SCC0202 – Algoritmos e Estrutura de Dados I

Listas Lineares Simplesmente Encadeadas

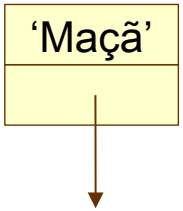
Prof.: Dr. Rudinei Goularte
(rudinei@icmc.usp.br)
Instituto de Ciências Matemáticas e de Computação - ICMC
Sala 4-229

Conteúdo

- Listas Encadeada - Discussão Intuitiva
- TAD Lista e Lista Encadeada
- Lista Encadeada – Implementação

Discussão Intuitiva

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

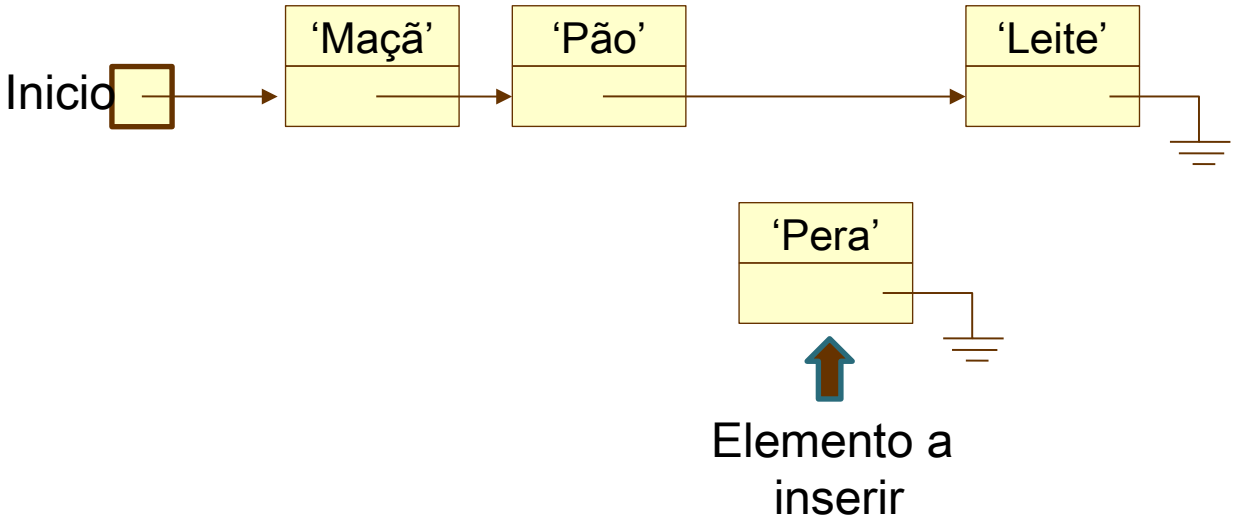


Discussão Intuitiva

- Listas encadeadas (ou 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

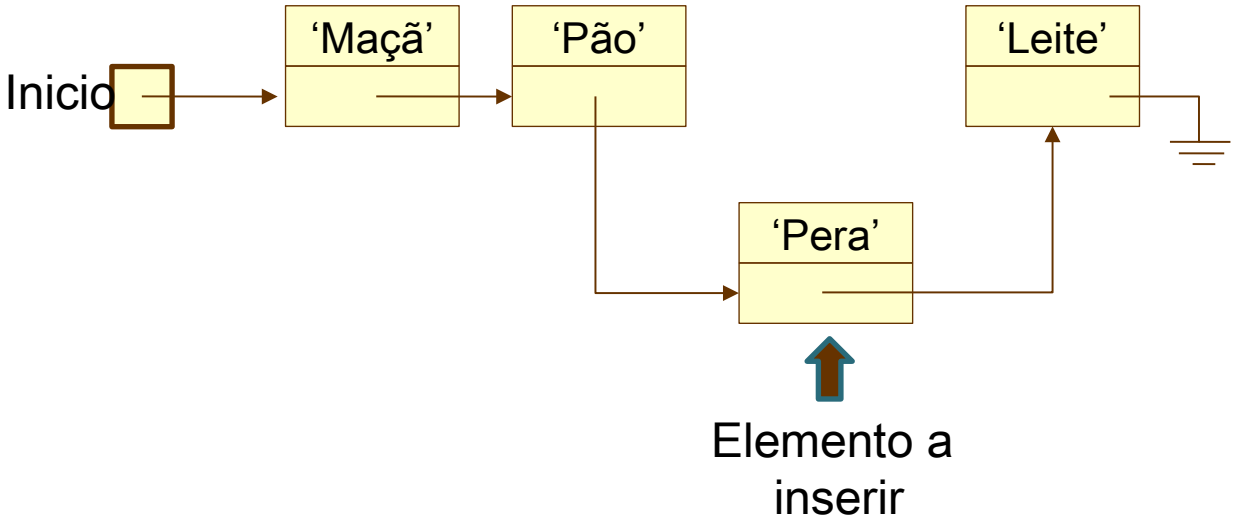
Discussão Intuitiva

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



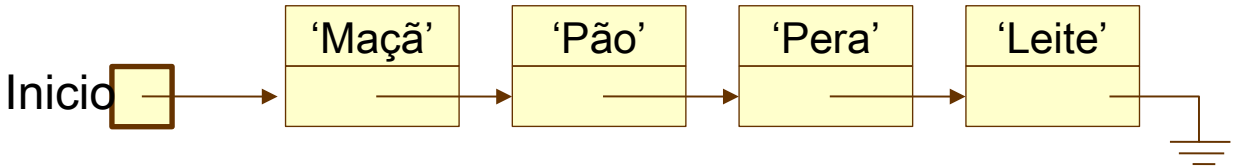
Discussão Intuitiva

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



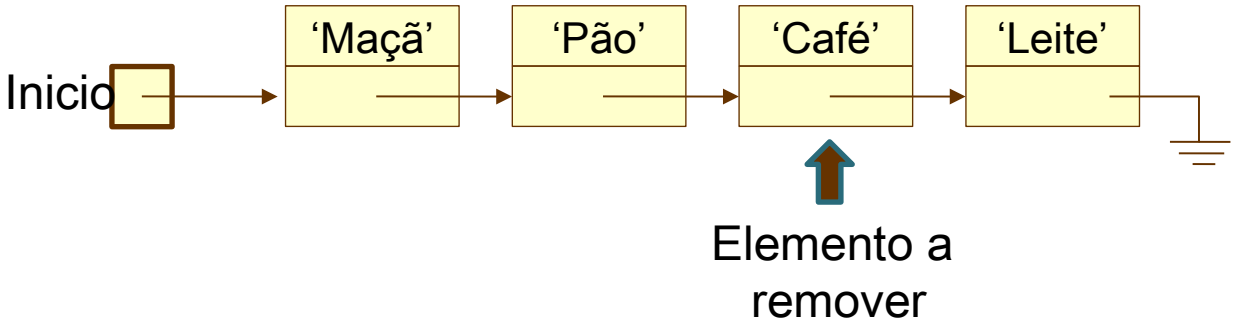
Discussão Intuitiva

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



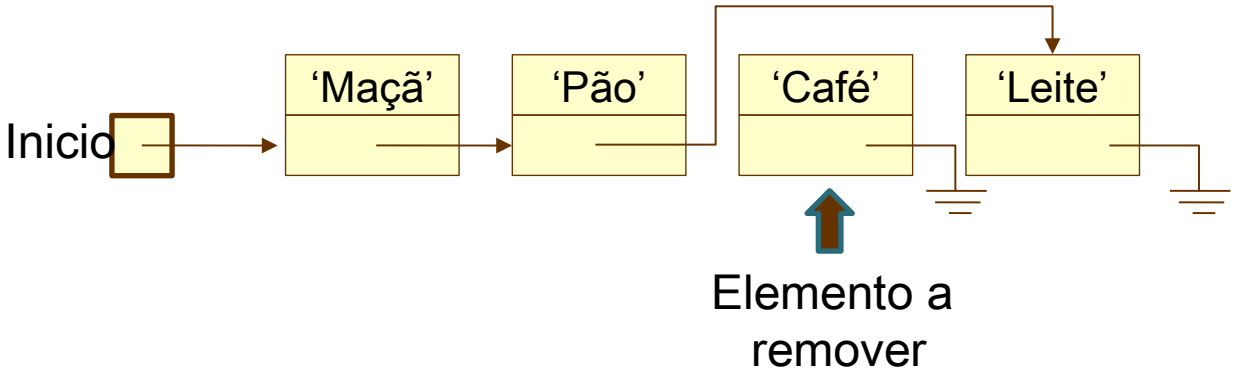
Discussão Intuitiva

- 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

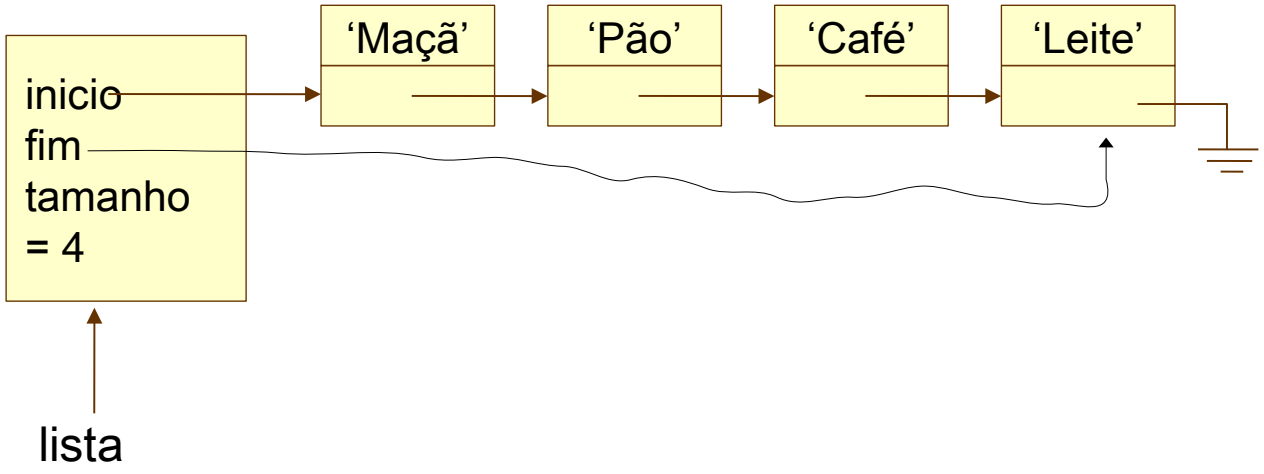


Relembrando: TAD Listas

- 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 Encadeadas

- 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 encadeada



TAD Listas e Listas Encadeadas

- 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 Encadeadas

- 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 encadeadas, 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
- 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
- Remover item (dado uma chave)
  - Pré-condição: a lista deve existir
  - Pós-condição: remove um determinado item da lista dado uma chave, retorna o item se a operação foi executada com sucesso, **NULL** caso contrário (o chamador é responsável pelo item, nesse caso)

TAD Listas III

- Recuperar item (dado uma chave)
  - É uma busca
  - Pré-condição: a lista deve existir
  - Pós-condição: recupera o item dada uma chave, retorna o item caso a chave tenha sido encontrada na lista, **NULL** caso contrário
- 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
- 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 Encadeadas - Implementação

```
1 #ifndef LISTA_H
2   #define LISTA_H
3   #define ORDENADA 0
4   #define ERRO -32000
5   #include “item.h”
6
7   typedef struct lista_ LISTA;
8
9   LISTA *lista_criar(void);
10  bool lista_inserir(LISTA *lista, ITEM *item);
11  bool lista_apagar(LISTA **lista);
12  ITEM *lista_remover(LISTA *lista, int chave);
13  ITEM *lista_busca(LISTA *lista, int chave);
14  int lista_tamanho(LISTA *lista);
15  bool lista_vazia(LISTA *lista);
16  bool lista_cheia(LISTA *lista);
17  void lista_imprimir(LISTA *lista);
18#endif
```

Listas Encadeadas

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

```
1 typedef struct no_ NO;
2 struct no_{
3    ITEM *item;
4    NO *proximo;
5 };
```

Listas Encadeadas

- 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 tamanho

```
1 struct lista_{
2    NO *inicio;
3    NO *fim;
4    int tamanho;  //tamanho da lista
5 };
```

Criar lista

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

Antes

?

Depois

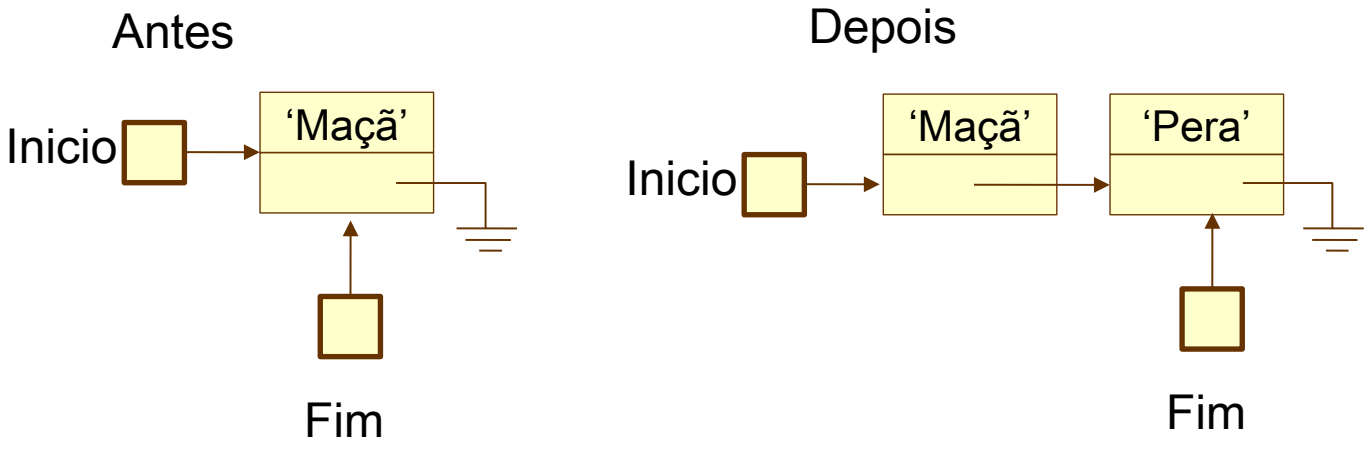
?

Criar lista

```
/*pré-condição: existir espaço na memória.*/
1 LISTA *lista_criar(void){
2   LISTA *lista = (LISTA *) malloc(sizeof(LISTA));
3   if(lista != NULL) {
4      lista->inicio = NULL;
5      lista->fim = NULL;
6      lista->tamanho = 0;
7   }
8   return (lista);
9 }
```

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

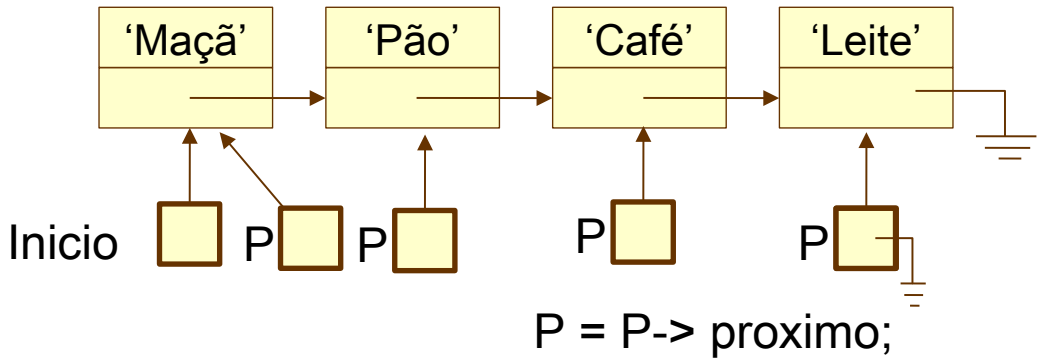
- Diferente da implementação com vetores, a lista ligada não requer especificar um tamanho para a estrutura
- Entretanto, a memória *heap* não é ilimitada e é sempre importante verificar se existe memória disponível ao chamar *malloc()*
- Em C, o procedimento *malloc()* atribui o valor NULL à variável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

Inserir item (última posição)

```
1 /*Insere um novo nó no fim da lista. PARA LISTAS NÃO ORDENADAS*/
2 bool lista_inserir_fim(LISTA *lista, ITEM item){
3
4    if ((!lista_cheia(lista)) && (lista != NULL)) {
5        NO *pnovo = (NO *) malloc(sizeof (NO));
6        pnovo->item = item;
7        pnovo->proximo = NULL;
8
9        if (lista->inicio == NULL)
10              lista->inicio = pnovo;
11       else
12           lista->fim->proximo = pnovo;
13       lista->fim = pnovo;
14       lista->tamanho++;
15
16       return (true);
17   } else
18       return (false);
19 }
```

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. Não remove o item da lista!



Recuperar item (dada uma chave)

```
1 ITEM *lista_busca(LISTA *lista, int chave){
2     NO *p;
3     if (lista != NULL){
4         p = lista->inicio;
5         while (p != NULL) {
6             if (item_get_chave(p->item) == chave)
7                 return (p->item);
8             p = p->proximo;
9         }
10     }
11     return(NULL);
12 }
```

Verificar se a lista está vazia

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

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

Remover item (dada 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)

```
1 ITEM *lista_remover(LISTA *lista, int chave) {
2   if (lista != NULL){
3      NO *p = lista->inicio;  NO *aux = NULL;
4      while(p != NULL && (item_get_chave(p->item)) != chave){/*procura até achar chave ou fim lista*/
5         aux = p;         /*aux - guarda posição anterior ao nó sendo pesquisado (p)*/
6         p = p->proximo;
7      }
8      if(p != NULL) {
9           if(p == lista->inicio) { /*se a chave está no 1o nó (Exceção a ser tratada!)*/
10             lista->inicio = p->proximo;
11             p->proximo = NULL;
12          }
13          else {
14             aux->proximo = p->proximo;
15             p->proximo = NULL;
16          }
17          if(p == lista->fim)    /*se chave está no último nó*/
18               lista->fim = aux;
19          ITEM *it = p→item;
20          lista->tamanho--;  free(p);
21          return (it);
22    }
23  }
24  return (NULL);
25}
```

Exercícios

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

Referências

- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de Dados e seus Algoritmos, Livros Técnicos e Científicos, 1994.

- TENEMBAUM, A.M., e outros Data Structures Using C, Prentice-Hall, 1990.

- ZIVIANI, N., Projeto de Algoritmos com Implementações em Pascal e C., Thompson, 2a. Ed, São Paulo, 2004.

- Material baseado nos originais produzidos pelos professores:
  - Gustavo E. de A. P. A. Batista
  - Fernando V. Paulovich