Aplicações de Pilhas

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará
Campus Quixadá

1º semestre/2021

Introdução

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([ ]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([ ]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Como usar pilha para testar se a sequência é bem-formada?

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([ ]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([ ]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

1. leu (ou [: empilha o símbolo lido

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([ ]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

1. leu (ou [: empilha o símbolo lido
2. leu]: desempilha [

Balanceamento de parênteses e colchetes

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Dizemos que uma sequência de parênteses e colchetes é **balanceada** ou **bem-formada** se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([ ]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Como usar pilha para testar se a sequência é bem-formada?

Para testar, leia cada símbolo e se:

1. leu (ou [: empilha o símbolo lido
2. leu]: desempilha [
3. leu): desempilha (

Implementação em C++

Implementação em C++


```
1  // Esta funcao devolve true se exp contiver uma
2  // sequencia bem-formada de parenteses e chaves e
3  // devolve false se a sequencia estiver malformada.
4  bool bemFormada(std::string exp) {
5    std::stack<char> pilha;
6
7    for(int i = 0; i < exp.size(); i++) {
8      switch (exp[i]){
9      case ')': if(!pilha.empty() && pilha.top() == '(')
10               pilha.pop();
11             else return false;
12             break;
13     case ']': if(!pilha.empty() && pilha.top() == '[')
14               pilha.pop();
15             else return false;
16             break;
17     default : pilha.push(exp[i]);
18       break;
19     }
20   }
21   return pilha.empty();
22 }
```

Notação de expressões

Notação de expressões

Notação de expressões

Exemplo 1:

Notação de expressões

Exemplo 1:

- Infixa: `a + b`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`

Notação de expressões

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

Notação de expressões

Universidade Federal do Ceará
Campus Quixadá

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. Posfixa: é notação polonesa reversa (RPN), das calculadoras HP.

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Prefixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. Posfixa: é notação polonesa reversa (RPN), das calculadoras HP.
   - Operador sucede operandos

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( (2 + 1) * 4 + 1)`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( 3 * 4 + 1)`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * ( 12 + 1)`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

`2 * 13`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 2

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

`2 2 1`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$2\ 2\ 1\ +$$

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 3

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

`2 3 4`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

`2 3 4 *`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 12

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$2 \;\; 12 \;\; 1$$

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

`2 12 1 +`

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação posfixa:

2 13

Exemplo de cálculo de expressão na notação posfixa

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

`2 13 *`

Exemplo de cálculo de expressão na notação posfixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
- Posfixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

$$26$$

Resolvendo com notação posfixa:

$$26$$

Calculando expressões posfixas

Algoritmo:

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:
     - empilha n

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_2 \oplus operando_1$

Calculando expressões posfixas

Algoritmo:

1. Para cada elemento lido na sequência de entrada:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_2 \oplus operando_1$

2. Ao final, desempilha o único valor contido na pilha e retorna.

Calculando expressões posfixas

Problema: Suponha dada uma expressão aritmética em notação posfixa sujeita às seguintes restrições:

1. cada número consiste nos inteiros do conjunto `0,1,...,9`;
2. os únicos operadores são $+$, $-$, $*$, $/$

Escreva uma função que calcule o valor da expressão.

Solução — Implementação em C++

Solução — Implementação em C++


```
1  // Supoe que 'posfix' contem expressao posfixa valida
2  double calculaPosfixa(std::string posfix) {
3      double a, b;
4      stack<double> pilha;
5
6      for(unsigned i = 0; i < posfix.size(); i++) {
7          if( isdigit(posfix[i]) ) {
8              char ch = posfix[i];
9              pilha.push( atof(&ch) );
10         }
11         else {
12             a = pilha.top(); pilha.pop();
13             b = pilha.top(); pilha.pop();
14             switch(posfix[i]) {
15                 case '+': pilha.push(b + a); break;
16                 case '-': pilha.push(b - a); break;
17                 case '*': pilha.push(b * a); break;
18                 case '/': pilha.push(b / a); break;
19             }
20         }
21     }
22     return pilha.top();
23 }
```

Converter uma expressão completamente parentizada para a notação posfixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Objetivo:

`(1 + (((2 * 3) / 4) * 5))` $\Rightarrow$ `1 2 3 * 4 / 5 * +`

Converter uma expressão completamente parentizada para a notação posfixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Objetivo:

`(1 + (((2 * 3) / 4) * 5))` $\Rightarrow$ `1 2 3 * 4 / 5 * +`

Algoritmo:

- Copiamos os números diretamente na saída
- Quando aparecer '(' na entrada, ignoramos
- Quando aparecer um novo operador na entrada:
  - empilhamos o operador novo
- Quando aparecer ')' na entrada:
  - desempilhamos um operador, copiando para a saída

Solução — Implementação em C++

Solução — Implementação em C++

Universidade Federal do Ceará
Campus Quixadá

```
1  std::string ParentizadaParaPosfixa(std::string exp) {
2      std::string posfix;
3      stack<char> pilha; // guarda os operadores
4
5      for(int i = 0; i < exp.size(); i++) {
6          switch(exp[i]) {
7              case '(': break;
8              case ')': posfix += pilha.top();
9                        pilha.pop();
10                       break;
11             case '+':
12             case '-':
13             case '*':
14             case '/': pilha.push(exp[i]);
15                       break;
16             default : posfix += exp[i];
17         }
18     }
19     return posfix;
20 }
```

Para casa: Convertendo de infixa para pós-fixa

Universidade Federal do Ceará
Campus Quixadá

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Para casa: Convertendo de infixa para pós-fixa

Objetivo:

$$\texttt{1 + 2 * 3 / 4 * 5 - 6} \Rightarrow \texttt{1 2 3 * 4 / 5 * + 6 -}$$

Ideia:

Para casa: Convertendo de infixa para pós-fixa

Universidade Federal do Ceará
Campus Quixadá

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` $\Rightarrow$ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída

Para casa: Convertendo de infixa para pós-fixa

Universidade Federal do Ceará
Campus Quixadá

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` $\Rightarrow$ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:

Para casa: Convertendo de infixa para pós-fixa

Universidade Federal do Ceará
Campus Quixadá

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada

Para casa: Convertendo de infixa para pós-fixa

Universidade Federal do Ceará
Campus Quixadá

Objetivo:

```
1 + 2 * 3 / 4 * 5 - 6  ⇒  1 2 3 * 4 / 5 * + 6 -
```

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída

Para casa: Convertendo de infixa para pós-fixa

Universidade Federal do Ceará
Campus Quixadá

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` $\Rightarrow$ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo

Para casa: Convertendo de infixa para pós-fixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Objetivo:

$$\texttt{1 + 2 * 3 / 4 * 5 - 6} \Rightarrow \texttt{1 2 3 * 4 / 5 * + 6 -}$$

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Para casa: Convertendo de infixa para pós-fixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Objetivo:

$$\texttt{1 + 2 * 3 / 4 * 5 - 6} \quad \Rightarrow \quad \texttt{1 2 3 * 4 / 5 * + 6 -}$$

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Pergunta:

Para casa: Convertendo de infixa para pós-fixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` $\Rightarrow$ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparecer um operador na entrada:
  - enquanto o operador no topo tiver precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Pergunta:

- Como generalizar para o caso em que a expressão tem parênteses?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Pilhas e Recursão

Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

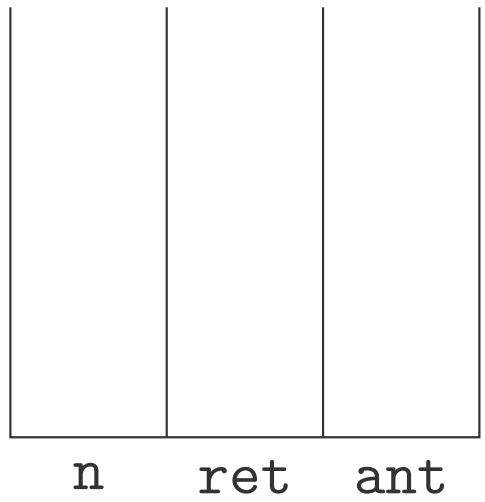
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

Vamos tentar descobrir simulando uma chamada: `fat(4)`

Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

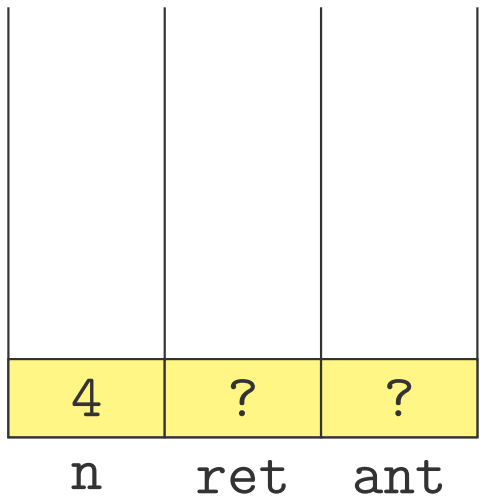
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

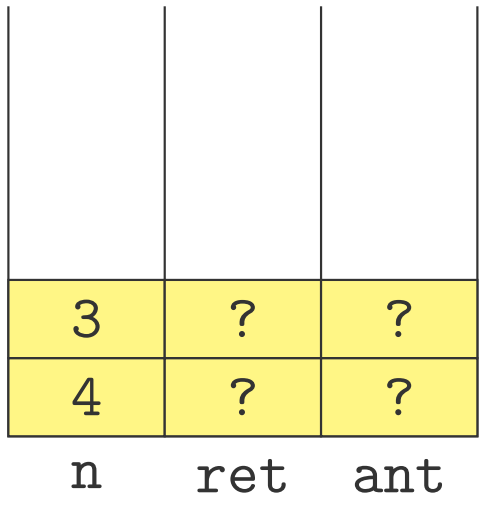
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

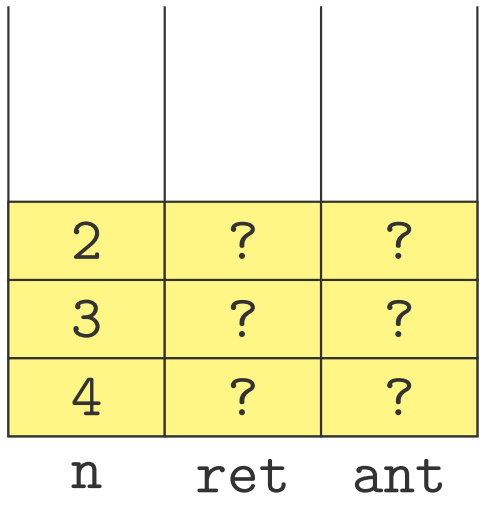
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

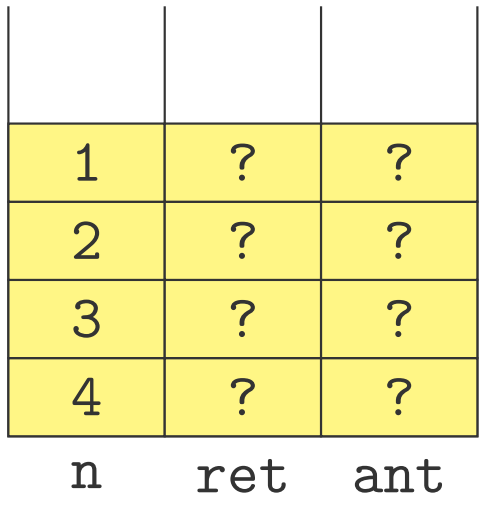
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

0	?	?
1	?	?
2	?	?
3	?	?
4	?	?
n	ret	ant

Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

n	ret	ant
0	1	?
1	?	?
2	?	?
3	?	?
4	?	?

Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

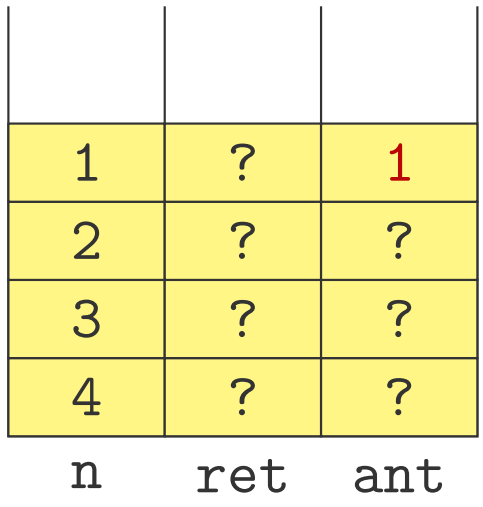
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

n	ret	ant
0	1	?
1	?	?
2	?	?
3	?	?
4	?	?

Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

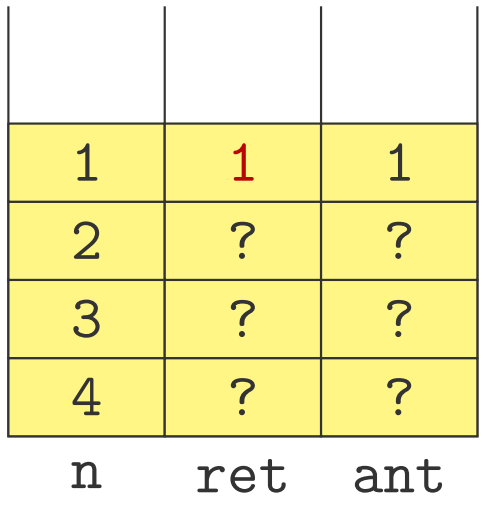
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

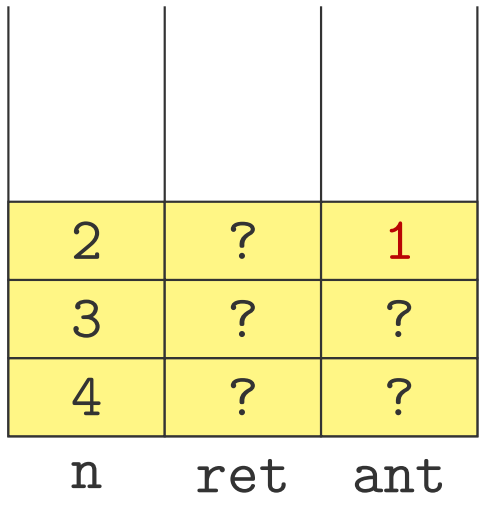
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

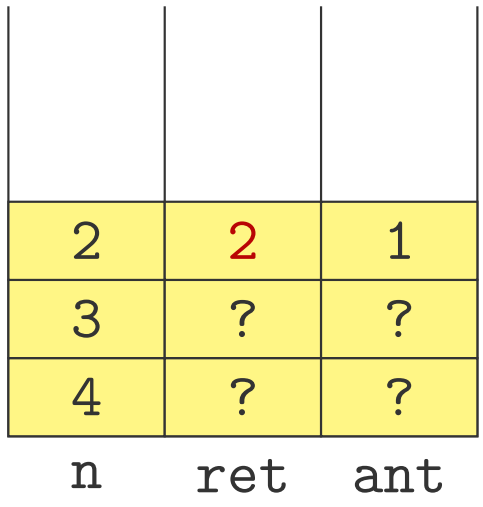
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

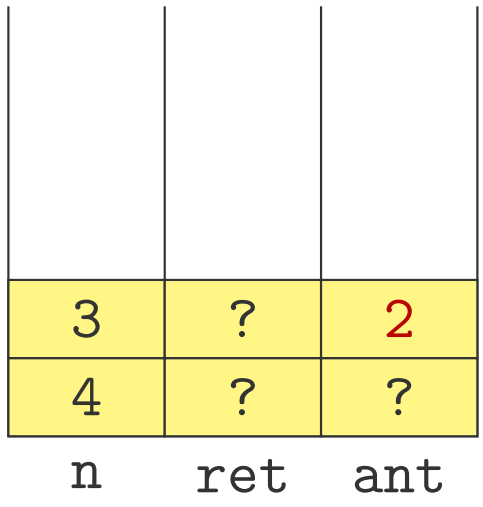
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

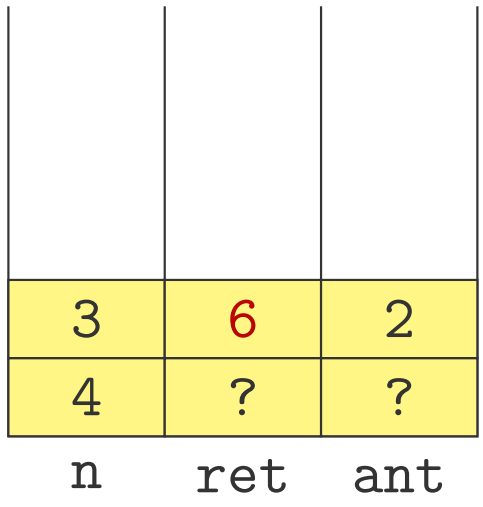
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

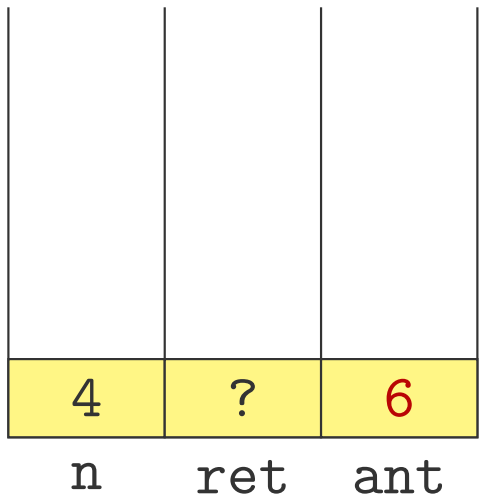
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

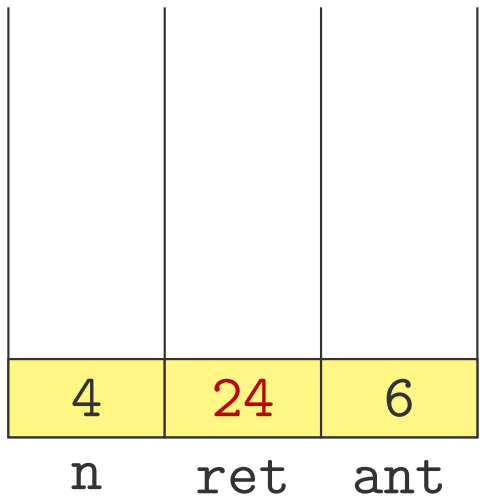
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilhas e recursão (continuando)

Quando empilhamos:

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

O conjunto de variáveis locais formam um elemento da pilha

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

O conjunto de variáveis locais formam um elemento da pilha

Isto é, a recursão pode ser simulada usando uma pilha de suas variáveis locais

Buscando um elemento em uma lista encadeada

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Buscando um elemento em uma lista encadeada

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Note que:

Buscando um elemento em uma lista encadeada

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Note que:

- A recursão é a última coisa antes do retorno da função.

Buscando um elemento em uma lista encadeada

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Note que:

- A recursão é a última coisa antes do retorno da função.
- Apenas retornamos o valor de `busca_rec(node->next, v)` sem manipulá-lo.

Buscando um elemento em uma lista encadeada

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Note que:

- A recursão é a última coisa antes do retorno da função.
- Apenas retornamos o valor de `busca_rec(node->next, v)` sem manipulá-lo.
- Exceto na base, o retorno não depende do valor das variáveis locais.

Buscando um elemento em uma lista encadeada

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Note que:

- A recursão é a última coisa antes do retorno da função.
- Apenas retornamos o valor de `busca_rec(node->next, v)` sem manipulá-lo.
- Exceto na base, o retorno não depende do valor das variáveis locais.
  - Depende apenas do valor da chamada recursiva.

Eliminação de Recursão

Podemos eliminar o uso de recursão na nossa função

Eliminação de Recursão

Podemos eliminar o uso de recursão na nossa função

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Eliminação de Recursão

Podemos eliminar o uso de recursão na nossa função

Versão recursiva:

```
1 Node* busca_rec(Node* node, int v) {
2     if (node == NULL || node->dado == v)
3         return node;
4     else
5         return busca_rec(node->next, v);
6 }
```

Eliminando a recursão:

```
1 Node* busca_iterativa(Node* node, int v) {
2     while(node != NULL && node->key != v)
3         node = node->next;
4     return node;
5 }
```

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`
- usando um `while`

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`
- usando um `while`
- até chegar em uma das bases da recursão

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`
- usando um `while`
- até chegar em uma das bases da recursão

Chamamos esse tipo de recursão de recursão de cauda

Recursão geral

Note que `hanoi` não tem recursão de cauda e ainda faz duas chamadas recursivas

Recursão geral

Note que `hanoi` não tem recursão de cauda e ainda faz duas chamadas recursivas

```
1 void hanoi(int n, char orig, char dest, char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     cout << "move de " << orig << " para " << dest << endl;
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

Recursão geral

Note que `hanoi` não tem recursão de cauda e ainda faz duas chamadas recursivas

```
1 void hanoi(int n, char orig, char dest, char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     cout << "move de " << orig << " para " << dest << endl;
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

Recursões que não são de cauda também podem ser eliminadas

Recursão geral

Note que `hanoi` não tem recursão de cauda e ainda faz duas chamadas recursivas

```
1 void hanoi(int n, char orig, char dest, char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     cout << "move de " << orig << " para " << dest << endl;
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

Recursões que não são de cauda também podem ser eliminadas

- Porém é necessário utilizar uma pilha

Recursão geral

Note que `hanoi` não tem recursão de cauda e ainda faz duas chamadas recursivas

```
1 void hanoi(int n, char orig, char dest, char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     cout << "move de " << orig << " para " << dest << endl;
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

Recursões que não são de cauda também podem ser eliminadas

- Porém é necessário utilizar uma pilha
- E o processo é mais trabalhoso

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos
- Não precisamos empilhar registros a cada iteração

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos
- Não precisamos empilhar registros a cada iteração

Eliminação de recursão de cauda é uma ótima forma de otimização

- E é feita automaticamente por alguns compiladores

Exercícios

Exercício

- Utilizando uma pilha, escreva uma função que verifique se uma string de entrada é da forma

$$str_1 C str_2$$

tal que str_1 é uma string composta apenas por caracteres A e B e str_2 é a string reversa de str_1.

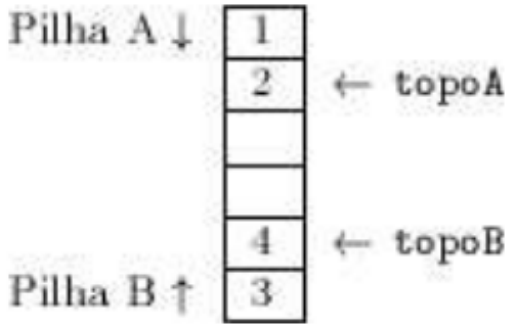
- Por exemplo, a cadeia $ABABBACABBABA$ é do formato especificado, enquanto as cadeias $ABABBACABB$, ABA, $BBBB$, AAA, $BBBBCAA$ e $ABBACBAABBBBBAB$ não seguem o formato.
- Sua função deve obedecer o seguinte protótipo:
`bool str1Cstr2(std::string& str);`
- Restrição: A string dada como entrada para a função deve ser percorrida uma única vez da esquerda para a direita.

Exercício

- Faça um programa em C++ para ler um número inteiro maior que zero, converter este número de decimal para binário, usando pilha e apresentar na tela, o resultado da conversão.

Exercício – Duas Pilhas em um vetor

Duas pilhas A e B podem compartilhar o mesmo vetor, como esquematizado na figura abaixo:



Faça as declarações de constantes e tipos necessárias e escreva as seguintes rotinas:

(a) `criaPilhas()`, que inicia os valores de topoA e topoB.

(b) `vaziaA()` e `vaziaB()`.

(c) `empilhaA(int x)` e `empilhaB(int x)`.

(d) `desempilhaA()` e `desempilhaB()`.

Observação: Só deve ser emitida uma mensagem de pilha cheia se todas as posições do vetor estiverem ocupadas.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM