CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Aplicações de Pilhas
Parte 2

Disciplina: Estrutura de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Aplicações de Pilhas (Stacks)

- Estudaremos quatro grupos de aplicações de pilhas:
  - Inversão de dados;
  - Análise sintática (*Parsing*);
  - Adiamento do uso de dados;
  - *Backtracking*.

Aplicações de Pilhas (Stacks)

- Adiamento do uso de dados
  - Converter notação infixa em posfixa;
  - Avaliar expressões posfixas;
- *Backtracking*
  - Escolher entre dois o mais caminhos;
  - Problema das oito rainhas.

Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa

- Método Manual.

```
A + B * C
```

Step 1 results in

```
( A + ( B * C ) )
```

Step 2 moves the multiply operator after C

```
( A + ( B C * ) )
```

and then moves the addition operator to between the last two closing parentheses. This change is made because the closing parenthesis for the plus sign is the last parenthesis. We now have

```
( A (B C * ) +)
```

Finally, step 3 removes the parentheses.

```
A B C * +
```

Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa

- Método Manual.

```
                    ( A + B ) * C  + D + E * F - G
```

Step 1 adds parentheses.

```
      ( ( ( ( ( A + B ) * C ) + D ) + ( E * F ) ) - G )
```

Step 2 then moves the operators.

```
      ( ( ( ( ( A B + ) C * ) D + ) ( E F * ) + ) G - )
```

Step 3 removes the parentheses.

```
                    A B + C * D + E F * + G -
```

Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa

- Transformação Algorítmica.

```
A * B    converts to    A B *
```

  - Onde usar a pilha? O que deverá ser guardado?
  - Como usar?

  - Empilhar todos os operadores? Desempilhar todos e aplicar?
  - Empilhar somente até achar outro operador. Desempilhar o mais antigo. Empilhar o mais novo?

```
A * B + C    converts to    A B * C +
```

  - E a regra de precedência entre operadores?

```
A + B * C    converts to    A B C * +
```

Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa

- Transformação Algorítmica.
  - E a regra de precedência entre operadores?

`A + B * C` converts to `A B C * +`

1. Copy operand `A` to output expression.
2. Push operator + into stack.
3. Copy operand `B` to output expression.
4. Push operator * into stack. (Priority of * is higher than +.)
5. Copy operand `C` to output expression.
6. Pop operator * and copy to output expression.
7. Pop operator + and copy to output expression.

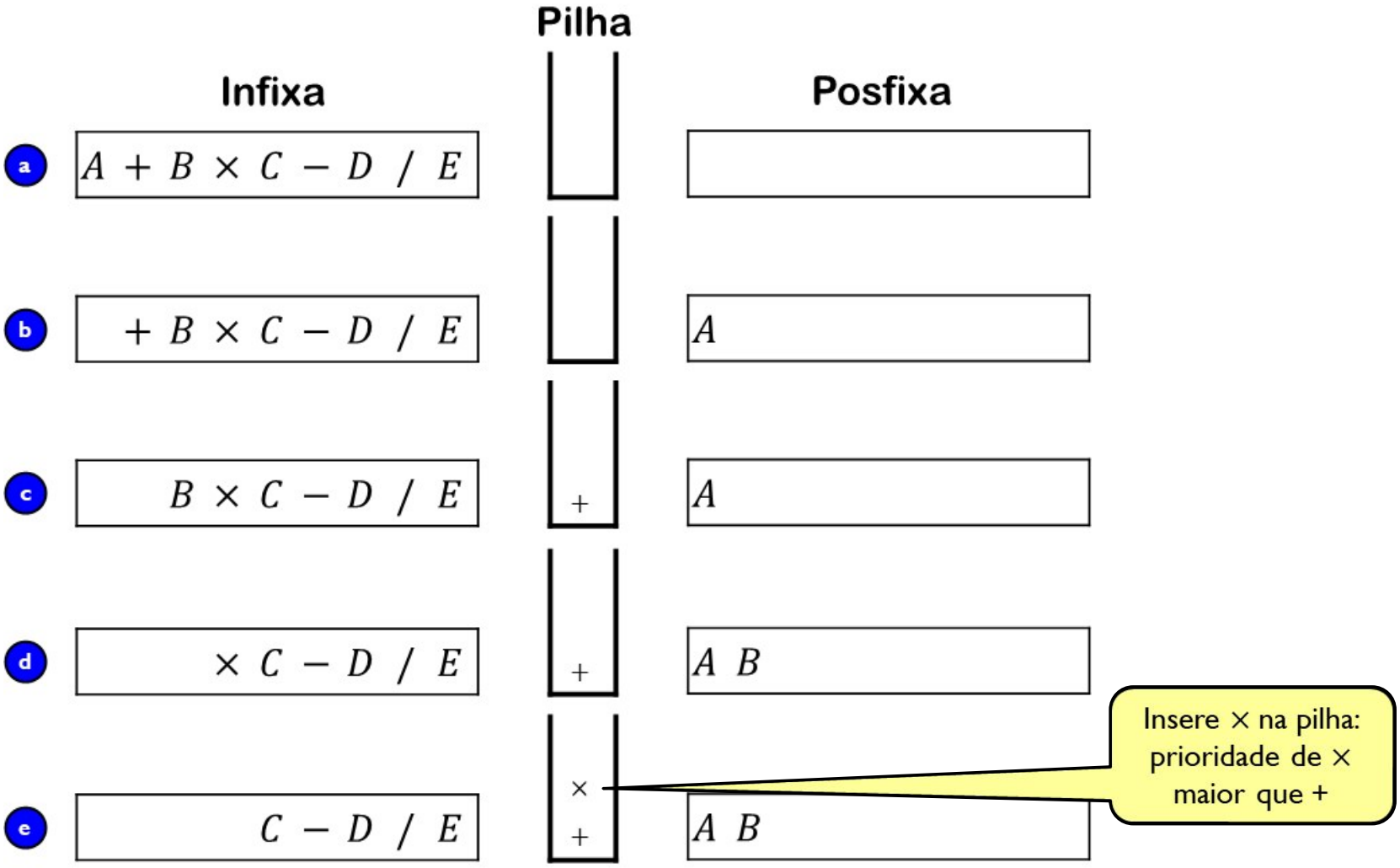
Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa

- Transformação Algorítmica.
  - Ao encontrar um novo operador com prioridade inferior ou igual ao que está no topo da pilha, devemos desempilhar o operador do topo e colocar ele na expressão posfixa.
  - O novo topo pode pela sua vez ter prioridade sobre o novo operador. Com isso o processo pode-se repetir várias vezes.
  - O novo operador somente será empilhado após remover todos os operadores com prioridade maior ou igual na pilha.

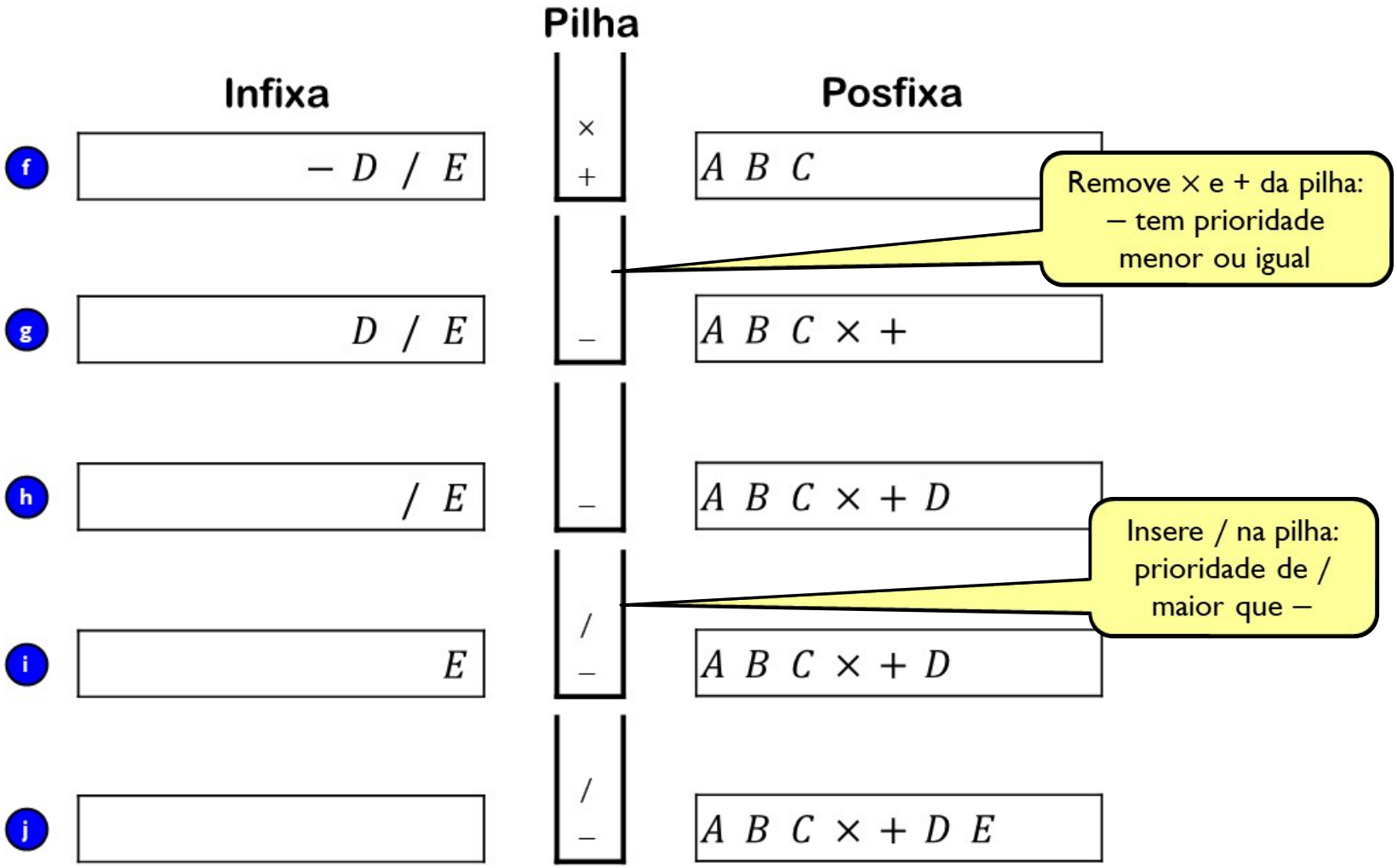
Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa



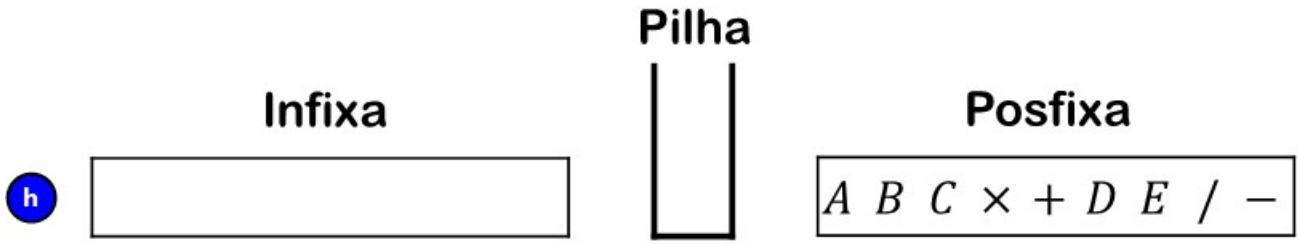
Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa



Aplicações de Pilhas (Stacks)

Adiamento do uso de dados - Notação infixa em posfixa



Aplicações de Pilhas (Stacks)

Notação Infixa em posfixa - Implementação

- Na primeira parte do programa principal definem-se as variáveis locais para armazenar o caráter lido e a expressão posfixa.
- Utiliza-se uma pilha para armazenar os operadores considerando a sua ordem de precedência.

P3-18.c

```
 7  #include <stdio.h>
 8  #include <string.h>
 9  #include "stacksADT.h"
10
11  // Prototype Declarations
12  int  priority (char token);
13  bool isOperator (char token);
14
15  int main (void)
16  {
17  // Local Definitions
18     char   postfix [80] = {0};
19     char   temp [2] = {0};
20     char   token;
21     char*  dataPtr;
22     STACK* stack;
23
24  // Statements
25     // Create Stack
26     stack = createStack ();
27
28     // read infix formula and parse char by char
29     printf("Enter an infix formula: ");
```

Expressão posfixa

String para um operador

Caracter da expressão infixa

Cria a pilha de operadores

P3-18.c (Continuação...)

- O loop principal do programa para transformar uma expressão infixa em posfixa consiste na leitura dos caracteres da expressão infixa.
- O caracter é processado segundo 4 casos possíveis, se for:
  - um ‘(‘ ;
  - um ‘)’;
  - um operador;
  - ou um operando.

```
31  while ((token = getchar ())!= '\n')
32     {
33      if (token == '(')                          // Caso1
34         {
35          dataPtr  = (char*) malloc (sizeof(char));
            *dataPtr = token;
            pushStack (stack, dataPtr);            // Empilha token
38         } // if
39      else if (token == ')')                     // Caso2
40         {
            dataPtr = (char*)popStack (stack);     // Desempilha
42          while (*dataPtr != '(')
               {
                temp [0]= *dataPtr;                // Concatena no final de expressão posfixa
                strcat (postfix , temp);
                dataPtr = (char*)popStack (stack);
46             } // while
47         } // else if
48      else if (isOperator (token))               // Caso3
49         {
            // test priority of token at stack top
            dataPtr = (char*)stackTop (stack);     // Consulta topo da pilha
            while (!emptyStack (stack)
              && priority (token) <= priority (*dataPtr))
               {
                dataPtr  = (char*)popStack (stack);  // Desempilha
                temp [0] = *dataPtr;
                strcat (postfix , temp);           // Concatena no final de expressão posfixa
                dataPtr  = (char*)stackTop (stack);
60             } // while
61          dataPtr  = (char*) malloc (sizeof (char));
            *dataPtr = token;
            pushStack (stack , dataPtr);           // Empilha token
64         } // else if
65      else                                       // character is operand
           {                                       // Caso4
            temp[0]= token;
            strcat (postfix , temp);               // Concatena no final de expressão posfixa
            } // else
70   } // while get token
```

Aplicações de Pilhas (Stacks)

Notação Infixa em posfixa - Implementação

- Na última parte do programa principal após a leitura e processamento de todos os caracteres da expressão infixa, examina-se a pilha de operadores.
- Enquanto houver operadores na pilha, eles serão desempilhados, e concatenados no final da expressão posfixa já existente.

P3-18.c (Continuação...)

```
71
72      // Infix formula empty. Pop stack to postfix
73      while (!emptyStack (stack))
74          {
75           dataPtr = (char*)popStack (stack);
76           temp[0] = *dataPtr;
77           strcat (postfix , temp);
78          } // while
79
80      // Print the postfix
81      printf ("The postfix formula is: ");
82      puts (postfix);
83
84      // Now destroy the stack
85      destroyStack (stack);
86      return 0;
87  } // main
```

Desempilha operador

Concatena no final de expressão posfixa

Imprime a expressão posfixa

Destrói a pilha

Aplicações de Pilhas (Stacks)

Notação Infixa em posfixa - Implementação

- A função **priority()** determina a prioridade de um operador.
- Quanto maior o valor retornado maior será a prioridade do operador.

P3-18.c (Função de prioridade do operador)

```
 88 /* ================== priority ====================
 89    Determine priority of operator.
 90       Pre  token is a valid operator
 91       Post token priority returned
 92 */
 93 int priority (char token)
 94 {
 95 // Statements
 96    if (token == '*' || token == '/')
 97        return 2;
 98    if (token == '+' || token == '-')
 99        return 1;
100    return 0;
101 } // priority
```

Aplicações de Pilhas (Stacks)

Notação Infixa em posfixa - Implementação

- A função **isOperator()** verifica se um determinado caráter corresponde a um dos operadores: +, −, *, /.
- Em caso afirmativo, retorna true. Caso contrário retorna false.

P3-18.c (Função que determina se o caráter é operador)

```
102 /* ================== isOperator ====================
103    Determine if token is an operator.
104       Pre  token is a valid operator
105       Post return true if operator; false if not
106 */
107 bool isOperator (char token)
108 {
109 // Statements
110    if (token == '*'
111       || token == '/'
112       || token == '+'
113       || token == '-')
114        return true;
115    return false;
116 }  // isOperator
```

Aplicações de Pilhas (Stacks)

Notação Infixa em posfixa - Resultado

- A figura mostra dois exemplos da transformação de uma expressão infixa em posfixa.

```
Results:
Run 1
   Enter an infix formula: 2+4
   The postfix formula is: 24+
Run 2
   Enter an infix formula: (a+b)*(c-d)/e
   The postfix formula is: ab+cd-*e/
```

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Ideia

- Mostraremos como usar uma pilha para avaliar uma expressão posfixa.
- Em uma expressão posfixa, os operandos aparecem antes dos operadores. Assim, utilizaremos a pilha para adiar o uso dos operandos até achar um operador.

- Considere a seguinte expressão posfixa:

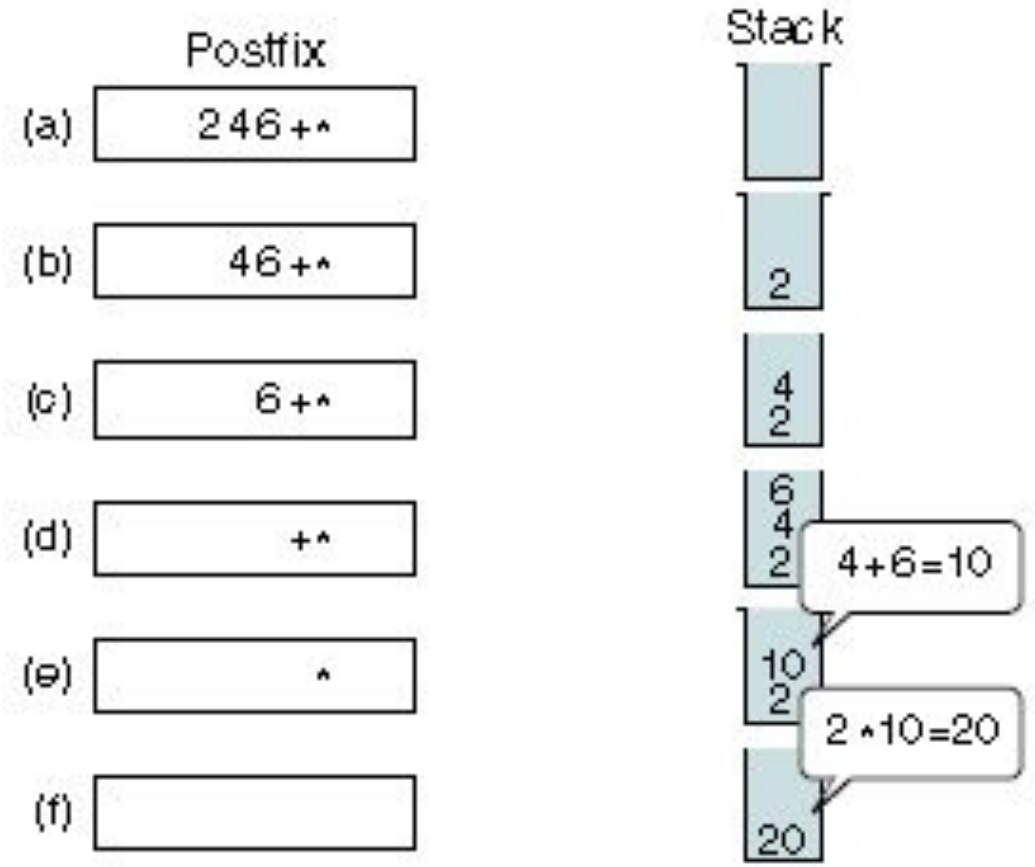
$$A\ B\ C\ +\ \times$$

- Os operandos são colocados em uma pilha até achar um operador.
- Quando achamos um operador, desempilhamos os dois operandos que se encontram no topo da pilha e executamos a operação indicada.
- Vale observar que: O primeiro operando desempilhado será de fato o segundo operando usado na operação, enquanto o segundo operando desempilhado corresponderá ao primeiro operando usado na operação. Respeitar a ordem é fundamental em operações como: $-, \div$.
- Após calcular o resultado da operação, empilhamos esse valor novamente para uso posterior.

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Exemplo

- A figura ilustra o uso da pilha para avaliação da expressão posfixa:



- Após a avaliação da expressão, o resultado fica armazenado na pilha.

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Algoritmo

- O algoritmo para avaliação da expressão posfixa é o seguinte:

```
Algorithm postFixEvaluate (expr)
This algorithm evaluates a postfix expression and returns its
value.
   Pre    a valid expression
   Post   postfix value computed
   Return value of expression
1  createStack (stack)
2  loop (for each character)
   1   if (character is operand)
       1  pushStack (stack, character)
   2   else
       1  popStack (stack, oper2)
       2  popStack (stack, oper1)
       3  operator = character
       4  set value to calculate (oper1, operator, oper2)
       5  pushStack (stack, value)
   3   end if
3  end loop
4  popStack (stack, result)
5  return (result)
end postFixEvaluate
```

Criar Pilha

Ler Carater

Empilhar Operando

Desempilhar 2 operandos

Executar Operação

Empilhar resultado da operação

Desempilhar resultado final

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Implementação

P3-19.c

- A primeira parte do programa principal define as variáveis locais o tipo de dado a ser armazenado e cria a pilha.

```
1  /* This program evaluates a postfix expression and
2     returns its value. The postfix expression must be
3     valid with each operand being only one digit.
4        Written by:
5        Date:
6  */
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include "stacksADT.h"
10
11 // Prototype Declarations
12 bool isOperator (char token);
13 int  calc       (int operand1, int oper, int operand2);
14
15 int main (void)
16 {
17 // Local Definitions
18    char   token;
19    int    operand1;
20    int    operand2;
21    int    value;
22    int*   dataPtr;
23    STACK* stack;
24
25 // Statements
26    // Create Stack
27    stack = createStack ();
28
```

Caracter para armazenar o operador

Define os dados a serem armazenados como inteiros

Cria a pilha

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Implementação

- O loop principal no programa principal consiste em:
  - Ler um caracter;
  - Identificar se é um operando ou operador;
  - Se for um operando, empilhar;
  - Se for um operador, desempilhar dois operandos, executar a operação e empilhar o resultado.

P3-19.c (Continuação...)

```
29    // read postfix expression, char by char
30    printf("Input formula: ");
31    while ((token = getchar ())!= '\n')
32       {
33        if (!isOperator (token))
34           {
35            dataPtr  = (int*) malloc (sizeof (int));
36            *dataPtr = atoi (&token);
37            pushStack (stack, dataPtr);
38           } // while
39
40        else
41           // character is operand
42           {
43            dataPtr  = (int*)popStack (stack);
44            operand2 = *dataPtr;
45            dataPtr  = (int*)popStack (stack);
              operand1 = *dataPtr;
47            value    = calc(operand1, token, operand2);
48            dataPtr  = (int*) malloc (sizeof (int));
              *dataPtr = value;
50            pushStack (stack, dataPtr);
51           } // else
52       } // while
```

Ler caracter

Empilha operando

Desempilha 2 operandos

Calcula operação

Empilha resultado da operação

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Implementação

- A parte final do programa principal, recupera o valor armazenado na pilha e imprime o resultado da expressão posfixa.
- Finalmente, a pilha é destruída.

P3-19.c (Continuação...)

```
53
54      // The final result is in stack. Pop it print it
55      dataPtr = (int*)popStack (stack);
56      value = *dataPtr;
57      printf ("The result is: %d\n", value);
58
59
60      // Now destroy the stack
61      destroyStack (stack);
62      return 0;
63  }   // main
```

Recupera o valor armazenado na pilha

Destrói a pilha

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Implementação

- A função isOperator () verifica se o token é um operador válido: +, −, ×, /.
- Em caso afirmativo, a função retorna o valor true, caso contrário o valor false.

P3-19.c (Função que determina se o caráter é um operador)

```
64  /* ==================== isOperator =================
65     Validate operator.
66        Pre  token is operator to be validated
67        Post return true if valid, false if not
68  */
69  bool isOperator (char token)
70  {
71  // Statements
72     if (token == '*'
73         || token == '/'
74         || token == '+'
75         || token == '-')
76        return true;
77     return false;
78  }  // isOperator
```

Verifica se o Token é um operador válido

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Implementação

- A função calc() realiza o calculo de uma operação binária do tipo: +, −, ×, /.
- A função retorna o resultado da operação.

P3-19.c (Função que calcula o resultado da expressão)

```
79  /* ================== calc =================
80     Given two values and operator, determine value of
81     formula.
82        Pre  operand1 and operand2 are values
83             oper is the operator to be used
84        Post return result of calculation
85  */
86  int calc (int operand1, int oper, int operand2)
87  {
88  // Local Declaration
89     int result;
90
91  // Statements
92     switch (oper)
93        {
94        case '+' : result  = operand1 + operand2;
95                   break;
96        case '-' : result  = operand1 - operand2;
97                   break;
98        case '*' : result  = operand1 * operand2;
99                   break;
100       case '/' : result  = operand1 / operand2;
101                  break;
102       } // switch
103    return result;
104 } // calc
```

Identifica o operador e executa a operação correspondente

Retorna o resultado

Aplicações de Pilhas (Stacks)

Avaliar expressão posfixa - Resultado

- A figura ilustra o resultado para uma expressão posfixa de exemplo.

```
Results:
Input formula: 52/4+5*2+
The result is 32
```

Aplicações de Pilhas (Stacks)

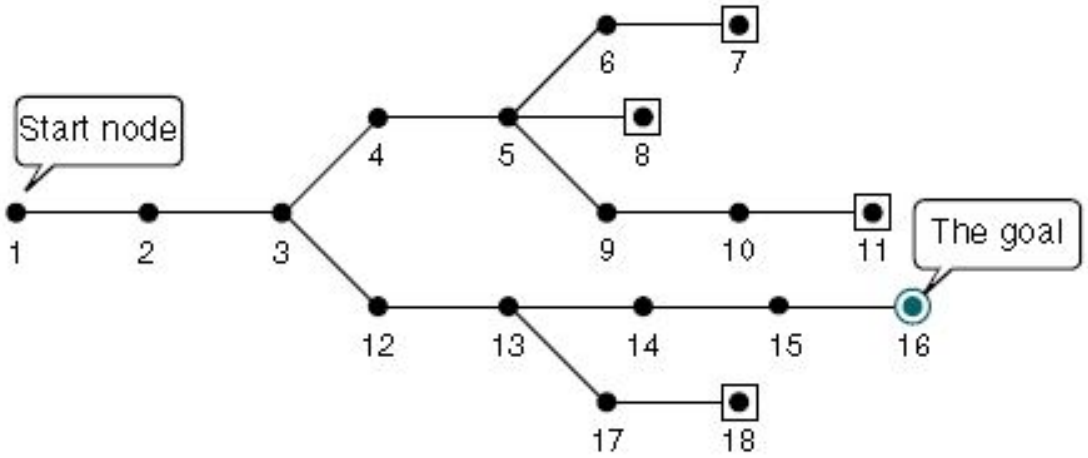
Backtracking

- O ***Backtracking*** é uma outra aplicação das pilhas usada em aplicações tais como jogos computacionais, analise de decisão e sistemas expertos.
- A ideia do Backtracking consiste em poder explorar caminhos alternativos e para isso ter a capacidade de desfazer um percurso já feito e lembrar daquele percurso que ficou pendente.
- As pilhas são boas estruturas para lembrar das tarefas pendentes.
- Mostraremos duas aplicações de Backtracking: A busca de um objetivo (destino) e o problemas das 8 rainhas.

Aplicações de Pilhas (Stacks)

Backtracking - A busca de um objetivo (Destino)

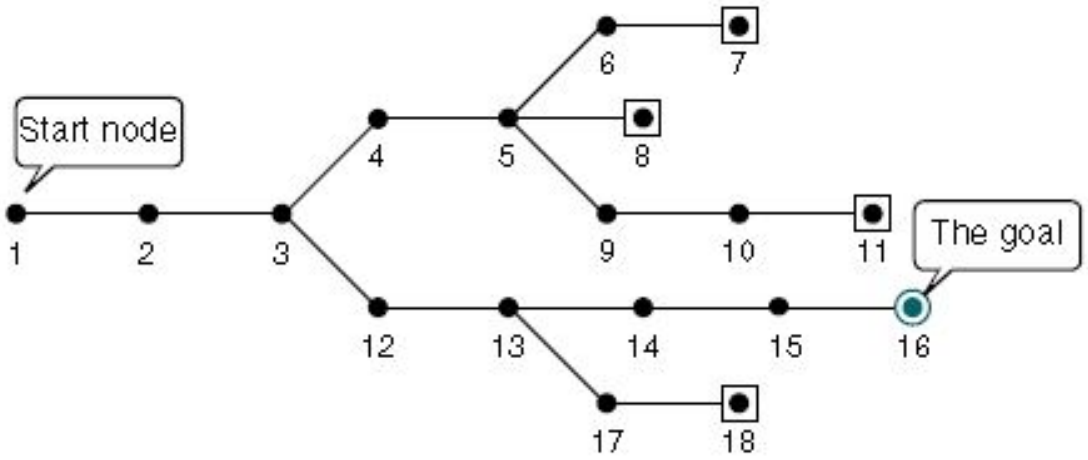
- A figura ilustra um exemplo de uma busca por objetivo.
- Temos um grafo, mas especificamente uma árvore que começa no nó 1 e apresenta uma série de caminhos alternativos a partir dele.
- Apenas um destes caminhos nos leva ao nosso objetivo no nó 16.
- Precisamos de um algoritmo para encontrar o caminho correto.



Aplicações de Pilhas (Stacks)

Backtracking - A busca de um objetivo (Destino)

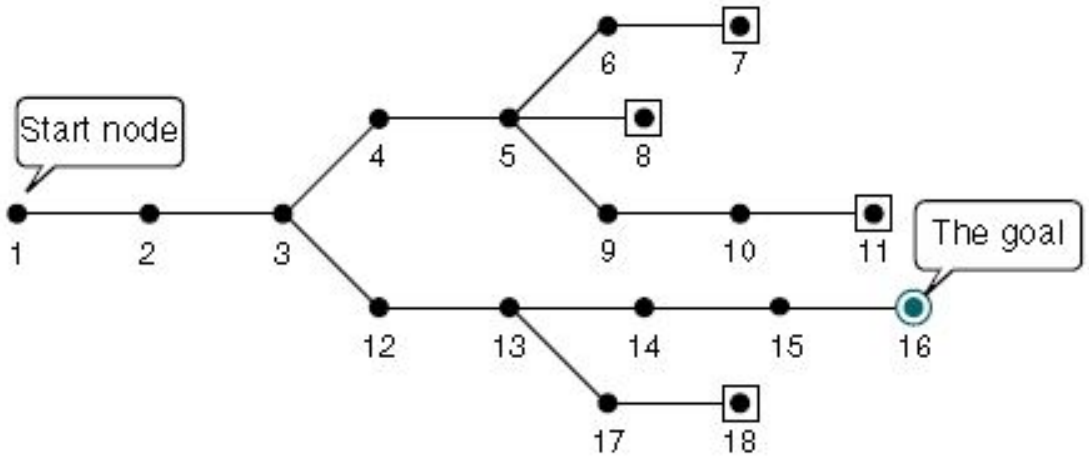
- Cada vez que chegamos a um ponto de bifurcação em que precisamos decidir para onde ir, precisamos lembrar onde ele fica de maneira que possamos voltar a ele caso seja necessário. Isto é chamado de *Backtrack*.
- Fazer *Backtrack* significa voltar ao ponto mais próximo em que tomamos uma decisão.
- Não queremos voltar ao ponto de partida.
- Para implementar o *Backtrack* usamos estruturas LIFO (*Last Input First Output*) como a pilha.



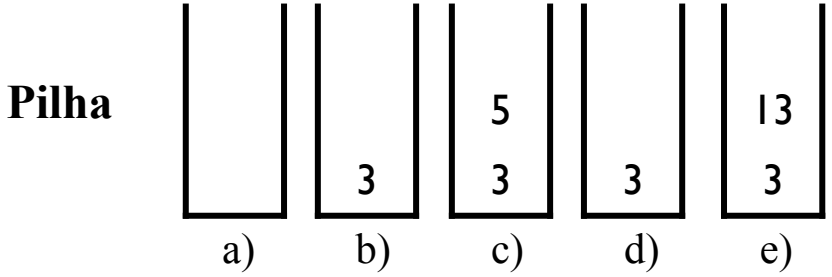
Aplicações de Pilhas (Stacks)

Backtracking - A busca de um objetivo (Destino)

- A questão aqui é o devemos colocar na pilha?
- Se queremos apenas localizar nosso objetivo, podemos empilhar os nós onde realizamos uma bifurcação.



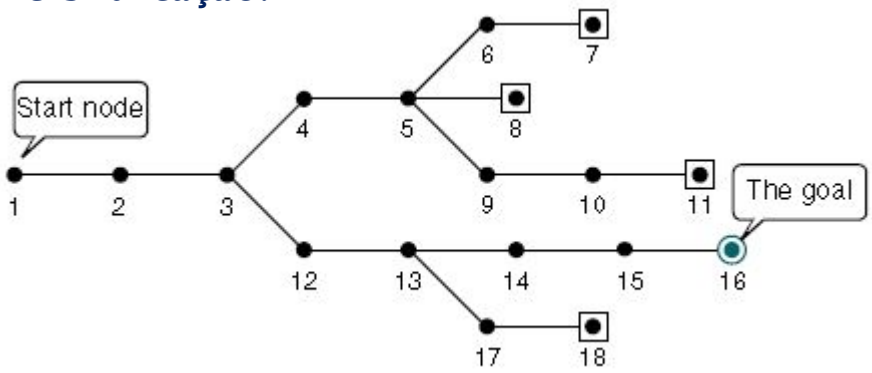
- Temos assim, a seguinte evolução da pilha:



Aplicações de Pilhas (Stacks)

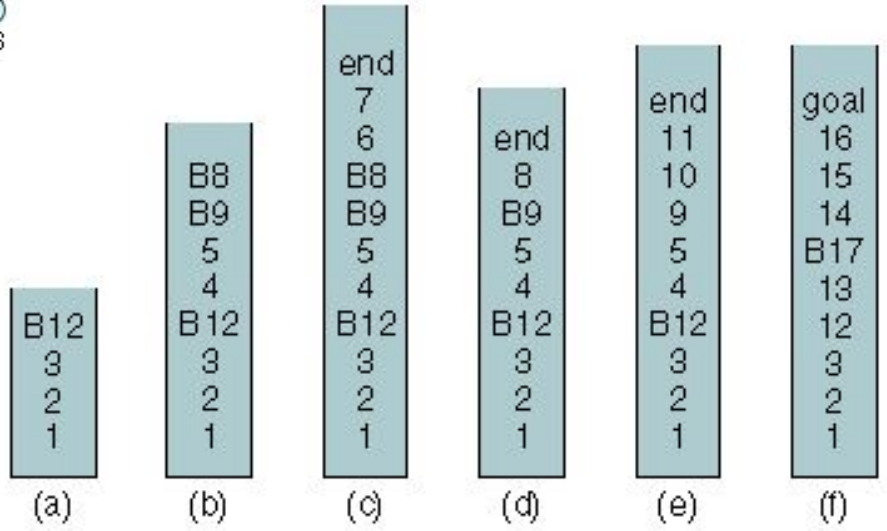
Backtracking - A busca de um objetivo (Destino)

- No entanto, se também queremos imprimir o caminho que nos leva ao nosso objetivo, teremos que empilhar tanto os nós que fazem parte de um caminho válido como os nós resultantes de uma bifurcação.
- Distinguimos entre estes dois tipos de nós usando uma marcação ou flag, para os nós de bifurcação.



- Na solução final ignora-se os nós resultantes da bifurcação.

- A pilha resultante neste caso é a seguinte:

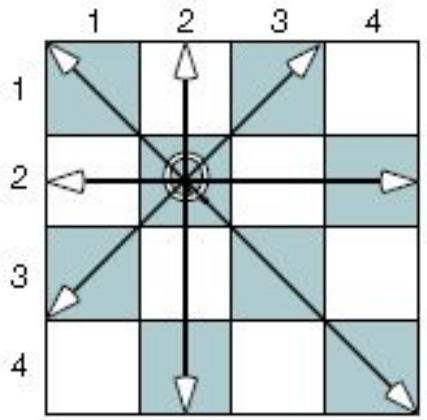


Aplicações de Pilhas (Stacks)

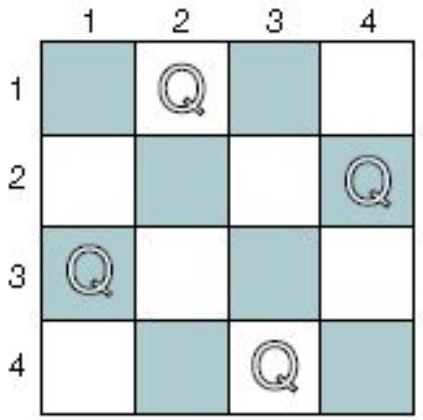
Backtracking – Problema da 8 rainhas

- O problema das 8 rainhas é um problema clássico associado ao jogo de xadrez quem consiste em posicionar 8 rainhas no tabuleiro de xadrez de maneira que nenhuma rainha possa atacar (capturar) qualquer outra rainha.
- A solução computacional para este problema requer que localizemos uma rainha em uma posição do tabuleiro e analisemos todas as possíveis direções de ataque, verificando que não existe outra rainha capaz de capturar a última rainha posicionada.
- Caso a posição da nova rainha esteja ameazada, tenta-se outra posição.
- A figura ilustra o caso simplificado com 4 rainhas e um tabuleiro 4x4.

Direções de ataque para a rainha localizada em [2,2].



(a) Queen capture rules



(b) First four queens solution

Solução para o problema das 4 rainhas.

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Implementação

- Na primeira parte do **programa principal** define-se uma estrutura para armazenar uma posição no tabuleiro.
- Uma estrutura de pilha será formada com nós que incluem ponteiros a estruturas desse tipo.
- Declaram-se protótipos das funções para:
  - obter o tamanho do tabuleiro;
  - preencher o tabuleiro e a pilha;
  - imprimir a pilha;
  - verificar se uma posição do tabuleiro está ameaçada.

P3-20.c (Funções para o tabuleiro)

```
 8  #include <stdio.h>
 9  #include <stdlib.h>
10  #include "P4StkADT.h"
11
12  // Structure Declarations
13     typedef struct
14        {
15         int  row;
16         int  col;
17        } POSITION;
18
19  // Prototype Declarations
20  int  getSize   (void);
21
22  void fillBoard  (STACK* stack, int boardSize);
23  void printBoard (STACK* stack, int boardSize);
24
25  bool guarded (int board[][9], int row,
26                int col,        int boardSize);
```

Define a estrutura para armazenar posições de tabuleiro

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Implementação

- O **programa principal** usa uma pilha para armazenar as posições das rainhas no tabuleiro.
- Além disso usará uma matriz de posições.

P3-20.c (Programa principal)

```
27
28  int main (void)
29  {
30  // Local Definitions
31     int  boardSize;
32
33     STACK* stack ;
34
35  // Statements
36     boardSize = getSize ();
37     stack     = createStack ();
38
39     fillBoard    (stack, boardSize);
40     printBoard   (stack, boardSize);
41     destroyStack (stack);
42
43     printf("\nWe hope you enjoyed Eight Queens.\n");
44     return 0;
45  }  // main
```

Define o tamanho do tabuleiro

Cria a pilha de posições

Preenche o tabuleiro e a pilha

Imprime o tabuleiro a partir da pilha

Destrói a pilha

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Implementação

- A função **getSize()** permite a definição do tamanho do tabuleiro.
- O tamanho é definido pelo usuário e retornado pela função.
- São válidos tamanhos entre 4 e 8.

Leitura do tamanho do tabuleiro

P3-21.h (Função que define o tamanho do tabuleiro)

```
 6  int getSize (void)
 7  {
 8  // Local Definitions
 9     int boardSize;
10
11  // Statements
12     printf("Welcome to Eight Queens. You may select\n"
13            "a board size from 4 x 4 to 8 x 8. I will\n"
14            "then position a queen in each row of the\n"
15            "board so no queen may capture another\n"
16            "queen. Note: There are no solutions for \n"
17            "boards less than 4 x 4.\n");
18     printf("\nPlease enter the board size: ");
19     scanf ("%d", &boardSize);
20     while (boardSize < 4 || boardSize > 8)
21        {
22         printf("Board size must be greater than 3 \n"
23                "and less than 9. You entered %d.\n"
24                "Please re-enter. Thank you.\a\a\n\n"
25                "Your board size: ", boardSize);
26         scanf ("%d", &boardSize);
27        } // while
28     return boardSize;
29  } // getSize
```

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Implementação

- A função **fillBoard()** realiza a tarefa de preenchimento do tabuleiro com rainhas em posições seguras (não ameaçadas).
- A função utiliza uma estratégia de tentativa e erro para colocar o maior número de rainhas no tabuleiro (No máximo uma por linha).
- Cada posição possível em uma linha é testada, usando a função **guarded()**, para garantir que não exista ameaça de outras rainhas.

P3-22.h (Função que posiciona as rainhas)

```
 1  /* ================== fillBoard ===================
 2     Position chess queens on game board so that no queen
 3     can capture any other queen.
 4        Pre  boardSize number of rows & columns on board
 5        Post Queens' positions filled
 6  */
 7  void fillBoard (STACK* stack, int boardSize)
 8  {
 9  // Local Definitions
10     int  row;
11     int  col;
12     int  board[9][9] = {0};  // 0 no queen: 1 queen
13                              // row 0 & col 0 !used
14     POSITION* pPos;
15
```

Matriz de posições

Ponteiro a uma posição

P3-22.h (Continuação...)

- A função **fillBoard()** assume que deve existir uma rainha em cada linha da solução.
- Para cada linha procura-se uma posição segura em uma coluna.
- Caso exista essa posição segura, registra ela na matriz, empilha e analisa a próxima linha.
- Caso contrário, faz o *backtrack*, libera a posição da última rainha posicionada, na matriz e na pilha.

```
16  // Statements
17     row = 1;
18     col = 0;
19
20     while (row <= boardSize)
21       {
22        while (col <= boardSize && row <= boardSize)
23          {
24           col++;
25           if (!guarded(board, row, col, boardSize))
26              {
27               board[row][col] = 1;
28
29              pPos = (POSITION*)malloc(sizeof(POSITION));
30               pPos->row = row;
31               pPos->col = col;
32
33               pushStack(stack, pPos);
34
35               row++;
36               col = 0;
37              } // if
38           while (col >= boardSize)
39              {
40               pPos = popStack(stack);
41               row  = pPos->row;
42               col  = pPos->col;
43               board[row][col] = 0;
44               free (pPos);
45              } // while col
46          } // while col
47       } // while row
48     return;
49  }  // fillBoard
```

Preenche a posição segura na matriz

Empilha uma posição segura

Backtrack

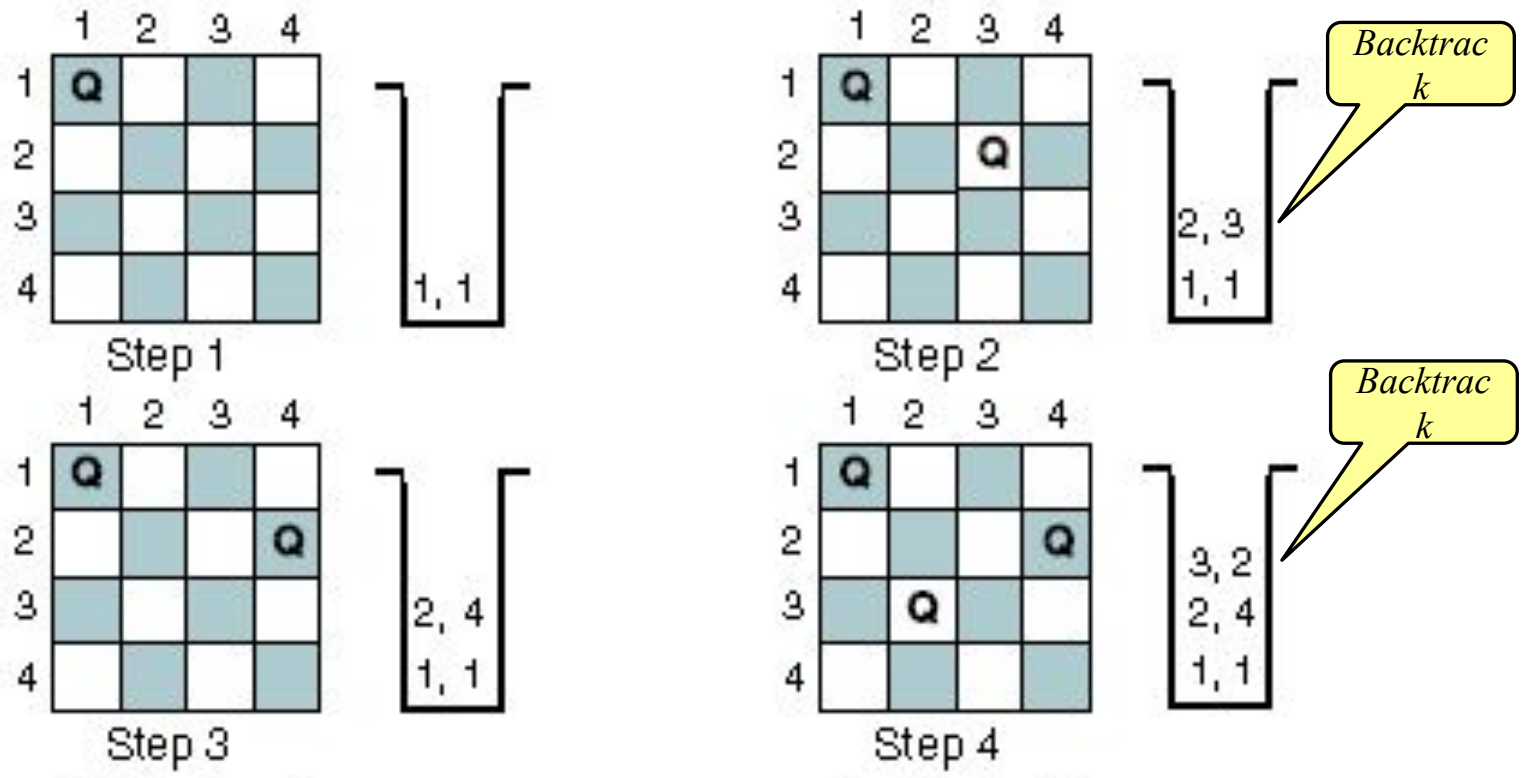
Desempilha a última posição

Libera a última posição na matriz

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Exemplo

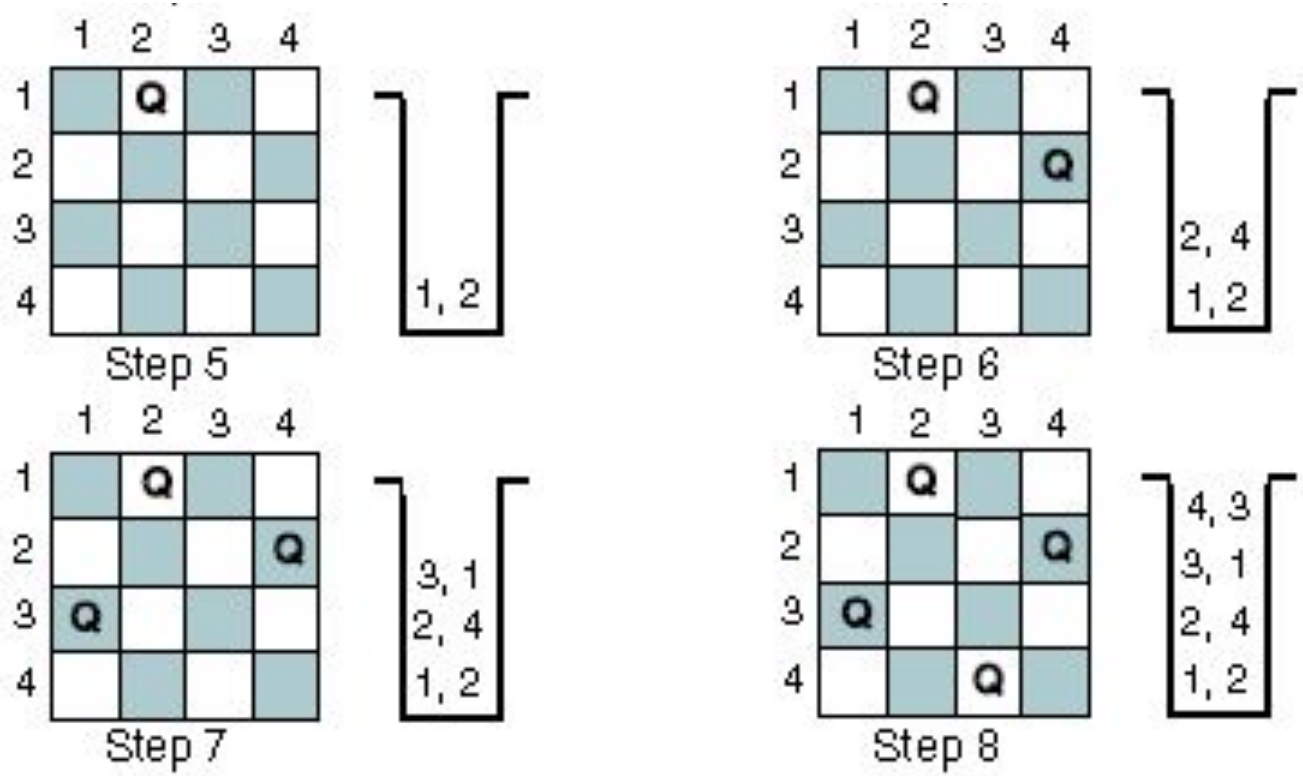
- A figura ilustra o uso da pilha para armazenamento das posições seguras para as rainhas. Assim como para desfazer o posicionamento da última rainha, caso ela impeça o posicionamento de novas rainhas na próxima linha.
- Observe os casos de *Backtrack*.



Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Exemplo

- A figura ilustra o uso da pilha para armazenamento das posições seguras para as rainhas.
- Os passos 5, 6, 7 e 8 mostram uma sequência posições seguras até achar a solução final.



P3-23.h (Função que determina se a posição esta ameaçada)

- A função **guarded()** verifica se existe uma ameaça para a posição da nova rainha, localizada na posição [chkrow, chk col].
- Caso a posição esteja ameaçada retorna true; Caso contrário, retorna false.
- A função recebe o tabuleiro preenchido com as rainhas anteriores.
- Somente é necessário testar ameaças vindas de cima.
- A função usa força bruta.

```
 9  bool guarded (int board[][9], int chkRow,
10                int chkCol,     int boardSize)
11  {
12  // Local Definitions
13     int row;
14     int col;
15
16  // Statements
17
18     // Check current col for a queen
19     col = chkCol;
20     for (row = 1; row <= chkRow; row++)
21         if (board[row][col] == 1)
22            return true;
23
24     // Check diagonal right-up
25     for (row = chkRow - 1, col = chkCol + 1;
26          row > 0 && col <= boardSize;
27          row--, col++)
28         if (board[row][col] == 1)
29            return true;
30
31     // Check diagonal left-up
32     for (row = chkRow - 1, col = chkCol - 1;
33          row > 0 && col > 0;
34          row--, col--)
35         if (board[row][col] == 1)
36            return true;
37
38     return false;
39  } // guarded
```

Ameaças na mesma coluna

Ameaças na diagonal à direita

Ameaças na diagonal à esquerda

P3-24.h (Imprime o tabuleiro a partir da pilha)

- A função **printBoard** () imprime o posicionamento das rainhas no tabuleiro.
- As posições armazenadas na pilha encontram-se ordenadas de maior a menor linha. Esta ordem é invertida usando uma pilha auxiliar.

```
 1  /* ================== printBoard ====================
 2     Print positions of chess queens on a game board
 3     Pre  stack contains positions of queen
 4          boardSize is the number of rows and columns
 5     Post Queens' positions printed
 6  */
 7  void printBoard (STACK* stack, int boardSize)
 8  {
 9  // Local Definitions
10     int col;
11
12     POSITION* pPos;
13     STACK*    pOutStack;
14
15  // Statements
16     if (emptyStack(stack))
17          {
18          printf("There are no positions on this board\n");
19           return;
20          } // if
21
22     printf("\nPlace queens in following positions:\n");
23
24     // Reverse stack for printing
25     pOutStack = createStack ();
26     while (!emptyStack (stack))
27         {
28          pPos = popStack (stack);
29          pushStack (pOutStack, pPos);
30         } // while
```

Pilha vazia

Cria uma segunda pilha para inverter a primeira

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Exemplo

- A função **printBoard()** imprime o posicionamento das rainhas no tabuleiro.
- Usam-se as posições armazenadas na segunda pilha de menor a maior linha.

P3-24.h (Continuação...)

```
31
32      // Now print board
33      while (!emptyStack (pOutStack))
34          {
35           pPos = popStack (pOutStack);
36           printf("Row %d-Col %d: \t|",
37                   pPos->row, pPos->col);
38           for (col = 1; col <= boardSize; col++)
39               {
40                if (pPos->col == col)
41                    printf(" Q |");
42                else
43                    printf("   |");
44               } // for
45           printf("\n");
46          } // while
47      destroyStack(pOutStack);
48      return;
49  } // printBoard
```

Extrai a posição

Imprime a posição

Imprime Q na coluna apropriada

Destrói a pilha

Aplicações de Pilhas (Stacks)

Problema da 8 rainhas - Resultado

- A figura ilustra a solução para o problema das 4 rainhas.

```
Results:
Welcome to Eight Queens. You may select
a board size from 4 x 4 to 8 x 8. I will
then position a queen in each row of the
board so no queen may capture another
queen. Note: There are no solutions for
boards less than 4 x 4.

Please enter the board size: 4

Place queens in following positions:
Row 1-Col 2:     | Q |   |   |
Row 2-Col 4:     |   |   | Q |
Row 3-Col 1:   Q |   |   |   |
Row 4-Col 3:     |   | Q |   |

We hope you enjoyed Eight Queens.
```

Referências

- Gilberg, R.F. e Forouzan, B. A. Data Structures_A Pseudocode Approach with C. Capítulo 3. Stacks. Segunda Edição. Editora Cengage, Thomson Learning, 2005.