Notas de aula – AED 1 – listas encadeadas
Prof. Jefferson T. Oliva

Nas aulas anteriores, vimos estruturas elementares em sua forma estática, como listas, pilhas e filas.

Na alocação estática, o tamanho dessas listas já é predefinido e o espaço é alocado estaticamente.

A principal vantagem dessas estruturas serem implementadas estaticamente é a simplicidade para realização dessa tarefa. Também, pode haver economia de memória, caso a mesma seja bem gerenciada.

No entanto, listas estáticas têm as seguintes desvantagens:
- 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)

Para contornar esses problemas, as listas podem ter cada elemento alocado dinamicamente. Esse tipo de lista é denominado lista encadeada.

Na lista encadeada, cada elemento é armazenado em uma estrutura conhecida como célula, que veremos daqui a pouco.

Por fim, existem diversos tipos de listas encadeadas: simples, duplamente encadeadas e circulares.

A seguir, veremos algumas definições para a alocação dinâmica de listas.

Alocação Dinâmica de Structs

A partir do uso de ponteiros é possível alocar dinamicamente vetores e matrizes de struct.

Para isso, podem ser utilizadas as mesmas funções (e da mesma) em relação às utilizadas em variáveis comuns, como vimos em outras aulas:
- Malloc()
- Calloc()
- Realloc()
- free()

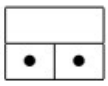
Exemplo de código

```
#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;
}
```

Em listas encadeadas, cada elemento é denominado célula ou nó.

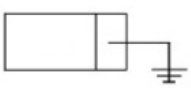
Cada célula é composta por um elemento (estrutura ou apenas chave) e a informação necessária (ponteiro) para acessar uma outra célula. Assim, em uma lista encadeada simples, o ponteiro dentro da célula aponta para a próxima célula.

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

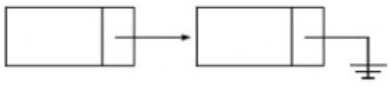


Exemplos de células:

Exemplo de célula apontando para NULL



Exemplo de célula apontando para outra célula



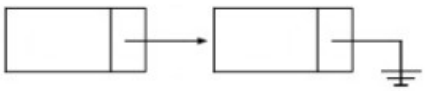
Nota: A célula deve apontar para outra célula do mesmo tipo.

```
typedef struct {
        int key;
}Item;

typedef struct Cell Cell;

struct Cell{
        Item item;
        Cell* next;
};
```

Exemplo 1:

```
Cell *p = (Cell*) malloc(sizeof(Cell));
Item item;

item.key = 10;

p->item = item;

p->next = NULL;
```

Exemplo 2:



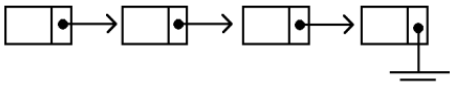
```
Cell *p1 = (Cell*) malloc(sizeof(Cell));
Cell *p2 = (Cell*) malloc(sizeof(Cell));
Item item1, item2;

item1.key = 10;
item2.key = 20;

p1->item = item1;
p2->item = item2;

p2->next = NULL;
p1->next = p2;
```

Listas Encadeadas

Cada item contém informação necessária para acessar o próximo elemento. Nesse tipo de lista, não é necessária a definição do tamanho de lista, pois é possível aumentar o tamanho de uma lista encadeada.

Como uma lista encadeada é composta por células. Para aumentar a lista em uma unidade, por exemplo, basta alocar espaço para uma célula e incorporá-la na lista.

A navegação em listas encadeadas simples é sempre feita a partir da primeira célula, não sendo possível retornar para uma célula anterior. Assim, deve ser utilizada uma variável ponteiro auxiliar para evitar a perda da lista.



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

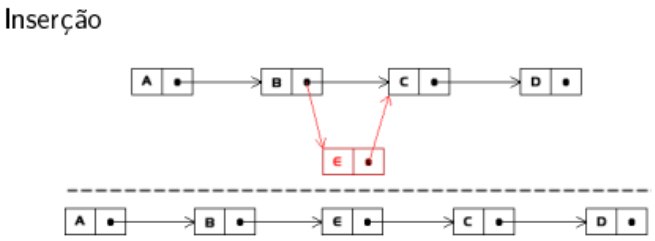
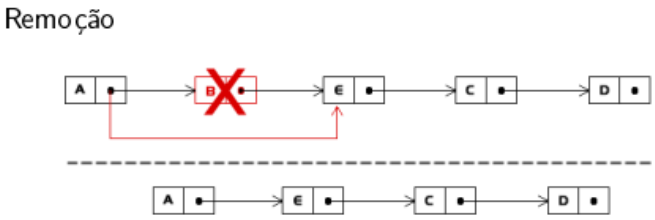


Figure 1: Operação de inserção.



Podem haver combinações entre listas de arranjos e encadeadas. Essa combinação vocês irão ver em matrizes esparsas nessa disciplina e em grafos, na disciplina de AED2.

A principal desvantagem da lista encadeada é a necessidade de alocar um espaço extra na memória para armazenar uma referência (endereço). Esse espaço é um ponteiro e é utilizado para apontar para uma célula ou recebe valor nulo.

Há outros tipos de listas encadeadas: duplamente encadeada e circular.

Lista duplamente encadeada

Cada nó (célula) possui dois ponteiros: um para apontar para a próxima célula e o outro para a célula anterior. Exemplo, instalador de software.

Caso essa lista não seja circular (a última célula aponta para primeira e vice-e-versa): o ponteiro previous da primeira célula e ponteiro next da última são apontadas para NULL.

Em listas encadeadas, os elementos podem estar ordenados (ou não) em ordem crescente ou decrescente.

A principal vantagem da lista encadeada em relação à simples é a facilidade de navegação, que pode ser feita em ambas direções, facilitando a implementação das operações de inserção e remoção.

Por outro lado, se não há a necessidade e percorrer a lista de trás para frente, a lista encadeada simples é mais vantajosa.

Lista circular

Na lista é formado um ciclo: a última célula aponta para a primeira em uma lista circular simples.

Nesse tipo de lista não existe o primeiro ou último elemento, mas ainda é necessária a existência de um ponteiro para a localização de algum elemento.

TAD Lista Encadeada Simples

Alguma 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
- etc.

Arquivo .h

```
typedef struct {
        int key;
}Item;

typedef struct Cell Cell;

struct Cell{
        Item item;
        Cell* next;
};

typedef struct{
        Cell *head;
}List;


List* create_list();

Cell* create_cell(int key);

int is_empty(List *l);

Cell* search(List *l, int key);

void insert_first(List *l, int key);

void insert_last(List *l, int key);

int remove(List *l, int key);

void print(List *l);

void release(List *l);

int length(List *l);
```

Arquivo .c

```
#include <stdlib.h>
#include <stdio.h>
#include "List.h"

List* create_list(){
	List *l = (List*) malloc(sizeof(List));
	l->head = NULL;
	return l;
}


Cell* create_cell(int key){
	Item item;
	Cell* c = (Cell*) malloc(sizeof(Cell));

	item.key = key;

	c->item = item;
	c->next = NULL;

	return c;
}


int is_empty(List *l){
	return l->head == NULL;
}


Cell* search(List *l, int key){
	Cell* aux = NULL;

	if (l != NULL){
		aux = l->head;

		while ((aux != NULL) && (aux->item.key != key))
			aux = aux->next;
	}

	return aux;
}


void insert_first(List *l, int key){
	Cell *node = create_cell(key);

	if (l == NULL)
		l = create_list();

	node->next = l->head;
	l->head = node;
}
```

```
void insert_last(List *l, int key){
	Cell *aux;
	Cell *node = create_cell(key);

	if (l == NULL)
		l = create_list();

	if (is_empty(l))
		l->head = node;
	else{
		aux = l->head;

		while (aux->next != NULL)
			aux = aux->next;

		aux->next = node;
	}
}

int remove(List *l, int key){
	Cell *auxPre, *auxActual;

	if ((l != NULL) && (!is_empty(l))){
		if (l->head->item.key == key){
			auxActual = l->head;
			l->head = auxActual->next;
			free(auxActual);
		}else{
			auxPre = l->head;
			auxActual = auxPre->next;

			while ((auxActual != NULL) && (auxActual->item.key != key)){
				auxPre = auxActual;
				auxActual = auxActual->next;
			}

			if (auxActual != NULL){
				auxPre->next = auxActual->next;
				free(auxActual);
			}
		}
		return 1;
	}

	return 0;
}


void print(List *l){
	Cell *aux = l->head;

	while (aux != NULL){
		printf("%d\n", aux->item.key);
		aux = aux->next;
	}
}
```

```
void release(List *l){
	Cell *aux;

	while (l->head != NULL){
		aux = l->head;
		l->head = aux->next;
		free(aux);
	}

	free(l);
}


int length(List *l){
	int i = 0;
	Cell *aux = l->head;

	while (aux != NULL){
		i++;
		aux = aux->next;
	}

	return i;
}
```

Listas Encadeadas

Vantagens: não é necessária a definição do tamanho máximo da lista (ou seja, a lista pode crescer); e 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 (na lista estática é a mesma coisa); e é utilizada maior quantidade de memória, pois em cada célula deve ter pelo menos um ponteiro (se for encadeada simples) para apontar para outra célula.

Exercícios

Exercício 1: para a TAD de lista estática apresentada neste material, 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 uas listas (l2 e l3)

e) *List* merge(List *l1, List *l2)*: as listas l1 e l2 devem ser concatenadas em uma nova lista

f) *List* intercalate(List *l1, List *l2)*: as listas l1 e l2 devem ser intercaladas em uma nova lista

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

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.