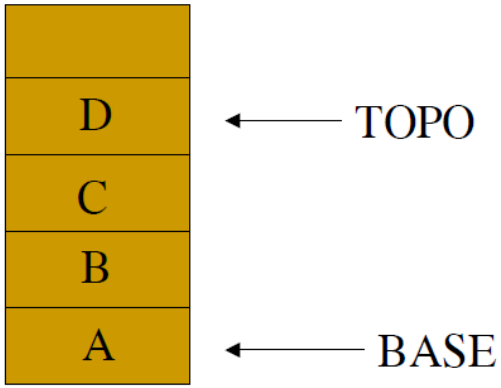
Estrutura de Dados

Pilhas

Pilhas

- Pilha é uma lista linear em que a inserção e eliminação de elementos somente podem ocorrer em uma das extremidades, que é chamada de TOPO da pilha.



- Dada uma Pilha P = (a1, a2, ..., na), dizemos que a1 é o elemento da base da pilha; na é o elemento do topo; e a_{i+1} está acima de a_i na pilha.

- São também conhecidas como listas do tipo LIFO (last in, first out).

Exemplos

- Comportamento dos retornos de chamadas de procedimento
- Comportamento da Lista DISPO
- Avaliação de expressões Posfix

Implementação Sequencial X Encadeada

- O problema da lista sequencial era quanto às movimentações de itens nas operações de inserção e remoção.

- No caso das pilhas, elas não ocorrem. A alocação sequencial é mais vantajosa na maioria das vezes.

- Encadeada (dinâmica): para evitar problemas de precisão de memória.

TAD Pilha - operações

- void define (pilha *p);

/* Cria pilha vazia. Deve ser usada antes de qqr outra operação */

- boolean push (tipo_info item, pilha *p);

/* Insere item no topo da pilha. Retorna true se sucesso, false c.c. */

- boolean vazia (pilha *p);

/* Retorna true se pilha vazia, false c.c. */

- void esvaziar (pilha *p);

/* Reinicializa pilha */

- tipo_elem top (pilha *p);

/* Devolve o elemento do topo sem removê-lo. Chamada apenas se pilha não vazia */

- void pop_up (pilha *p);

/* Remove item do topo da pilha. Chamada apenas se pilha não vazia */

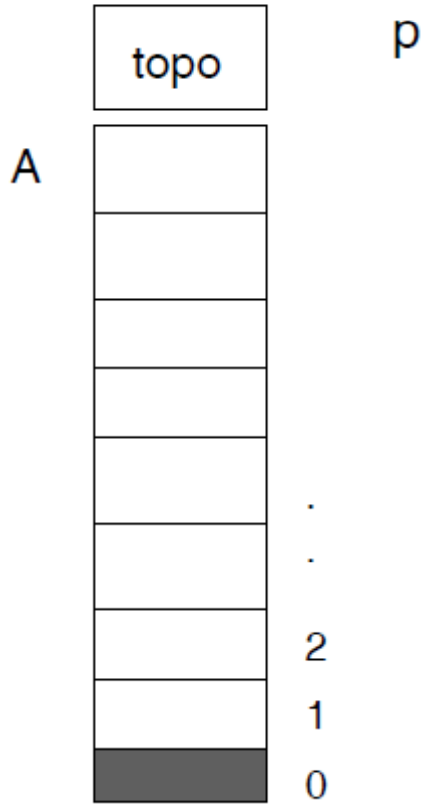
- tipo_elem pop (pilha *p);
- /* Remove e retorna o item do topo da pilha. Chamada apenas se pilha não vazia */

Operações – alocação sequencial

```
#define MAXP 1000
#define indice int

typedef struct{
      Tipo_info info;
}tipo_elem;

typedef struct{
      tipo_elem A[MAXP+1];
      indice topo;
}pilha;
```



1. define (P) - cria uma pilha P vazia

```
void define (pilha *p){
        p->topo = -1;
}
```

2. insere x no topo de P (empilha): push (x, P)

```
boolean push (tipo_elem x, pilha *p){
        if (p->topo == MAXP)  /* pilha cheia */
                return FALSE;

        p->topo ++;
        p->A[p->topo].info = x;
        return TRUE;
}
```

3. testa se P está vazia

```
boolean vazia (pilha *p){
        return (p->topo == -1);
}
```

4. acessa o elemento do topo da pilha (sem remover) - testar antes se a pilha não está vazia!!!

```
tipo_elem top (pilha *p){
        return p->A[p->topo];
}
```

5. remove o elemento no topo de P sem retornar valor (desempilha, v. 1) – testar antes se pilha não está vazia!!!

```
void pop_up (pilha *p){
    p->topo --;
}
```

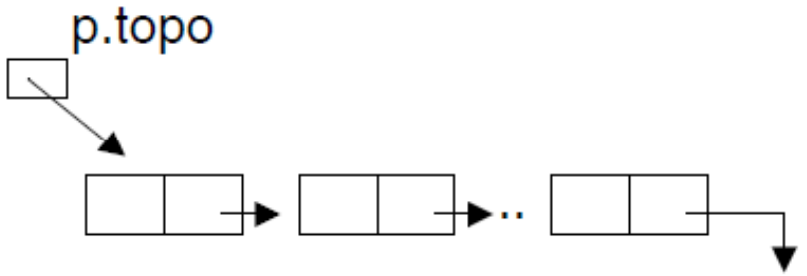
6. Remove e retorna o elemento (todo o registro) eliminado (desempilha, v. 2) – testar antes se pilha não está vazia!!!

```
tipo_elem pop (pilha *p){
    tipo_elem x = p->A[p->topo];
    p->topo --;
    return x;
}
```

Operação – alocação encadeada dinâmica

```
typedef struct elem{
        tipo_info info;
        struct elem *lig;
}tipo_elem;

typedef struct{
        tipo_elem *topo;
}pilha;
```

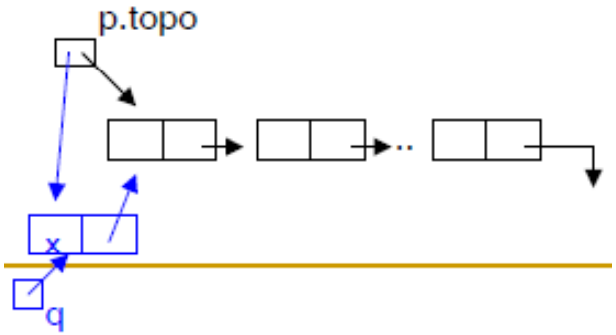


1. define (P) - cria uma pilha P vazia

```
void define (pilha *p){
        p->topo = NULL;
}
```

2. insere x no topo de P (empilha): push (x, P)

```
boolean push (tipo_info x, pilha *p){
        tipo_elem *q = malloc(sizeof(tipo_elem));
        if (*q == NULL)
        /*não possui memória disponível*/
                return FALSE;
        q->info = x;
        q->lig = p->topo
        p->topo = q;
        return TRUE;
}
```



3. testa se P está vazia

```
boolean vazia (pilha *p){
        return (p->topo == NULL);
}
```

4. acessa o elemento do topo da pilha (sem remover) -
testar antes da chamada se a pilha não está vazia!!!

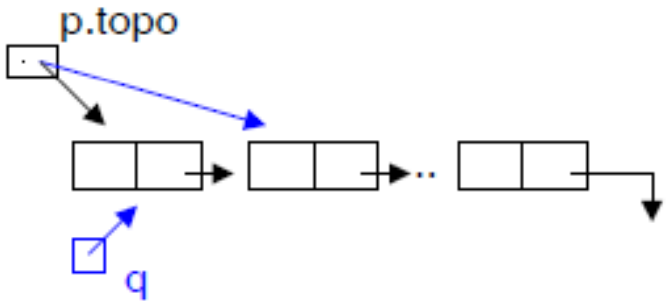
```
tipo_elem *topo (pilha *p){
        return p->topo;
}
```

5. remove o elemento no topo de P sem retornar valor
(desempilha, v. 1) – testar antes se pilha não está vazia!!!

```
void pop_up (pilha *p){
        tipo_elem *q = p->topo;
        p->topo = p->topo->lig;
        free(q);
}
```

6. Remove e retorna o elemento (todo o registro) eliminado
(desempilha, v. 2) – testar antes se pilha não está vazia!!!

```
tipo_elem *pop (pilha *p){
        tipo_elem *q = p->topo;
        p->topo = p->topo->lig;
        return q;
}
```



Exercícios:
Implementar as operações dos TAD:
-Pilha estática
-Pilha dinâmica

Pilhas

Reflita:

- Há vantagens de se implementar as Pilhas de forma dinâmica? Quais?

- Há desvantagens? Quais?

- Há vantagens em implementá-las estaticamente no array?

- Há desvantagens? Quais?

Exemplo de aplicação – Editor de texto

- Editores de texto sempre permitem que algum caracter (por exemplo *backspace*) tenha o efeito de cancelar os caracteres anteriores: caracter de apagamento.

- Se “#” é o caracter de apagamento, então a string “abc#d##e” é, na verdade, a string “ae”

- Editores têm também um caracter de eliminação de linha: o efeito é cancelar todos os caracteres anteriores da linha corrente. Suponha que ele seja o “@”

Exemplo: editor de texto

- Um editor de texto pode processar uma linha de texto usando uma pilha. O editor lê um caracter por vez (até ler o caracter de fim de linha), e
  - Se o caractere lido não é nem o de apagamento nem o de eliminação, ele é inserido na pilha
  - Se for o de apagamento, remove um elemento da pilha, e
  - Se for o de eliminação, o editor torna a pilha vazia.

- Faça um programa que execute estas ações utilizando o TAD Pilha.

```
#include "pilha.h"
void editor(){
        tipo_info c; pilha p;
        define(&p);
        system("cls"); /* limpa a tela */
        while (1){
                c = getch();
                switch (c){
                        case 0x0d: /* tecla ENTER */
                                return;
                        case '#': /* apaga */
                                if (vazia(&p)){
                                printf("erro");
                                return;
                                }
                        pop_up(&p);
                        break;
                        case '@': /* esvazia a pilha */
                                define(&p);
                        break;
                        default: /* tenta empilhar */
                                if (!push(c, &p)){
                                        printf("erro");
                                        return;
                                }// end if
                        } // end switch
                        system("cls");
                imprimir_em_ordem_reversa(&p);
                } // end while
} // end
```

```
void
imprimir_em_ordem_reversa(pilha
*p){
/* considera a implementação
sequencial */
int i;
for (i = 1; i <= p->topo; i++){
printf("%c", p->A[i].info);
}
}
```

Exemplo de aplicação: avaliação de expressões aritméticas

- Uma representação para expressões aritméticas conveniente do ponto de vista computacional é de interesse, por exemplo, para o desenvolvimento de compiladores.

- A notação tradicional (infixa) é ambígua e, portanto, obriga o pré-estabelecimento de regras de prioridade.

- Exemplo:

Tradicional: A*B-C/D

Parentizada: ((A*B)-(C/D))

- Notação Polonesa (prefixa): operadores aparecem imediatamente antes dos operandos. Esta notação especifica quais operadores, e em que ordem, devem ser calculados. Por esse motivo, dispensa o uso de parênteses.

Ex:

Tradicional: A*B-C/D

Polonesa: -*AB/CD

- Notação Polonesa Reversa (posfixa): operadores aparecem após os operandos.

Exemplo:

Tradicional: A*B-C/D

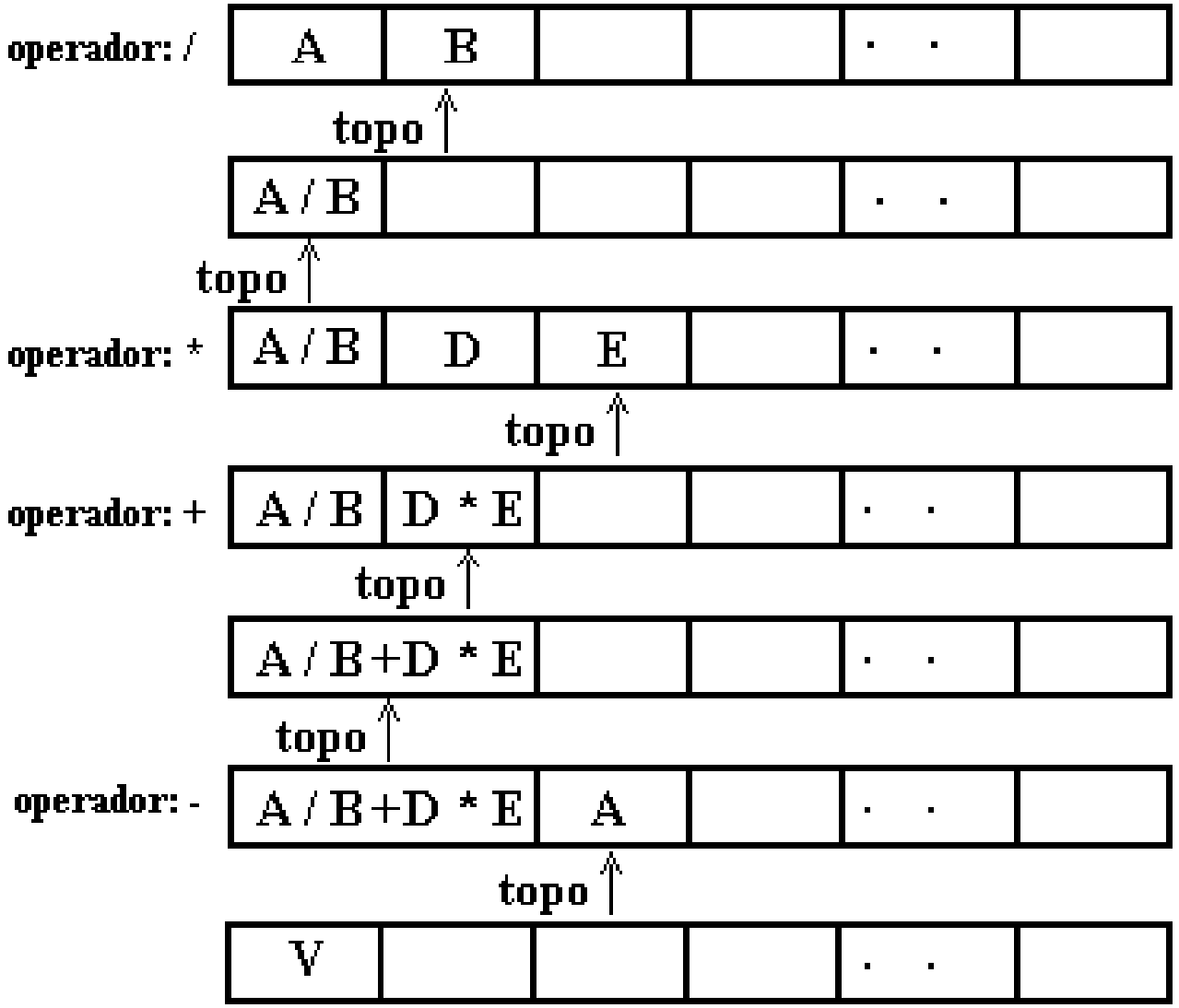
Polonesa reversa: AB*CD/-

Avaliação de Expressões Aritméticas Notação Posfixa

- Algoritmo

- Percorre sequencialmente a expressão, e empilha operandos até encontrar um operador;
- Desempilha o número correspondente de operandos;
- Calcula e empilha o valor resultante até chegar ao final da expressão.

Expressão: AB/DE+A-



Exercícios