SCC0502 - ALGORITMOS E ESTRUTURAS DE DADOS I

Listas Lineares Encadeadas

Prof.: Leonardo Tórtoro Pereira
leonardop@usp.br

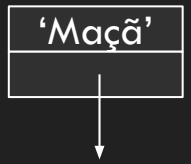
Baseado nos slides do Prof. Rudinei Goularte

Conteúdo

- ➔ Listas Ligadas - Discussão Intuitiva
- ➔ TAD Lista e Lista Ligada
- ➔ Lista Ligada – Implementação

Discussão Intuitiva

- Ponteiros podem ser usados para construir estruturas, tais como listas, a partir de componentes simples chamados nós



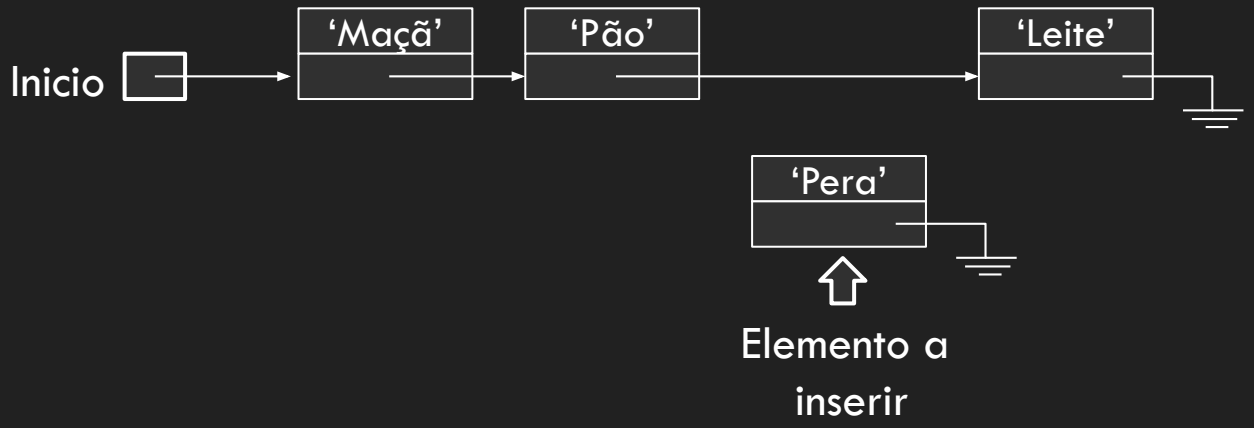
Discussão Intuitiva

- ➔ Listas ligadas são úteis pois podem ser utilizadas para implementar o TAD lista. Nesse caso, as operações inserção (ordenada) e remoção no meio da lista podem ser mais eficientes
- ➔ Uma segunda vantagem é o fato de não ser necessário informar o número de elementos em tempo de compilação

Inserção

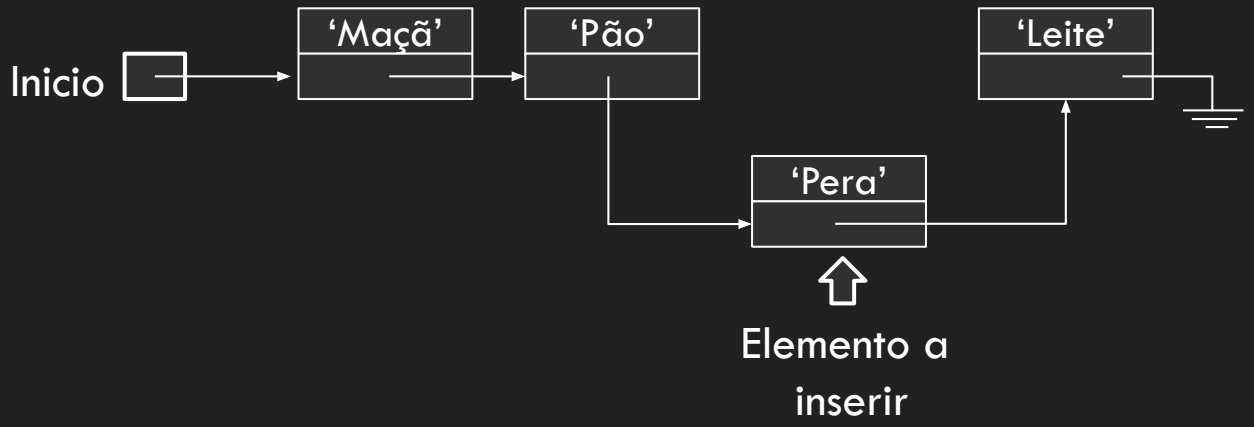
Discussão Intuitiva

➔ Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



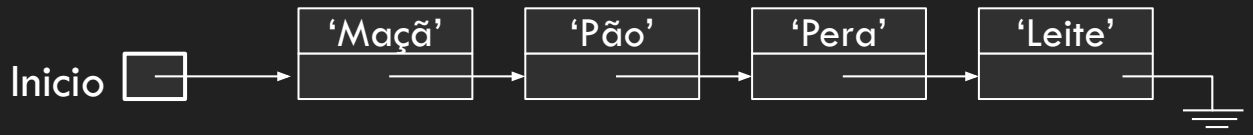
Discussão Intuitiva

- Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



Discussão Intuitiva

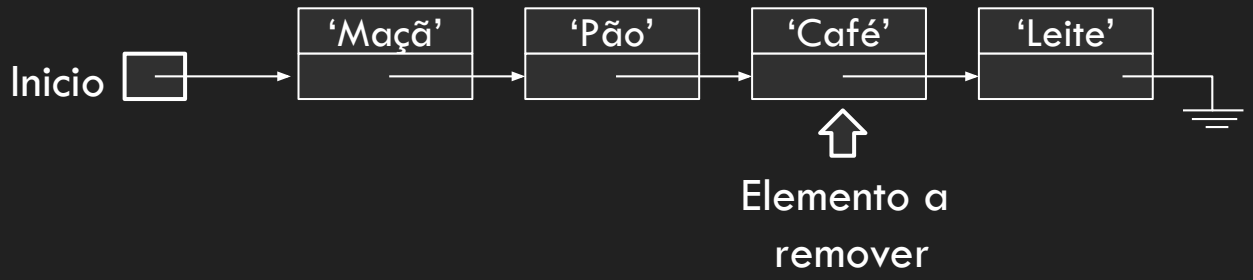
➔ Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



Remoção

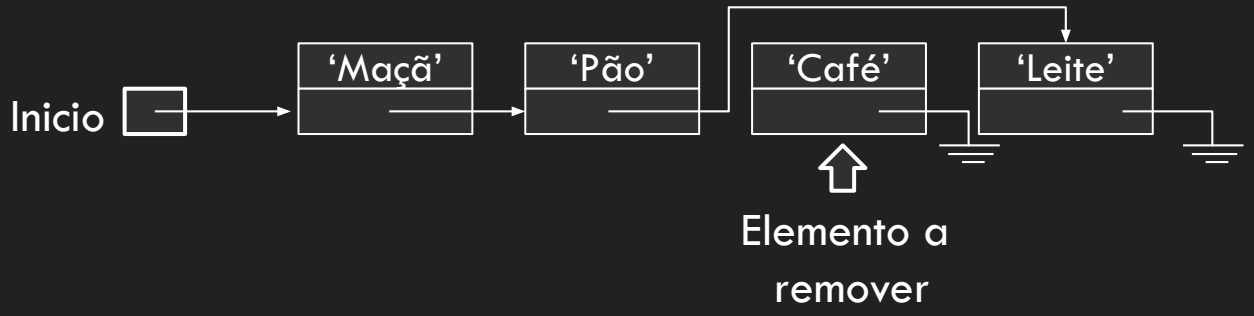
Discussão Intuitiva

- Por exemplo, uma operação de **remoção** pode ser feita da seguinte maneira



Discussão Intuitiva

- Por exemplo, uma operação de **remoção** pode ser feita da seguinte maneira

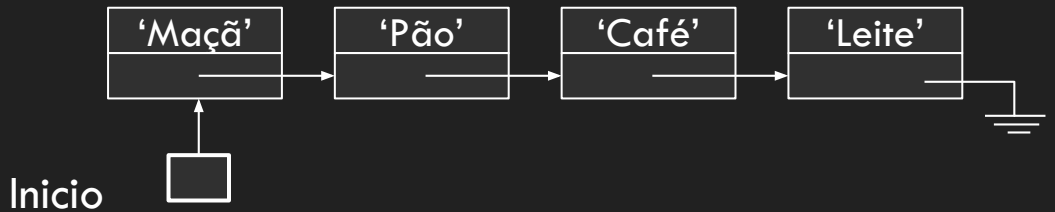


Relembrando: TAD Listas

- Principais operações
  - Criar lista
  - Apagar lista
  - Inserir item (última posição)
  - Remover item (dado uma chave)
  - Busca item (dado uma chave)
  - Contar número de itens
  - Verificar se a lista está vazia
  - Verificar se a lista está cheia
  - Imprimir lista

TAD Listas e Listas Ligadas

- Antes de começarmos, precisamos definir como a lista será representada
- Uma forma bastante comum é manter uma variável ponteiro para o primeiro elemento da lista ligada



TAD Listas e Listas Ligadas

- ➔ Convenciona-se que essa variável ponteiro deve ter valor NULL quando a lista estiver vazia
- ➔ Portanto, essa deve ser a inicialização da lista e também a forma de se verificar se ela se encontra vazia

TAD Listas e Listas Ligadas

- Outro detalhe importante é quanto às posições
  - Na implementação com vetores, uma posição é um valor inteiro entre 0 e o campo fim
  - Com listas ligadas, uma posição passa ser um ponteiro que aponta um determinado nó da lista
- Vamos analisar cada uma das operações do TAD Lista

TAD Listas I

- Criar lista
  - Pré-condição: existir espaço na memória
  - Pós-condição: inicia a estrutura de dados

TAD Listas I

- Limpar lista
  - Pré-condição: lista não pode estar vazia
  - Pós-condição: remove a estrutura de dados da memória

TAD Listas II

- Inserir item
  - Pré-condição: deve existir a lista e existir memória disponível
  - Pós-condição: insere um item na última posição, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas II

- Remover item (dado uma chave)
  - Pré-condição: a lista deve existir
  - Pós-condição: remove um determinado item da lista dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas III

- Recuperar item (dado uma chave)
  - Pré-condição: a lista deve existir
  - Pós-condição: recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas III

- Contar número de itens
  - Pré-condição: a lista deve existir
  - Pós-condição: retorna o número de itens na lista

TAD Listas IV

- Verificar se a lista está vazia
  - Pré-condição: a lista deve existir
  - Pós-condição: retorna **true** se a lista estiver vazia e **false** caso-contrário

TAD Listas IV

- Verificar se a lista está cheia
  - Pré-condição: a lista deve existir
  - Pós-condição: retorna **true** se a lista estiver cheia e **false** caso-contrário

TAD Listas V

- Imprimir lista
  - Pré-condição: a lista deve existir
  - Pós-condição: imprime na tela os itens da lista

Listas Ligadas - Implementação

```
#ifndef LISTADINAMICA_H
#define LISTADINAMICA_H
#define TRUE 1 /*define valor booleano – não existe na linguagem C*/
#define FALSE 0
#define boolean int //define um tipo booleano
#define inicial 0
#define ERRO -32000
typedef struct lista_ LISTA;
LISTA *lista_criar(void);
boolean lista_inserir_fim(LISTA *lista, ITEM i);
boolean lista_inserir_ordenado(LISTA *lista, ITEM i);
void lista_apagar(LISTA **ptr);
boolean lista_remover(LISTA *lista, int chave);
ITEM lista_busca(LISTA *lista, int chave);
int lista_tamanho(LISTA *lista);
boolean lista_vazia(LISTA *lista);
boolean lista_cheia(LISTA *lista);
void lista_imprimir(LISTA *lista);
#endif
```

Listas Ligadas

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

```
typedef struct no_ NO;
struct no_{
   ITEM *item;
   NO *proximo;
};
```

Lista Ligada

- Considerando a estrutura NO, para a definição da lista ligada o que falta é a indicação da posição de memória do primeiro nó
- Também incluiremos a posição para o último nó para acelerar a inserção de itens no final da lista e uma variável

```
struct lista_{
   NO *inicio;
   NO *fim;
   int tamanho;  //tamanho da lista
};
```

Criar lista

- Pré-condição: existir memória
- Pós-condição: inicia a estrutura de dados

Antes

?

Depois

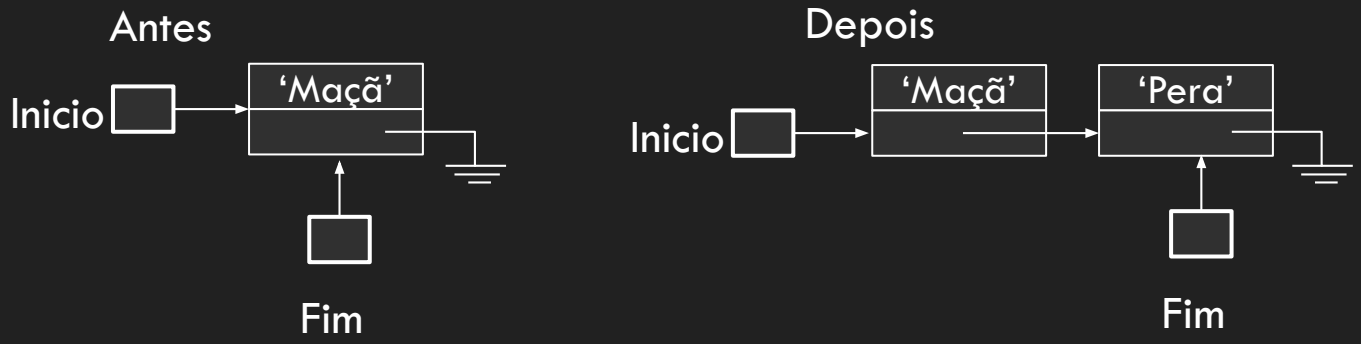
?

Criar lista

```
/*pré-condição: existir espaço na memória.*/
LISTA *lista_criar(void){
  LISTA *lista = (LISTA *) malloc(sizeof(LISTA));
  if(lista != NULL) {
     lista->inicio = NULL;
     lista->fim = NULL;
     lista->tamanho = 0;
  }
  return (lista);
}
```

Inserir item (última posição)

- Pré-condição: existe memória disponível
- Pós-condição: insere um item na última posição, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



Memória Disponível

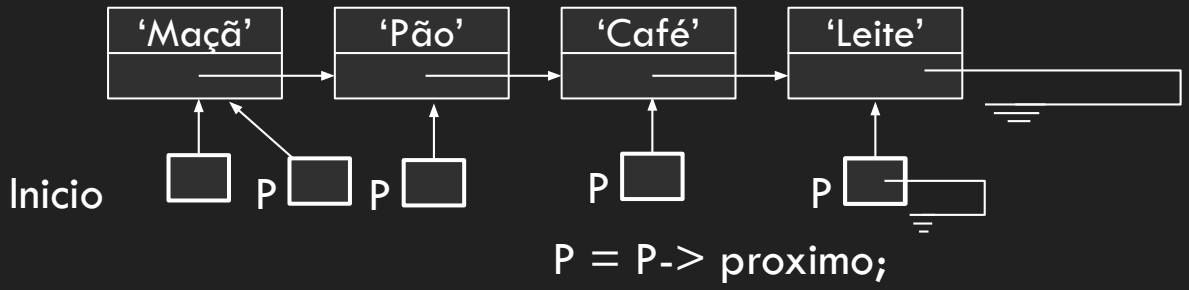
- Diferente da implementação com vetores, a lista ligada não requer especificar um tamanho para a estrutura
- Entretanto, a memória *heap* não é ilimitada e é sempre importante verificar se existe memória disponível ao chamar *malloc()*
- Em C, o procedimento *malloc()* atribui o valor NULL à variável ponteiro quando não existe memória disponível

Inserir item (última posição)

```
/*Insere um novo nó no fim da lista. PARA LISTAS NÃO ORDENADAS*/
boolean lista_inserir_fim(LISTA *lista, ITEM item){
     if ((!lista_cheia(lista)) && (lista != NULL)) {
          NO *pnovo = (NO *) malloc(sizeof (NO));
          if (lista->inicio == NULL){
               pnovo->item = item;
               lista->inicio = pnovo;
               pnovo->proximo = NULL;
          } else {
               lista->fim->proximo = pnovo;
               pnovo->item = item;
               pnovo->proximo = NULL;
          }
          lista->fim = pnovo;
          lista->tamanho++;
          return (TRUE);
     } else
          return (FALSE);
}
```

Recuperar item (dada uma chave)

- Pré-condição: a lista deve existir
- Pós-condição: recupera o item dada uma chave *x*, retorna **o item** cuja chave é *x* se a operação foi executada com sucesso, **NULL** caso contrário



Recuperar item (dada uma chave)

```
ITEM lista_busca(LISTA *lista, int chave){
  NO *p;
  if (lista != NULL){
     p = lista->inicio;
     while (p != NULL) {
        if (item_get_chave(p->item) == chave)
           return (p->item);
        p = p->proximo;
     }
   }
   return(NULL);
}
```

Verificar se a lista está vazia

- Pré-condição: nenhuma
- Pós-condição: retorna **true** se a lista estiver vazia e **false** caso-contrário

```
1 boolean lista_vazia(LISTA *lista){
2    if((lista != NULL) && lista->inicio == NULL)
3       return (TRUE);
4    return (FALSE);
5 }
```

Remover item (dado uma chave)

- ➔ Pré-condição: a lista deve existir e não estar vazia
- ➔ Pós-condição: remove um determinado item da lista dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Remover item (dada uma chave)

```
boolean lista_remover_item(LISTA *lista, int chave) {
	if (lista != NULL){
		NO *p = lista->inicio;  NO *aux = NULL;
		while(p != NULL && (item_get_chave(p->item)) != chave) { /*procura até achar chave ou fim lista*/
			aux = p;		/*aux - guarda posição anterior ao nó sendo pesquisado (p)*/
			p = p->proximo;
		}
		if(p != NULL) {
			if(p == lista->inicio) { /*se a chave está no 1o nó (Exceção a ser tratada!)*/
				lista->inicio = p->proximo;
				p->proximo = NULL;
			}
		else {
			aux->proximo = p->proximo;
			p->proximo = NULL;
		}
		if(p == lista->fim)    /*se chave está no último nó*/
			lista->fim = aux;
			lista->tamanho--;  free(p);
			return (TRUE);
		}
	}
	return (FALSE);
}
```

Exercícios

- ** Implementar a operação busca de modo recursivo **
- Implementar as demais operações do TAD Lista
  - Apagar lista
  - Inserir item (ordenadamente)
  - Remover item (ordenadamente)
  - Recuperar item
  - Contar número de itens
  - Imprimir lista

Referências

- ZIVIANI, N. Projeto de Algoritmos, Thomson, 2a. Edição, 2004.