

Lista Encadeada

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados 2 (AE43CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

- Alocação Dinâmica de *Structs*
 - Célula
- Listas Encadeadas
 - Lista duplamente encadeada
 - Lista encadeada circular
- TAD Lista Encadeada Simples

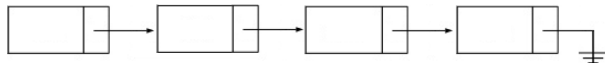
- Recapitulação sobre listas estáticas
 - Os itens são armazenados em vetores
 - A alocação é feita de forma estática
 - Simplex: a estrutura contém uma variável para indicar a posição final da lista



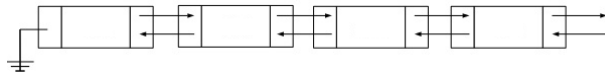
- TAD listas estáticas: criar uma lista vazia, inserir, remover, acessar ...

- Vantagens das listas estáticas
 - Simples implementação
 - Economia de memória (se houver um bom gerenciamento)
- Desvantagens das listas estáticas
 - Custo para retirar itens
 - Caso a lista atinja o limite de armazenamento, não é possível realocar memória
 - Pode ocorrer desperdício de memória se for alocado muito espaço e for utilizado pouco do mesmo
 - Na realidade, não há a operação de remoção (os elementos são geralmente apenas deslocados, mas a cópia do último elemento é mantido)

- Listas encadeadas
 - Cada elemento é armazenado em uma estrutura conhecida como célula
 - Tipos de listas encadeadas
 - Encadeada simples



- Duplamente encadeada



- Encadeada circular



Alocação Dinâmica de *Structs*

Alocação Dinâmica de *Structs*

- A partir do uso de ponteiros é possível alocar dinamicamente vetores e matrizes de *struct*
- Os comandos seguem o mesmo padrão das variáveis comuns
- A alocação é também feita por meio de funções da *stdlib*:
 - *Malloc()*
 - *Calloc()*
 - *Realloc()*
 - *free()*

- Exemplo

```
#include <stdlib.h>

typedef struct{
    char nome[120];
    char RG[10];
    char CPF[14];
    Data nasc;
    Endereco end;
}Pessoa;

Pessoa* alocar_vetor(int n){
    Pessoa *p;

    p = (Pessoa*) malloc(n * sizeof(Pessoa));

    return p;
}
```


Alocação Dinâmica de *Structs*

Célula

- Também conhecida como nó (*node*)
- É composta por um elemento (estrutura) e a informação necessária (ponteiro) para acessar outro elemento
 - Em listas encadeadas simples, o ponteiro aponta para o próximo elemento

Elemento →

	•
--	---

 ← **Ponteiro para o próximo elemento**

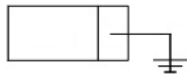
- Em listas duplamente encadeadas, a célula contém dois ponteiros: um apontando para o próximo elemento e o outro apontando para o anterior



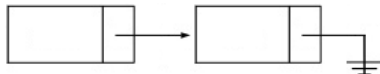
Alocação Dinâmica de *Structs*

Célula

- Exemplo de célula apontando para NULL



- Exemplo de célula apontando para outra célula

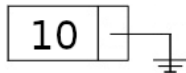


- Exemplo de estrutura para uma célula

```
typedef struct Cell Cell;  
  
struct Cell{  
    int item;  
    Cell* next;  
};
```

Alocação Dinâmica de *Structs*

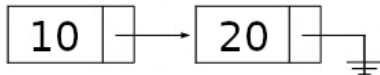
Célula



```
Cell *p = (Cell*) malloc(sizeof(Cell));  
p->item = 10;  
p->next = NULL;
```

Alocação Dinâmica de *Structs*

Célula

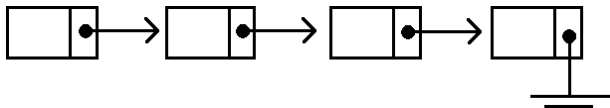


```
Cell *p1 = (Cell*) malloc(sizeof(Cell));  
Cell *p2 = (Cell*) malloc(sizeof(Cell));  
p1->item = 10;  
p2->item = 20;  
  
p2->next = NULL;  
p1->next = p2;
```

Listas Encadeadas

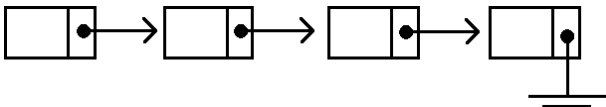
Listas Encadeadas

- Cada item (célula) contém informação necessária para acessar o próximo elemento
- Possibilidade de utilizar posições não contíguas de memória
- Exemplos de aplicações: listas, filas, matrizes esparsas, tabelas de símbolos, gerenciamento de memória, ...
- O primeiro elemento (1ª célula) de uma lista encadeada é denominada cabeça (*head*)



Listas Encadeadas

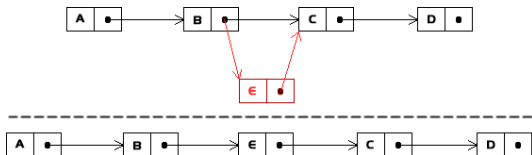
- A navegação em listas encadeadas simples é sempre feita a partir da primeira célula, não sendo possível retornar para uma célula anterior
 - Para não perder a lista, a navegação é feita por meio do uso de uma variável (do mesmo tipo da lista) ponteiro auxiliar



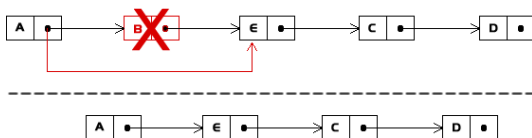
Listas Encadeadas

- A inserção e a remoção de elementos podem ser realizadas sem a necessidade de realocar os demais itens da lista

- Inserção



- Remoção



TAD Lista Encadeada Simples

TAD Lista Encadeada Simples

- Algumas operações com listas encadeadas:
 - Criar uma lista vazia
 - Inserir um item
 - Remover um item
 - Acessar um item
 - Verificar se a lista está vazia
 - Imprimir a lista
 - Liberar lista

- Exemplo de arquivo .h para estrutura do tipo célula:

```
typedef Cell Cell;
```

```
Cell* criar_celula(int key);
```

```
int acessar_celula(Cell* cell);
```

TAD Lista Encadeada Simples

- Exemplo de arquivo .h para estrutura do tipo listaE:

```
typedef struct ListaE ListaE;  
  
ListaE* criar_listaE();  
  
int listaE_vazia(ListaE *l);  
  
int procurar(int key, ListaE *l);  
  
void inserir_primeiro(int key, ListaE *l);  
  
void inserir_ultimo(int key, ListaE *l);  
  
int remover(int key, ListaE *l);  
  
void imprimir(ListaE *l);  
  
int liberar_LE(ListaE *l);
```

- As implementações do TAD acima estão disponíveis no GitHub:
https://github.com/jefferson-oliva/material_grad

- Complexidade de tempo das três principais operações em listas encadeadas:

Operação	Melhor caso	Caso Médio	Pior Caso
Busca	$\Omega(1)$	$\Omega(n)$, $\Theta(n)$ ou $O(n)$	$O(n)$
Inserção	$\Omega(1)$	$\Omega(n)$, $\Theta(n)$ ou $O(n)$	$O(n)$
Remoção	$\Omega(1)$	$\Omega(n)$, $\Theta(n)$ ou $O(n)$	$O(n)$

- Nas implementações (iterativas) apresentadas em aula possuem complexidade de espaço (extra) na ordem de $\Theta(1)$

- Exercício 1: para o TAD de lista encadeada apresentado na aula, implemente as seguintes funcionalidades:
 - a) – Inserção ordenada de elementos de forma crescente
 - b) – Remoção no início
 - c) – Remoção no fim
 - d) – *void split(List *l1, List *l2, List *l3)*: a lista *l1* é dividida em duas listas (*l2* e *l3*)
 - e) – *void concatenate(List *l1, List *l2)*: a lista *l2* deve ser concatenada em *l1*
 - f) – *List* merge(List *l1, List *l2)*: as listas *l1* e *l2* devem ser intercaladas em uma nova lista

Listas Encadeadas (continuação)

- O tamanho da lista não precisa ser definido previamente
- Podem haver combinações de listas de arranjos e encadeadas
- Na implementação de listas encadeadas simples, cada célula aponta para a próxima célula ou NULL
- Há outros tipos de listas encadeadas
 - Duplamente encadeada
 - Circular

Listas Encadeadas

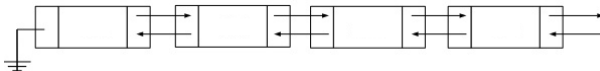
Lista duplamente encadeada

- Cada nó (célula) possui dois ponteiros
 - Um que aponta para o nó antecessor (*previous*) e outro que aponta para o nó sucessor (*next*)
 - Se não for uma lista circular
 - O ponteiro *previous* do primeiro nó é apontado para NULL
 - O ponteiro *next* do último nó é apontado para NULL
- Uma lista duplamente encadeada pode ser circular

Listas Encadeadas

Lista duplamente encadeada

- Exemplo de lista duplamente encadeada



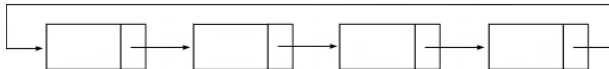
- A principal vantagem da lista encadeada em relação à simples é a facilidade de navegação, que pode ser feita em ambas direções
 - As operações de inserção e remoção são facilitadas
- Se não haver a necessidade de percorrer a lista de trás para frente, a lista encadeada simples é mais vantajosa

- Na lista é formado um ciclo
 - O último elemento aponta para o primeiro em uma lista encadeada circular simples
 - Em uma lista duplamente encadeada circular, além do caso anterior, o primeiro elemento também aponta para o último
- Apesar de não existir, na realidade, primeiro ou último elemento, ainda é necessária a existência de um ponteiro para algum elemento

Listas Encadeadas





Lista circular

- Exemplo de lista encadeada circular simples



- A lista circular pode ser útil quando:
 - A busca pode ser feita a partir de qualquer elemento
 - Não há ordenação na lista

- Vantagens?
 - Não é necessária a definição do tamanho máximo da lista
 - Em operações de inclusão e remoção não é necessário o deslocamento de outras células
- Desvantagens?
 - Pode ser necessário percorrer a lista inteira (mesmo se tiver ordenada) para encontrar um item
 - Cada elemento utiliza maior quantidade de memória, pois em cada célula deve ter pelo menos um ponteiro (se for encadeada simples) para apontar para outra célula

-  Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S.
Algoritmos: teoria e prática.
Elsevier, 2012.
-  Pereira, S. L.
Estrutura de Dados e em C: uma abordagem didática.
Saraiva, 2016.
-  Szwarcfiter, J.; Markenzon, L.
Estruturas de Dados e Seus Algoritmos.
LTC, 2010.
-  Tenenbaum, A.; Langsam, Y.
Estruturas de Dados usando C.
Pearson, 1995.



Ziviani, M.

Projetos de Algoritmos: com implementações em Pascal e C.
Thomson, 2004.