ESTRUTURAS DE DADOS

prof. Alexandre César Muniz de Oliveira

Sugestão bibliográfica:

- **ESTRUTURAS DE DADOS USANDO C**
Aaron M. Tenenbaum, et alli

- **DATA STRUCTURES, AN ADVANCED APPROACH USING C**
Jeffrey Esakov & Tom Weiss

- **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA (2ED)**
Michael Godrich & Roberto Tamassia

INTRODUÇÃO

1. Tipos de dados

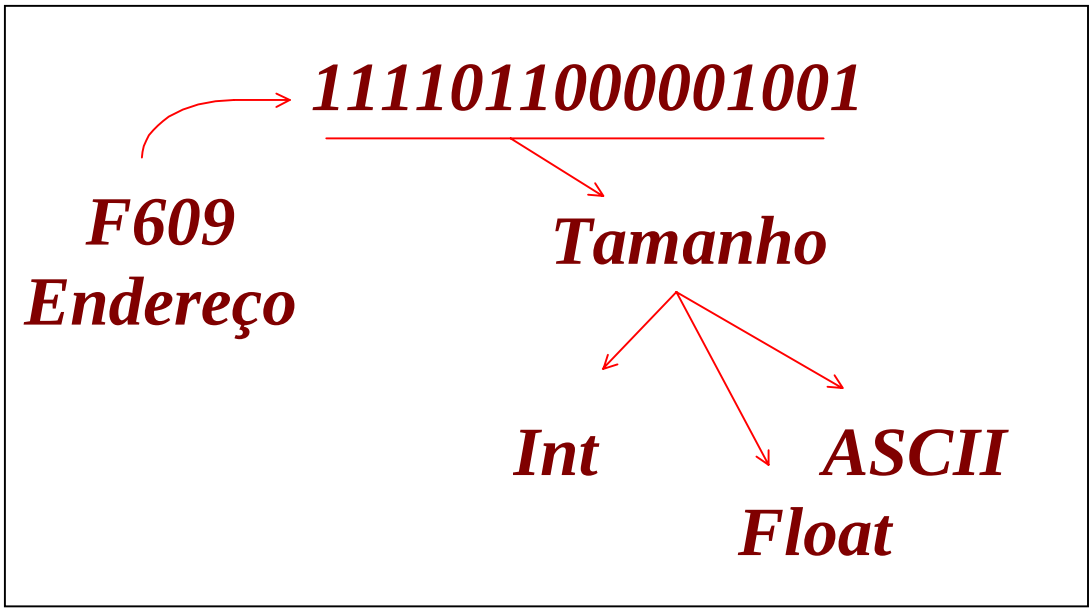
Especifica o conjunto de possíveis valores e o conjunto de possíveis operações.

Ex.: Número inteiro
{-∞, ..., -2, -1, 0, 1, 2, ..., +∞}
Operações aritméticas

Cadeia de caracteres
{'0', ..., '9', 'A', ..., 'Z', ...}
Concatenação,
Subdivisão,
Comprimento, etc. ...

Fisicamente, um dados nada mais é que uma seqüência de bits, armazenados em um local conhecido de memória.

A forma de manipulação depende da definição do tipo.



INTRODUÇÃO

2. Implementação de tipo
Forma como um tipo de dado será implementado em nível de hardware / software

2.1. Implementação de hardware
Circuito Físico que efetuará as operações. É elaborado e construído como parte do computador.

2.2. Implementação de software
Um programa, consistindo de instruções de hardware já existentes, é criado para interpretar cadeias de bits na forma desejada, efetuando operações.

3. Tipos de dados abstratos
Forma de especificar propriedades de um tipo de dado.

Tipo dado := Valores + Operações
Implementável por hardware / software.

T.D.A. → Conceito matemático básico que define completamente o tipo de dado.
Tempo e espaço → Implementável.

Obs.: *Nem sempre é possível implementar um T.D.A. em um determinado hardware ou utilizando um certo software.*

INTRODUÇÃO

Exemplo(1):

Numero_Racional = (A/B, B<>0)

```
# typedef struct {
        int a;
        int b;
        condição: b<>0;
  } Racional;

# Multip_Racional (x, y, z)
      Racional x, y, z;
  {
        z.a := x.a * y.a;
        z.b := x.b * y.b;
  }

# Igual_Racional (x, y)
        Racional x, y;
  {     return (x.a * y.b = x.b * y.a?V:F); }
```

Exemplo(2):

Seqüência = $<S_0, S_1, ..., S_{n-1}>$
(S) ou $<S_1, S_2, ..., S_n>$
onde S tem tamanho n

Operações:

- ***len (S) $\rightarrow n$***
- ***first (S) $\rightarrow S_0$***
- ***last (S) $\rightarrow S_{n-1}$***
- ***equal (S, T) $\rightarrow$ .V. se $S_i = T_i \ \forall i < n-1$***
- ***sub (S, i, f) $\rightarrow T=<T_0, T_1, ..., T_{n-1},>$***
  onde $T_0 = S_i$, $T_1 = S_{i+1}$, $T_{n-1} = S_f$,
- ***concat (S, T) $\rightarrow R = <R_0, ..., R_{n-1}>$***
  onde $R_0 = S_0$, $R_{k-1} = S_{n-1}$, $R_k = T_0$, ..., $R_{n-1} = T_{n-1}$

INTRODUÇÃO

4. Conceitos sobre estruturas de dados

Como organizar dados de forma eficiente considerando uma dada aplicação

- requisitos de espaço de armazenamento,
- operações e
- tempo de acesso

Estruturar de dados ⇒ Entidade Abstrata + Operações

Objetivos do curso

I. Como utilizar na solução de problemas computacionais.
II. Como implementar, usando os tipos de dados já existentes.

O primeiro objetivo diz respeito a construir algoritmos usando abstrações clássicas, encontradas na literatura, bem como aquelas de criação do próprio aluno, inovadoras, específicas para um dado problema.

O segundo objetivo diz respeito a implementação dessas abstrações em termos de uma linguagem de programação alvo. Depende dos tipos básicos implementados na linguagem.

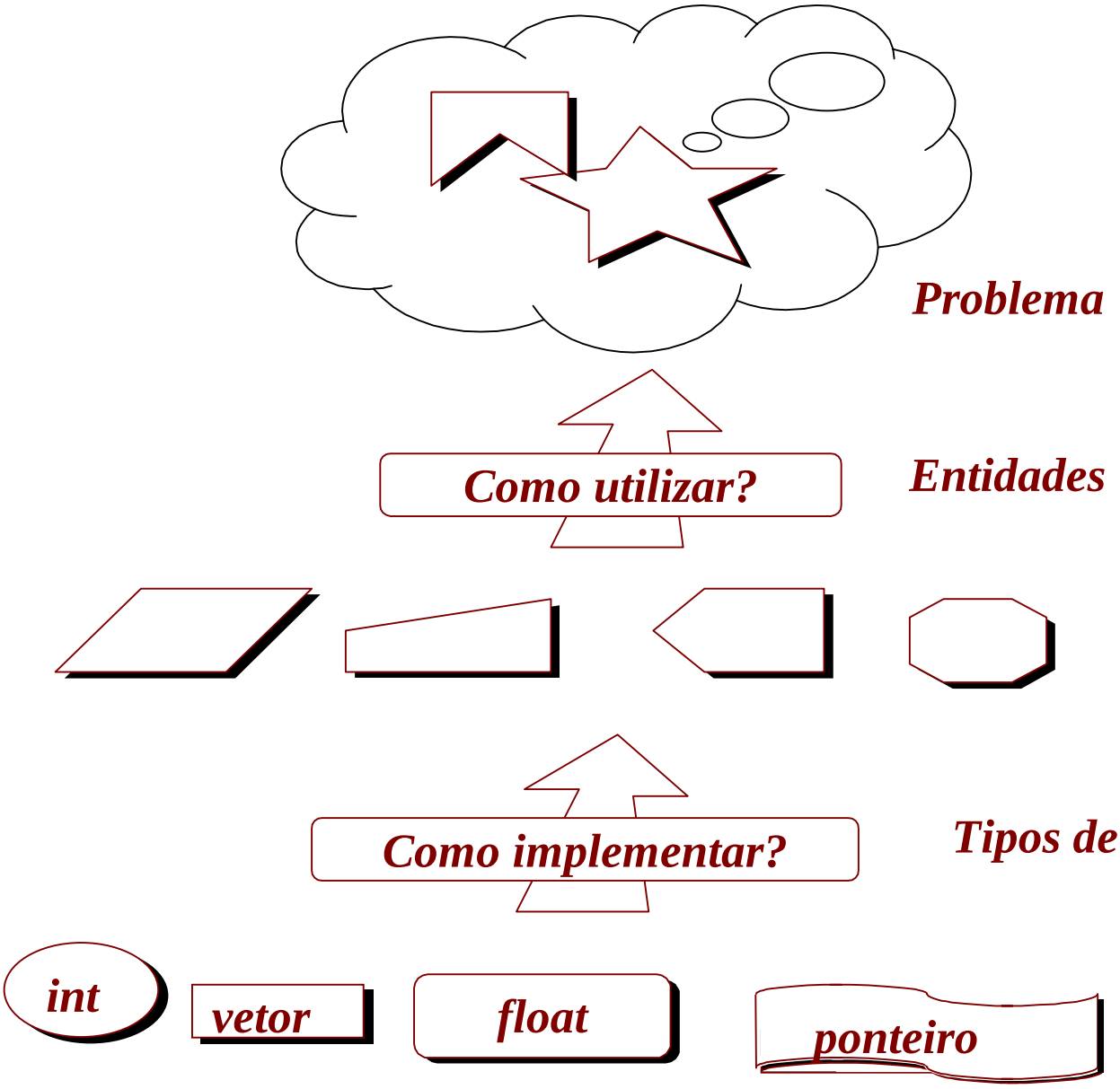
5. Tipos de dados em C, C++

- int, float, char, double
- short, long, unsigned
- ponteiros near, far
- vetor
- estruturas
- união

Podem ser usados para implementar as estruturas de dados: Pilha, Fila, etc... na solução de problemas computacionais.

INTRODUÇÃO

A Figura a seguir ilustra os níveis de atuação do programador em face de um problema computacional. Abstrações de alto-nível permitem uma abordagem de forma mais direta, voltando-se exclusivamente para os detalhes do problema propriamente dito. Níveis de abstrações permitem o progressivo detalhamento do algoritmo a ser empregado para a solução do problema.



INTRODUÇÃO

6. Vetor

Conjunto finito e limitado de elementos homogêneos.

6.1. Forma de definição

- **Nome**
- **Limite inicial**
- **Tipo**
- **Tamanho**
- **Limite final**

6.2. Forma de acesso

- Vetor definido
- Posição

Armazenamento e recuperação de qualquer posição dentro do vetor em qualquer tempo.

6.3. Tipos

- Unidimensional
- Bidimensional
- Tridimensional

Matrizes

...

INTRODUÇÃO

6.4. Vetor como um T.D.A.

- Vetor de floats: $x \in \Re$ e precisão (x) < limite

Operações:

- Armazenar (vetor, posição, valor)
  - Condição: tipo vetor = tipo valor
    tipo posição = inteiro
    inicial ≤ posição ≤ final
  - Processo: vetor [posição] := valor

- Acessar
  - Condição: tipo índice = inteiro
    inicial ≤ índice ≤ final
  - Processo: retorna (vetor [índice])
- Operações relativas às seqüências

6.5. Implementação de vetores

** Em C: int a [100];*

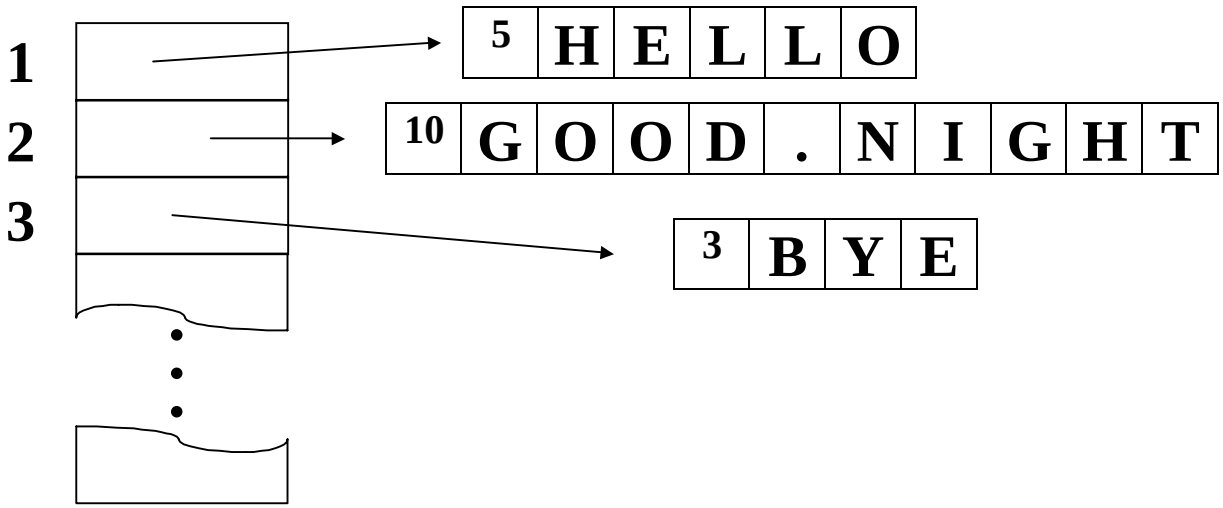
- *Reserva n posições sucessivas de memória*
- *Cada posição contém um único elemento*
- *O endereço da primeira posição é a base a*
- *Intervalo de 0 a (n-1)*
- *Forma geral:*

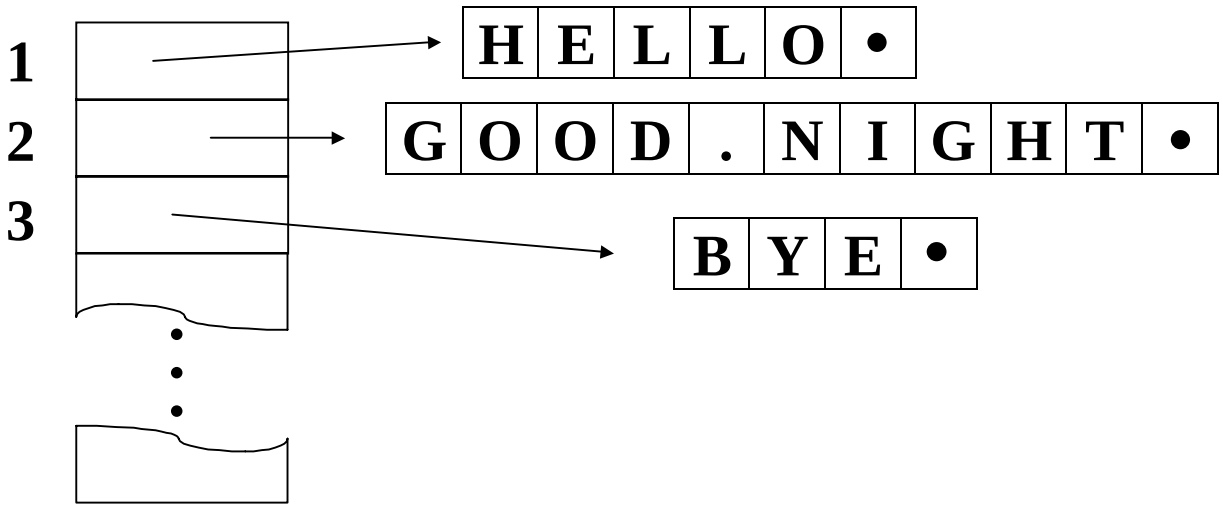
*a[pos]→ conteúdo { base(a) + pos * size - t }*

Todos os elementos possuem o mesmo tamanho, facilitando as operações básicas e a geração de código compacto e veloz.

INTRODUÇÃO

6.6. Implementação de vetores de elementos com tamanho variável





INTRODUÇÃO

7. Matrizes

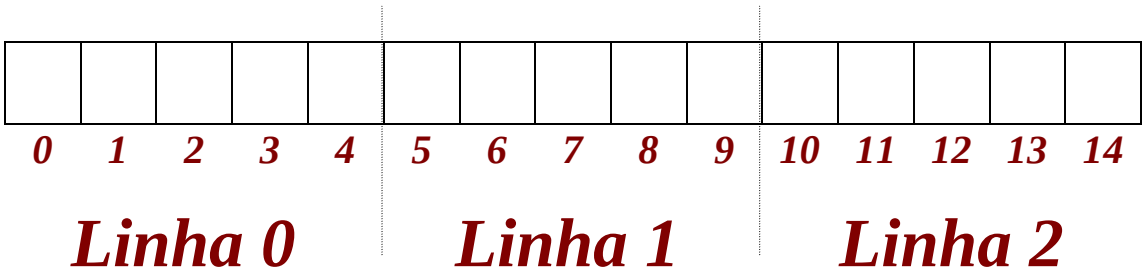
Nome usual para se definir vetores n-dimensionais.

71. Implementação de matrizes

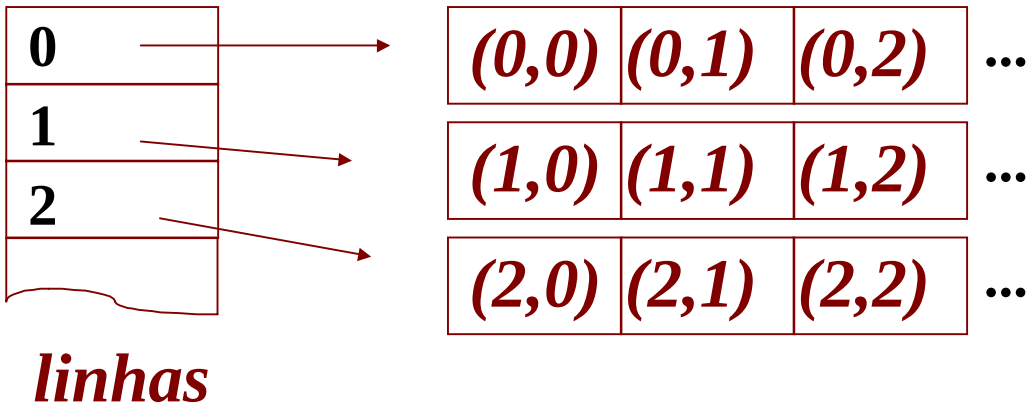
a[i][j] := @base (a) + i * numcol + j

Exemplo(1):

Matriz A [3] [5]



ou:



INTRODUÇÃO

7. Estruturas

Estrutura que agrega diferentes tipos de dados (heterogênea)

Exemplo (1):

```
Struct Teste{
        Int k;
        Char l[7];
        Double m;
        Char n[3];
} ;

Teste sopa;
```

sopa.<campo>:= @base (sopa) + int(<campo>)

INTRODUÇÃO

8. Desenvolvimento de software

8.1 Etapas:

- Especificação
- Análise
- Projeto
- Implementação
- Teste
- Manutenção

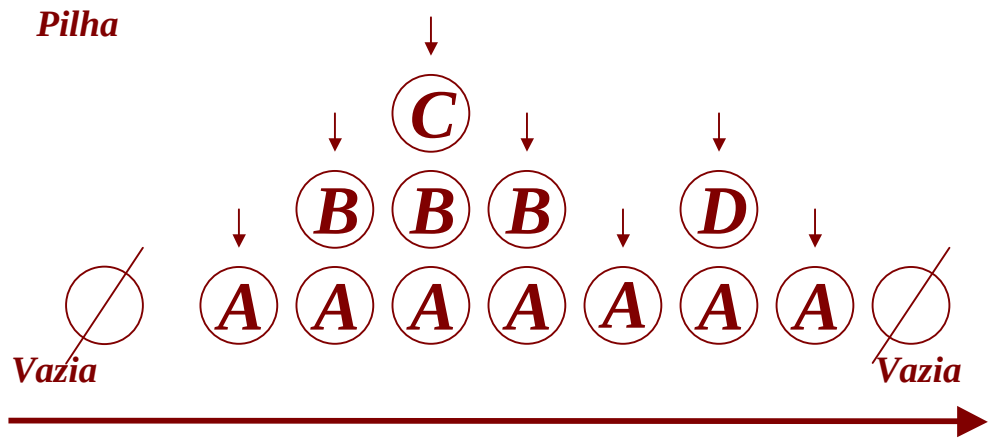
8.2 Dicas:

- Independência entre módulos facilita reutilização;
- Substituir código ruim;
- Análise: Escreva várias possibilidades;
- Documentação;
- Discuta soluções, observe estilos;
- Isole E/S em módulos específicos;
- Não traduzir uma linguagem em outra;
- Clareza acima de tudo;
- Evite variáveis globais;
- Evite GOTOs;
- Escreva funções genéricas;
- Critique os dados de entrada;
- Use todos os recursos disponíveis;
- Evite literais explícitos.

PILHA

1. Conceito

Conjunto de itens, no qual itens podem ser inseridos ou removidos em uma extremidade chamada topo.



empilha (A) (B) (C)
desempilha (B) (C)
empilha (D)
desempilha (D) (A)

LAST-IN FIRST-OUT

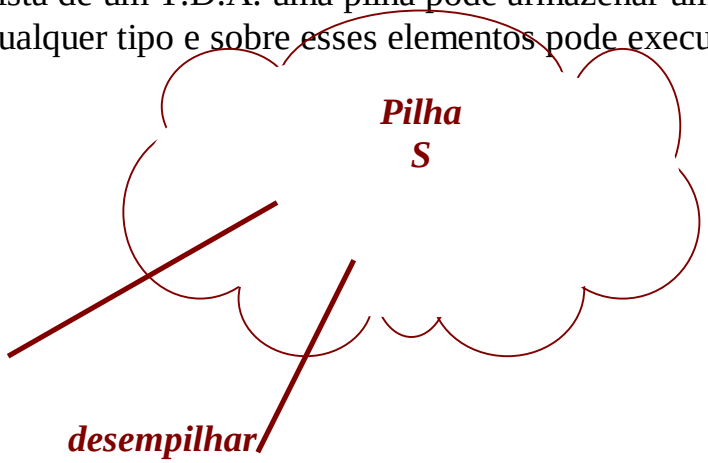
Armazenar = Empilhar = Push
Recuperar = Desempilhar = Pop

Pilha armazena de 0 a ∞ elementos de qualquer tipo.

PILHA

2. Operações primitivas

Do ponto de vista de um T.D.A. uma pilha pode armazenar um número de itens de qualquer tipo e sobre esses elementos pode executar as seguintes operações:



Seja
S uma pilha

1) Empilhar o item i na pilha S:
Push (S, i) → Armazena
2) Desempilhar o item do topo da pilha S:
Pop (S) → Remove

Push (S, I) - Sempre

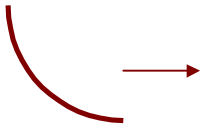
I = Pop (S) - Se lenght (S) > 0

PILHA

Outras primitivas:

Empty (S) ⇒ ***True - Se pilha vazia***
[Vazia (S)] ***False - Se pilha contém elemento***

Stacktop (S) ⇒ ***Retorna o item do topo da pilha sem removê-lo.***
[TopoPilha (S)]



I = Pop (S)
Push (S, I)

Obs.: Tentativa de desempilhar uma pilha vazia ⇒ Stack Underflow

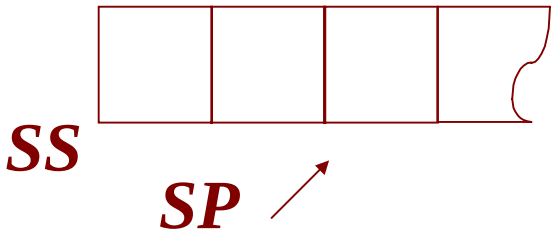
Exemplo:

Operação	***Retorno***	***Pilha***
Push (S, 'A')	***-***	***→ A***
Push (S, 'B')	***-***	***→ B A***
Pop (S)	***B***	***→ A***
Push (S, 'C')	***-***	***→ C A***
Empty (S)	***FALSO***	***→ C A***
Stacktop (S)	***C***	***→ C A***

PILHA

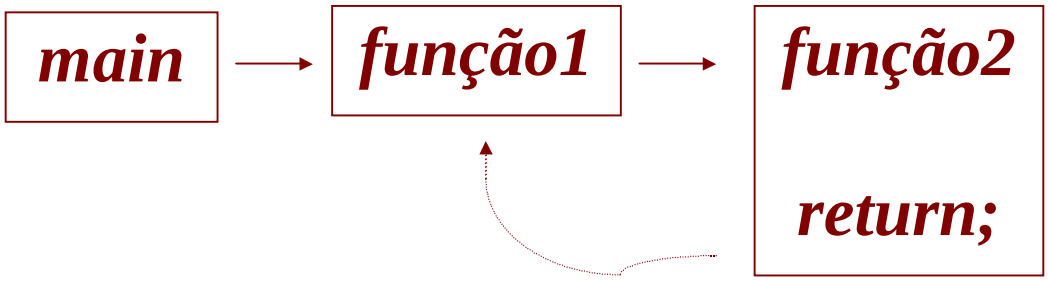
3. Aplicações

- Hardware do PC
Seção de Pilha (SS)

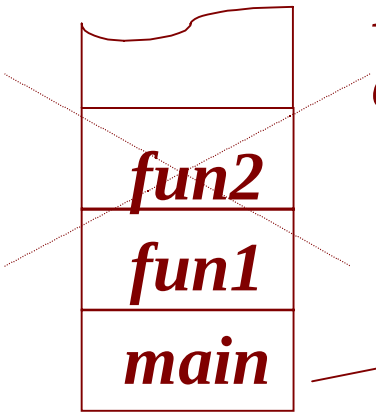


** PUSH AX*
MOV [SS], AX
INC SP
** POP AX*

- Pilha de Registro de Ativação



Retorno para último ambiente.



Cada registro possui a tabela de símbolos e o conteúdo das variáveis do módulo

PILHA

4. Verificação de parênteses

( ) [ ] { } { [( )] } [( ])

Parêntese à esquerda: abre escopo
Parêntese à direita: fecha escopo
Profundidade de aninhamento: abertos

<u>Exemplo(1):Um único tipo de parêntese</u>

*7 - ((x * ((x + y) / (j - 3)) + y) / 2)*
0 1 2 3 4 3 4 3 2 1 0

O.k.

(A + B)) - (C + D
1 0 -1 0

ñ O.k.

((A + B)
1 2 1

ñ O.k.

> ***É suficiente um programa com incremento / decremento para parênteses.***

<u>Exemplo(2):N tipos de parênteses</u> *([{*

Rastrear não só os escopos abertos, mas também suas correspondências

PILHA

Verificação de parênteses de vários tipos

Algoritmo básico:

Até final da expressão

1: Se início do escopo empilha.

Guarda, na ordem do último aberto, os escopos

2: Se fim de escopo desempilha

Examina se o finalizador atual corresponde ao último aberto

3: Se pilha vazia ou não coincidir → erro.

Pseudo código

```
válido := TRUE;
inicializa pilha S;
enquanto (não fim de expressão) {
        lê_próximo (símbolo) da expressão
        se (simbolo = '(' ou símbolo='[' ou símbolo = '{') faça
                push (S, símbolo);
        se (simbolo = ')' ou símbolo=']' ou símbolo = '}') faça
                se (empty (S)) faça
                        válido := FALSE;
                senão {
                        I = pop (S);
                        se (I não corresponde a símbolo) faça
                                válido := FALSE;
                        }
}
se (não empty (S)) válido := FALSE;
se (válido) ok
```

PILHA

5. Implementação da estrutura usando vetor:

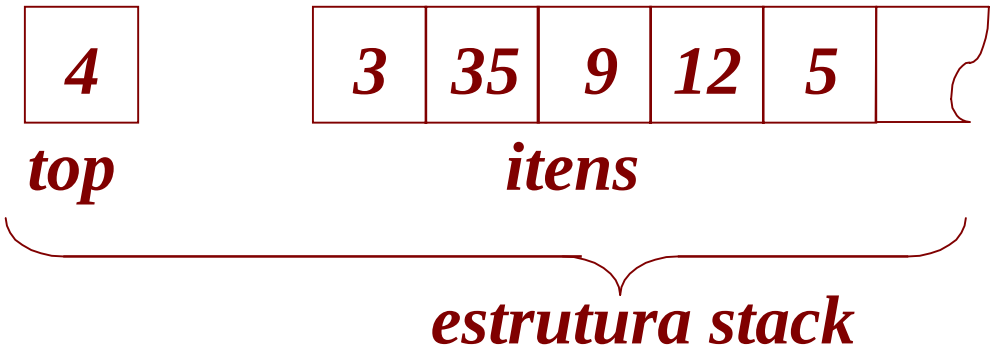
- *Limita a pilha a um tamanho físico*
- *Simplicidade*
- *Melhoria: Alocação dinâmica*

Estrutura para pilha de inteiros

```
# define STACKSIZE 100
 struct stack {
      int top;
      int itens [STACKSIZE];
 };
```

Tipo definido como pilha de inteiros

```
# define STACKSIZE 100
 typedef {
      int top;
      int itens [STACKSIZE];
 }Pilha;
```



struct stack S;
S.top = -1; → *Inicializa a pilha.*

OU MELHOR:

Pilha S; → *usa uma abstração de dados*
Inicia(S); → *usa uma abstração de código*

PILHA

Estrutura para pilha genérica (3 tipos de dados diferentes)

```
# define INT 1
# define FLT 2
# define STR 3
struct elementopilha {
   int e_tipo;
   union {
     int ival;
     float fval;
    char *sval;
   } elemento;
};
struct stack {
   int top;
   struct elementopilha itens[MAX];
}
struct stack S;
```

Melhor: usando typedef em vez de estruturas

6. Implementação das primitivas para pilha de inteiros

```
struct stack s;
s.top = -1;
if (s.top == -1) puts ("vazia")
else puts ("pilha não vazia")
```

para empilhar:
1) incrementar s.top
2) armazena

para desempilhar:
1) armazena
2) decrementa s.top;

Primitivas são procedimentos ou funções que ocultam detalhes de programação e não podem ser descritos em função de outras primitivas

PILHA

Implementação de primitivas

```
→ empty (  struct stack *os)
{       if (ps -> top == -1)
                return (TRUE);
                else return (FALSE);
}

→ pop (struct stack *ps)
{       if (empty (ps))
        {       print ("Stack underflow");
                exit (1);    }
        return (ps -> itens [ps -> top - -]);
}

→ push (struct stack *ps, int x)
{       ps -> itens [++(ps -> top)] = x;
        return (1);
 }
```

O problema das N-Rainhas

Suponha que você tenha 8 rainhas de um jogo de xadrez. As rainhas podem ser dispostas no tabuleiro de modo que duas rainhas não se ataquem. Duas rainhas não são permitidas na mesma linha ou na mesma coluna. O número de Rainhas e o tamanho do tabuleiro podem variar.

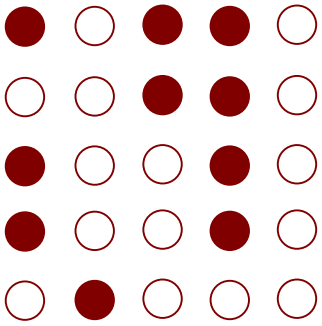
Fazer um programa que tenta encontrar uma maneira de colocar N Rainhas num tabuleiro N x N. (Permitir que sejam colocadas até mais de N rainhas caso seja possível)

O programa usa uma pilha para guardar onde cada rainha é colocada. Cada vez que o programa decide colocar uma rainha no tabuleiro, a posição da nova rainha é guardada em um registro que é colocado na pilha. Também temos um inteiro para guardar quantas linhas foram preenchidas até o momento (L). Cada vez que o programa decide colocar uma rainha no tabuleiro, a posição da nova rainha é guardada em um registro que é colocado na pilha. Se há conflito com outra rainha, mudamos a nova rainha para a próxima coluna à direita. Se outro conflito acontece, a rainha é novamente mudada para a próxima coluna. Quando não há mais conflitos, paramos e adicionamos 1 ao valor de L.

Se não for possível colocar uma rainha em uma dada coluna, após vários deslocamentos, concluímos que o posicionamento atual das demais rainhas está impedindo-nos. Daí, temos que fazer um chamado *backtracking* (retrocesso) para a **última** rainha colocada e tentar coloca-la numa nova posição, caso seja possível, e continuar o restante do processo a partir dessa nova posição. O retrocesso é feito retirando a posição da última rainha da pilha. Este raciocínio é seguidamente repetido até que encontremos uma configuração sem ataques.

PILHA

7. Uma aplicação gráfica: Algoritmo de pintura



1 0 1 1 0	2 0 4 6 0
0 0 1 1 0	0 0 1 4 0
1 0 0 1 0	5 0 0 8 0
1 0 0 1 0	1 0 0 8 0
0 1 0 0 0	0 1 0 0 0

imagem formada por PIXELS — *matricial monocromática* — *policromática*

Pixels conectados: vizinhos com mesma cor
Para pintar uma região:
- *Um ponto inicial dentro da região*
- *Uma nova cor*

2 0 4 6 0	2 7 4 6 7
0 0 1 4 0	7 7 1 4 7
5 0 0 8 0	5 7 7 8 7
1 0 0 8 0	1 7 7 8 7
0 1 0 0 0	0 1 7 7 7

0 → 7
a partir de (1, 1)

Vizinhança
4 - Conectada

PILHA

Algoritmo de pintura

Sugestão: Usar pilha para armazenar temporariamente pontos a serem pintados

```
pinta (int x, int y, int cor)
   push inicial
  guardar cor_antiga
   enquanto pilha não vazia
      pop ponto_p
     se cor do ponto_p = cor_antiga
        push 4-conectados
        pinta ponto_p com cor
     fim
fim
```

Questão de implementação em C

```
    c = getpixel (x, y)
          putpixel (x, y, c)
```

Obs.: Pilha armazena uma coordenada

PILHA

8. Notação Infixa, Posfixa, Prefixa

Referente à posição do operador em relação aos operandos

A+B *+AB* *AB+*

Outro exemplo: add (A, B)

8.1 Conversão de notação

→ Infixa para posfixa

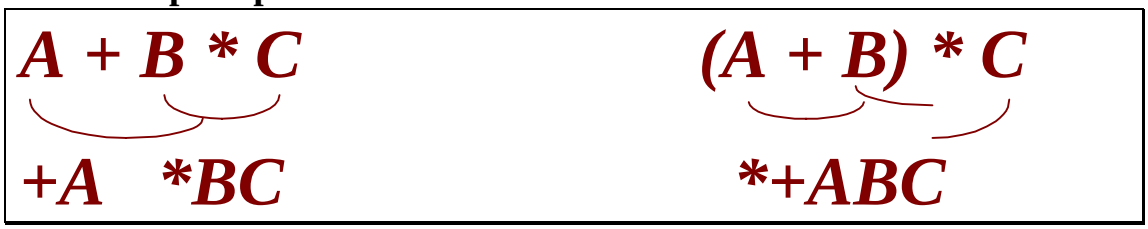
*A + B * C*	*(A + B) * C*
A BC +*	*AB+ C**

- *Operações com maior precedência são convertidas antes das demais*
- *Desnecessário uso de parênteses*
- *Tabela de precedência:*

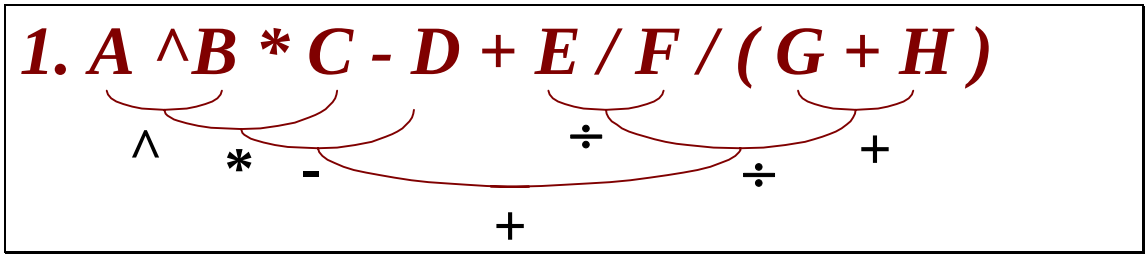
Exponenciação	^
Multiplicação / Divisão	* /
Adição / Subtração	+ -

PILHA

→ **Infixa para prefixa**



Exemplos:



AB^ EF/ GH+

AB^C* EF/GH+/

AB^C*D-EF/GH+/+

Posfixa.

^AB /EF +GH

***^ABC //EF+GH**

+-*^ABCD//EF+GH

Prefixa.

*2. ((A+B)*C-(D-E))^(F+G)*
*3. A-B/(C*D^E)*
4. (A+B)(C-D)*

PILHA

8.2. Avaliando uma expressão posfixa

Idéia básica: ***Se for operando, empilha.***

Logo: se operador, desempilha os 2 operandos da pilha, aplica o operador e empilha o resultado para os próximos operadores.

Ex.: 6 2 3 + - 3 8 2 / + * 2 ^ 3 +

termo	*ação*	*pilha*
6	*empilha*	*6 ←*
2	*Empilha*	*6 2 ←*
3	*Empilha*	*6 2 3 ←*
+	*desempilha e soma*	*6 5 ←*
-	*desempilha e subtrai*	*1 ←*
3	*Empilha*	*1 3 ←*
8	*Empilha*	*1 3 8 ←*
2	*Empilha*	*1 3 8 2 ←*
/	*desempilha e divide*	*1 3 4 ←*

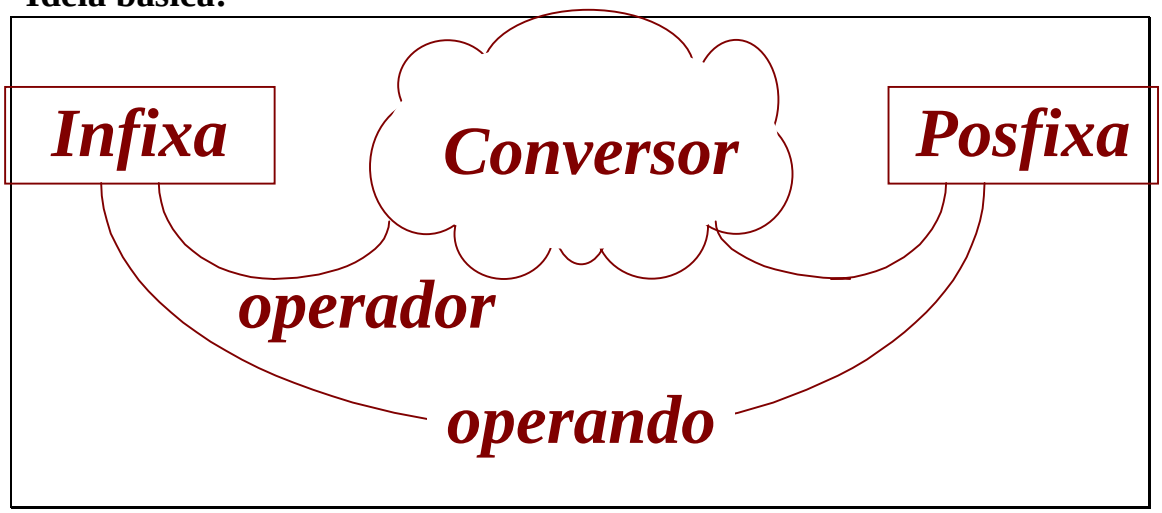
resposta: 52

Algoritmo de avaliação de expressão posfixa.

```
str := expressão_posfixa
s := pilha_vazia
enquanto (não fim str)
   elemento := le_elemento (str)
   se elemento é operando
      empilha (s, elemento)
   senão
      operando1 := desempilha (S)
      operando2 := desempilha (S)
      valor := aplica (operando1, operando2, elemento)
      empilha (S, valor)
   fim_se
fim_enquanto
resultado := desempilha (S)
```

PILHA

8.3. Conversão de infixa para posfixa
Idéia básica:



Se precedência do operador_infixa for maior que operador_pilha então empilha operador_infixa

Exemplo: ***A + B * C***

termo	*posfixa*	*pilha*
A	*A*	. ←
+	*A*	+ ←
B	*AB*	+ ←
*	*AB*	+* ←
C	*ABC*	+* ←
.	*ABC**	+ ←
.	*ABC*+*	. ←

PILHA

Algoritmo de conversão infixa para posfixa

```
s := pilha_vazia
infixa := expressao_infixa
enquanto (nao fim infixa)
   simbolo := le_infixa()
   se simbolo for operando
      inclui em posfixa
   senão
      enquanto (não pilha vazia  &  precedência > (stacktop (s), simbolo))
            simbolo_topo := pop (s)
            inclui simbolo_topo em posfixa
      fim_enq
       push (s, simbolo)
  fim_se
fim_enq
enquanto ( não empty (s) )
   simbolo_topo := pop (s)
  inclui simbolo_topo em posfixa
fim_enq
```

Enquanto (operador topo precede operador símbolo) desempilha)

Infixa para posfixa com parênteses

Alterações no algoritmo anterior

- *Função precedência > (pilha, símbolo)*
- *Quando símbolo = '(' ele é introduzido na pilha. Para isso:*
  Precedência > (pilha, '(') - .F.
  i.e., qualquer elemento na pilha é inferior a '('

Quando símbolo = ')' retira todos da pilha até o '('. Basta:
Precedência > (pilha, ')') - .V.
i.e., qualquer elemento na pilha é superior a ')'. Exceto '('

Ex.: ⇒ *(A+B)*C*

termo	*posfixa*	*pilha*
(	*.*	*(←*
A	*A*	*(←*
+	*A*	*(+ ←*
B	*AB*	*(+ ←*
)	*AB+*	*. ←*
***	*AB+*	** ←*
C	*AB+C*	** ←*
.	*AB+C**	*. ←*

PILHA

Infixa para posfixa com parêntese

substituição de *push (S, simbolo)* por:

```
if ( empty (S) ou simbolo ≠ ')' )
   push (S, simbolo)
else
   pop (S)
```

Ex.: ***((A-(B+C))*D)^E+F)***

item	*posfixa*	*pilha*
(	*.*	*(*
(	*.*	*((*
A	*A*	*((*
-	*A*	*((-*
(	*A*	*((-(*
B	*AB*	*((-(*
+	*AB*	*((-(+*
C	*ABC*	*((-(+*
)	*ABC+*	*((-*
)	*ABC+-*	*(*
***	*ABC+-*	*(**
D	*ABC+-D*	*(**
)	*ABC+-D**	*.*
^	*ABC+-D**	^
E	*ABC+-D*E*	^