Pilhas

Armazenamento de dados:

Variáveis

- struct (classe)
- Primitivas ou não

Lista encadeada

- É uma forma de armazenamento
- Alternativo ao vetor

Estrutura de dados

- É uma regra de manipulação de dados

Técnicas de estrutura de dados:

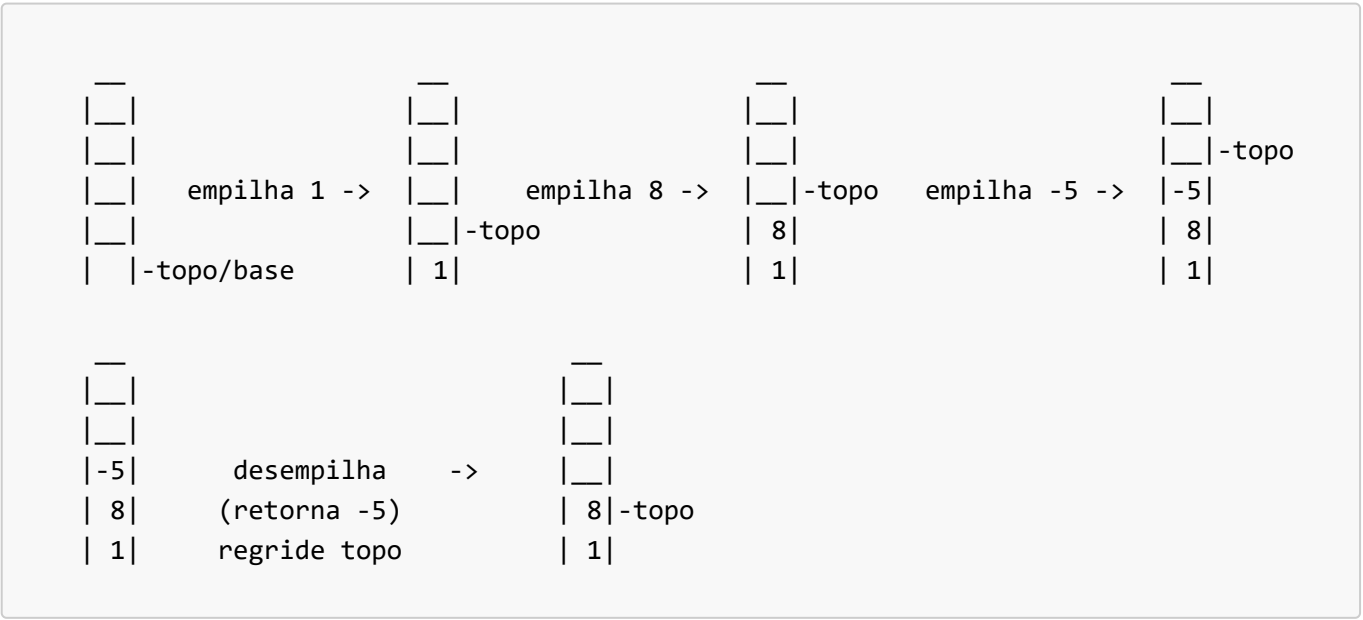
- Armazenar os dados de forma conveniente a uma aplicação específica
- Ex: Vetor ordenado (motivos: para busca eficiente, busca binária....)
- Principais tipos: **Pilhas, filas e árvores**

Pilha

- Colocar um elemento SOBRE o outro
- Remove e insere o elemento de CIMA

REGRA: O último elemento que entra é o primeiro que sai (LIFO: last-in, first-out)

Premissa: Elementos são inseridos ou removidos UM POR VEZ



Implementação

- Forma de armazenamento (vetor ou lista encadeada)
- Indicador para o topo

*Operações: Numa pilha as operações permitidas são apenas **INSERÇÃO** e **REMOÇÃO***

Implementação com VETORES

- É bom evitar a realocação - Dá trabalho para o sistema operacional
- Elementos necessários: Vetor, tamanho e indicador do topo

1. Representação:

```
typedef struct {
   int *dado; // vetor
   int N; // tamanho do vetor
   int topo;  // indicador do topo
} pilha // nome da struct
```

2. Criando a pilha:

- TOPO INDICA A POSIÇÃO VAZIA
- Topo = -1 ---> 1° incrementa o topo; 2° insere o elemento na posição topo (indica que não tem ninguém)
- Topo = 0 ---> 1° insere o elemento; 2° incrementa topo

```
// Função de criação da pilha: Para isso é usado ponteiros - 1° aloca depois
desaloca
    pilha *cria_pilha(){
        pilha *p = malloc(sizeof(pilha)); // criando a pilha
        p->n=10; // definindo o tamanho
        p->dado=malloc(p->N*sizeof(int)); // alocando o vetor e definindo o tipo
        p->topo = 0;//inicializando o top (0 ou -1)
        return p;
    }
```

3. Inserção na pilha

- Topo = 0
- Para inserir: É preciso colocar o elemento na posição top (posição livre) e depois incrementar o topo (+1)

```
    void empilha(pilha *p, int x){ // pilha *p = sctruct = vetor com
 tamanho, topo,  já alocada/inicializada
```

```
        p->dado[p->topo]=x; // pega a pilha e coloca o x a posição topo
        p->topo++;
    }
```

- *Quando empliha pode falhar? Quando o topo atingir o tamanho do vetor* ***(topo==N)***

```
    // pilha responsiva -> Retornando um int
    int empilha (pilha *p, int x){
        if (p->topo==p->n){ //topo é igual a n?
            // se for
            p-> dado= realloc(p->dado, 2*p->N); // vetor, novo tamanho

            // Caso de errado retorna 0
            if (p->dado==NULL) return 0;// quando o realloc da errado
 ele retorna NULL

            //Se deu certo é preciso atualizar o valor de N
            p->N*=2;
        }
        p->dado[p->topo]=x;
        p->topo++;
        return 1; // retornando 1 ta tudo certo
    }
```

4. Remoção

```
       int desempilha(pilha *p, int x){ // int, pois precisa retornar o
valor desempilhado
           p->topo--; // 1° decrementa o topo pois ele aponta para posição
vazia
           return p-> dado[p->topo];
       }
```

- *Quando empliha pode falhar?? Quando a pilha estiver vazio (p->topo==0)*

```
        int desempilha(pilha *p, int *y){
            if (p->topo==0) return 0; //não tem o q fazer

            p->topo--;
            *y = p-> dado[p->topo];
            return 1;
        }
```

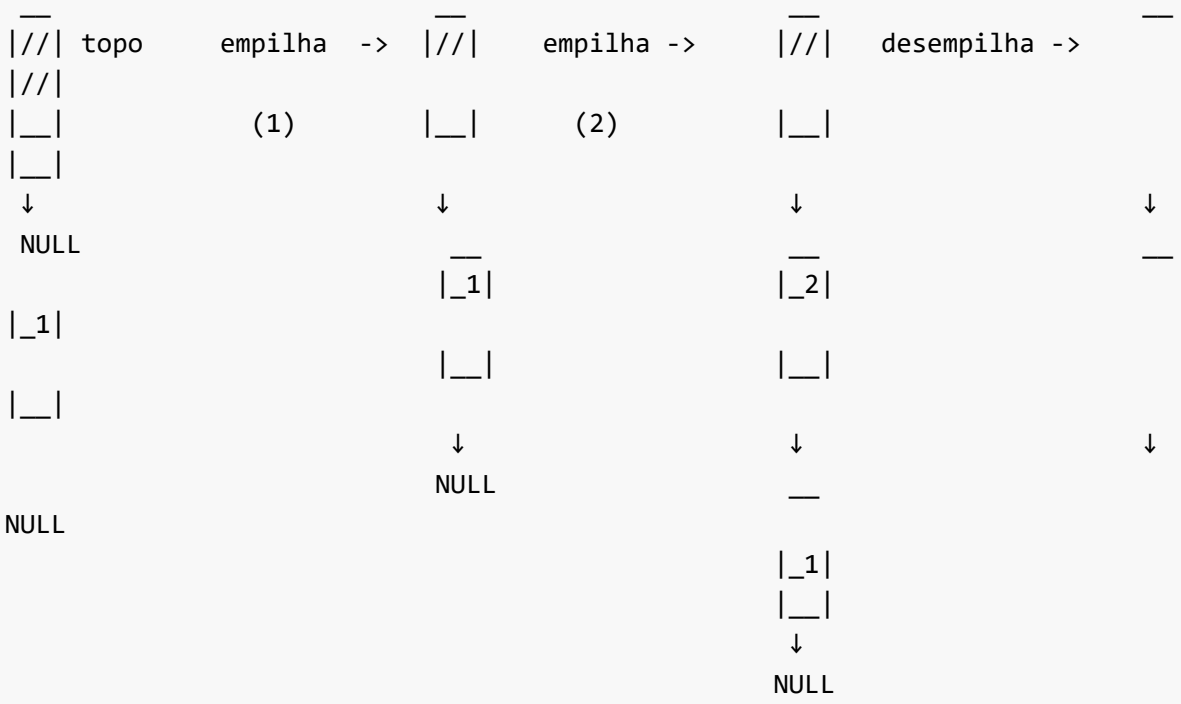
5. Como utilizar?

```
    pilha *p = cria_pilha();
    empilha(p,1);
    empilha (p,10);
    empilha (p,13);
    int y;
    desempilha (p,&y); // y=13 que foi empilhado por último
    // Podemos verificar ainda os retornas das funções

    // Quando terminar é preciso limpar o que alocamos
    free(p->dado);
    free(p);
```

Implementação com LISTA ENCADEADA

- Mais vantajoso que o vetor por NÃO TEM LIMITE de tamanho - favore a inserção e remoção de elementos
- O nó cabeça representa o topo da pilha
- Para empilhar é preciso inserir o elemento no início
- Para remover, removemos do início



Usando uma lista encadeada

1. Representação

```
typedef struct no{
    int dado;
    struct no *prox;
} no;
```

2. Criação

```
// alocar o nó cabeça
no *cria_pilha(){
    no *topo=malloc(sizeof(no));
    topo -> prox=NULL;
}
```

3. Inserção

```
int empilha(no *topo, int x){
    no *novo=malloc(sizeof(no));
    if(novo==NULL) return 0;//DA ERRADO
    novo->dado = x;
    novo->prox = topo->prox;
    topo->prox = novo;
    return 1;
}
```

4. Remoção

```
int desempilha(pilha *topo, int *y){
    no *lixo = topo->prox;
    if (lixo==NULL) return 0;
    *y = lixo->dado;
    topo=>prox = lixo->prox;
    free(lixo);
    return 1;
}
```

5. Usando na main

```
no *topo=cria_pilha();
int y;
empilha(topo, 1);
empilha(topo,10);
desempilha(topo,&y);
...
// limpar
```

```
    while (desempilha(topo,&y)); //while vazio que chama o desempilha
enquanto retornar 1
    free(topo);
```

OBS: Segmentação

- Tenta acessar algo que não existe ou algo que não tem
- NULL não aponta para nada -> não existe struct

Aplicação da pilha

Para que serve uma pilha?

- Utilizamos uma pilha quando queremos construir uma memória de dados e recupeá-los na ordem inversa da qual foram salvos.
- Para construir uma memória de tal forma que os dados sejam recuperados na ordem inversa a que foram armazenados.
- **Exemplos**: Empilhar uma string e imprimir na tela a sequência de desempilhamento considere a string "roma"
    - Empilho todos os caracteres da string
    - Desempilho e imprimo até esvaziar

```
 __
|_a|-> topo
|_m|          = "amor"
|_o|
|_r|
```

- **Exemplos comuns:**
    - Inverter um vetor qualquer;
    - Ctrl-z e refazer; Torre de Hanoi;
    - *RECURSÃO e CHAMADA DE FUNÇÃO* - Gerenciamento de processos, Chamadas de sistemas operacionais...
    - A forma mais natural de se escrever um algoritmo recursivo de forma iterativa é usando pilhas

Considere o código:

```
int f3(){
     ⋮
}

void f2(){
     ⋮
}

void f1(){
     ⋮
}

int main(){
     ⋮
    f1;
     ⋮
}
```

↳ Pilha de execução de sistema

- Na ida vai empilhar, na volta vai desempilhar
- Toda interação recursiva pode ser escrito como pilha

```
 ____
|_f3_|
|_f2_|
|_f1_|
|main|
```

Validação/execução de expressão matemática

- Ex: Os fechas aparecem na ordem inversa dos abre
  - { A + [C - D * (A + E * C) + J] } ✓
  - A + (B - C]) ✗
  - A + [(B - C]) ✗
  - A + [(B + C) ✗

    Abre: { → [→ (

```
|   |
| ( | ↓ desempilha === fecha => ( → [ → {
| [ | ↓
| { | ↓
```