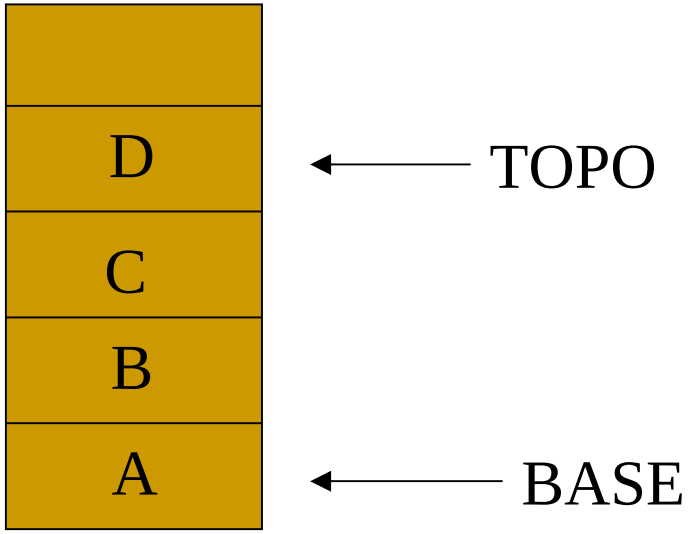
SCC 202– Algoritmos e Estruturas de Dados I

Pilhas (*Stacks*)

Pilhas

- Pilha: lista linear em que inserção e eliminação de elementos só ocorrem em uma das extremidades (TOPO da pilha)



Pilhas

- Dada uma Pilha P = (a_1, a_2, ..., a_n), dizemos que a_1 é o elemento na base; a_n 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*)

Ex: Pilhas de bandejas no Bandejão

1) Bandejas inicialmente são empilhadas
2) Pega-se (remove-se) bandeja no topo
3) Se não há mais bandejas, a pilha está vazia e não podemos remover mais
4) Uma vez que podemos ver a bandeja no topo, se ela estiver suja podemos não querer pegar (remover)

Este ex. faz referência a 4 operações de pilhas:

1) Inserir = *push*
2) Remover = *pop*
3) Se a pilha não contém elementos a verificação de vazia retorna *true*, caso contrário *false*
4) Topo_da_pilha = *top*, retorna o topo da pilha (cópia) para ser examinado, sem removê-lo

Exemplos

- Comportamento dos retornos de chamadas a procedimentos
- Avaliação de expressões aritméticas expressas na forma Posfixa

- A/B + D*E – A meta➔ ((A/B)+(D*E))-A

  Como ocorre: AB/DE*+A-

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 */`

TAD Pilha - operações

- `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 */`

Implementação: sequencial x encadeada

- Problema da implementação sequencial de listas: necessidade de movimentações de itens em inserções e remoções
  - No caso das pilhas, tais movimentações não ocorrem!
- Alocação sequencial vantajosa na maioria das situações...
- Alocação dinâmica interessante para pilhas cujo tamanho não pode ser antecipado, ou é muito variável...

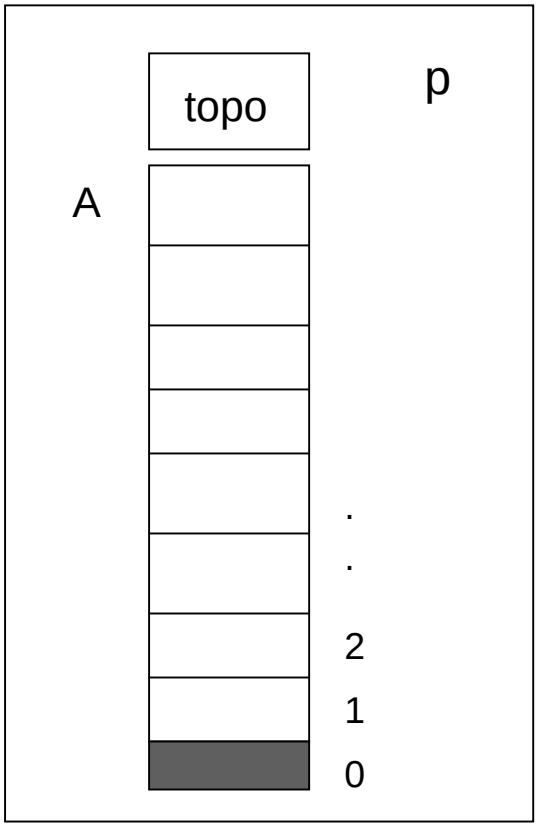
Operações – alocação sequencial

```
#define MAXP 1000

#define indice int

typedef struct{
    tipo_info info;
}tipo_elem;

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




```
pilha p; /*exemplo de declaração*/
```

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

```
void define (pilha *p){
    p->topo = 0;
}
```

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

```
boolean push (tipo_info 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 == 0);
}
```

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!!!

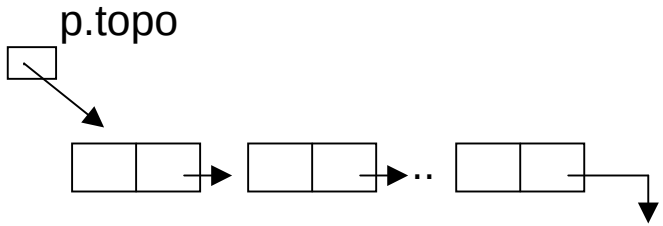
```
tipo_elem pop (pilha *p){
    tipo_elem x = p->A[p->topo];
    p->topo --;
    return x;
}
```

Operações – alocação encadeada dinâmica

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

typedef struct{
    tipo_elem *topo;
}pilha;

pilha p;
```



Obs: p.topo dá o endereço do elemento no topo

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

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

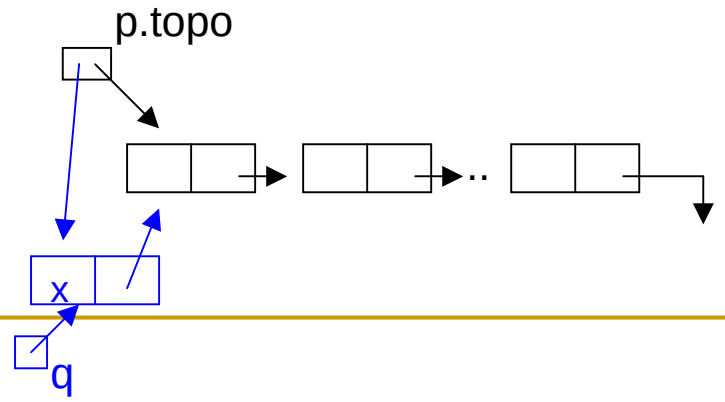
2. insere x no topo de P (empilha): push (x, P)

```
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!!!

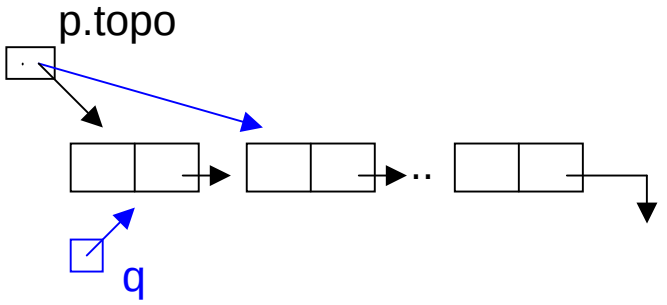
```
tipo_elem *topo (pilha *p){
    return p->topo;
}
```

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 TADs:
  - pilha estática
  - pilha dinâmica

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 (p.ex. *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
- Façam um programa que execute estas ações utilizando o TAD Pilha

```c
#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;
                }
        }
        system("cls");
        imprimir_em_ordem_reversa(&p);
    }
}
```

```
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))

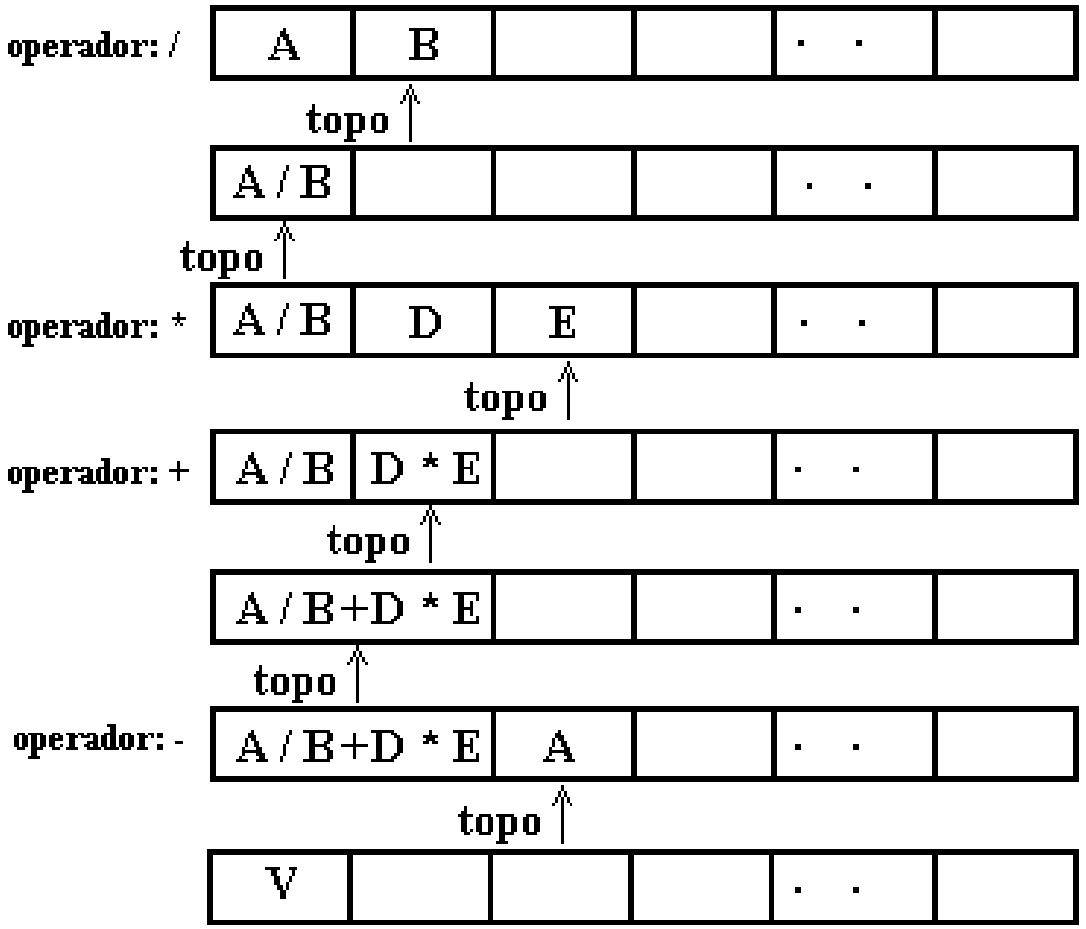
- **Notação Polonesa (prefixa):** operadores aparecem imediatamente antes dos operandos. Esta notação especifica quais operadores, e em que ordem, devem ser calculados. Por esse motivo, dispensa o uso de parênteses.
  Exemplo:
  tradicional: A * B - C / D
  polonesa: - * A B / C D

- **Notação Polonesa Reversa (posfixa):** operadores aparecem após os operandos.
  Exemplo:
  tradicional: A * B - C / D
  polonesa reversa: A B * C D / -

Avaliação de Expressões Aritméticas: Notação Posfixa

- Algoritmo
  - percorre seqüencialmente 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: A B / D E * + A -



Avaliação de Expressão Posfixa

- Expressão posfixa: seqüência de caracteres, que ou são operandos (por ex., valores inteiros) ou são operadores binários (+, -, *, /)
  - Se operandos definem operações inteiras, resultado da avaliação é integer, senão é real... (TipoValor)
- Definidos
  - Tipo de dado Expressao (por ex., string ou vetor de char)
  - **function** proxsimb(E: expressao): TipoOperador;
    - Quando retorna um operando (número), precisa converter para TipoValor (por exemplo, converter char para integer...)
- Teste ‘x é operando’ deve ser implementado por alguma função booleana que verifica se x é um número...

```
function valor ( E: expressão): TipoValor;   {supõe E uma expressão
    posfix bem-formada}
var x : TipoOperador;  ok: boolean;
     op1, op2: TipoValor; P: Pilha;
 begin
    define(P); { P é uma pilha que armazena TipoValor...}
    while not acabou(E) do
    begin
        x := proxsimb(E);
        if  x é operando then ok:= push(x,P)    { empilha }
        else begin   {  x é operador: desempilha operandos,
                        executa operação, empilha resultado }
              op2:= pop(P);
              op1:= pop(P);
              valor := resultado da operação { op1 x op2 };
              ok := push(valor, P)
         end;
    end; { while }
    valor:= pop(P);
end;
```

Conversão para notação Posfixa

- Converte expressão de entrada dada na notação tradicional parentizada para a notação posfixa
- Expressões (tradicional e posfixa): seqüências de caracteres
  - Caracteres ou são operandos (por ex., valores inteiros) ou são operadores (+, -, *, /), ou são parêntesis (abre e fecha)
  - Todas as operações devem ser delimitadas por parêntesis

Conversão para notação Posfixa

- Definidos
  - Tipo de dado Expressao (por ex., string ou vetor de char)
  - **function** proxsimb(E: expressao): **char**;
    - retorna o próximo caracter da expressão
  - **Procedure** insere(x: **char**; **var** E: expressão)
    - Insere elemento x no final da expressão E
- Testes ‘x é operando’ e ‘x é operador’ devem ser implementados por alguma função booleana...

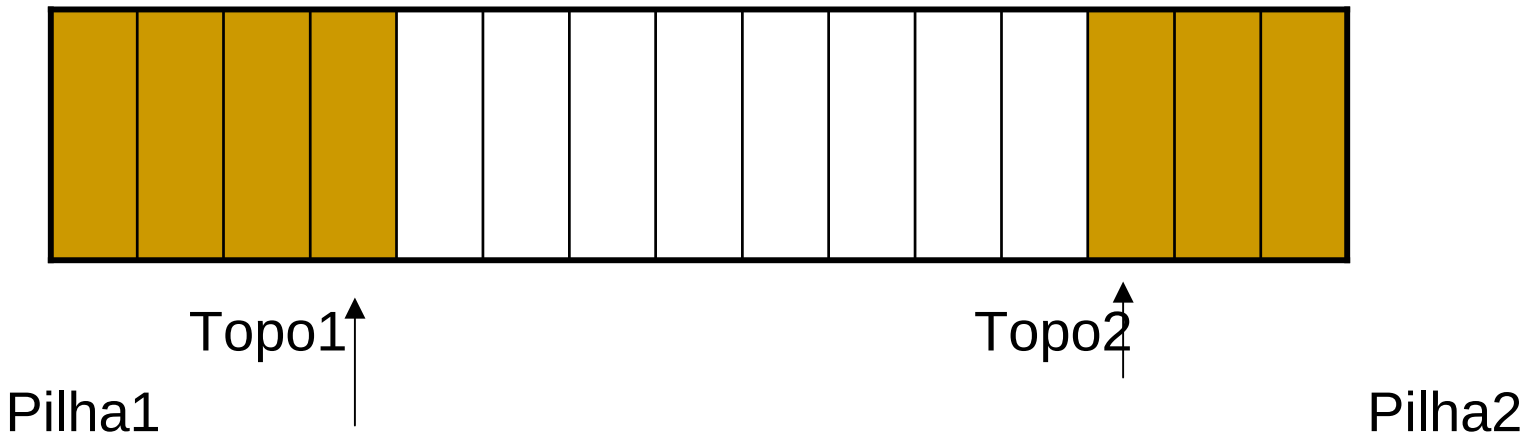
```
procedure converte (exp: expressao,var exp_pol: expressao);
var x ,operador: char;  ok: boolean; P: pilha;
begin
    define(P); x:= proxsimb(exp); { Pilha armazena char)
    while not acabou(exp) do
    begin
       while x = '(' do x:= proxsimb(exp);
       if (x é operando) then
         insere(x,exp_pol)   { copia para expressao polonesa}
       else if (x é operador) then   { empilha }
                    ok := push(x,P)
              else if x = ')' then
                    if not vazia(P) then { forma operacao } begin
                        operador:= pop(P);
                        insere(operador,exp_pol)
                    end
                    else "erro na expressão";
         x:= proxsimb(exp);
      end; {while}
      esvaziar(P);
end;
```

Alocação Múltipla de Pilhas

2 pilhas com elementos do mesmo tipo

P1[1..m1] P2[1..m2]

1 único *array*: a[1..M]; M > m1+m2



Conseqüências

- *Overflow* só ocorre se o no. total de elementos de ambas exceder M
- Bases fixas

    início: P.Topo1 = 0 cresce à direita

    P.Topo2 = M+1 cresce à esq.

Inserção na Pilha 1

```
if P. topo1  < P. topo2 – 1 then
   begin
         P.topo1:= P.topo1+1;
         P.A[topo1]:= x
   end
else
   “overflow”
```

Inserção na Pilha 2

```
if P.topo2  > P.topo1 + 1 faça
   begin
         P.topo2:= P.topo2-1;
         P.A[topo2]:= x
   end
else
   “overflow”
```

N Pilhas

- Bases não podem ser fixas para garantir o melhor aproveitamento do espaço