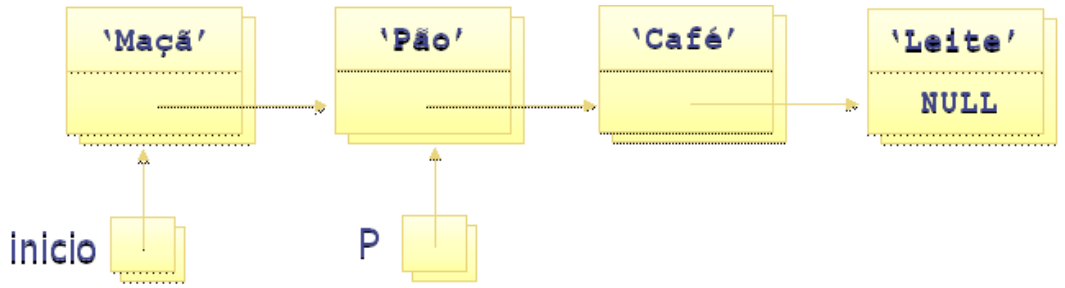
Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



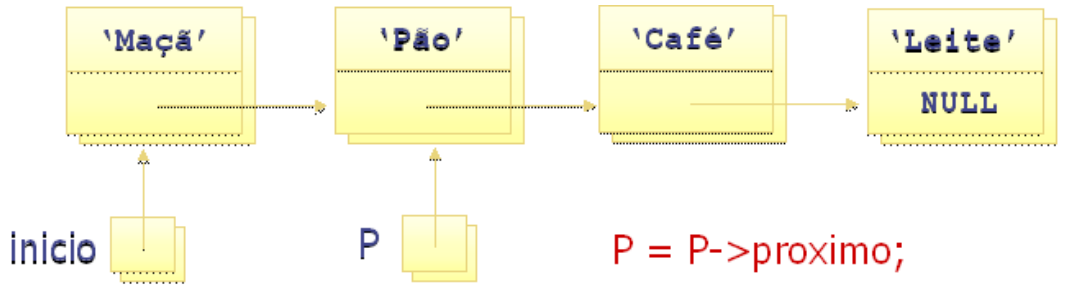
Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



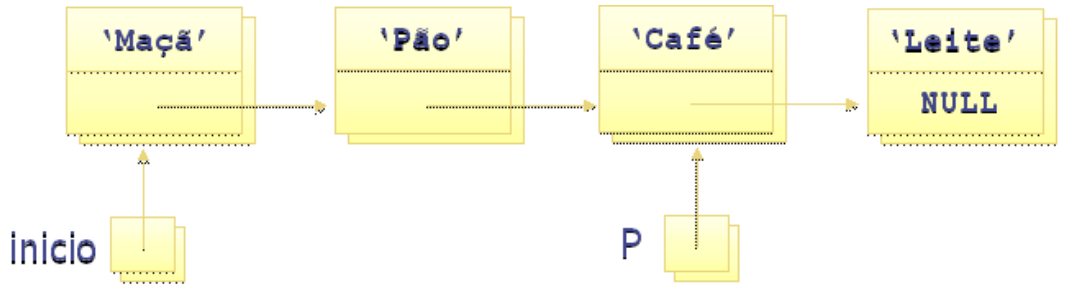
Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



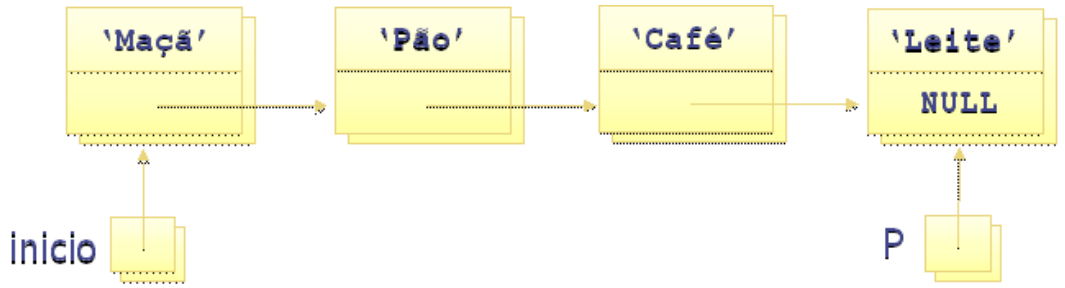
Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



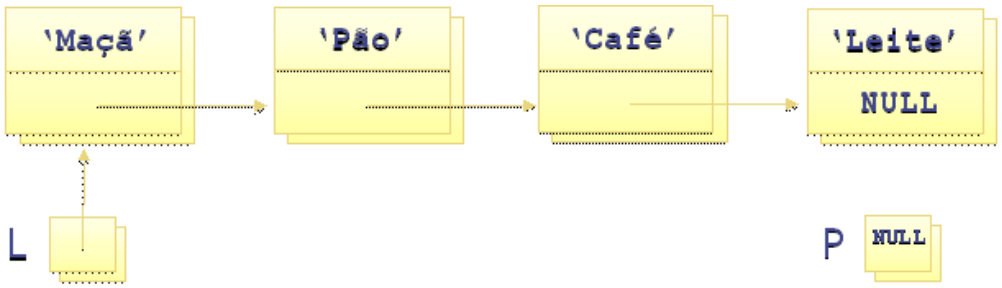
Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



Busca de um item (dad uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário



Busca de um item (dada uma chave)

- Buscar um item
  - Percorre a lista sequencialmente, comparando o valor desejado com o valor em cada nó
    - Se o valor for encontrado, a função retorna o nó que contém o item desejado;
    - Caso contrário, ela retorna NULL

```
// Função para recuperar um item da lista
struct No* buscaItem(struct No* lista, int n) {
        struct No* atual = lista;
        while (atual != NULL) {
                if (atual->dado == n) {
                        return atual;
                }
                atual = atual->proximo;
        }
        return NULL; // O item não foi encontrado na lista.
}
```

Removendo um elemento na lista

- É necessário Buscar o elemento na lista!

```
// Função para remover um nó da lista
struct No* removeLista(struct No* lista, int n) {
	if (lista == NULL) {
		return NULL; // A lista está vazia, não há nada a ser removido.
	}
	if (lista->dado == n) {
		struct No* novaLista = lista->proximo;
		free(lista);
		return novaLista;
	}
	struct No* atual = lista;
	while ((lista->proximo != NULL) && (atual->proximo->dado != n)) {
		 atual = atual->proximo;
	}
	if (atual->proximo != NULL) {
		struct No* temp = atual->proximo;
		atual->proximo = atual->proximo->proximo;
		free(temp);
	}
	return lista;
}
```

Outras funções

- Imprimir lista

```
// Função para imprimir os elementos da lista
void printLista(struct No* lista) {
        struct No* atual = lista;
        while (atual != NULL) {
                printf("%d -> ", atual->dado);
                atual = atual->proximo;
        }
        printf("NULL\n");
}
```

- Libera a lista

```
// Função para liberar a memória alocada para a lista
void freeList(struct No* lista) {
        struct No* atual = lista;
        while (atual != NULL) {
                struct No* proximo = atual->proximo;
                free(atual);
                atual = proximo;
        }
}
```

Outras funções

```
// Programa Principal
int main() {
        struct No* lista = NULL;       // Inicializando a lista vazia
        lista = insereInicio(lista, 1);
        lista = insereInicio(lista, 2);
        lista = insereFim(lista, 3);
        printf("Lista encadeada: ");
        printList(lista);

        lista = removeLista(Lista, 2);         // Remover um item da lista
        printf("Lista após a remoção do item 2: ");
        printList(lista);

        struct No* item = buscaItem(lista, 3);   // Recuperar um item da lista
        if (item != NULL) {
                printf("Item 3 encontrado na lista.\n");
        }
        else {
                printf("Item 3 não encontrado na lista.\n");
        }
        freeList(lista);
        return 0;
}
```

Exercício

- Apagar lista
- Inserir item (por posição)
- Remover item (por posição)
- Recuperar item (por posição)
- Contar número de itens Imprimir lista

Operações sobre Lista Encadeada

- Vantagens:
  - Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
  - Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido a priori).
- Desvantagem:
  - Utilização de memória extra para armazenar os apontadores.