Unit 1: Introduction and basic concepts (Chap. 1, 2)

Unit 2: Linear lists

Stacks (chap. 3)

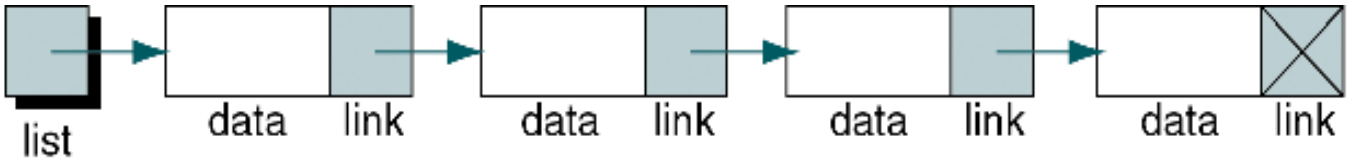
Queues (chap. 4)

General linear lists (chap.5)

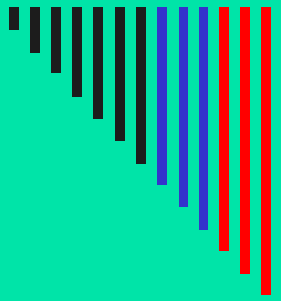
Unit 3: Non-Linear Lists (Chap. 6, 7, 8, 9, 10, 11)

Unit 4: Sorting (Chap. 12)

Unit 5: Searching (Chap. 13)



- Restricted list: addition and deletion of data are restricted to the ends of the list
  - Stack: last in-first out (LIFO)
  - Queue: first in-first out (FIFO)
- General list: data can be inserted and deleted anywhere in the list



Chapter 3

Stacks

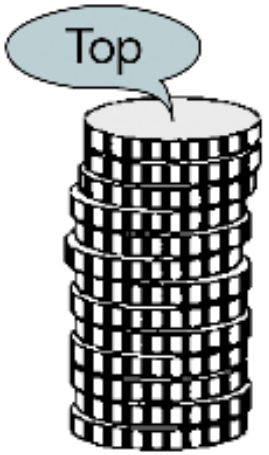
Objectives

Upon completion you will be able to

- **Explain the design, use, and operation of a stack**
- **Implement a stack using a linked list structure**
- **Understand the operation of the stack ADT**
- **Write application programs using the stack ADT**
- **Discuss reversing data, parsing, postponing and backtracking**

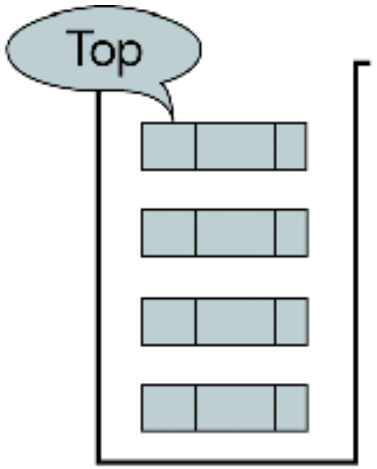
What is Stack ?

- A stack is a last in-first out data structure in which all insertions and deletions are restricted to one end, called top



Stack of coins

Stack of books



Computer stack

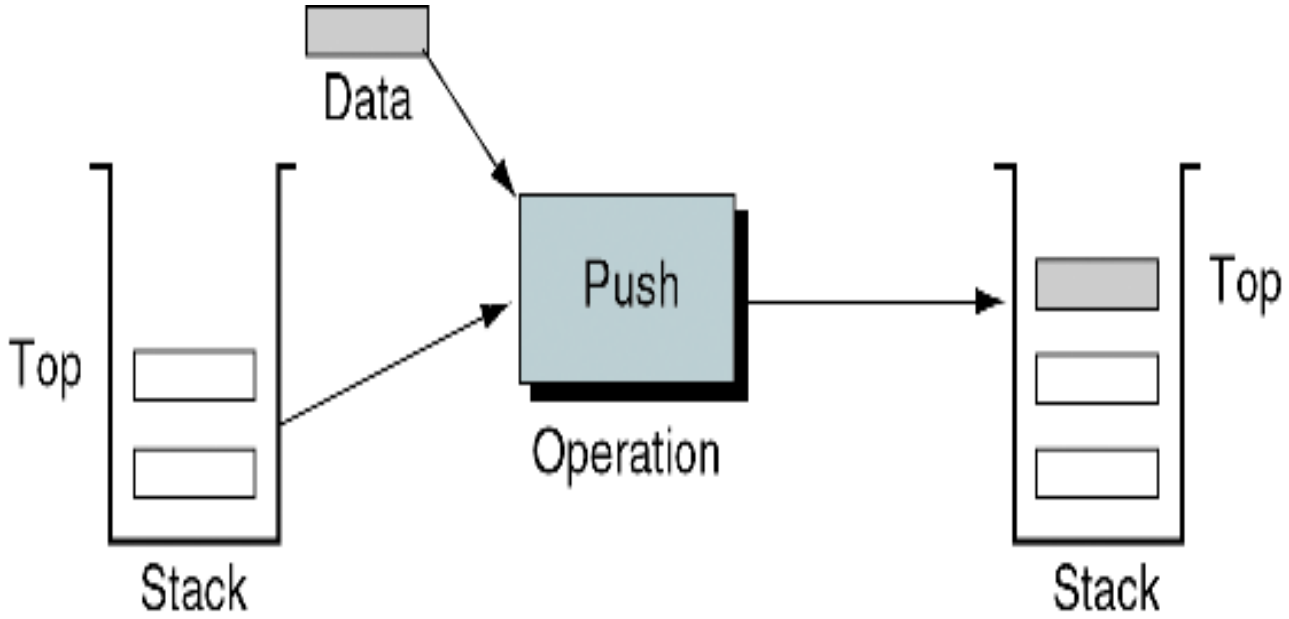
3-1 Basic Stack Operations

The stack concept is introduced and three basic stack operations are discussed.

- **Push**
- **Pop**
- **Stack Top**

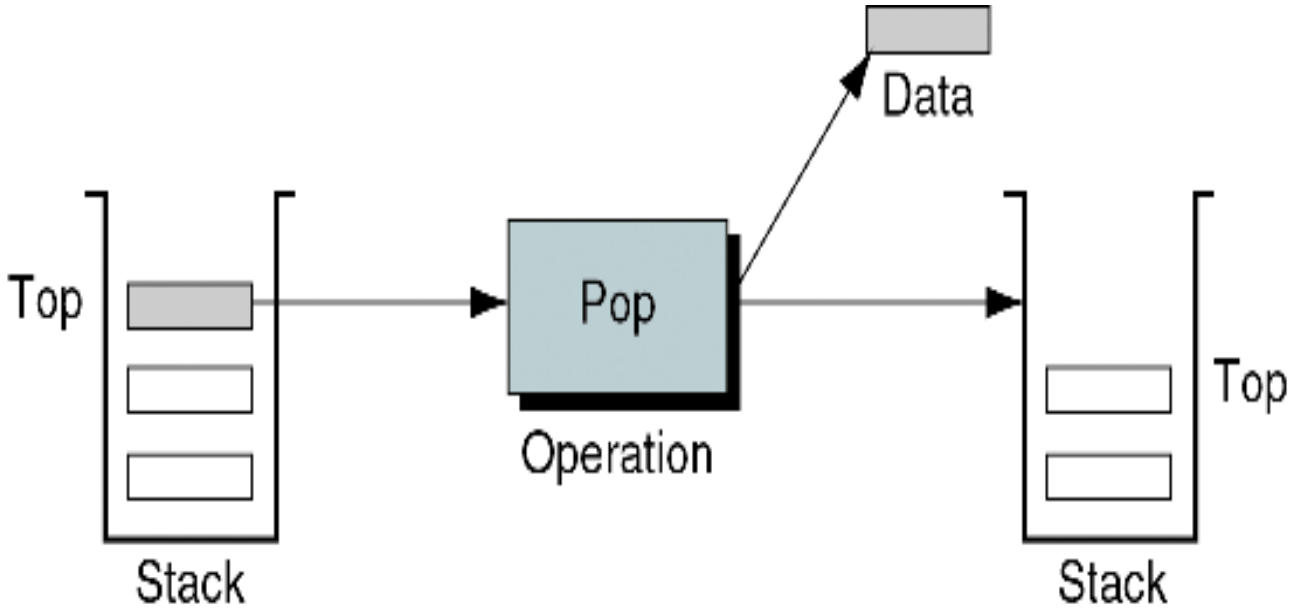
Push Stack Operation

- If there is not enough room, the stack is in an *overflow* state and the item cannot be added



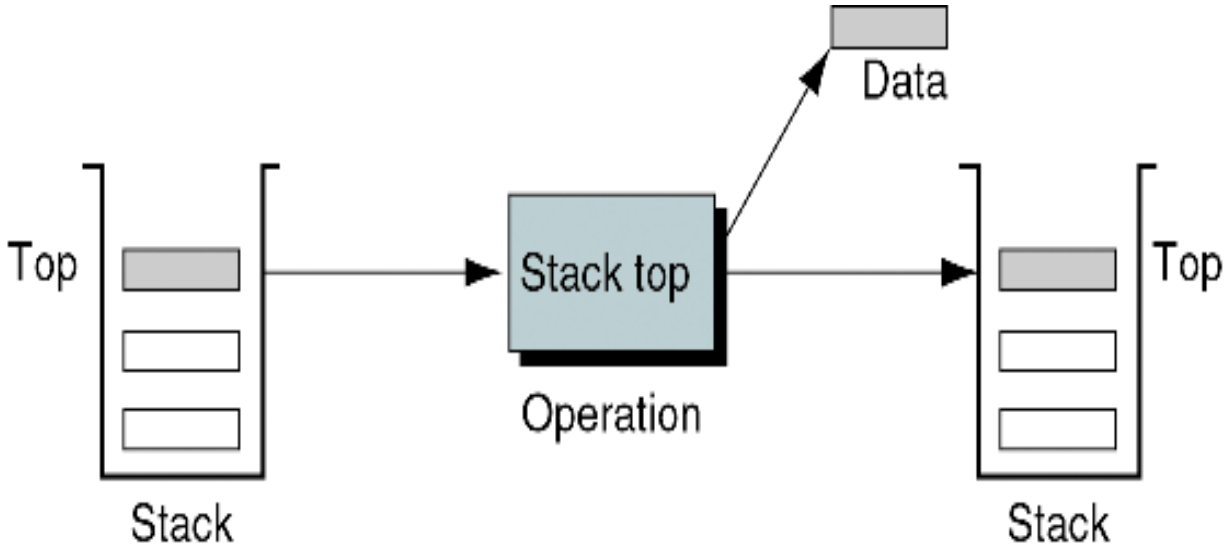
Pop Stack Operation

- If pop is called when the stack is empty, it is in an *underflow* state

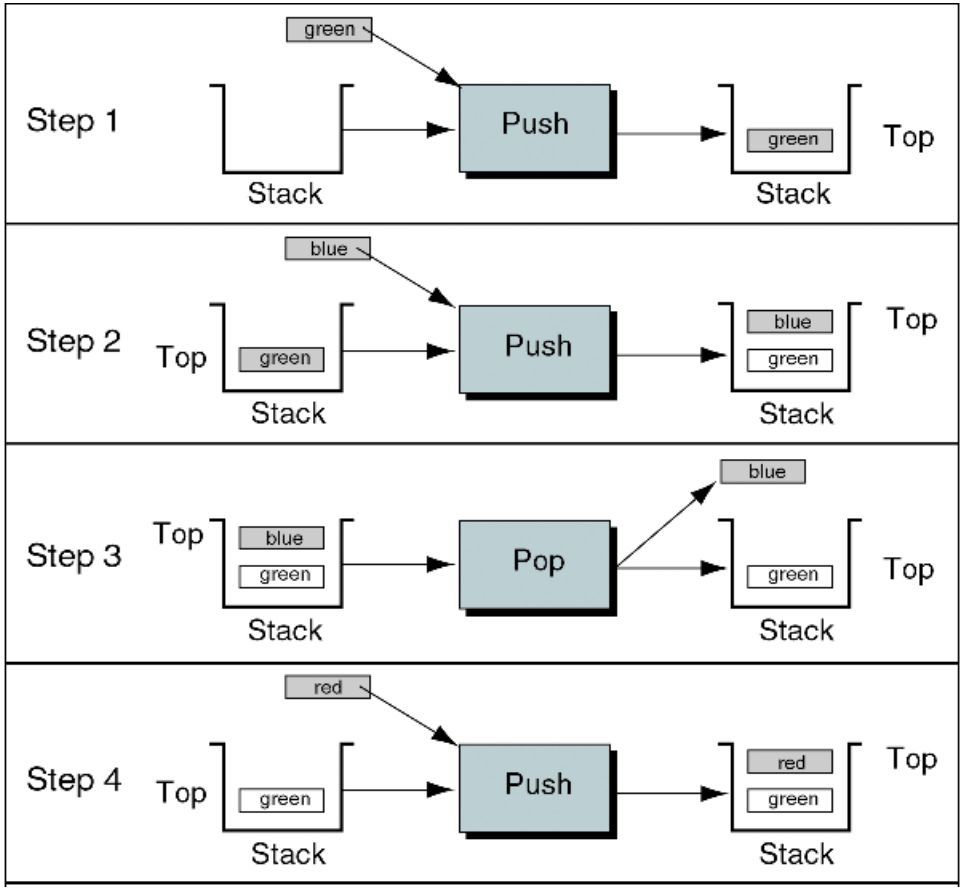


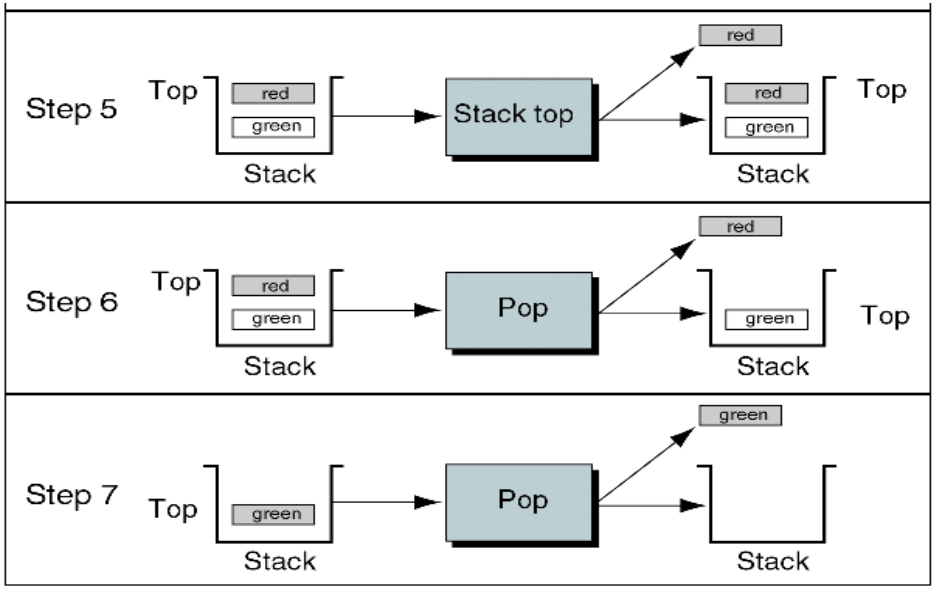
Stack Top Operation

- Stack top copies the item at the top of stack but does not delete it
- Stack top can also result in underflow if the stack is empty



Stack Example





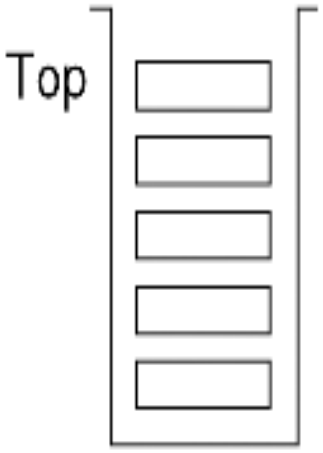
3-2 Stack Linked List Implementation

In this section we present a linked-list design for a stack. After developing the data structures, we write pseudocode algorithms for the stack ADT.

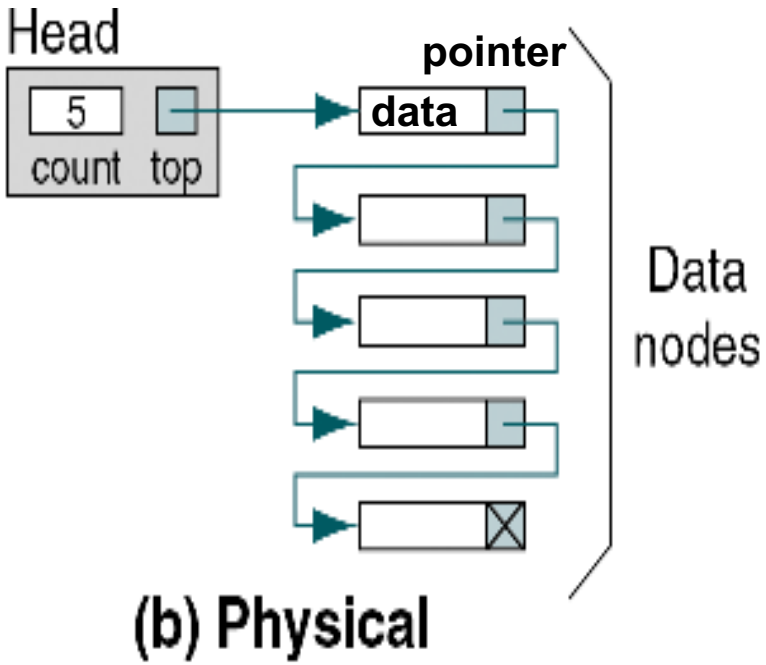
- **Data Structure**
- **Algorithms**

Conceptual & Physical Stack Implementation

- Implement the stack as linked list
- Need two different data structures: head & data node



(a) Conceptual

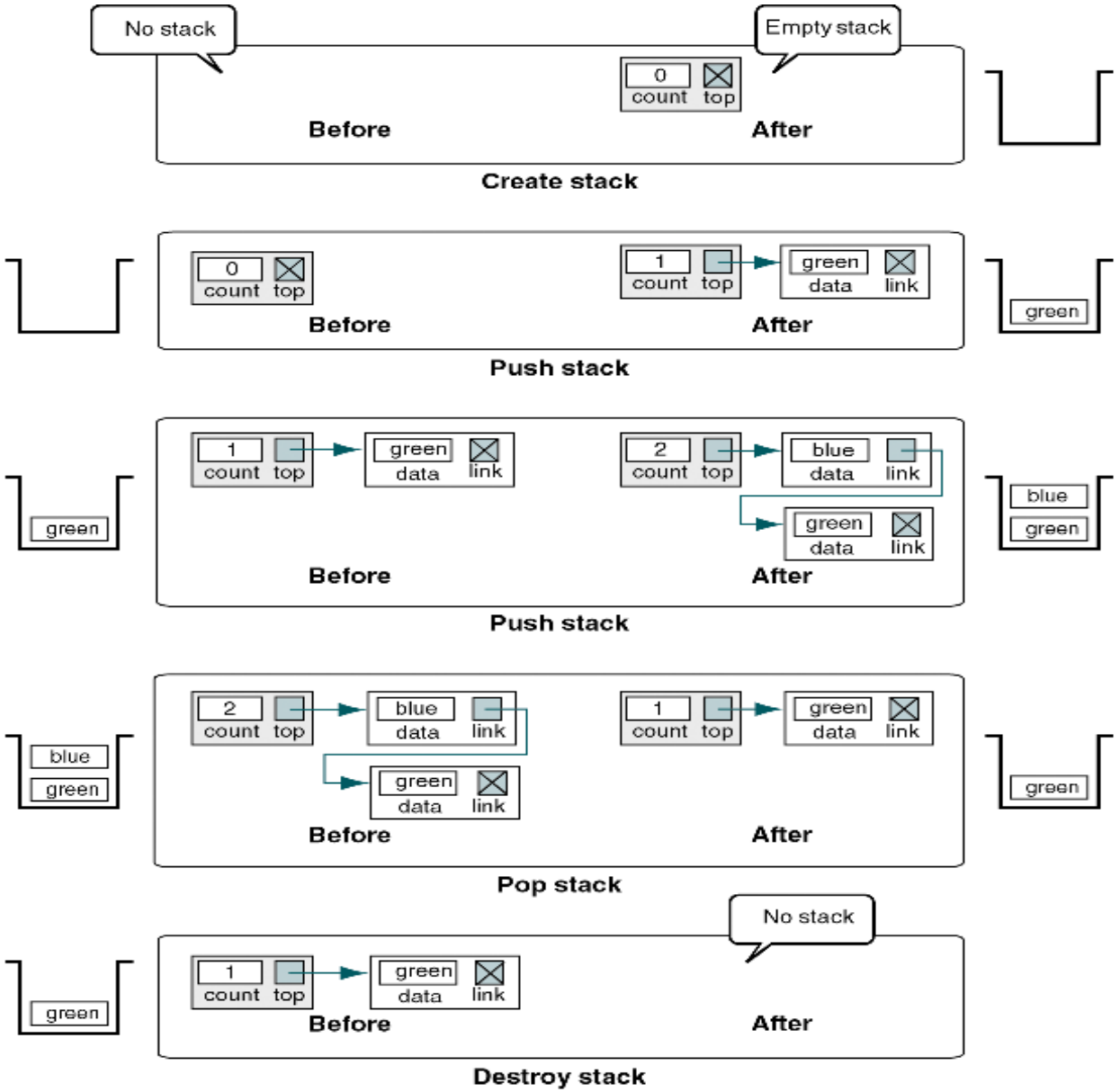


(b) Physical

```
stack
  count
  top
end stack
node
  data
  link
end node
```

Data Structure

Stack Operations

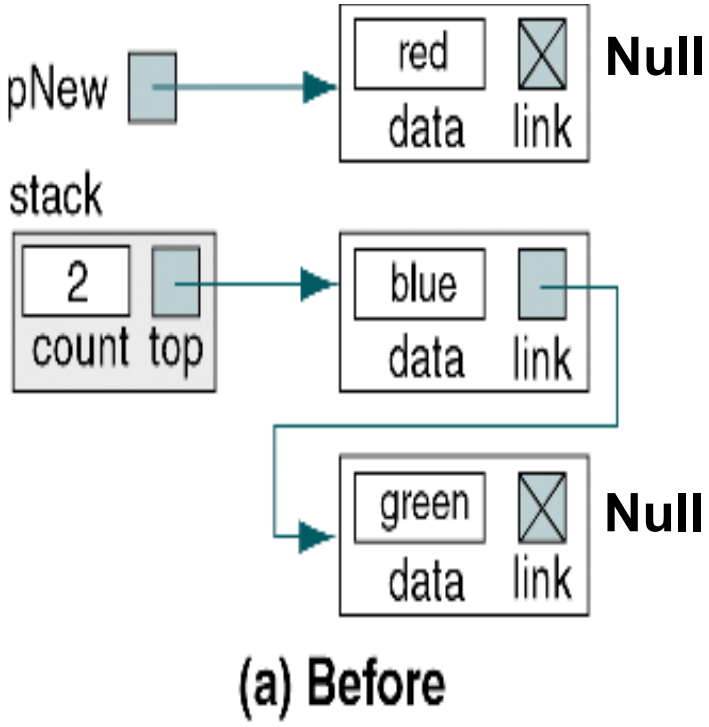


Create Stack Algorithm

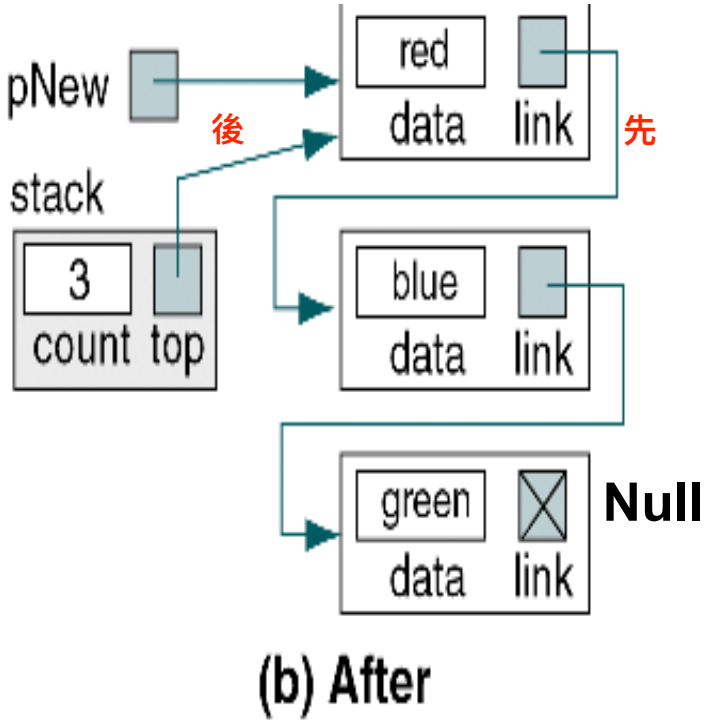
```
Algorithm createStack
Creates and initializes metadata structure.
   Pre    Nothing
   Post   Structure created and initialized
   Return stack head
 1 allocate memory for stack head
 2 set count to 0
 3 set top to null
 4 return stack head
end createStack
```

Push Stack Example

- Three conditions should be considered
  - Insertion into an empty stack
  - Insertion into a stack with data
  - Insertion into a stack when the available memory is exhausted



(a) Before

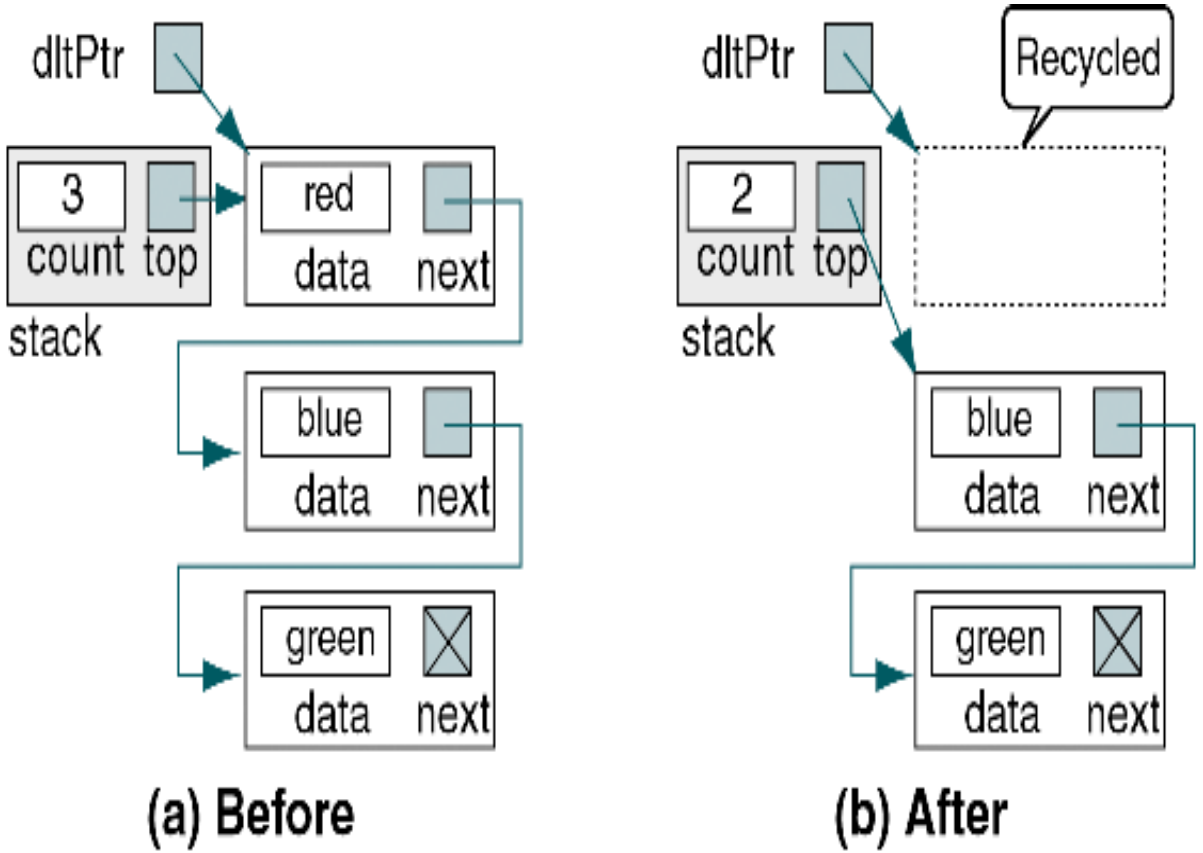


(b) After

Push Stack Algorithm

```
Algorithm pushStack (stack, data)
Insert (push) one item into the stack.
   Pre  stack passed by reference
        data contain data to be pushed into stack
   Post data have been pushed in stack
 1 allocate new node
 2 store data in new node
 3 make current top node the second node
 4 make new node the top
 5 increment stack count
end pushStack
```

Pop Stack Example



Pop Stack Algorithm

```
Algorithm popStack (stack, dataOut)
This algorithm pops the item on the top of the stack and
returns it to the user.
   Pre    stack passed by reference
          dataOut is reference variable to receive data
   Post   Data have been returned to calling algorithm
   Return true if successful; false if underflow
 1 if (stack empty)
   1  set success to false
 2 else
   1  set dataOut to data in top node
   2  make second node the top node
   3  decrement stack count
   4  set success to true
 3 end if
 4 return success
end popStack
```

Stack Top Algorithm

```
Algorithm stackTop  (stack, dataOut)
This algorithm retrieves the data from the top of the stack
without changing the stack.
   Pre    stack is metadata structure to a valid stack
          dataOut is reference variable to receive data
   Post   Data have been returned to calling algorithm
   Return true if data returned, false if underflow
1  if (stack empty)
   1   set success to false
2  else
   1   set dataOut to data in top node
   2   set success to true
3  end if
4  return success
end stackTop
```

Empty Stack

```
Algorithm emptyStack (stack)
Determines if stack is empty and returns a Boolean.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack status
   Return true if stack empty, false if stack contains data
1  if (stack count is 0)
   1   return true
2  else
   1   return false
3  end if
end emptyStack
```

Full Stack

```
Algorithm fullStack  (stack)
Determines if stack is full and returns a Boolean.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack status
   Return true if stack full, false if memory available
1  if (memory not available)
   1   return true
2  else
   1   return false
3  end if
end fullStack
```

Stack Count

```
Algorithm stackCount (stack)
Returns the number of elements currently in stack.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack count
   Return integer count of number of elements in stack
 1 return (stack count)
end stackCount
```

Destroy Stack

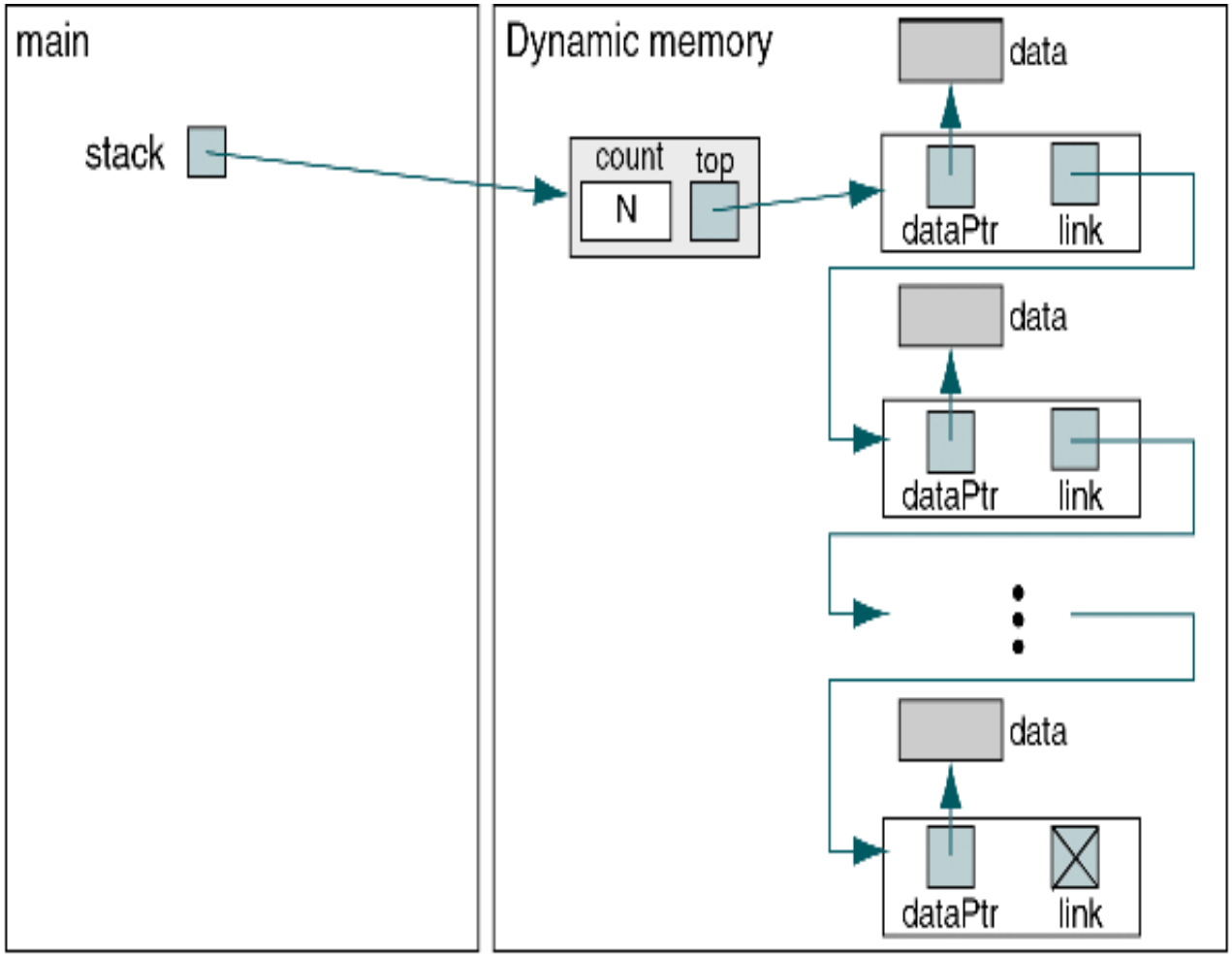
```
Algorithm destroyStack (stack)
This algorithm releases all nodes back to the dynamic memory.
   Pre    stack passed by reference
   Post   stack empty and all nodes deleted
 1 if (stack not empty)
   1   loop (stack not empty)
       1  delete top node
   2   end loop
 2 end if
 3 delete stack head
end destroyStack
```

3-4 Stack ADT

We begin the discussion of the stack ADT with a discussion of the stack structure and its application interface. We then develop the required functions.

- **Data Structure**
- **ADT Implementation**

Stack ADT Structural Concepts



Stack ADT Definitions

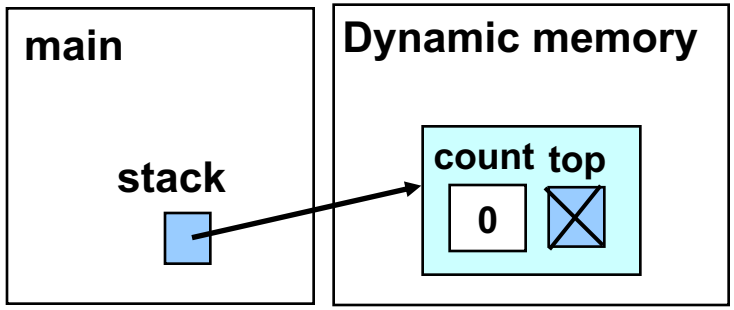
```
1   // Stack ADT Type Defintions
2      typedef struct node
3         {
4          void*        dataPtr;
5          struct node* link;
6         } STACK_NODE;
7
8      typedef struct
9         {
10         int          count;
11         STACK_NODE* top;
12        } STACK;
```

Stack head structure

ADT Create Stack

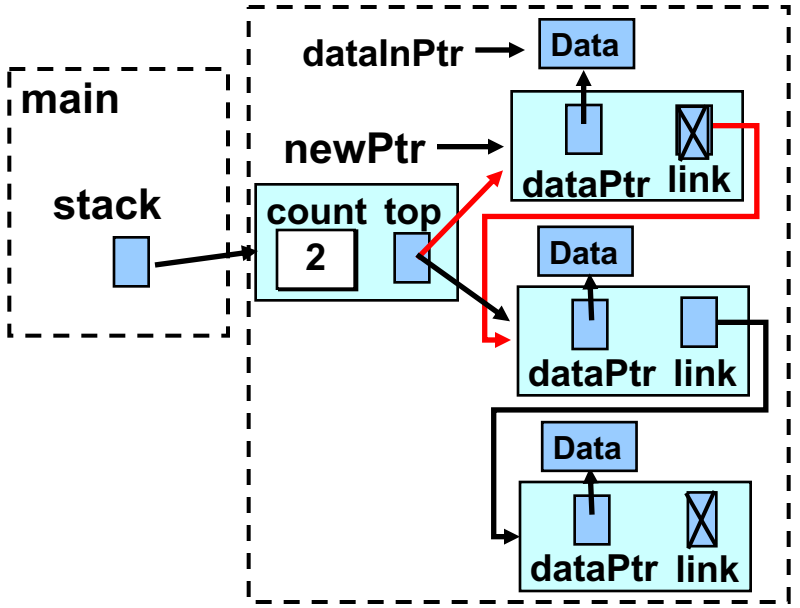
- Allocate a stack head node, initializes the top pointer to null, and zeros the count field
- The call to create a stack must assign the return pointer value to a stack pointer as: *stack = creatstack()*

```
 7  STACK* createStack (void)
 8  {
 9  // Local Definitions
10     STACK* stack;
11
12  // Statements
13     stack = (STACK*) malloc( sizeof (STACK));
14     if (stack)
15        {
16         stack->count = 0;
17         stack->top   = NULL;
18        } // if
19     return stack;
20  } // createStack
```



Push Stack

```
 9  bool pushStack (STACK* stack, void* dataInPtr)
10  {
11  // Local Definitions
12     STACK_NODE* newPtr;
13
14  // Statements
15     newPtr = (STACK_NODE* ) malloc(sizeof( STACK_NODE));
16     if (!newPtr)
17         return false;
18
19     newPtr->dataPtr = dataInPtr;
20
21     newPtr->link    = stack->top;
22     stack->top      = newPtr;
23
24     (stack->count)++;
25     return true;
26  } // pushStack
```

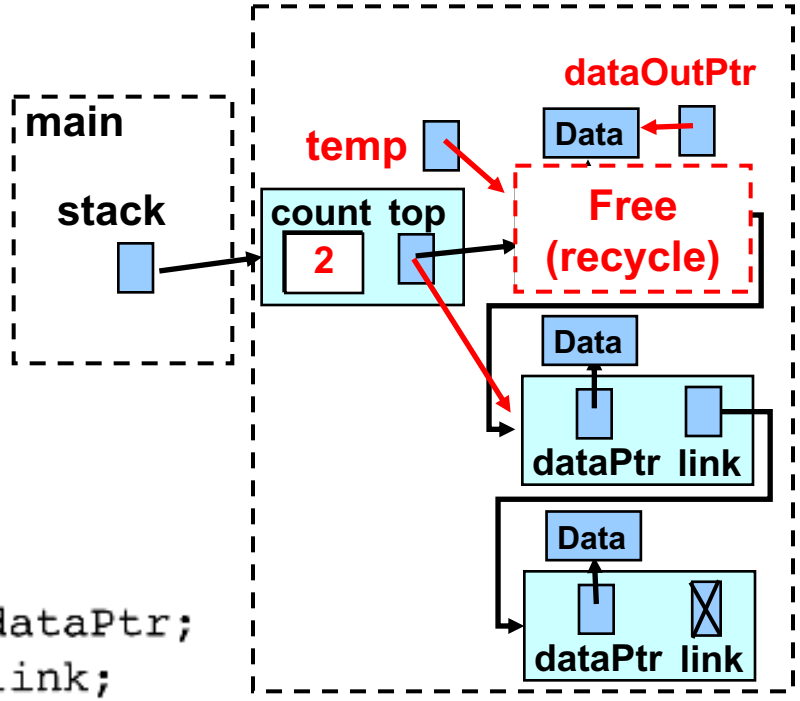


ADT Pop Stack

```
1  /* ================== popStack ================
2     This function pops item on the top of the stack.
3        Pre  stack is pointer to a stack
4        Post Returns pointer to user data if successful
5                     NULL if underflow
6  */
7  void* popStack (STACK* stack)
8  {
9  // Local Definitions
10    void*       dataOutPtr;
```

ADT Pop Stack (conti.)

```
11      STACK_NODE* temp;
12
13  // Statements
14      if (stack->count == 0)
15          dataOutPtr = NULL;
16      else
17          {
18           temp        = stack->top;
19           dataOutPtr = stack->top->dataPtr;
20           stack->top = stack->top->link;
21           free (temp);
22           (stack->count)--;
23          } // else
24      return dataOutPtr;
25  }   // popStack
```



Retrieve Stack Top

```
 8  void* stackTop (STACK* stack)
 9  {
10  // Statements
11     if (stack->count == 0)
12         return NULL;
13     else
14         return stack->top->dataPtr;
15  }  // stackTop
```

Empty Stack

```
1  /* ================ emptyStack ================
2     This function determines if a stack is empty.
3        Pre  stack is pointer to a stack
4        Post returns 1 if empty; 0 if data in stack
5  */
6  bool emptyStack (STACK* stack)
7  {
8  // Statements
9     return (stack->count == 0);
10 }  // emptyStack
```

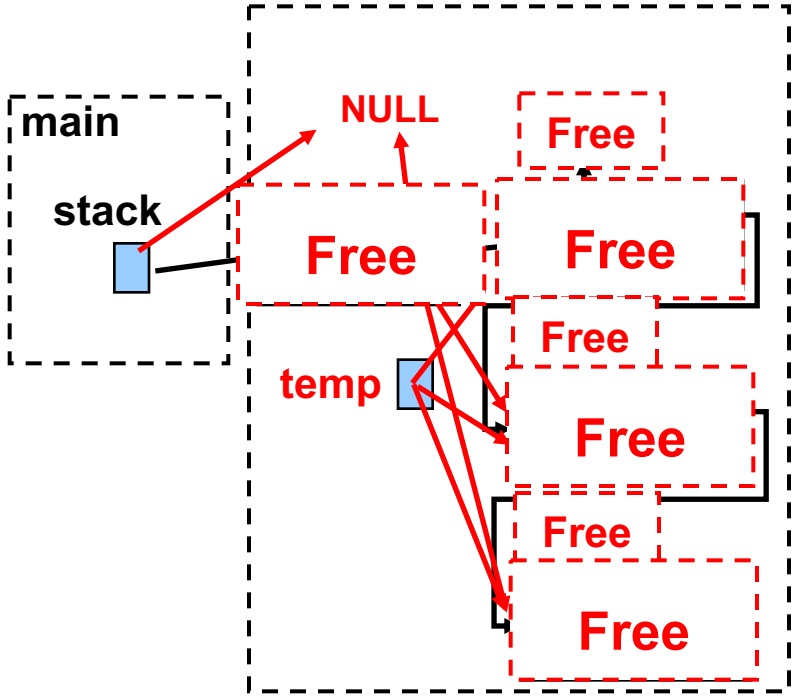
Full Stack

```
 8  bool fullStack (STACK* stack)
 9  {
10  // Local Definitions
11  STACK_NODE* temp;
12
13  // Statements
14     if ((temp =
15        (STACK_NODE*)malloc (sizeof(*(stack->top)))))
16        {
17         free (temp);
18         return false;
19        } // if
20
21     // malloc failed
22     return true;
23  }  // fullStack
```

Stack Count

```
1  /* ================== stackCount =================
2     Returns number of elements in stack.
3        Pre  stack is a pointer to the stack
4        Post count returned
5  */
6  int stackCount (STACK* stack)
7  {
8  // Statements
9     return stack->count;
10 }  // stackCount
```

Destroy Stack

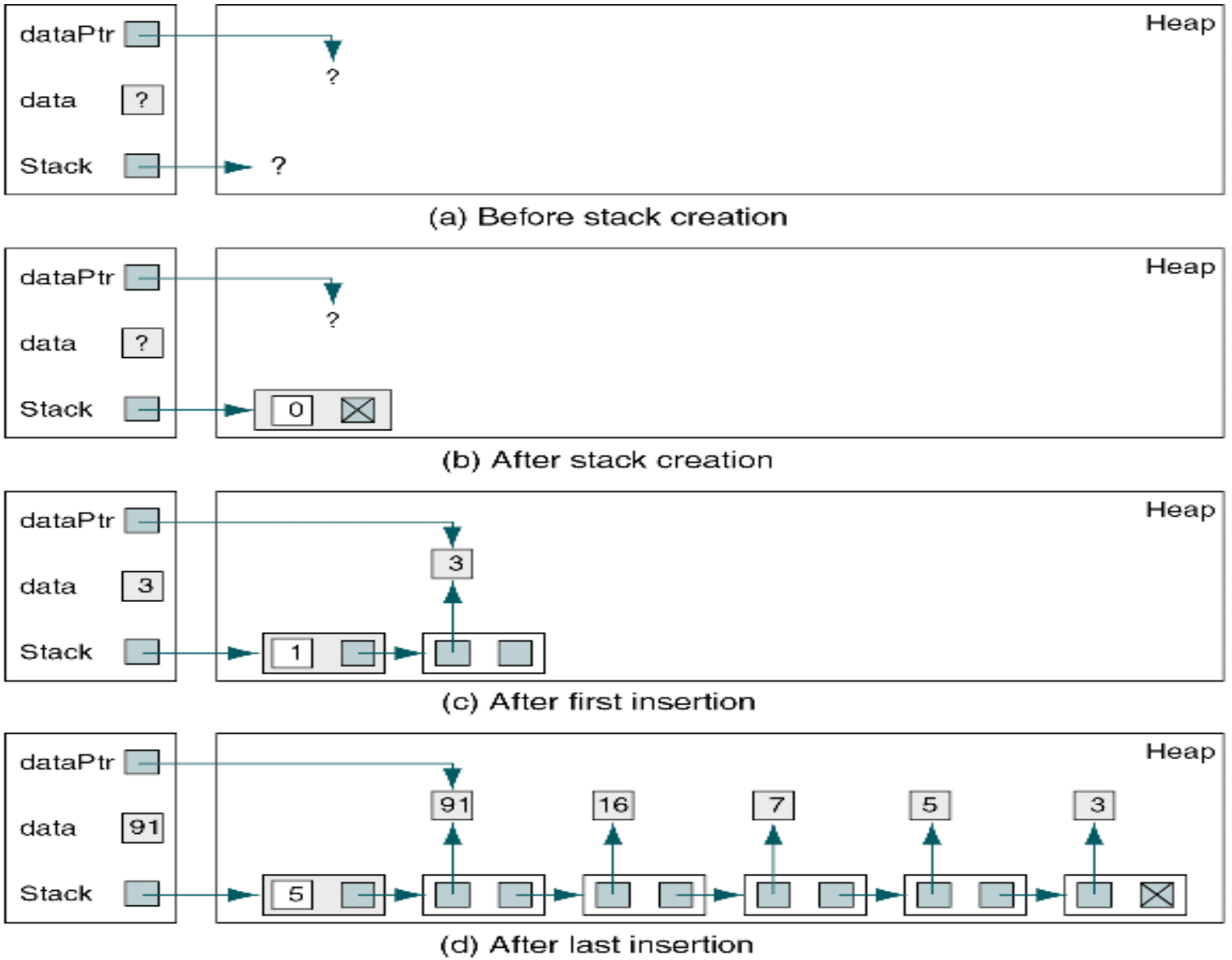


3-5 Stack Applications

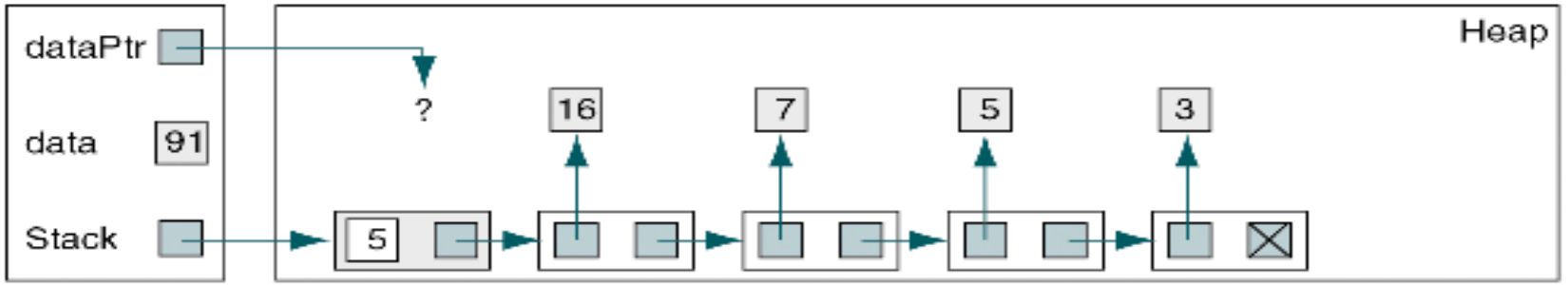
Three basic application problems-parsing, postponement, and backtracking-are discussed and sample programs developed. In addition, several other stack applications are presented, including the classic Eight Queens problem.

- Reversing Data
- Converting Decimal to Binary
- Parsing
- Postponement
- Backtracking

Reverse a Number Series



Reverse a Number Series (cont.)



(e) After first print

(f) After last print

(g) After stack destruction

Reverse a Integer List: Program

```
1  /* This program reverses a list of integers read
2     from the keyboard by pushing them into a stack
3     and retrieving them one by one.
4        Written by:
5        Date:
6  */
7  #include <stdio.h>
8  #include <stdbool.h>
9  #include "stacksADT.h"
10
11 int main (void)
12 {
13 // Local Definitions
14    bool done = false;
15    int* dataPtr;
16
17    STACK* stack;
18
19 // Statements
20    // Create a stack and allocate memory for data
21    stack = createStack ();
22
23    // Fill stack
24    while (!done)
25       {
```

Reverse a Integer List: Program (cont.)

```
26          dataPtr = (int*) malloc (sizeof(int));
27          printf ("Enter a number: <EOF> to stop: ");
28          if ((scanf ("%d" , dataPtr)) == EOF
29                  || fullStack (stack))
30             done = true;
31          else
32             pushStack (stack, dataPtr);
33        } // while
34
35  // Now print numbers in reverse
36     printf ("\n\nThe list of numbers reversed:\n");
37     while (!emptyStack (stack))
38        {
39          dataPtr = (int*)popStack (stack);
40          printf ("%3d\n", *dataPtr);
41          free (dataPtr);
42        } // while
43
```

Reverse a Integer List: Program (cont.)

```
44  // Destroying Stack
45     destroyStack (stack);
46     return 0;
47  }  // main
```

```
Results:
Enter a number: <EOF> to stop: 3
Enter a number: <EOF> to stop: 5
Enter a number: <EOF> to stop: 7
Enter a number: <EOF> to stop: 16
Enter a number: <EOF> to stop: 91
Enter a number: <EOF> to stop:

The list of numbers reversed:
 91
 16
  7
  5
  3
```

Convert Decimal to Binary

```
1   read (number)
2   loop (number > 0)
      1   set digital to number modulo 2
      2   print (digital)
      3   set number to quotient of number  / 2
3   end loop
```

For example: Number = 10

loop 1: digital = 10 mod 2 = 0	output → 0
number = 5	
loop 2: digital = 5 mod 2 = 1	output → 01
number = 2	
loop 3: digital = 2 mod 2 = 0	output → 010
number = 1	
loop 4: digital = 1 mod 2 = 1	output → 0101
number = 0	end loop

Note: output should be **1010** instead of 0101

Decimal to Binary: Using Stack

```
1  /* This program reads an integer from the keyboard
2     and prints its binary equivalent. It uses a stack
3     to reverse the order of 0s and 1s produced.
4        Written by:
5        Date:
6  */
7  #include <stdio.h>
8  #include "stacksADT.h"
9
10 int main (void)
11 {
12 // Local Definitions
13    unsigned int    num;
14             int*   digit;
15             STACK* stack;
16
17 // Statements
18    // Create Stack
19    stack = createStack ();
20
21    // prompt and read a number
22    printf ("Enter an integer:      ");
23    scanf ("%d", &num);
24
25    // create 0s and 1s and push them into the stack
26    while (num > 0)
```

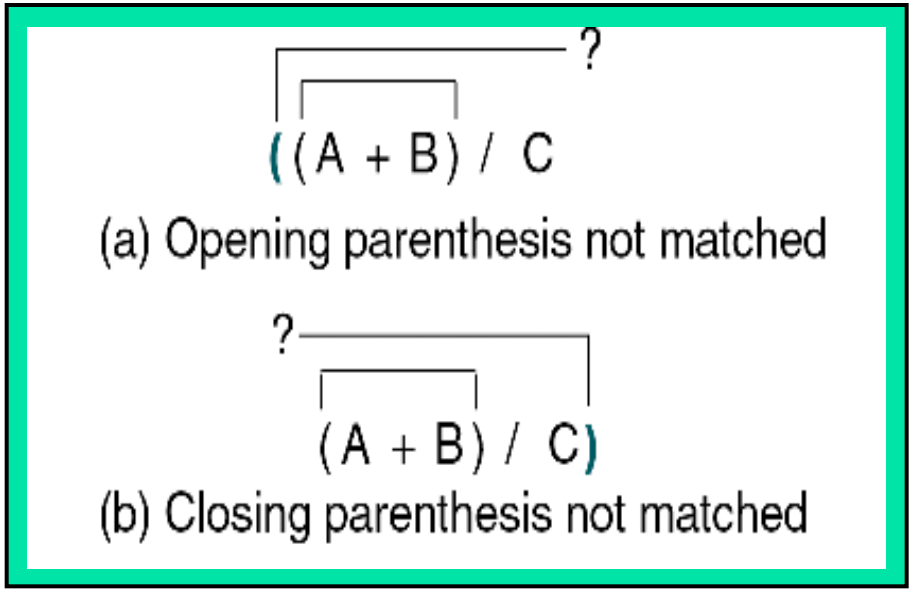
Decimal to Binary: Using Stack (cont.)

```
27          {
28           digit  = (int*) malloc (sizeof(int));
29           *digit = num % 2;
30           pushStack (stack, digit);
31           num = num /2;
32          } // while
33
34       // Binary number created. Now print it
35       printf ("The binary number is : ");
36       while (!emptyStack (stack))
37          {
38           digit = (int*)popStack (stack);
39           printf ("%1d", *digit);
40          } // while
41       printf ("\n");
42
43       // Destroying Stack
44       destroyStack (stack);
45       return 0;
46   }  // main
```

```
Results:
Enter an integer:       45
The binary number is : 101101
```

Parsing

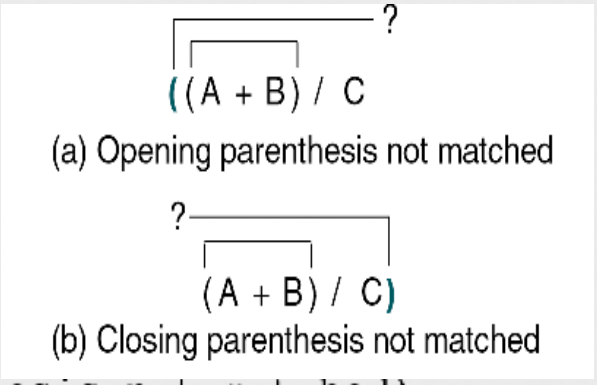
- **Parsing** is any logic that **breaks data into independent pieces** for futher processing
  - E.g., compiler translate a source program to machine language by parseing the program into keywords, names, and tokens



Unmatched parentheses examples

Parse Parentheses

```
Algorithm parseParens
This algorithm reads a source program and parses it to make
sure all opening-closing parentheses are paired.
1  loop (more data)
   1   read (character)
   2   if (opening parenthesis)
       1   pushStack (stack, character)
   3   else
       1   if (closing parenthesis)
           1   if (emptyStack (stack))
               1   print (Error: Closing parenthesis not matched)
           2   else
               1   popStack(stack)
           3   end if
       2   end if
   4   end if
2  end loop
3  if (not emptyStack (stack))
   1   print (Error: Opening parenthesis not matched)
end parseParens
```



Verify Parentheses Paired

```
 1  /* This program reads a source program and parses it to
 2     make sure all opening-closing parentheses are paired.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include "stacksADT.h"
 8
 9  // Error Messages
10  const char closMiss[] = "Close paren missing at line";
11  const char openMiss[] = "Open paren missing at line";
12
13  int main (void)
14  {
15  // Local Definitions
16     STACK* stack;
17     char   token;
18     char*  dataPtr;
```

Verify Parentheses Paired (cont.)

```
19      char   fileID[25];
20      FILE*  fpIn;
21      int    lineCount = 1;
22
23  // Statements
24      // Create Stack
25      stack = createStack ();
26      printf ("Enter file ID for file to be parsed: ");
27      scanf  ("%s", fileID);
28
29      fpIn = fopen (fileID, "r");
30      if (!fpIn)
31          printf("Error opening %s\n", fileID), exit(100);
32
```

Verify Parentheses Paired (cont.)

```
33    // read characters from the source code and parse
34    while ((token = fgetc (fpIn)) != EOF )
35       {
36        if (token == '\n')
37           lineCount++;
38        if (token == '(' )
39           {
40            dataPtr = (char*) malloc (sizeof (char));
41            pushStack (stack, dataPtr);
42           } // if
43        else
44           {
45            if (token == ')')
46               {
47                if (emptyStack (stack))
48                   {
49                    printf ("%s %d\n",
50                             openMiss, lineCount);
51                    return 1;
52                   } // if true
53                else
54                   popStack (stack);
55               } // token ==
56           } // else
57       } // while
```

Verify Parentheses Paired (cont.)

```
59    if (!emptyStack (stack))
60       {
61        printf ("%s %d\n", closMiss, lineCount);
62        return 1;
63       } // if
64
65    // Now destroy the stack
66    destroyStack (stack);
67    printf ("Parsing is OK: %d Lines parsed.\n",
68             lineCount);
69    return 0;
70 } // main
```

```
Results:
Run 1:
   Enter file ID for file to be parsed: no-errors.txt
   Parsing is OK: 65 Lines parsed.
Run 2:
   Enter file ID for file to be parsed: close-match.txt
   Close paren missing at line 46
Run 3:
   Enter file ID for file to be parsed: open-match.txt
   Open paren missing at line 23
```

Postponement: Infix to Postfix

Manual Transformation

One to one mapping

postfix對電腦來說執行起來比較快

好像是因為不用判斷先乘除後加減

Example 1: A + B * C

Step 1: (A + (B * C))

Step 2: (A + (B C *))

Step 3: (A (B C *) +)

Example 2: (A + B) * C + D + E * F – G

Step 1: (((((A + B) * C) + D) + (E * F)) – G)

Step 2: (((((A B +) C *) D +) (E F *) +) G –)

Step 3: A B + C * D + E F * + G –

Algorithm transformation

A * B → A B * : push operator * into stack and pop the stack after the whole infix expression has been read

A * B + C → A B * C + : push operator * into stack and pop the stack before we get another operator +

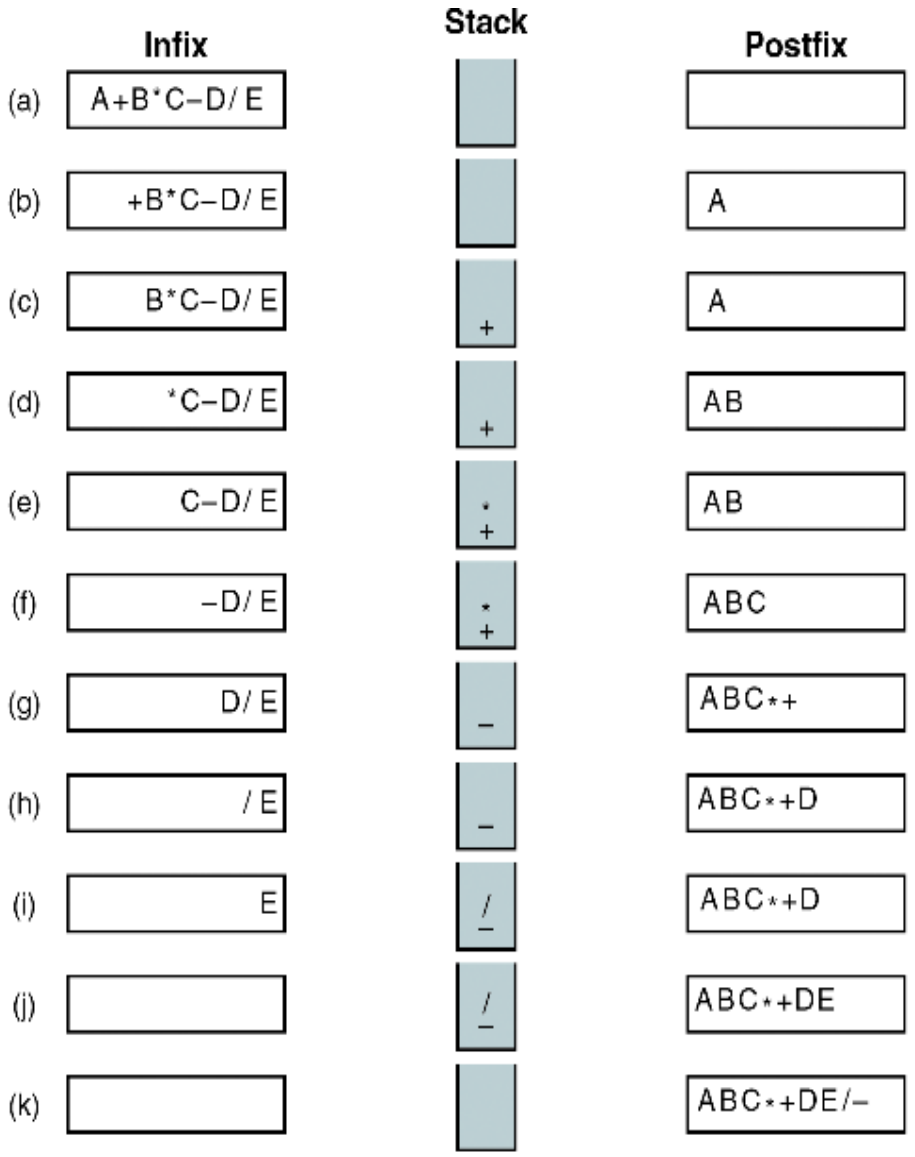
A + B * C → A B C * + : ?

Postponement: Infix to Postfix

A + B * C → A B C * +

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

Note 1: several operators may be popped before pushing the new operator into stack
Note 2: Priority order
Priority 2: * /
Priority 1: + -



Convert Infix to Postfix

```
Algorithm inToPostFix (formula)
Convert infix formula to postfix.
   Pre    formula is infix notation that has been edited
          to ensure that there are no syntactical errors
   Post   postfix formula has been formatted as a string
   Return postfix formula
1  createStack (stack)
2  loop (for each character in formula)
   1   if (character is open parenthesis)
       1  pushStack (stack, character)
   2   elseif (character is close parenthesis)
       1  popStack (stack, character)
       2  loop (character not open parenthesis)
          1  concatenate character to postFixExpr
          2  popStack (stack, character)
       3  end loop
   3   elseif (character is operator)
          Test priority of token to token at top of stack
       1  stackTop (stack, topToken)
       2  loop (not emptyStack (stack)
            AND priority(character) <= priority(topToken))
          1  popStack (stack, tokenOut)
          2  concatenate tokenOut to postFixExpr
          3  stackTop (stack, topToken)
```

Convert Infix to Postfix (cont.)

```
      3  end loop
      4  pushStack (stack, token)
   4  else
         Character is operand
      1 Concatenate token to postFixExpr
   5  end if
3  end loop
   Input formula empty. Pop stack to postFix
4  loop (not emptyStack (stack))
   1  popStack (stack, character)
   2  concatenate token to postFixExpr
5  end loop
6  return postFix
end inToPostFix
```

Convert Infix to Postfix in 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: ");
```

Convert Infix to Postfix in C (cont.)

```
30  括號的處理
31     while ((token = getchar ())!= '\n')
32         {
33          if (token == '(')
34             {
35              dataPtr  = (char*) malloc (sizeof(char));
36              *dataPtr = token;
37              pushStack (stack, dataPtr);
38             } // if
39          else if (token == ')')
40             {
41              dataPtr = (char*)popStack (stack);
42              while (*dataPtr != '(')
43                 {
44                  temp [0]= *dataPtr;
45                  strcat (postfix , temp);
46                  dataPtr = (char*)popStack (stack);
47                 } // while
48             } // else if
```

Convert Infix to Postfix in C (cont.)

```
49          else if (isOperator (token))
50             {
51              // test priority of token at stack top
52              dataPtr = (char*)stackTop (stack);
53              while (!emptyStack (stack)
54                && priority (token) <= priority (*dataPtr))
55                  {
56                   dataPtr  = (char*)popStack (stack);
57                   temp [0] = *dataPtr;
58                   strcat (postfix , temp);
59                   dataPtr  = (char*)stackTop (stack);
60                  } // while
61              dataPtr  = (char*) malloc (sizeof (char));
62              *dataPtr = token;
63              pushStack (stack , dataPtr);
64             } // else if
```

Convert Infix to Postfix in C (cont.)

```
65          else                    // character is operand
66             {
67              temp[0]= token;
68              strcat (postfix , temp);
69             } // else
70          } // while get token
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
```

Convert Infix to Postfix in C (cont.)

```
 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
```

Convert Infix to Postfix in C (cont.)

```
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
```

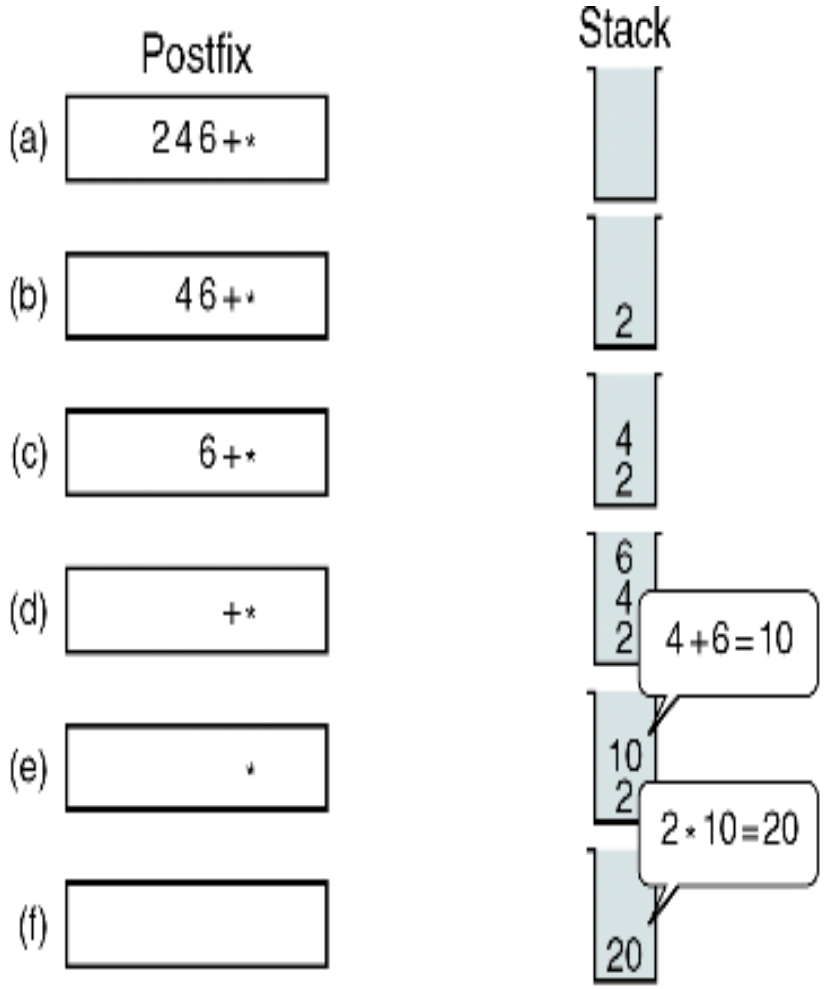
```
Results:
Run 1
   Enter an infix formula: 2+4
   The postfix formula is: 24+
Run 2
   Enter an infix formula: (a+b)*(c-d)/e
   The postfix formula is: ab+cd-*e/
```

Evaluation of Postfix Expression

Consider postfix expression A B C + * with A = 2, B = 4, and C = 6

- **Put the operands (values), instead of operators, into stack**
- **Pop two operands at top of the stack and perform the operation**
- **Push the value back into the stack**

Q: how to convert postfix to infix ?



Evaluation of Postfix Expression

```
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
```

Evaluation of Postfix Expression in C

```
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;
```

Evaluation of Postfix Expression in C

```
22      int*   dataPtr;
23      STACK* stack;
24
25  // Statements
26      // Create Stack
27      stack = createStack ();
28
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
```

Evaluation of Postfix Expression in C

```
39
40          else
41             // character is operand
42             {
43              dataPtr  = (int*)popStack (stack);
44              operand2 = *dataPtr;
45              dataPtr  = (int*)popStack (stack);
46              operand1 = *dataPtr;
47              value    = calc(operand1, token, operand2);
48              dataPtr  = (int*) malloc (sizeof (int));
49              *dataPtr = value;
50              pushStack (stack, dataPtr);
51             } // else
52          } // while
```

Evaluation of Postfix Expression in C

```
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
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)
```

Evaluation of Postfix Expression

```
70  {
71  // Statements
72     if (token == '*'
73         || token == '/'
74         || token == '+'
75         || token == '-')
76        return true;
77     return false;
78  }  // isOperator
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  */
```

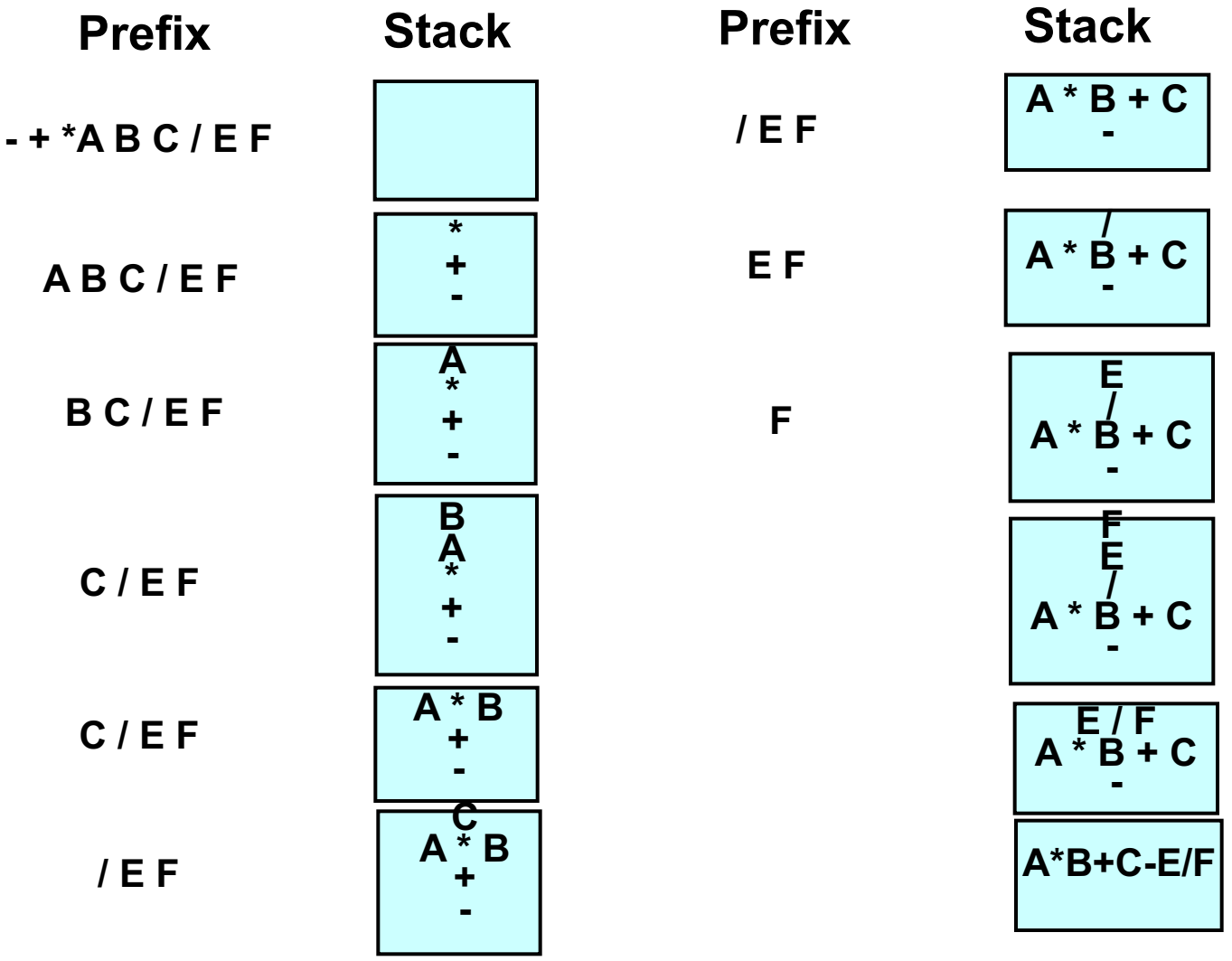
Evaluation of Postfix Expression

```
 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
```

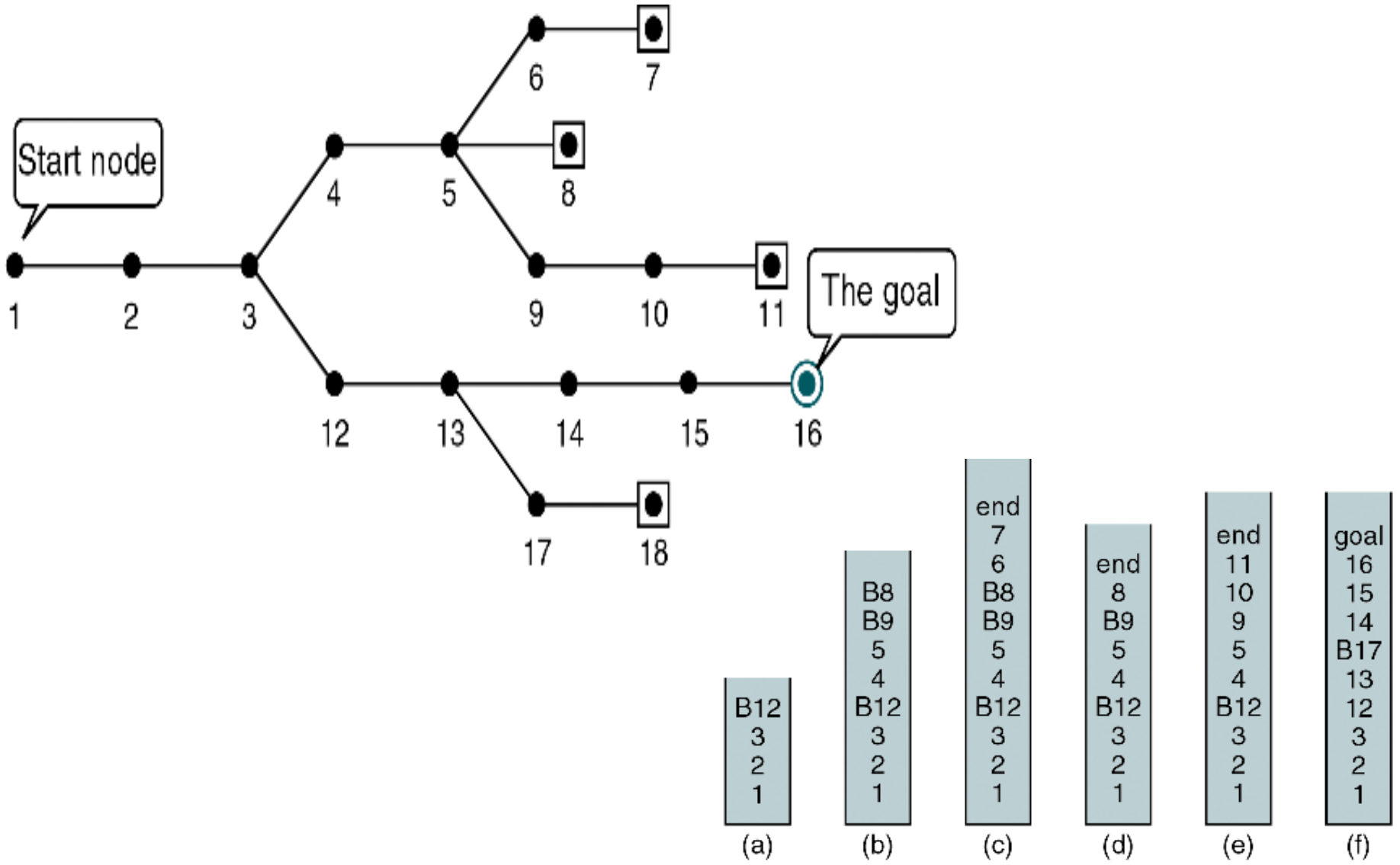
```
Results:
Input formula: 52/4+5*2+
The result is 32
```

Convert Prefix to Infix using Stack

Example: - + *A B C / E F → A * B + C – E / F



Backtracking: Goal Seeking



Goal Seeking Program

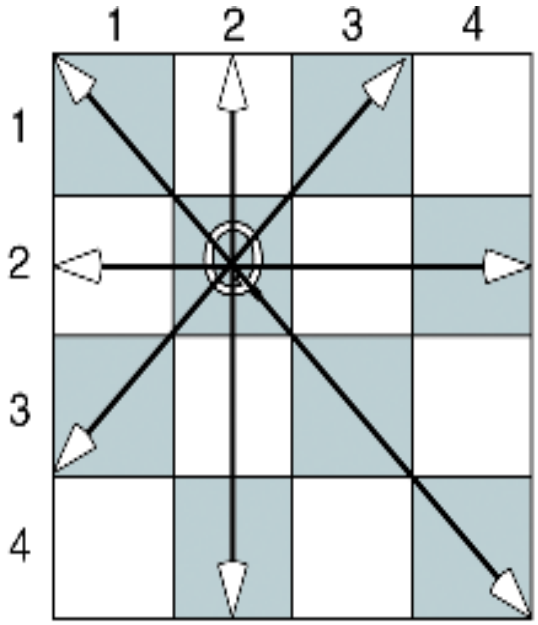
```
Algorithm seekGoal (map)
This algorithm determines the path to a desired goal.
   Pre  a graph containing the path
   Post path printed
1  createStack (stack)
2  set pMap to starting point
3  loop (pMap not null AND goalNotFound)
   1   if (pMap is goal)
       1   set goalNotFound to false
   2   else
       1   pushStack (stack, pMap)
       2   if (pMap is a branch point)
           1   loop (more branch points)
               1   create branchPoint node
               2   pushStack (stack, branchPoint)
           2   end loop
       3   end if
       4   advance to next node
   3   end if
4  end loop
```

Goal Seeking Program (cont.)

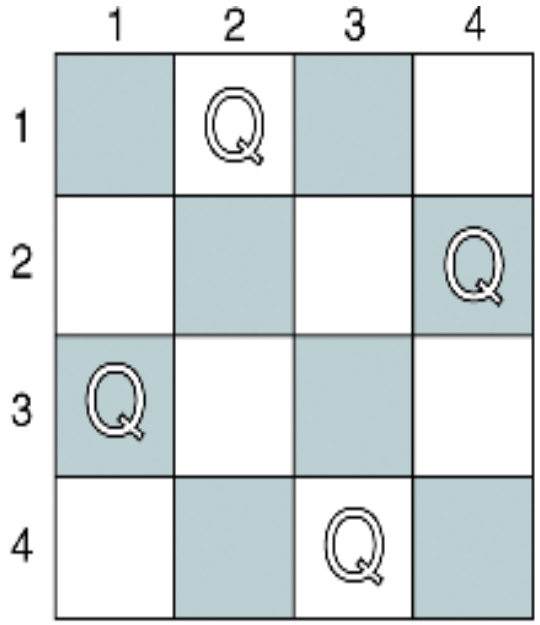
```
5  if (emptyStack (stack))
   1   print (There is no path to your goal)
6  else
   1   print (The path to your goal is:)
   2   loop (not emptyStack (stack))
       1   popStack (stack, pMap)
       2   if (pMap not branchPoint)
           1   print(map point)
       3   end if
   3   end loop
   4   print (End of Path)
7  end if
8  return
end seekGoal
```

Eight Queens Problem

For example: four queens problem and solution



(a) Queen capture rules

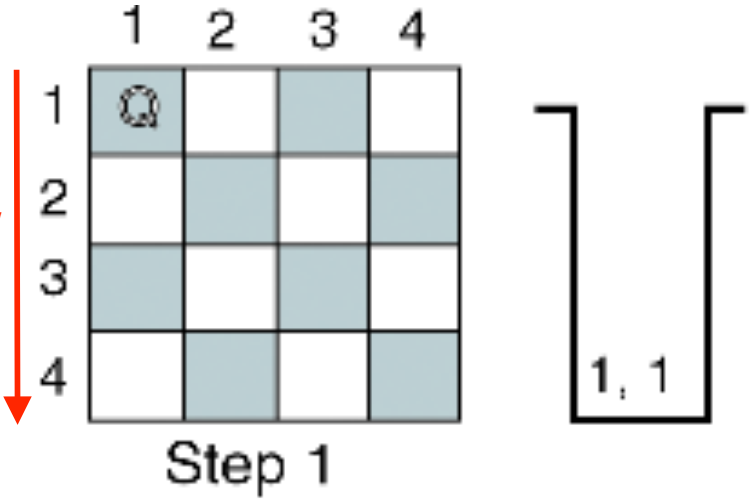


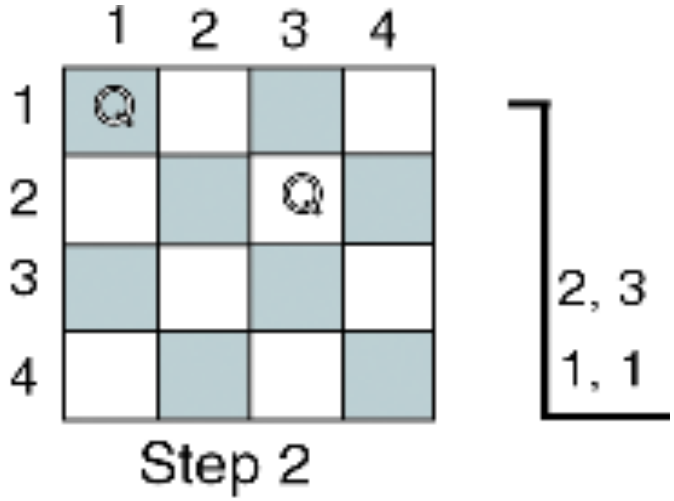
(b) First four queens solution

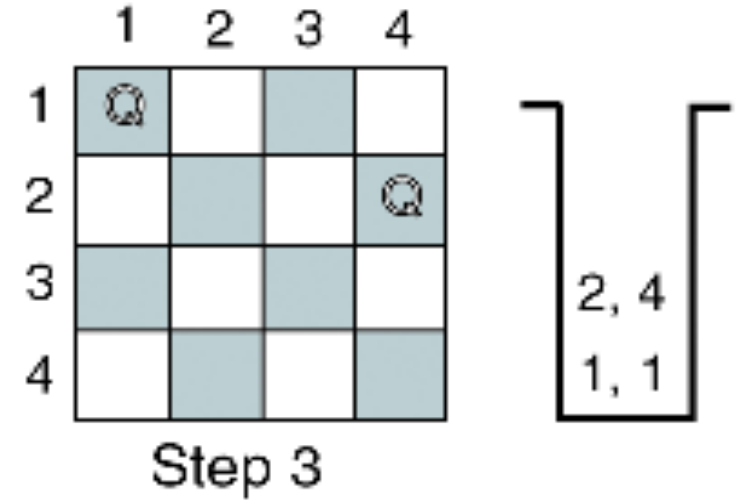
Four Queens Step-by-Step Solution

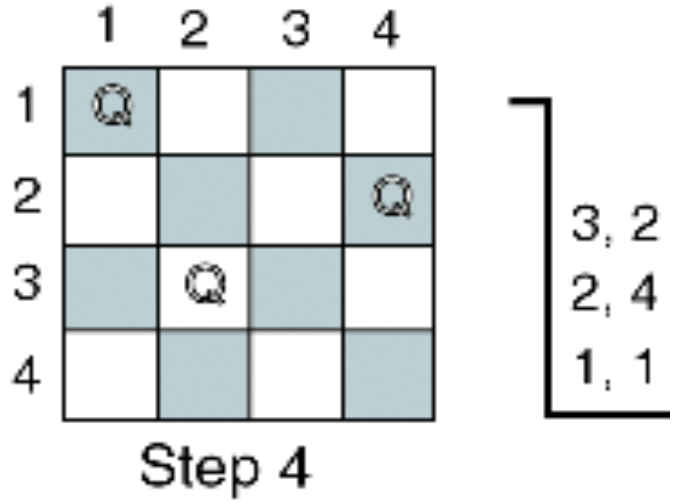
找不到解就backtrack（從stack裡pop出來）

每一個row只能擺一個

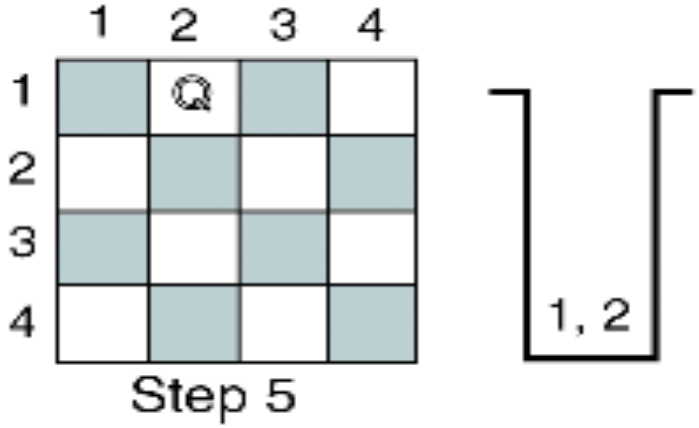




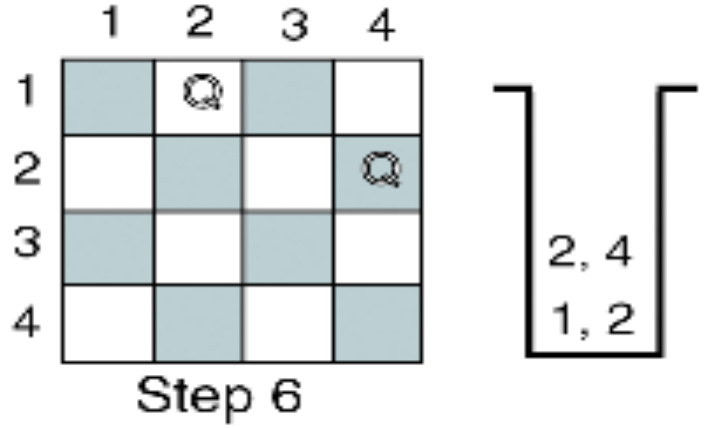




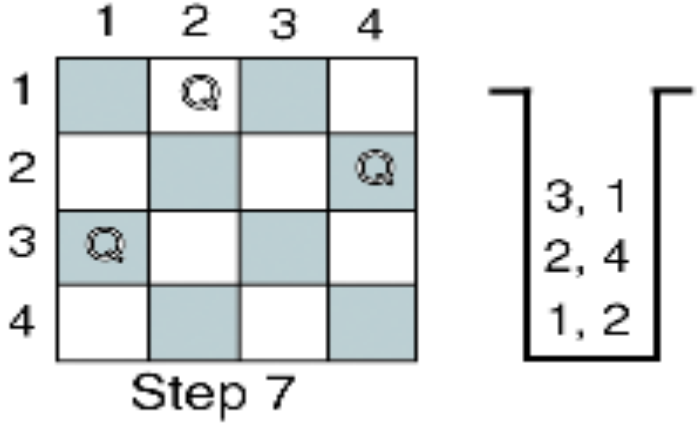
4 Queens Step-by-Step Solution (cont.)



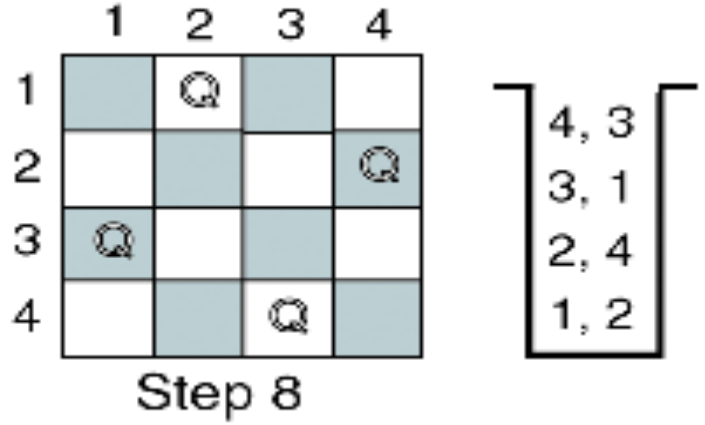
Step 5



Step 6



Step 7

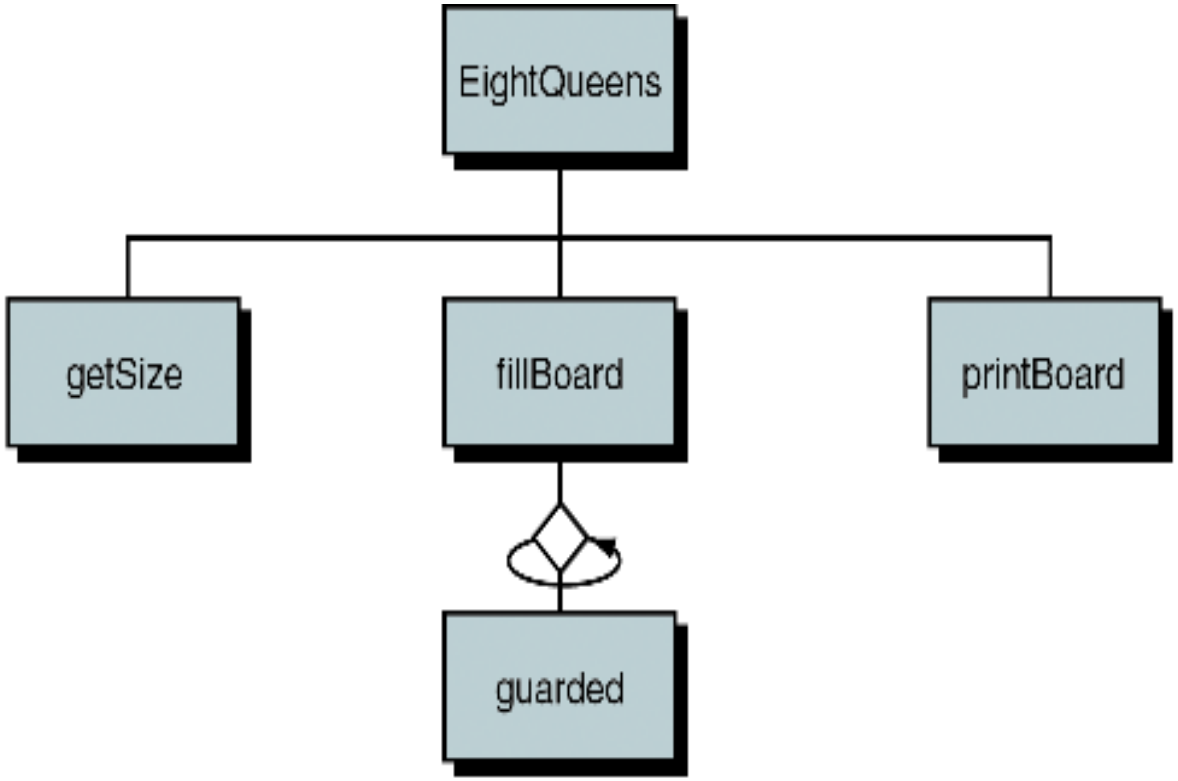


Step 8

Eight Queens Problem

```
Algorithm queens8 (boardSize)
Position chess queens on a game board so that no queen can
capture any other queen.
   Pre  boardSize is number of rows & columns on board
   Post queens' positions printed
 1 createStack (stack)
 2 set row to 1
 3 set col to 0
 4 loop (row <= boardSize)
   1   loop (col <= boardSize AND row <= boardSize)
       1   increment col
       2   if (not guarded (row, col))
           1   place queen at row-col intersection on board
           2   pushStack (row-col into stack)
           3   increment row
           4   set col to 0
       3   end if
           At end of row. Back up to previous position.
       4   loop (col >= boardSize)
           1   popStack (row-col from stack)
           2   remove queen from row-col intersection on board
       5   end loop
   2   end loop
 5 end loop
 6 printBoard (stack)
 7 return
end queens8
```

Design for Eight Queens



Eight Queens: Mainline

```
 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);
27
```

跟前面擺的有沒有衝突

Eight Queens: Mainline (cont.)

```
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
```

Eight Queens: Get Board Size

```
1  /* ===================== getSize =====================
2     Prompt user for a valid board size.
3        Pre  nothing
4        Post valid board size returned
5  */
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: ");
```

Eight Queens: Get Board Size (cont.)

```
 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
```

Eight Queens: Fill Board

```
 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
```

Eight Queens: Fill Board (cont.)

```
14      POSITION* pPos;
15
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
```

Eight Queens: Fill Board (cont.)

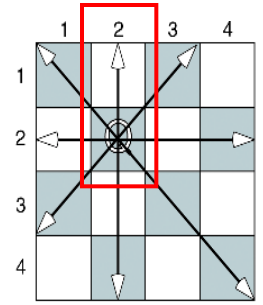
```
38              while (col >= boardSize)
39                {
40                 pPos = popStack(stack);
41                 row  = pPos->row;
42                 col  = pPos->col;
43                 board[row][col] = 0;
44                 free (pPos);
45                } // while col
46            } // while col
47        } // while row
48      return;
49  }  // fillBoard
```

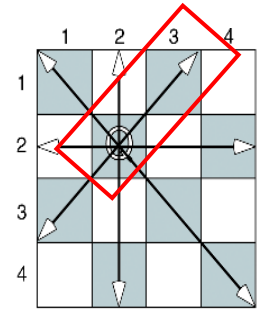
Eight Queens: Guarded

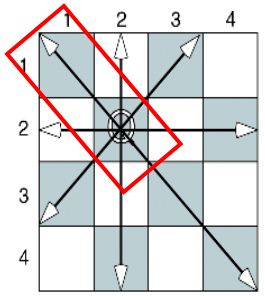
```
 1  /* ===================== guarded ==================
 2     Checks rows, columns, diagonals for guarding queens
 3
 4        Pre  board contains current positions for queens
 5             chkRow & chkCol are position for new queen
 6             boardSize is number of rows & cols in board
 7        Post returns true if guarded; false if not
 8  */
 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
```

Eight Queens: Guarded (cont.)

```
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
```







Eight Queens: Print Board

```
 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
```

Eight Queens: Print Board (cont.)

```
24      // Reverse stack for printing
25      pOutStack = createStack ();
26      while (!emptyStack (stack))
27         {
28          pPos = popStack (stack);
29          pushStack (pOutStack, pPos);
30         } // while
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
```

Eight Queens: Print Board (cont.)

```
47      destroyStack(pOutStack);
48      return;
49  }   // printBoard
```

```
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.
```

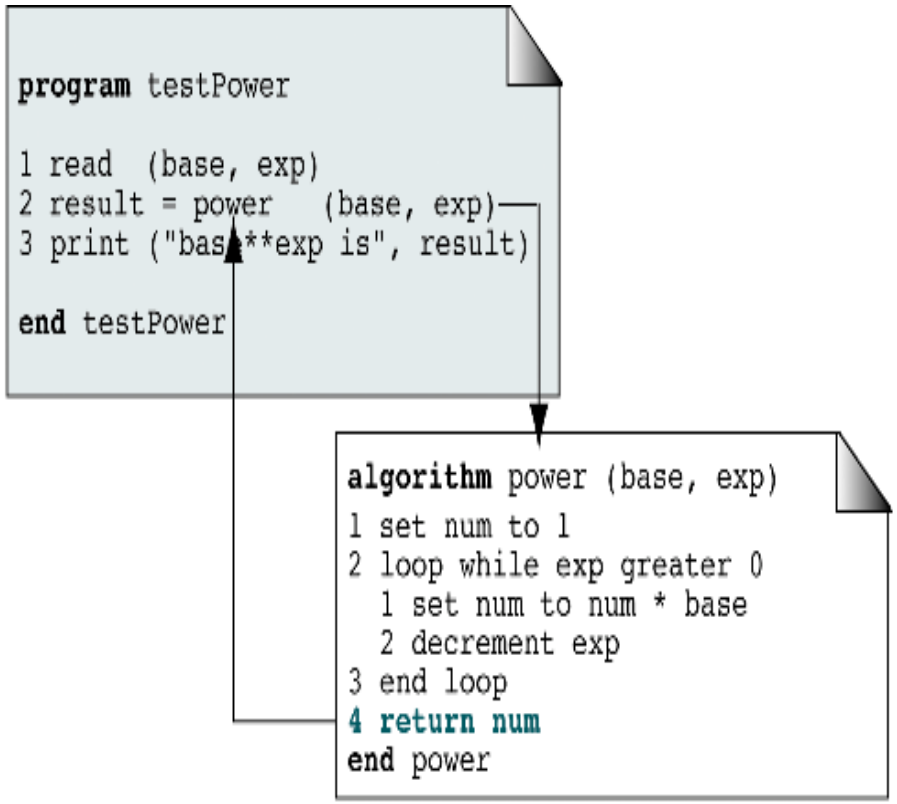
3-6 How Recursion Works

This section discusses the concept of the stack frame and the use of stacks in writing recursive software

Call and Return

A stackfrme contains four different elements

1. The parameters to be processed by the called algorithm
2. The local variables in the calling algorithm
3. The return statement in the calling algorithm
4. The expression that is to receive the return value(if any)



Recursive Power Algorithm

```
Algorithm power (base, exp)
This algorithm computes the value of a number, base, raised
to the power of an exponent, exp.
   Pre    base is the number to be raised
          exp is the exponent
   Post   value of base raised to power exp computed
   Return value of base raised to power exp returned
1  if (exp is 0)
   1   return (1)
2  else
   1   return (base * power (base, exp - 1))
3  end if
end power
```

Stackframes for Power (5, 2)