Pontifícia Universidade Católica
do Rio de Janeiro

INF 1007 – Programação II

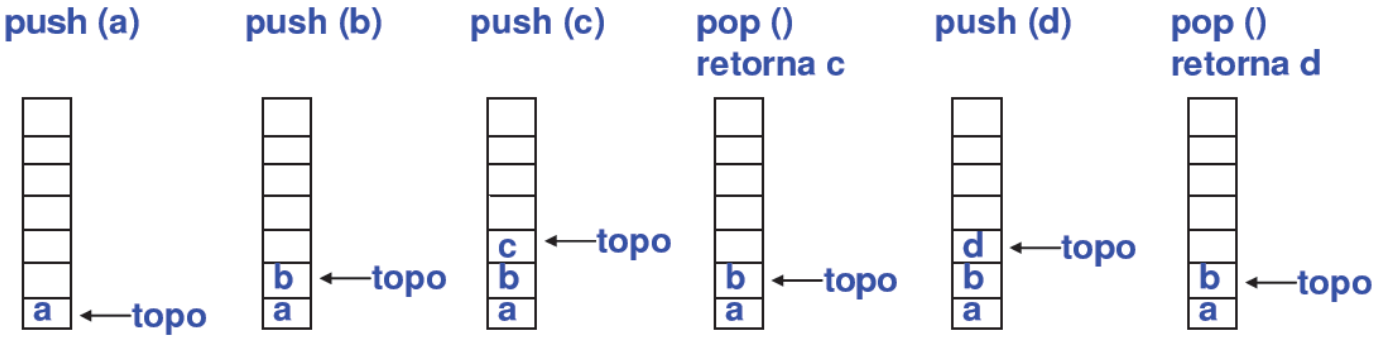
Aula 13 – Pilhas

Edirlei Soares de Lima
<elima@inf.puc-rio.br>

visionlab
visualization laboratory

Pilha

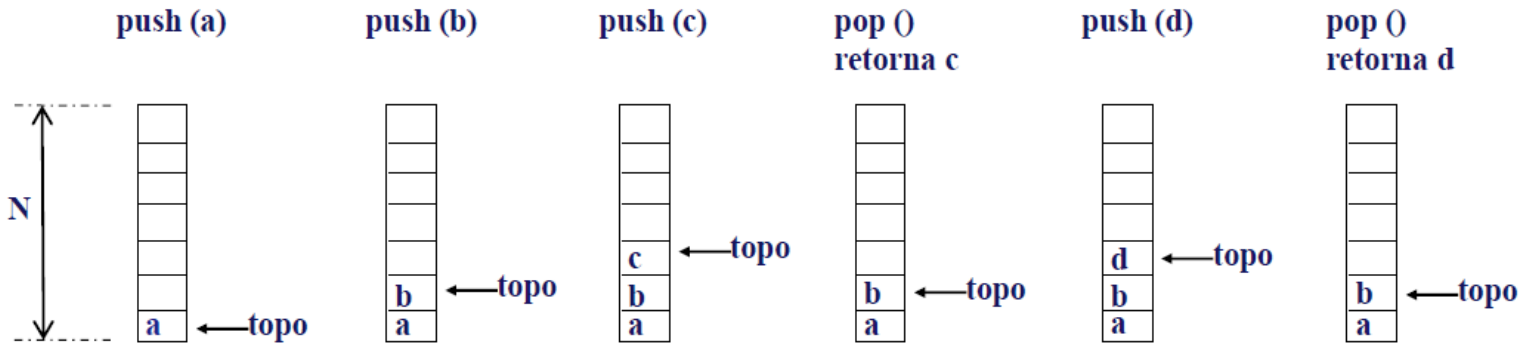
- Uma pilha é uma estrutura de dados dinâmica na qual novos elementos são sempre **inseridos no topo** da pilha e **acessados somente pelo topo**.
  - O único elemento que pode ser acessado e removido é o do topo;
  - Elementos são retirados na ordem inversa à ordem em que foram colocados;
  - O primeiro que sai é o último que entrou (LIFO – last in, first out)
- Operações básicas
  - Empilhar (**push**) um novo elemento, inserindo-o no topo;
  - Desempilhar (**pop**) um elemento, removendo-o do topo;



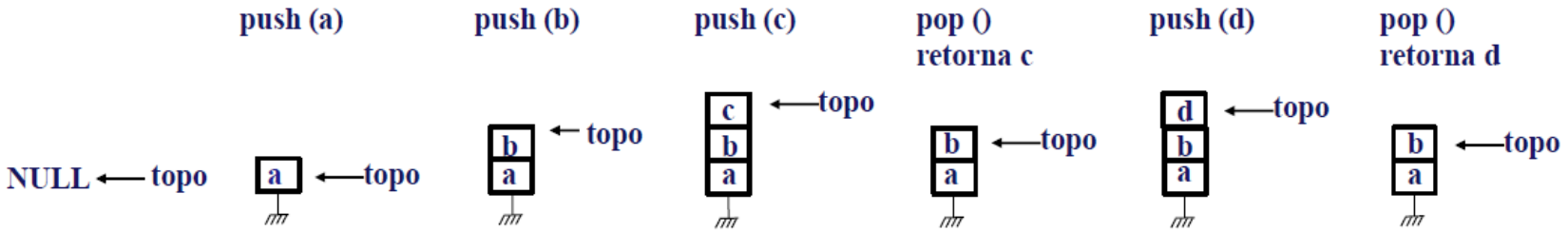
Pilha

- Uma estrutura do tipo Pilha pode ser implementada como **Vetor** ou **Lista Encadeada**:

- **Vetor:**



- **Lista Encadeada:**



Interface do Tipo Pilha

- **Interface do TAD Pilha**: `pilha.h`
  - função `pilha_cria`:
    - aloca dinamicamente a estrutura da pilha;
    - inicializa seus campos e retorna seu ponteiro;
  - funções `pilha_push` e `pilha_pop`:
    - inserem e retiram, respectivamente, um valor real na pilha;
  - função `pilha_vazia`:
    - informa se a pilha está ou não vazia;
  - função `pilha_libera`:
    - destrói a pilha, liberando toda a memória usada pela estrutura.

Interface do Tipo Pilha

```
/* TAD: pilha de valores reais (float) */

typedef struct pilha Pilha;
/* Tipo Pilha, definido na interface, depende da
implementação do struct pilha */

Pilha* pilha_cria();

void pilha_push(Pilha* p, float v);

float pilha_pop(Pilha* p);

int pilha_vazia(Pilha* p);

void pilha_libera(Pilha* p);
```

Exercício 1

- **Crie uma função para verificar expressões matemáticas:**
  - Considerando cadeias de caracteres com expressões matemáticas que podem conter termos entre parênteses, colchetes ou chaves, ou seja, entre os caracteres '(' e ')', ou '[' e ']', ou '{' e '}';
  - A função retorna 1, se os parênteses, colchetes e chaves de uma expressão aritmética são abertos e fechados corretamente, ou 0 caso contrário;
  - Para a expressão "`2*{3+4*(2+5*[2+3])}`" a função deve retornar 1;
  - Para a expressão "`2*(3+4+{5*[2+3}])`" a função deve retornar 0;

- Protótipo:

```
int verifica(char* exp);
```

Exercício 1

- A estratégia para resolver esse problema é percorrer a expressão da esquerda para a direita:
  - Se encontra '(', '[' ou '{', empilha;
  - Se encontra ')', ']' ou '}', desempilha e verifica o elemento no topo da pilha, que deve ser o caractere correspondente;
  - Ao final, a pilha deve estar vazia.

- Protótipo:

```
int verifica(char* exp);
```

```
char fecho(char c) {
  if(c == '}') return '{';
  if(c == ']') return '[';
  if(c == ')') return '(';
}
int verifica(char* exp) {
  Pilha* p = pilha_cria();
  int i;
  for(i=0; exp[i]!='\0'; i++)
  {
    if(exp[i] == '{' || exp[i] == '[' || exp[i] == '(')
      pilha_push(p, exp[i]);
    else if(exp[i] == '}' || exp[i] == ']' || exp[i] == ')')
    {
      if(pilha_vazia(p)) return 0;
      if(pilha_pop(p) != fecho(exp[i])) return 0;
    }
  }
  if(!pilha_vazia(p)) return 0;
  pilha_libera(p);
  return 1;
}
```

Implementação de Pilha com Vetor

- Implementação de pilha com **VETOR**:
  - vetor (vet) armazena os elementos da pilha;
  - elementos inseridos ocupam as primeiras posições do vetor;
    - elemento vet[n-1] representa o elemento do topo.

```
#define N 50        /* número máximo de elementos */

struct pilha {
  int topo;     /* vet[topo]: primeira posição livre do vetor */
  float vet[N];    /* vet[topo-1]: topo da pilha */
                   /* vet[0] a vet[N-1]: posições ocupáveis */
};
```

Implementação de Pilha com Vetor

- **função `cria_pilha`:**
  - aloca a estrutura da pilha;
  - inicializa a lista como sendo vazia;

```
Pilha* pilha_cria()
{
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->topo = 0;
  return p;
}
```

Implementação de Pilha com Vetor

- **função `pilha_push`:**
  - insere um elemento na pilha;
  - usa a próxima posição livre do vetor, se houver;

```
void pilha_push (Pilha* p, float v)
{
  if (p->topo == N) {     /* capacidade esgotada */
    printf("Capacidade da pilha estourou.\n");
    exit(1);              /* aborta programa */
  }
  /* insere elemento na próxima posição livre */
  p->vet[p->topo] = v;
  p->topo++;
}
```

Implementação de Pilha com Vetor

- **função `pilha_pop`:**
  - retira o elemento do topo da pilha, retornando o seu valor;
  - verificar se a pilha está ou não vazia;

```
float pilha_pop (Pilha* p)
{
  float v;
  if (pilha_vazia(p)) {
    printf("Pilha vazia.\n");
    exit(1); /* aborta programa */
  }
  /* retira elemento do topo */
  v = p->vet[p->topo-1];
  p->topo--;
  return v;
}
```

Implementação de Pilha com Vetor

- **função `pilha_vazia`:**
  - retorna 1, se a pilha está vazia, ou 0, caso contrário;

```
int pilha_vazia (Pilha* p)
{
  if(p->topo == 0)
    return 1;
  return 0;
}
```

Implementação de Pilha com Vetor

- **função `pilha_libera`:**
  - libera libera a pilha;

```
void pilha_libera (Pilha* p)
{
  free(p);
}
```

Implementação de Pilha com Lista

- Implementação de pilha com **LISTA**:
  - elementos da pilha armazenados na lista;
  - pilha representada por um ponteiro para o primeiro nó da lista;

```
/* nó da lista para armazenar valores inteiros */
struct elemento {
  float info;
  struct elemento *prox
};
typedef struct elemento Elemento;

/* estrutura da pilha */
struct pilha {
  Elemento* topo;  /* aponta para o topo da pilha */
};
```

Implementação de Pilha com Lista

- **função `cria_pilha`:**
  - aloca a estrutura da pilha;
  - inicializa a lista como sendo vazia;

```
Pilha* pilha_cria(void)
{
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->topo = NULL;
  return p;
}
```

Implementação de Pilha com Lista

- **função `pilha_push`:**
  - insere novo elemento n no início da lista;

```
void pilha_push (Pilha* p, float v)
{
  Elemento* n = (Elemento*) malloc(sizeof(Elemento));
  n->info = v;
  n->prox = p->topo;
  p->topo = n;
}
```

Implementação de Pilha com Lista

- **função `pilha_pop`:**
  - retira o elemento do início da lista;

```
float pilha_pop (Pilha* p)
{
  Elemento* t; float v;
  if (pilha_vazia(p)){
    printf("Pilha vazia.\n");
    exit(1);
  }
  t = p->topo;
  v = t->info;
  p->topo = t->prox;
  free(t);
  return v;
}
```

Implementação de Pilha com Lista

- **função `pilha_libera`:**
  - libera todos os elementos da lista e depois libera a pilha;

```
void pilha_libera (Pilha* p)
{
  Elemento *t, *q = p->topo;
  while (q!=NULL)
  {
    t = q->prox;
    free(q);
    q = t;
  }
  free(p);
}
```

Implementação de Pilha com Lista

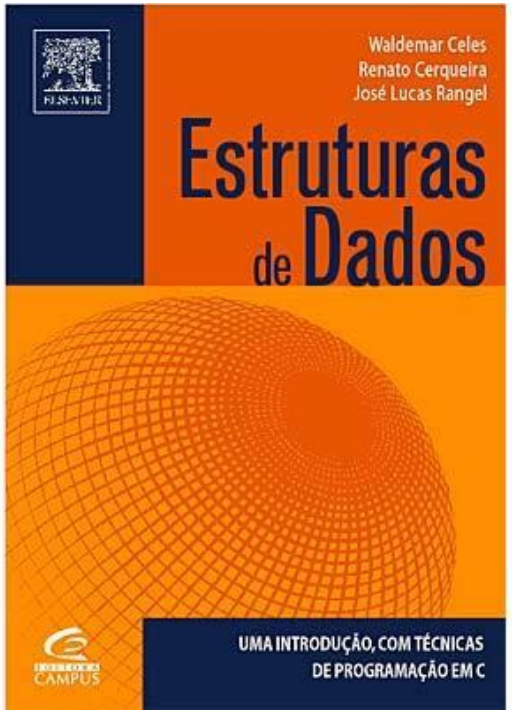
- **função `pilha_vazia`:**
  - retorna 1, se a pilha está vazia, ou 0, caso contrário;

```
int pilha_vazia (Pilha* p)
{
  if(p->topo == NULL)
    return 1;
  return 0;
}
```

Leitura Complementar

- Waldemar Celes, Renato Cerqueira, José Lucas Rangel, **Introdução a Estruturas de Dados**, Editora Campus (2004).

- **Capítulo 11 – Pilhas**



Exercícios

Lista de Exercícios 12 – Pilhas

http://www.inf.puc-rio.br/~elima/prog2/