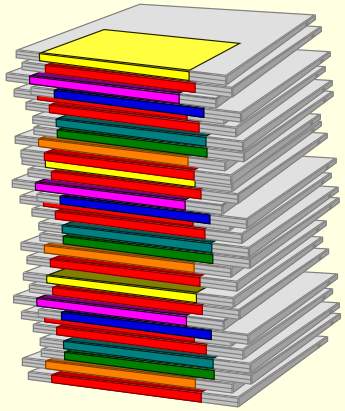
Pilha

Algoritmos e Estruturas de Dados I

Pilha

- O que é?
- Para que serve?



Problema: chamada de sub-rotinas

Rotina A
1 print “A”
2 call C
3 call B
4 call D
5 return

Rotina B
1 call C
2 print “B”
3 call D
4 call C
5 return

Rotina C
1 print “C”
2 call D
3 return

Rotina D
1 print “D”
2 return

Qual o resultado da execução da rotina A?

Problema: chamada de sub-rotinas

Rotina A
1 print “A”
2 call C
3 call B
4 call D
5 return

Rotina B
1 call C
2 print “B”
3 call D
4 call C
5 return

Rotina C
1 print “C”
2 call D
3 return

Rotina D
1 print “D”
2 return

Qual o resultado da execução da rotina A?

Qual a dificuldade para se fazer esse cálculo?

Problema: chamada de sub-rotinas

Rotina A
1 print “A”
2 call C
3 call B
4 call D
5 return

Rotina B
1 call C
2 print “B”
3 call D
4 call C
5 return

Rotina C
1 print “C”
2 call D
3 return

Rotina D
1 print “D”
2 return

Qual o resultado da execução da rotina A?

Qual a dificuldade para se fazer esse cálculo?

Possíveis soluções?

Problema: chamada de sub-rotinas

1. Um computador está executando a rotina X e, durante a execução de X, encontra uma chamada à rotina Y

2. Para-se a execução de X e se inicia a execução de Y

3. Quando se termina a execução de Y, o computador deve saber o que fazer, isto é, onde voltar na rotina X

Problema: chamada de sub-rotinas

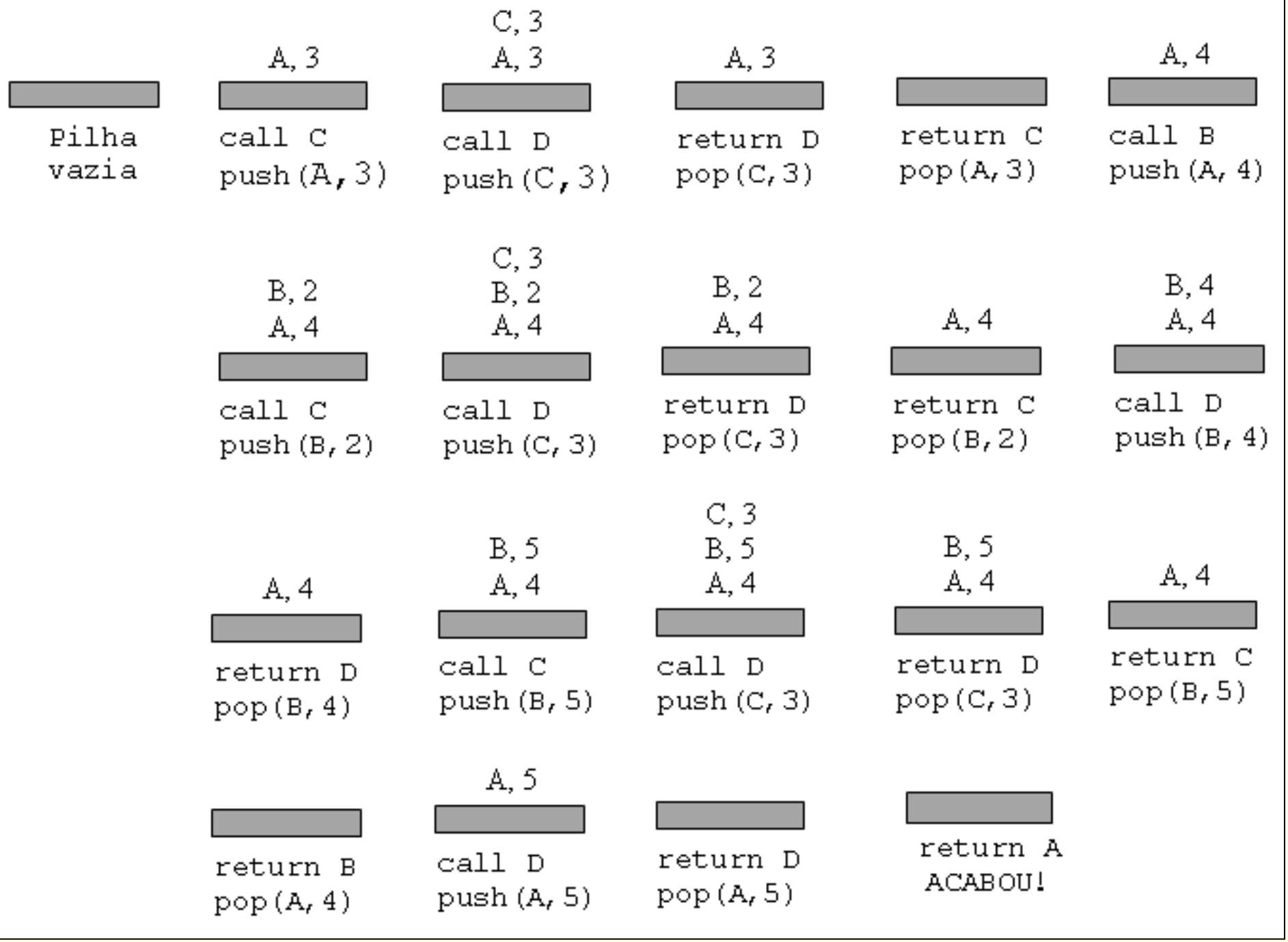
- Dificuldade
  - O que estava sendo executado quando uma sub-rotina foi interrompida? Para onde voltar agora que se chegou ao fim de uma sub-rotina?
- Solução
  - A cada chamada de sub-rotina, armazenar o endereço de retorno (rotina e número da linha, por exemplo)
  - Como armazenar o endereço de retorno de chamadas sucessivas: pilha

Pilha (*stack*)

- Definição
  - *Estrutura para armazenar um conjunto de elementos que funciona da seguinte forma*
    - Novos elementos sempre entram no “topo” da pilha
    - O único elemento que se pode retirar da pilha em um dado momento é o elemento do topo
- Para que serve
  - Para modelar situações em que é preciso “guardar para mais tarde” vários elementos e “lembrar” sempre do último elemento armazenado
- L.I.F.O.
  - *Last In, First Out*

Problema: chamada de sub-rotinas

- A cada comando call
  - Empilha (*push*) o endereço para retornar depois
  - Passa a executar a nova sub-rotina
- A cada comando return
  - Desempilha (pop) o último endereço armazenado
  - Passa a executar a partir do endereço desempilhado


Pilha vazia
A, 3
call C
push(A, 3)
C, 3
A, 3
call D
push(C, 3)
A, 3
return D
pop(C, 3)
return C
pop(A, 3)
A, 4
call B
push(A, 4)
B, 2
A, 4
call C
push(B, 2)
C, 3
B, 2
A, 4
call D
push(C, 3)
B, 2
A, 4
return D
pop(C, 3)
A, 4
return C
pop(B, 2)
B, 4
A, 4
call D
push(B, 4)
A, 4
return D
pop(B, 4)
B, 5
A, 4
call C
push(B, 5)
C, 3
B, 5
A, 4
call D
push(C, 3)
B, 5
A, 4
return D
pop(C, 3)
A, 4
return C
pop(B, 5)
return B
pop(A, 4)
A, 5
call D
push(A, 5)
return D
pop(A, 5)
return A
ACABOU!

Problema: chamada de sub-rotinas

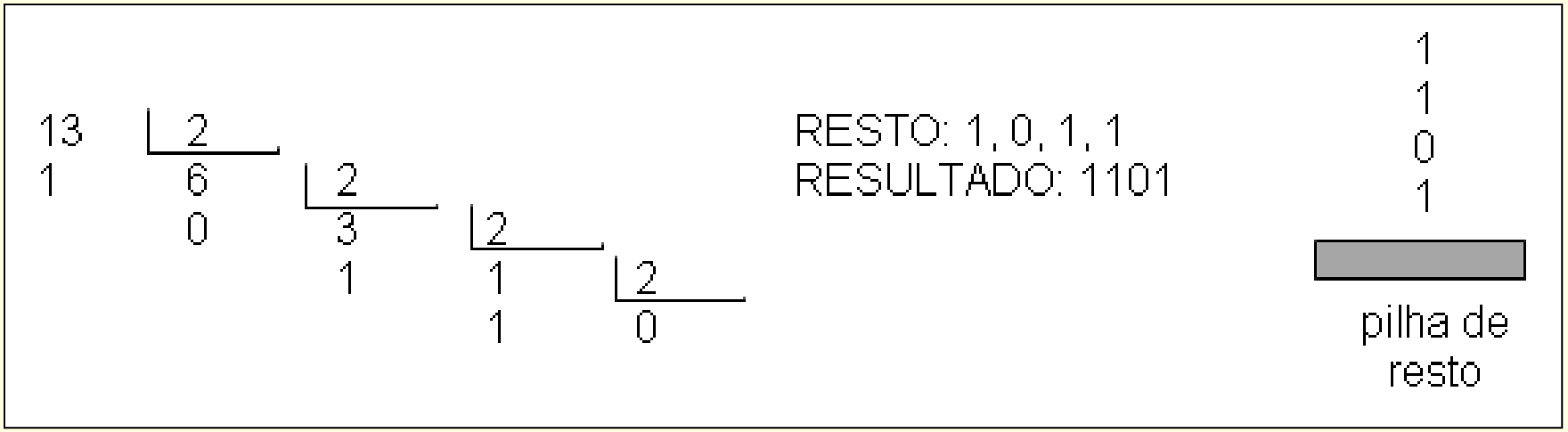
- Resultado
  - A, C, D, C, D, B, D, C, D, D

Pilha

- Operações usuais
  - Push(P,X): empilha o valor da variável X na pilha P
  - Pop(P,X): desempilha P e retorna em X o valor do elemento que estava no topo de P
  - X=top(P): acessa o valor do elemento do topo de P, sem desempilhar
  - Create(P): cria uma pilha vazia P
  - IsEmpty(P): é *true* se a pilha estiver vazia; *false* caso contrário
  - Empty(P): esvazia logicamente uma pilha P

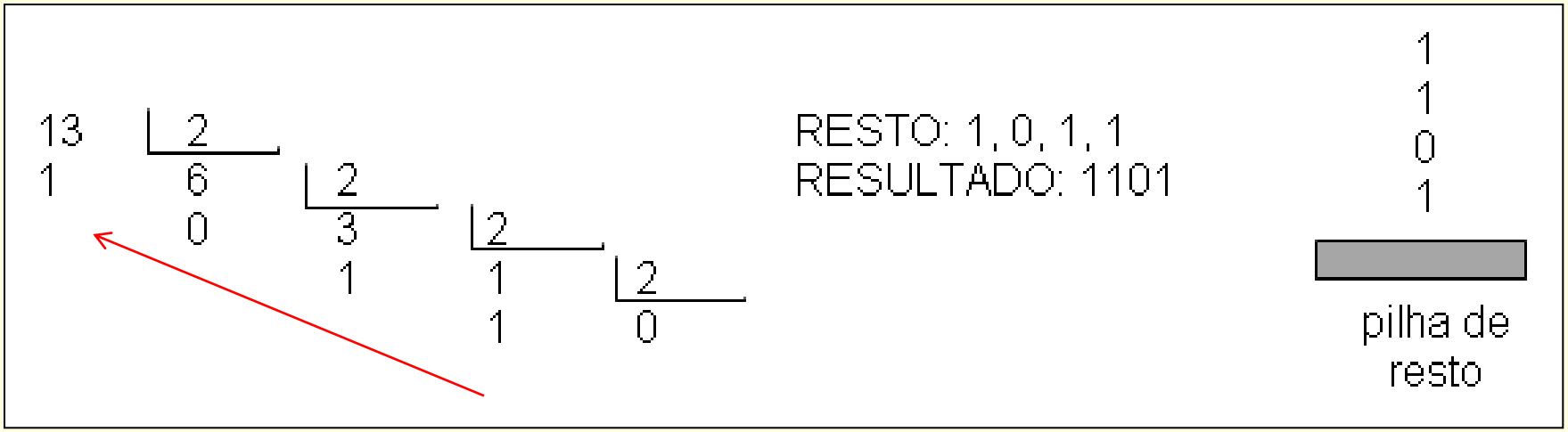
Exercício

- Fazer algoritmo de conversão decimal para binário usando pilha



Exercício

- Fazer algoritmo de conversão decimal para binário usando pilha



Estratégia de resolução:
• a cada divisão, empilha o resto
• quando acabar a divisão (quociente=0), desempilha e escreve todos os elementos

Considerar prontos: operações da pilha, resto(X,Y) e quociente(X,Y)

Exercício

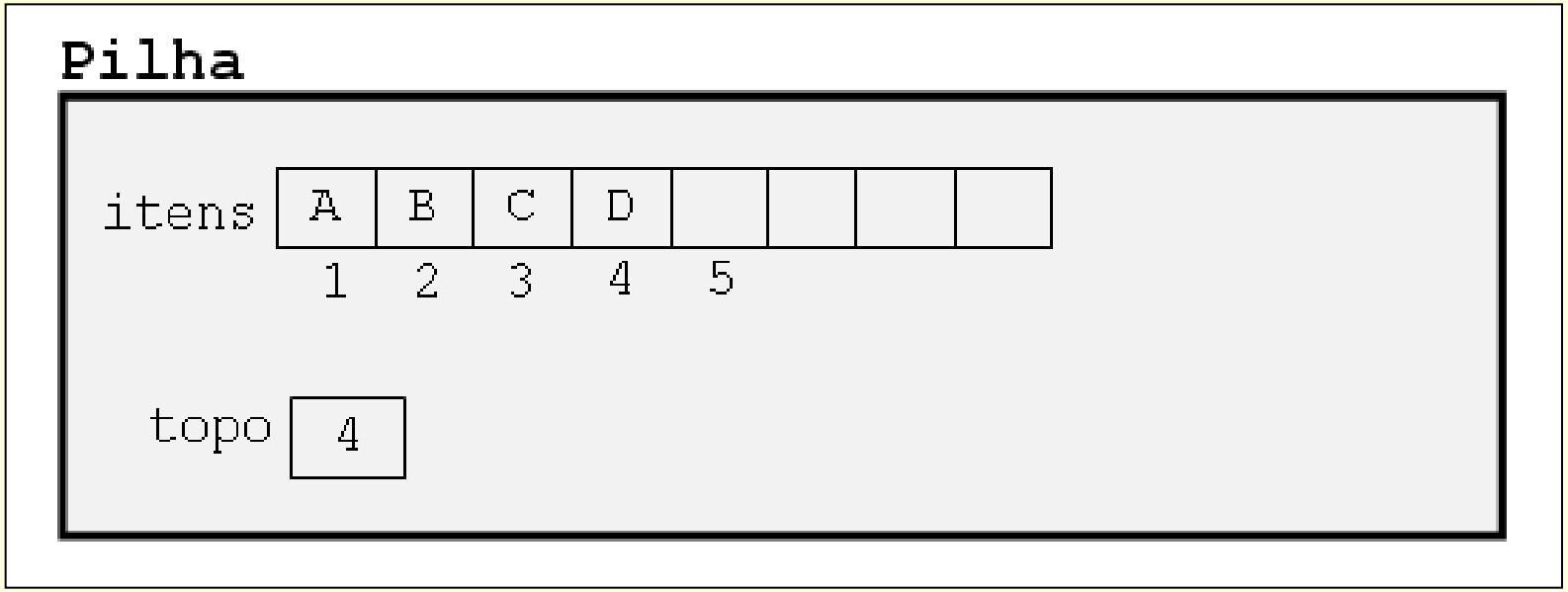
```
Variáveis
    P: pilha
    N: inteiro {número a ser convertido}
    X: inteiro {resto da divisão}

Início do algoritmo
    leia N
    create(P)
    repita
            X=resto(N,2)
            push(P,X)
            N=quociente(N,2)
    até que (N=0)
    escreva “o resultado é “
    enquanto (IsEmpty(P)=falso) faça
            pop(P,X)
            escreva X
Fim
```

Implementação da pilha

- Alocação seqüencial
  - Os elementos da pilha ficam, necessariamente, em seqüência (um ao lado do outro) na memória
- Alocação estática
  - Todo o espaço de memória a ser utilizado pela pilha é reservado (alocado) em tempo de compilação
  - Todo o espaço reservado permanece reservado durante todo o tempo de execução do programa, independentemente de estar sendo efetivamente usado ou não

Implementação da pilha

- Seqüencial e estática



Implementação da pilha

- Declaração em C

  ?

Implementação da pilha

- Declaração em C

```
#define TamPilha 100

typedef int elem;

typedef struct {
            int topo;
            elem A[TamPilha];
} Pilha;

Pilha P;
```

Exercício

- Implementar operações da pilha
  - Create
  - Empty
  - IsEmpty
  - IsFull
  - Push
  - Pop
  - Top

- Atenção: considerações sobre TAD
  - Arquivos .c e .h, parâmetros

Interface do TAD: Pilha.h

```
#define TRUE 1   /*define tipo boleano*/
#define FALSE 0
#define boolean int
typedef char elem;
typedef struct {
                    int topo;
                    elem A[TamPilha];
} Pilha;

void Create(Pilha*);
void Empty(Pilha*);
boolean IsEmpty(Pilha*);
boolean IsFull(Pilha*);
boolean Push(Pilha*, elem*);
boolean Pop(Pilha*, elem*);
```

Implementando as operações: Pilha.c

```
#include "pilha.h“
#define TRUE 1   /*define tipo boleano*/
#define FALSE 0
#define boolean int

void Create(Pilha *P) { /* cria uma pilha vazia P */
    P->topo=-1;
    return; }
void Empty(Pilha *P) { /* esvazia logicamente a pilha P */
    P->topo=-1;
    return; }
boolean IsEmpty(Pilha *P) { /* retorna true se a pilha estiver vazia; false caso
     contrário */
   return (P->topo==-1);
}
```

```
boolean IsFull(Pilha *P) { /* verifica se P cresceu até o final do array */
    return (P->topo==TamPilha-1);
}

boolean Push(Pilha *P, elem *X) { /* empilha o valor da variável X na pilha P */
     if (!IsFull(P)) {
          P->topo++;
          P->A[P->topo]=*X;
        return TRUE;
     }
    return FALSE;
}
```

```
boolean Pop(Pilha *P, elem *X) { /*desempilha P e retorna em X o valor do
    elemento que estava no topo de P; retorna TRUE se sucesso; FALSE, c.c. */
    if (!IsEmpty(P)) {
        *X=P->A[P->topo];
        P->topo--;
        return TRUE;
    }
    return FALSE;
}
elem Top(Pilha *P) { /* retorna o valor do elemento do topo de P, sem
    desempilhar – chamada apenas se Pilha P não estiver vazia!!! */
  return P->A[P->topo];
 }
```

Análise da Complexidade – Pilha Sequencial

- Acesso ao Topo:
  - Tempo constante – O(1)
- Inserção
  - Sempre no topo: tempo constante – O(1)
- Eliminação
  - Sempre no topo: tempo constante – O(1)

➔Portanto, toda operação tem custo independente do tamanho da pilha

- Espaço: exatamente o necessário para armazenar o número de elementos da pilha
- Desvantagem: espaço máximo fixado pelo tamanho do array pode ser excessivo ou insuficiente

Exercícios

- 1. Adicionar uma rotina ao TAD para verificar se duas pilhas são iguais
- 2. Adicionar uma rotina ao TAD para inverter a posição dos elementos de uma pilha

Editor de texto: exercício

- Considere que um editor de texto representa os caracteres digitados como uma pilha, sendo que o último caracter lido fica no topo
- Alguns comandos apagam caracteres. Por exemplo, o *backspace* apaga o último caractere lido
- Alguns comandos apagam tudo o que já foi lido anteriormente
- Considere que, no seu editor, # representa *backspace* e @ indica “apagar tudo”
- Faça um programa que execute essas ações usando o TAD pilha

```
#include <stdio.h>
#include "pilha.h"
int main(void) {
    elem c, x;
    Pilha P;
    Create(&P);
    printf("Digite seu texto: ");
    while ((c=getche())!='\r') {
        if (c=='#') {
           if (Pop(&P,&x))
              printf("(%c desempilhado) ",x);
           else printf("erro");
           else
        }
        else if (c=='@') {
            Empty(&P);
            printf("(pilha esvaziada) ");
        }
        else {
            if (!Push(&P,&c))
                printf("(erro) ");
        }
    }
...
```

```
...
    printf("\n\nDesempilhando tudo: ");
    while (!IsEmpty(&P)) {
        if (Pop(&P,&x))
            printf("%c ",x);
        else printf("erro ");
    }
    system("pause");
    return 0;
}
```

POSSÍVEL SOLUÇÃO

Notação posfixa: exercício

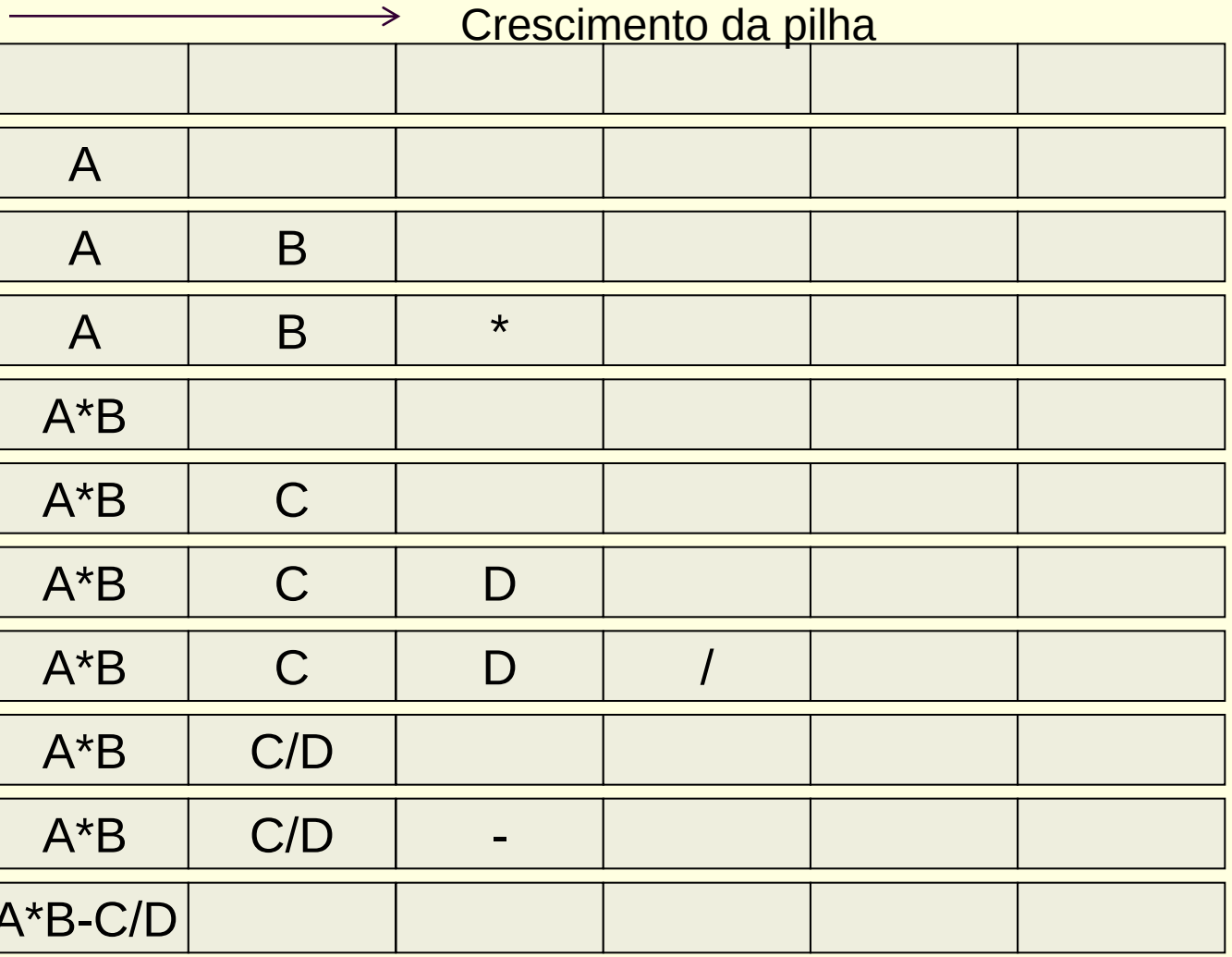
- Avaliação de expressões aritméticas
  - Às vezes, na aritmética tradicional, faz-se necessário usar parênteses para dar o significado correto à expressão
    - A*B-C/D → (A*B)-(C/D)
  - Notação polonesa (prefixa): operadores aparecem antes dos operandos e dispensa parênteses
    - -*AB/CD
  - Notação polonesa reversa (posfixa): operadores aparecem depois dos operandos e dispensa parênteses
    - AB*CD/-

Notação posfixa: exercício

- Interpretação da notação posfixa usando pilha
  - Empilha operandos até encontrar um operador
  - Retira os operandos, calcula e empilha o resultado
  - Até que se chegue ao final da expressão

Notação posfixa: exercício

- AB*CD/-



Exercício

- Implemente uma função que calcule o valor de uma expressão posfixa passada por parâmetro utilizando o TAD pilha

Exercício – resposta algorítmica

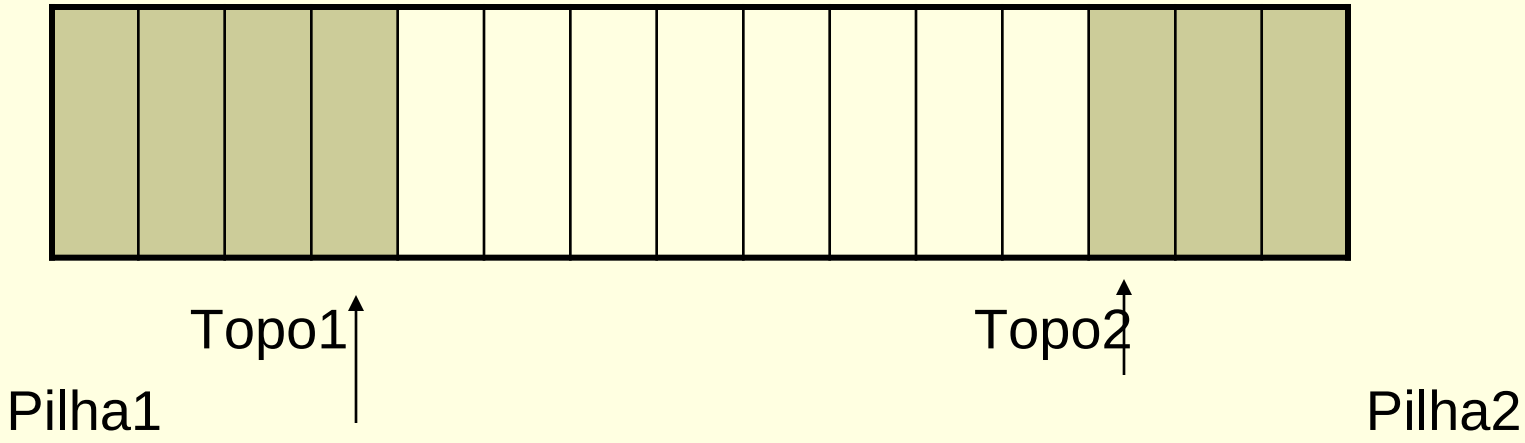
```
função valor(E: expressão): retorna real;
declare x real;
declare P pilha;
início
    Create(P)
    enquanto não acabou(E) faça
            início
            x=proxsimb(E);
            se x é operando então Push(P,x)
            senão início
                        remove operandos; {dois pops, em geral}
                        calcula o resultado da operação;
                        empilhe resultado;   {push}
                    fim
            fim
    valor=Top(P);
fim
```

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

```
se P. topo1  < P. topo2 – 1
  então
        P.topo1= P.topo1+1;
        P.A[topo1]= x
 senão
      “overflow”
```

Inserção na Pilha 2

```
se P.topo2  > P.topo1 + 1
   então
         P.topo2 = P.topo2-1;
         P.A[topo2] = x
    senão
       “overflow”
```

N Pilhas

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