Estruturas de Dados

Aula 6 – Listas Encadeadas, Filas e Pilhas

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia – Campus Feira de Santana

2026

Agenda

Listas

Encadeadas

O que são Listas Encadeadas?

Listas Encadeadas

Em ciência da computação, uma lista ou sequência é uma estrutura de dados abstrata que representa uma coleção ordenada de valores, em que o mesmo valor pode ocorrer mais de uma vez. Uma instância de uma lista é uma representação computacional do conceito matemático de uma sequência finita, que é, uma tupla. Uma lista é armazenada dinamicamente na memória RAM.

Uma lista encadeada possui algumas propriedades:

- Os nodos são criados dinamicamente, à medida que for necessário. Assim, a quantidade total de memória utilizada pela lista depende do número de dados nela armazenados.
- A lista não precisa ocupar uma área de memória contígua: como os nodos são alocados dinamicamente, podem ocupar áreas de memória arbitrárias, e não há nenhuma relação entre a localização dos nodos em memória e sua ordem na lista.

Propriedades das Listas Encadeadas

Ideia

Uma lista encadeada possui algumas propriedades:

- Não é possível indexar os nodos; por isso, para acessar um nodo, deve-se procurá-lo obrigatoriamente a partir do início da lista, seguindo cada nodo até chegar ao procurado.

Propriedades das Listas Encadeadas

Ideia

Uma lista encadeada possui algumas propriedades:

- Não é possível indexar os nodos; por isso, para acessar um nodo, deve-se procurá-lo obrigatoriamente a partir do início da lista, seguindo cada nodo até chegar ao procurado.
- Para adicionar um nodo, basta modificar a referência do nodo que o antecede na lista. Assim, não é necessário "empurrar" os nodos seguintes para frente (como ocorreria em um vetor).

Propriedades das Listas Encadeadas

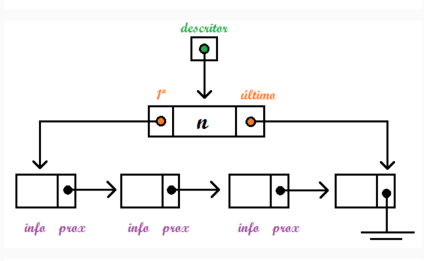
Ideia

Uma lista encadeada possui algumas propriedades:

- Não é possível indexar os nodos; por isso, para acessar um nodo, deve-se procurá-lo obrigatoriamente a partir do início da lista, seguindo cada nodo até chegar ao procurado.
- Para adicionar um nodo, basta modificar a referência do nodo que o antecede na lista. Assim, não é necessário "empurrar" os nodos seguintes para frente (como ocorreria em um vetor).
- Para remover um nodo é a mesma coisa: basta modificar a referência de seu nodo antecessor. Assim, não é necessário "deslocar pra trás" os nodos seguintes (como ocorreria em um vetor).

Listas Encadeadas

A lista possui um descritor que indica o nó inicial e o final da lista, além de indicar o número de elementos (n) que a lista contém.

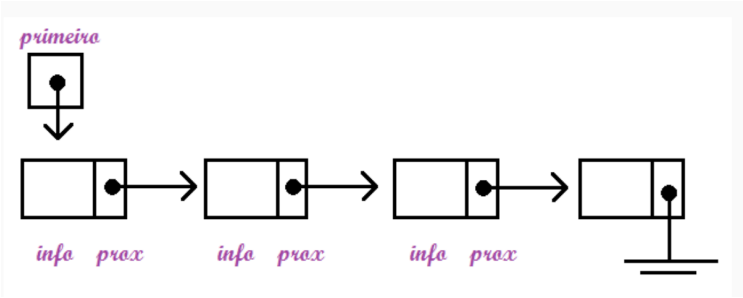


Listas Simplesmente

Encadeadas

Lista Simplesmente Encadeada

Uma Lista Simplesmente Encadeada é uma estrutura de dados em que os objetos estão organizados linearmente; porém, diferentemente de um vetor, não é indexada. É uma estrutura de dados em que os elementos são interligados por ponteiros. Desta forma, os nodos são encadeados de forma unidirecional, pode-se percorrer a lista do primeiro nodo em direção ao último nodo, mas não na direção contrária.



Lista Simplesmente Encadeada

Cada nodo, célula ou elemento da Lista Simplesmente Encadeada é composto pelo conteúdo do nodo e por um ponteiro para o próximo nodo. Assim, uma Lista Simplesmente Encadeada é, na verdade, uma lista de registros. Em linguagem C, uma lista de *Structs*.

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

typedef struct no{
    int         conteudo;
    struct      no *prox;
}celula;
```

Inserir:

O interessante de estruturas de dados que utilizam ponteiros é que não têm limites de tamanho. O limite é o tamanho da memória disponível para armazená-las. Cria-se o primeiro elemento, o topo ou cabeça, e, a partir dele, a lista cresce conforme a necessidade do usuário.

```
celula *inserirNoInicio(int x, celula *topo){
    celula *nova = (celula*) malloc(sizeof(celula));
    if (nova == NULL){
        printf("Erro de alocação!\n");
        exit(1);
    }
    nova->conteudo = x;
    nova->prox = topo;
    return nova;
}
```

Imprimir:

Exibir os elementos inseridos na lista é uma tarefa bem simples e pode ser realizada tanto recursivamente quanto iterativamente. Vejamos as duas soluções possíveis:

```
void imprimir(celula *lista){
    celula *p = lista;
    printf("Lista: ");
    while (p != NULL){
        printf("%d -> ", p->conteudo);
        p = p->prox;
    }
    printf("\n");
}
```

Inserir no FINAL:

Função INSERIR colocando os elementos sempre no final da lista

```
celula *inserirNoFinal(int x, celula *topo){
    celula *nova = (celula*) malloc(sizeof(celula));
    if (nova == NULL){
        printf("Erro de alocação!\n");
        exit(1);
    }
    nova->conteudo = x;
    nova->prox = NULL;
    if (topo == NULL) return nova;
    celula *p = topo;
    while (p->prox != NULL) p = p->prox;

    p->prox = nova;
    return topo;
}
```

Buscar:

A função buscar consiste em percorrer a lista até a célula que contém o conteúdo específico buscado. Assim, a função buscar deverá retornar um ponteiro para a célula da lista que contém o conteúdo de interesse. Esta função também pode ser construída de forma iterativa ou recursiva, vejamos as duas opções:

```
celula *buscar(int x, celula *lista){
    celula *p = lista;
    while (p != NULL && p->conteudo != x)
        p = p->prox;

    return p;
}
```

Buscar:

A função buscar consiste em percorrer a lista até a célula que contém o conteúdo específico buscado. Assim, a função buscar deverá retornar um ponteiro para a célula da lista que contém o conteúdo de interesse. Esta função também pode ser construída de forma iterativa ou recursiva, vejamos as duas opções:

```
celula *buscarRecursivo(int x, celula *lista){
    if (lista == NULL) return NULL;
    if (lista->conteudo == x) return lista;

    return buscarRecursivo(x, lista->prox);
}
```

Remover:

A função remover pode ser construída de uma forma simples, onde passa-se o ponteiro, por valor, do elemento anterior e deseja-se remover o próximo elemento:

```
void remover(celula *p){
   celula *lixo;
   lixo = p->prox;
   p->prox = lixo->prox;
   free (lixo);
}
```

Ou faz-se o buscar e remover:

```
celula *buscar_e_remover(int x, celula *lista){
    if (lista == NULL) return NULL;
    celula *p = lista;
    celula *ant = NULL;
    while (p != NULL && p->conteudo != x){
        ant = p;
        p = p->prox;
    }
    // Elemento não encontrado
    if (p == NULL) return lista;

    // Remoção do primeiro nó
    if (ant == NULL) lista = p->prox;
    else ant->prox = p->prox;

    free(p);
    return lista;
}
```

Listas Simplesmente

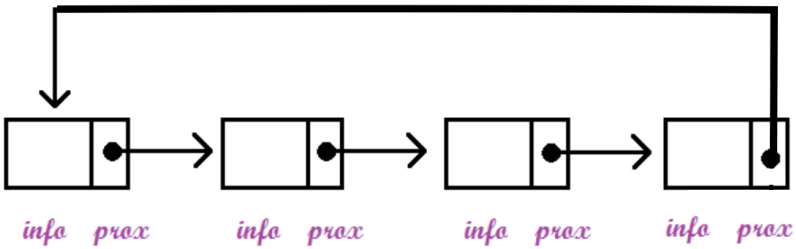
Encadeadas Circulares

Listas Encadeadas Circulares

Uma Lista Encadeada Circular é uma estrutura de dados em que os objetos estão organizados de forma linear, assim como a Lista Simplesmente Encadeada, porém a lista não tem início, meio ou fim. Ela pode iniciar a partir de qualquer um dos seus nós. Assim, o último nó apontará para o primeiro, formando um círculo. Os nodos são encadeados de forma unidirecional, pode-se percorrer a lista em apenas um sentido, mas não na direção contrária. [Cormen et al. 2009]

Listas Encadeadas Circulares

Uma Lista Encadeada Circular é uma estrutura de dados em que os objetos estão organizados de forma linear, assim como a Lista Simplesmente Encadeada, porém a lista não tem início, meio ou fim. Ela pode iniciar a partir de qualquer um dos seus nós. Assim, o último nó apontará para o primeiro, formando um círculo. Os nodos são encadeados de forma unidirecional, pode-se percorrer a lista em apenas um sentido, mas não na direção contrária. [Cormen et al. 2009]



Listas Encadeadas Circulares

Assim como a Lista Simplesmente Encadeada, a Lista Encadeada Circular é uma lista de registros, composta de conteúdo e um ponteiro para o próximo elemento da lista. Em linguagem C, uma lista de *Structs*.

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

typedef struct no{
    int         conteudo;
    struct      no *prox;
}celula;
```

O elemento pode ser inserido em qualquer posição da lista. O limite do tamanho da lista é o tamanho da memória disponível.

Listas Encadeadas Circulares

```
celula *inserir(int x, celula *topo){
    celula *nova = (celula*) malloc(sizeof(celula));
    if (nova == NULL){
        printf("Erro de memória\n"); exit(1);
    }
    nova->conteudo = x;
    if (topo == NULL){  // Lista vazia
        nova->prox = nova;
        return nova;
    }
    celula *aux = topo;  // Encontrar último nó
    while (aux->prox != topo){
        aux = aux->prox;
    }
    nova->prox = topo;
    aux->prox = nova;
    return nova; // novo topo
}
```

Imprimir:

Exibir os elementos inseridos na lista é uma tarefa bem simples:

```
void imprimir(celula *topo){
    if (topo == NULL){
        printf("Lista vazia\n");
        return;
    }
    celula *p = topo;
    do {
        printf("%d -> ", p->conteudo);
        p = p->prox;
    } while (p != topo);
    printf("(volta ao início)\n");
}
```

Buscar:

A função buscar consiste em percorrer a lista até a célula que contém o conteúdo específico buscado. Assim, a função buscar deverá retornar um ponteiro para a célula da lista que contém o conteúdo de interesse:

```
celula *buscar(int x, celula *topo){
    if (topo == NULL) return NULL;
    celula *aux = topo;
    do {
        if (aux->conteudo == x)
            return aux;
        aux = aux->prox;
    } while (aux != topo);
    return NULL;
}
```

Listas Encadeadas Circulares

```
celula *buscar_e_remover(int x, celula *topo){
    if (topo == NULL) return NULL;
    celula *p = topo;
    celula *ant = NULL;
    do {
        if (p->conteudo == x) break;
        ant = p;
        p = p->prox;
    } while (p != topo);

    // Não encontrou
    if (p->conteudo != x) return topo;

    // Caso: único elemento
    if (p == topo && p->prox == topo){
        free(p);
        return NULL;
    }
    //continua no próximo slide
```

```
    // Caso: remover topo
    if (p == topo){
        celula *ultimo = topo;
        while (ultimo->prox != topo){
            ultimo = ultimo->prox;
        }
        topo = topo->prox;
        ultimo->prox = topo;
        free(p);
        return topo;
    }
    // Caso geral
    ant->prox = p->prox;
    free(p);
    return topo;
}
```

Listas Duplamente

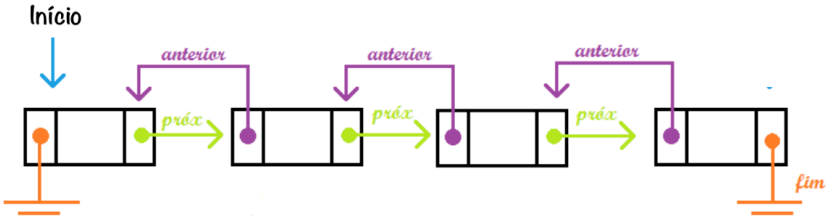
Encadeadas

Listas Duplamente Encadeadas

Uma Lista Duplamente Encadeada é uma estrutura de dados semelhante a uma lista simplesmente encadeada, porém, além de o nó apontar para o próximo, ele também aponta para o nó anterior. Assim, permite que a lista seja percorrida em ambos os sentidos. [Cormen et al. 2009]

Listas Duplamente Encadeadas

Uma Lista Duplamente Encadeada é uma estrutura de dados semelhante a uma lista simplesmente encadeada, porém, além de o nó apontar para o próximo, ele também aponta para o nó anterior. Assim, permite que a lista seja percorrida em ambos os sentidos. [Cormen et al. 2009]



Listas Duplamente Encadeadas

Cada nodo, célula ou elemento da Lista Duplamente Encadeada é composto pelo conteúdo do nodo e por dois ponteiros para o nodo seguinte e o nodo anterior. Assim, uma Lista Duplamente Encadeada é, na verdade, uma lista de registros. Em linguagem C, uma lista de *Structs*.

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

typedef struct reg{
    int         conteudo;
    struct      reg *prox;
    struct      reg *ant;
}celula;
```

Inserir:

Na Lista Duplamente Encadeada, deve-se tomar cuidado ao inserir, pois tanto o ponteiro do elemento anterior quanto o do elemento próximo devem ser manipulados corretamente.

```
celula *inserir(int x, celula *topo){
    celula *nova = (celula*) malloc(sizeof(celula));
    if (nova == NULL){
        printf("Erro de memória\n");
        exit(1);
    }
    nova->conteudo = x;
    nova->ant = NULL;
    nova->prox = topo;
    if (topo != NULL){
        topo->ant = nova;
    }
    return nova;}
```

Imprimir:

Exibir os elementos inseridos na lista é uma tarefa bem simples. No caso da Lista Duplamente Encadeada, pode ser feita em ordem crescente ou decrescente:

```
void imprimir(celula *lista){
    celula *p = lista;
    while (p != NULL){
        printf("%d <-> ", p->conteudo);
        p = p->prox;
    }
    printf("NULL\n");
}
```

Buscar:

A função buscar consiste em percorrer a lista até a célula que contém o conteúdo específico buscado. Assim, a função buscar deverá retornar um ponteiro para a célula da lista que contém o conteúdo de interesse. Esta função também pode ser construída de forma iterativa ou recursiva e no caso da Lista Duplamente Encadeada, a lista pode ser percorrida em qualquer sentido, vejamos as duas opções:

Buscar:

```
celula *buscar(int x, celula *lista){
    while (lista != NULL && lista->conteudo != x){
        lista = lista->prox;
    }
    return lista;
}

celula *busca_r(int x, celula *listaDuplaEncadeada){
   if (listaDuplaEncadeada == NULL)  return NULL;
   if (listaDuplaEncadeada->conteudo == x)  return listaDuplaEncadeada;
   return busca_r(x, listaDuplaEncadeada->prox);
}
```

Buscar e Remover:

A função remover pode ser construída informando o conteúdo a ser excluído e buscando-o na lista, excluindo a célula correspondente.

```
celula *buscar_e_remover(celula *topo, celula *p){
    if (p == NULL) return topo;
    // remover primeiro
    if (p->ant == NULL){
        topo = p->prox;
        if (topo != NULL)
            topo->ant = NULL;
    }else{
        p->ant->prox = p->prox;
        if (p->prox != NULL)
            p->prox->ant = p->ant;
    }
    free(p);
    return topo;}
```

Listas Duplamente

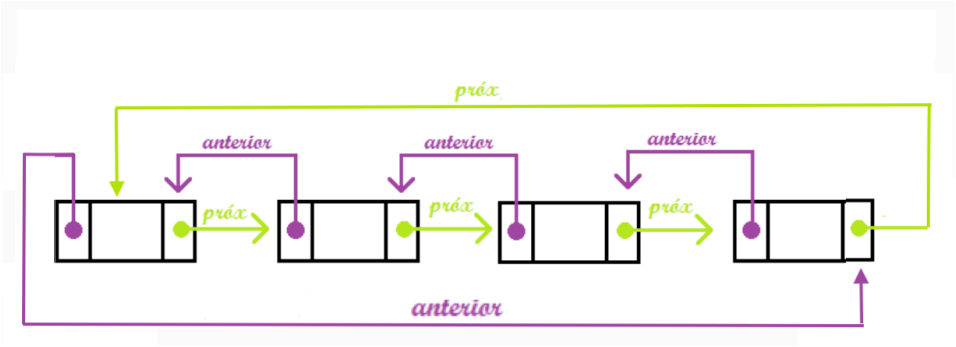
Encadeadas Circulares

Listas Duplamente Encadeadas Circulares

Uma Lista Duplamente Encadeada Circular é uma estrutura de dados semelhante a uma lista duplamente encadeada e a uma lista circular. É basicamente uma lista duplamente encadeada cujo último nó aponta para o primeiro; assim, não há início nem fim. [Cormen et al. 2009]

Listas Duplamente Encadeadas Circulares

Uma Lista Duplamente Encadeada Circular é uma estrutura de dados semelhante a uma lista duplamente encadeada e a uma lista circular. É basicamente uma lista duplamente encadeada cujo último nó aponta para o primeiro; assim, não há início nem fim. [Cormen et al. 2009]



Listas Duplamente Encadeadas Circulares

Cada nodo, célula ou elemento da Lista Duplamente Encadeada Circular é composto pelo conteúdo do nodo e por dois ponteiros para o nodo seguinte e para o nodo anterior. Assim, uma Lista Duplamente Encadeada Circular é, na verdade, uma lista de registros, assim como as anteriores. Em linguagem C, uma lista de *Structs*.

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

typedef struct registro{
    int conteudo;
    struct registro *ant;
    struct registro *prox;
}celula;
```

Inserir:

Na Lista Duplamente Encadeada Circular, deve-se tomar cuidado ao inserir, pois tanto o ponteiro do elemento anterior quanto o do elemento próximo devem ser manipulados corretamente; não há ponteiro nulo, como a lista duplamente encadeada. O primeiro elemento da lista, quando isolado, aponta para o anterior e para o próximo.

Listas Duplamente Encadeadas Circulares

```
celula *inserir(int x, celula *topo){
    celula *novo = (celula*) malloc(sizeof(celula));
    if (novo == NULL){
        printf("Erro de memória\n");
        exit(1);
    }
    novo->conteudo = x;
    // Lista vazia
    if (topo == NULL){
        novo->prox = novo;
        novo->ant  = novo;
        return novo;
    }
    celula *ultimo = topo->ant;
    novo->prox = topo;
    novo->ant  = ultimo;
    ultimo->prox = novo;
    topo->ant = novo;
    return novo; // novo topo
}
```

Imprimir:

Exibir os elementos inseridos na lista é uma tarefa simples. No caso da Lista Duplamente Encadeada Circular, pode ser feita em qualquer ordem, tomando cuidado para não entrar em loop:

```
void imprimir(celula *topo){
    if (topo == NULL){
        printf("Lista vazia\n");
        return;}
    celula *p = topo;
    printf("Lista: ");
    do{
        printf("%d <-> ", p->conteudo);
        p = p->prox;
    } while (p != topo);
    printf("(circular)\n");
}
```

Buscar:

A função buscar consiste em percorrer a lista até a célula que contém o conteúdo específico buscado. Assim, a função buscar deverá retornar um ponteiro para a célula da lista que contém o conteúdo de interesse. Esta função também pode ser construída de forma iterativa e no caso da Lista Duplamente Encadeada Circular, a lista pode ser percorrida em qualquer sentido:

```
celula *buscar(int x, celula *topo){
    if (topo == NULL) return NULL;
    celula *aux = topo;
    do{
        if (aux->conteudo == x)
            return aux;
        aux = aux->prox;
    } while (aux != topo);
    return NULL;
}
```

```
celula *buscar_remover(celula *topo, celula *lixo){
    if (topo == NULL || lixo == NULL) return topo;
    // único elemento
    if (lixo->prox == lixo){
        free(lixo);
        return NULL;
    }
    // ajustar vizinhos
    lixo->ant->prox = lixo->prox;
    lixo->prox->ant = lixo->ant;
    // se remover topo
    if (lixo == topo) topo = lixo->prox;

    free(lixo);
    return topo;
}
```

Filas

FIFO - First In First Out

A Fila consiste numa Estrutura de Dados que funciona segundo a ordem FIFO (First In First Out), ou seja, o primeiro a entrar é o primeiro a sair. A Fila pode ser implementada com vetores ou listas, ou seja, com ou sem ponteiros, contanto que obedeça a regra FIFO. Aqui, na disciplina, usaremos ponteiros em C para implementar nossa Fila.

As filas são amplamente utilizadas na computação em diversos campos, como filas de processos em Sistemas Operacionais, de pacotes em Redes de Computadores, entre outros exemplos. Então, nenhum profissional de computação deve se eximir de aprender, com maestria, a manipular filas. Na verdade, nenhuma estrutura de dados!

Basicamente, quando pensamos em filas, pensamos em duas ações importantes: **Enfileira** e **Desenfileira**, ou seja, o elemento chegou e entrará no fundo da fila (enfileira) e o elemento já esperou na fila e agora está no topo e chegou a hora de ser atendido (desenfileira).

Struct:

A *STRUCT* de uma fila pode ser construída de forma idêntica a uma lista duplamente encadeada.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct NODO{
    int     conteudo;
    struct  NODO *prox;
    struct  NODO *ant;
}celula;
```

Enfileirar: A função **Enfileirar**, que corresponde a função “inserir" na fila, deve ser realizada de modo que o elemento inserido seja sempre colocado no “FINAL" da fila.

```
celula *enfileirar(int x, celula *fila){
    celula *nova = (celula*) malloc(sizeof(celula));
    if (nova == NULL){
        printf("Erro de memória\n");
        exit(1);
    }
    nova->conteudo = x;
    nova->ant = NULL;
    nova->prox = fila;

    if (fila != NULL) fila->ant = nova;

    return nova; // novo início
}
```

```
celula *desenfileirar(celula *fila){
    if (fila == NULL){
        printf("Fila vazia!\n");
        return NULL;
    }
    celula *p = fila;
    // encontrar último
    while (p->prox != NULL) p = p->prox;
    // único elemento
    if (p->ant == NULL){
        free(p);
        return NULL;
    }
    // mais de um elemento
    p->ant->prox = NULL;
    free(p);
    return fila;
}
```

Buscar:

A função buscar na fila percorre a fila e retorna um ponteiro para o elemento buscado.

```
celula *buscar(int x, celula *fila){
    while (fila != NULL && fila->conteudo != x){
        fila = fila->prox;
    }
    return fila;
}
```

Imprimir:

A função imprimir exibe todos os elementos da fila na ordem do último para o primeiro, como mostrado na figura abaixo.

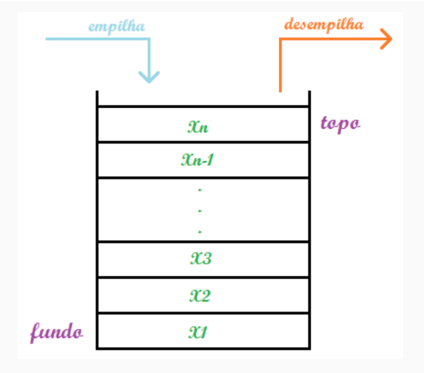

```
void imprimir(celula *fila){
    celula *p = fila;
    printf("Fila: ");
    while (p != NULL){
        printf("%d <- ", p->conteudo);
        p = p->prox;
    }
    printf("NULL\n");
}
```

Pilhas

LIFO - Last In First Out **OU** **FILO** - First In - Last Out

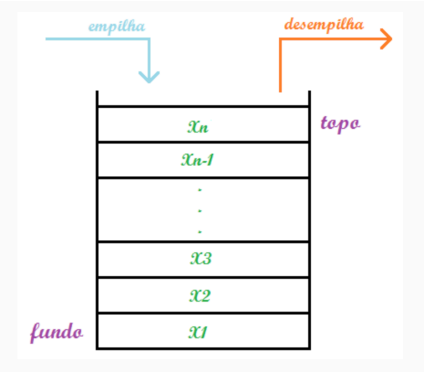
Pilhas

Pilha é uma estrutura de dados que obedece a regra de que os elementos são removidos na ordem inversa à de inserção, de modo que o último elemento a entrar é sempre o primeiro a ser retirado; por isso, este tipo de estrutura de dados é chamado LIFO (Last In - First Out) ou FILO (First In - Last Out).

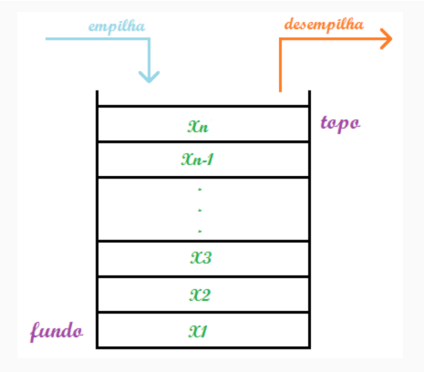


Pilhas

O exemplo mais prático para entender uma pilha é uma pilha de livros ou de pratos, em que, ao colocar diversos elementos uns sobre os outros, se quisermos pegar o livro ou o prato mais abaixo, devemos tirar todos os livros ou pratos que estão por cima. Outro exemplo é um elevador: a primeira pessoa a entrar, normalmente, é a última a sair.



Basicamente, quando pensamos em pilhas, pensamos em duas ações importantes: **Empilha** e **Desempilha**, ou seja, o elemento chegou e entrará em cima da pilha (empilha) e será atendido e sairá da pilha (desempilha).



Struct, Imprimir e Buscar:

A *STRUCT*, a função imprimir e a função buscar da Pilha são idênticas às da Fila; portanto, não as repetiremos aqui. Podemos consultar os Slides anteriores ou o Código fonte.

Empilhar:

A função **Empilhar**, que corresponde à função "inserir" na pilha, deve ser realizada de modo que o elemento inserido seja sempre colocado no "TOPO" da pilha.

```
celula *empilhar(int x, celula *pilha){
    celula *nova = (celula*) malloc(sizeof(celula));
    if (nova == NULL){
        printf("Erro de memória\n");
        exit(1);
    }
    nova->conteudo = x;
    nova->ant = NULL;
    nova->prox = pilha;
    if (pilha != NULL){
        pilha->ant = nova;
    }
    return nova; // novo topo
}
```

Desempilhar:

A função **Desempilhar**, que corresponde a função “remover" da pilha, deve ser realizada de modo que o elemento removido seja sempre retirado do “TOPO" da pilha.

```
celula *desempilhar(celula *pilha){
    if (pilha == NULL){
        printf("Pilha vazia!\n");
        return NULL;
    }
    celula *novoTopo = pilha->prox;
    if (novoTopo != NULL)
        novoTopo->ant = NULL;
    free(pilha);
    return novoTopo;
}
```

Referências I

CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262032937.