CSE 207

STACK

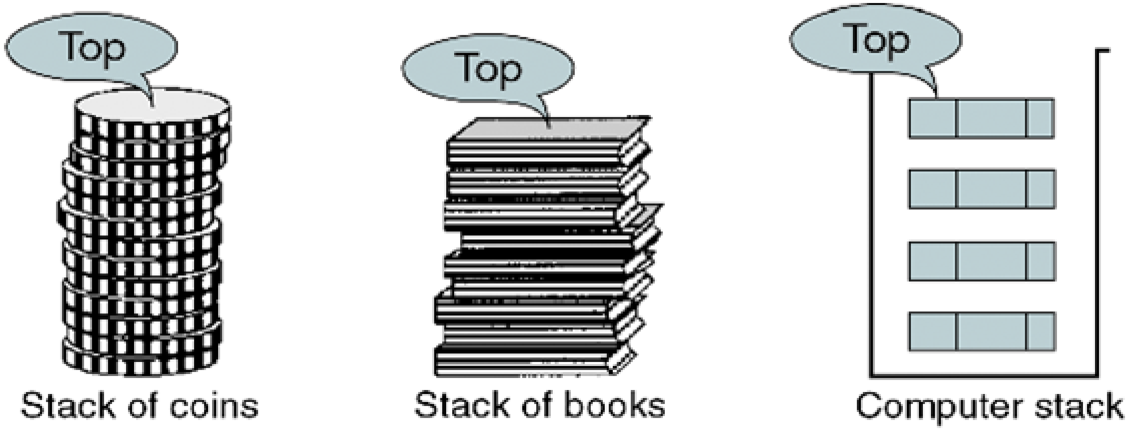
Stack

- A stack is linear list in which all additions and deletions are restricted to one end, called top

- If you insert a data series into a stack and then remove it, the order of the data will be reverse. i.e. data input as {5,10,15,20} is removed as {20,15,10,5}

- For this reversing attribute stack is called **LIFO- Last in First out**

Stack

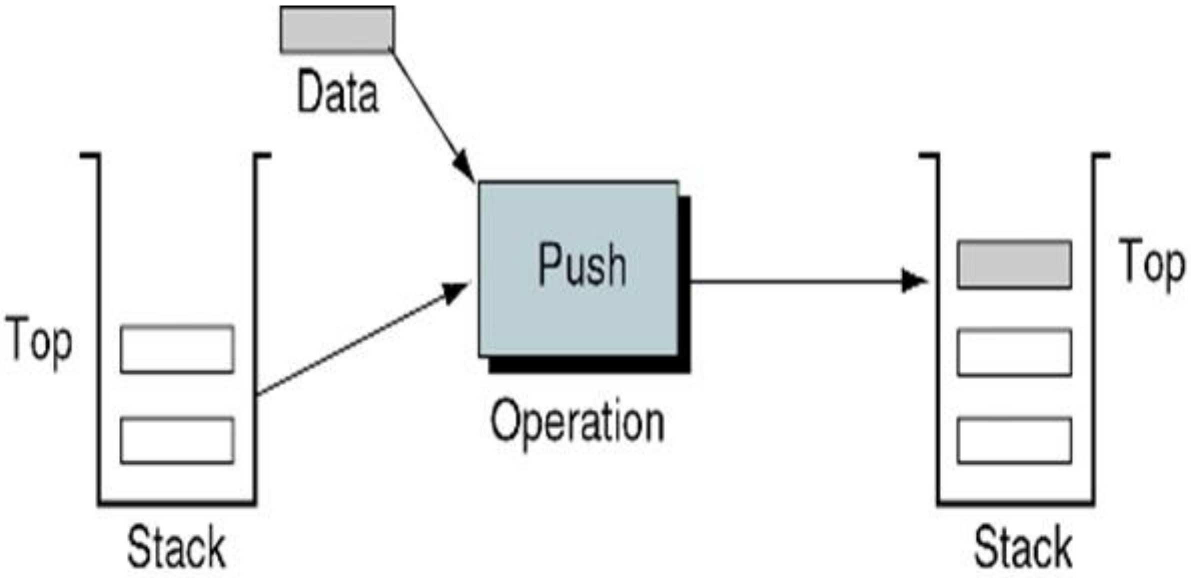


Basic Stack Operations

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

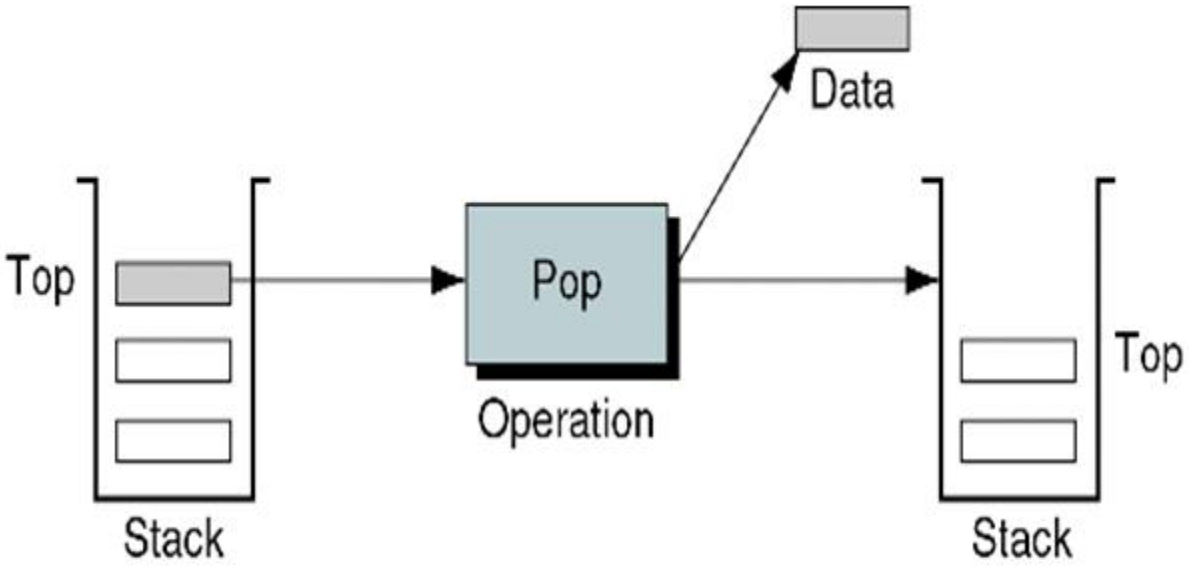
- **Push**
- **Pop**
- **Stack Top**

Push Operation



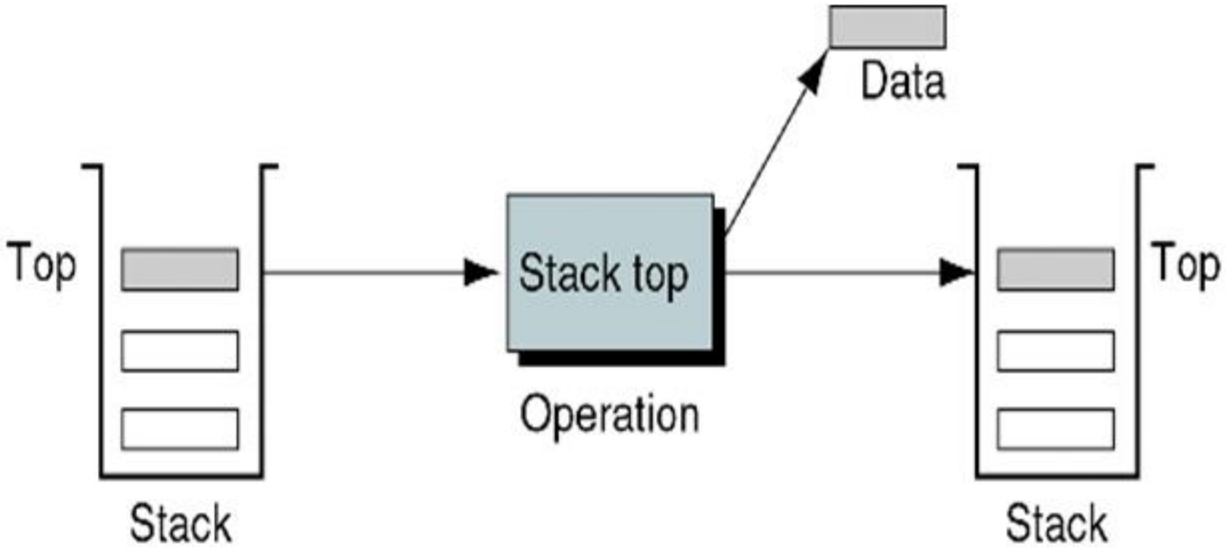
Push Stack Operation

POP Operation



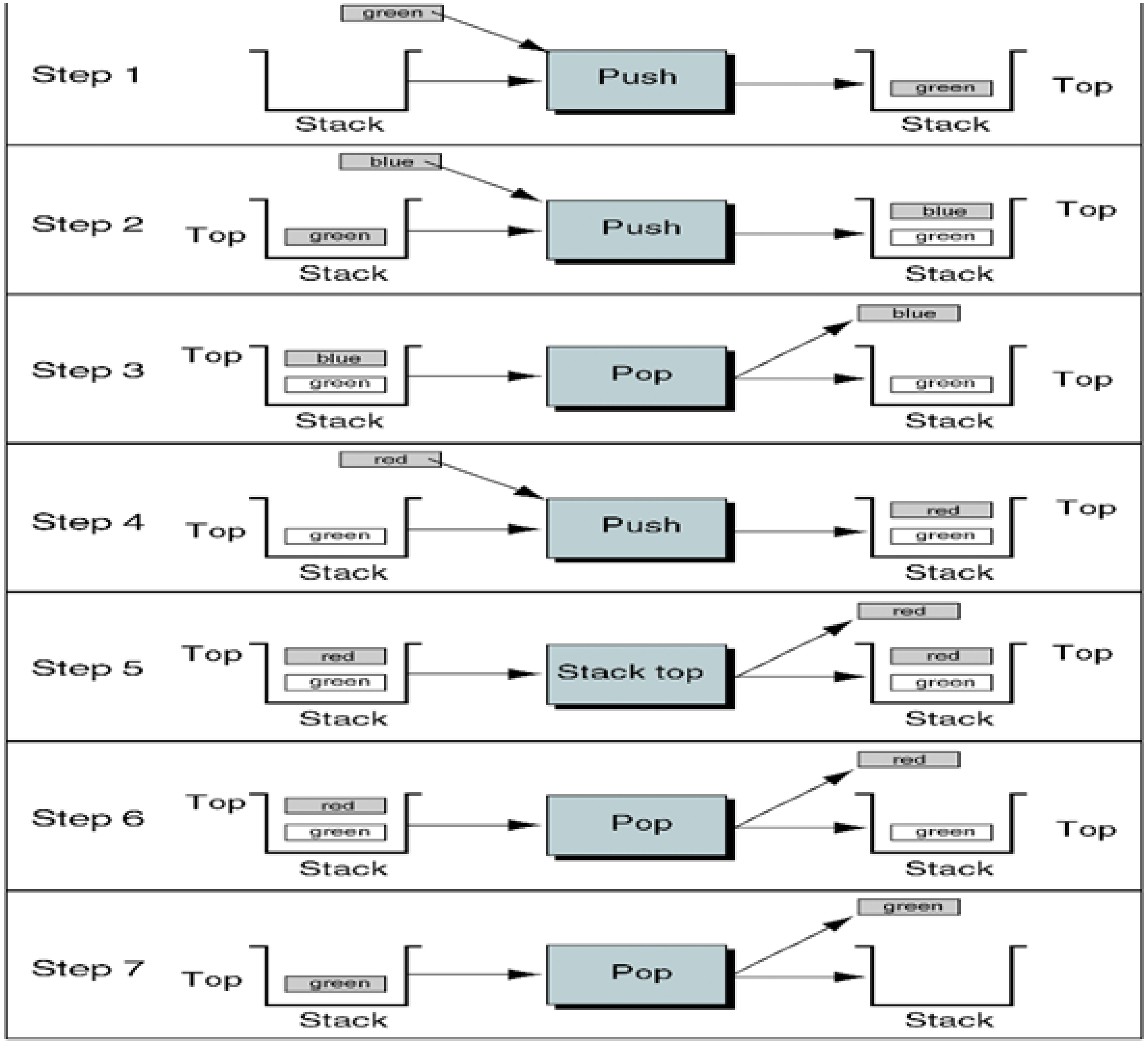
Pop Stack Operation

Stack Top Operation



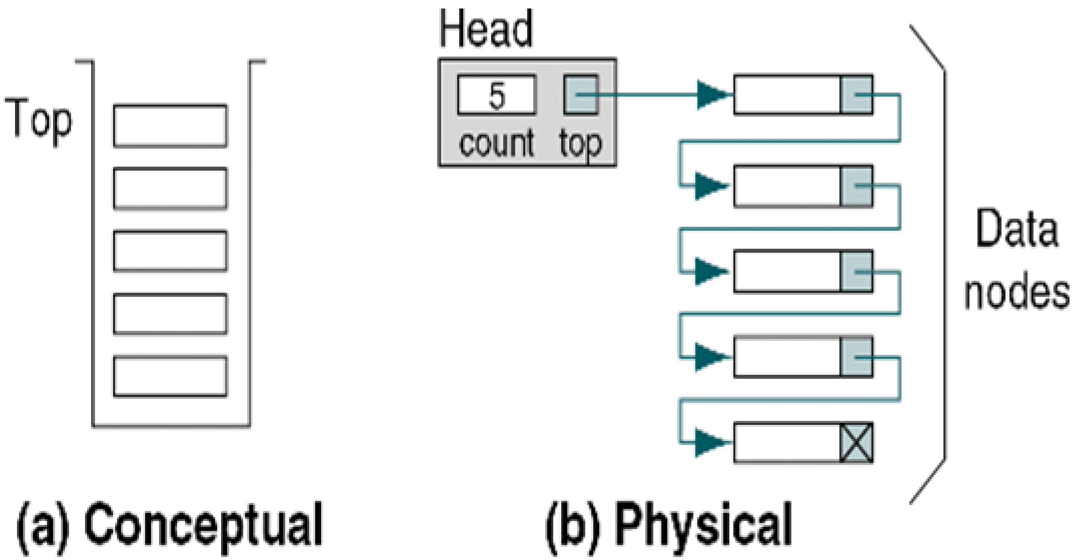
Stack Top Operation

Stack Example



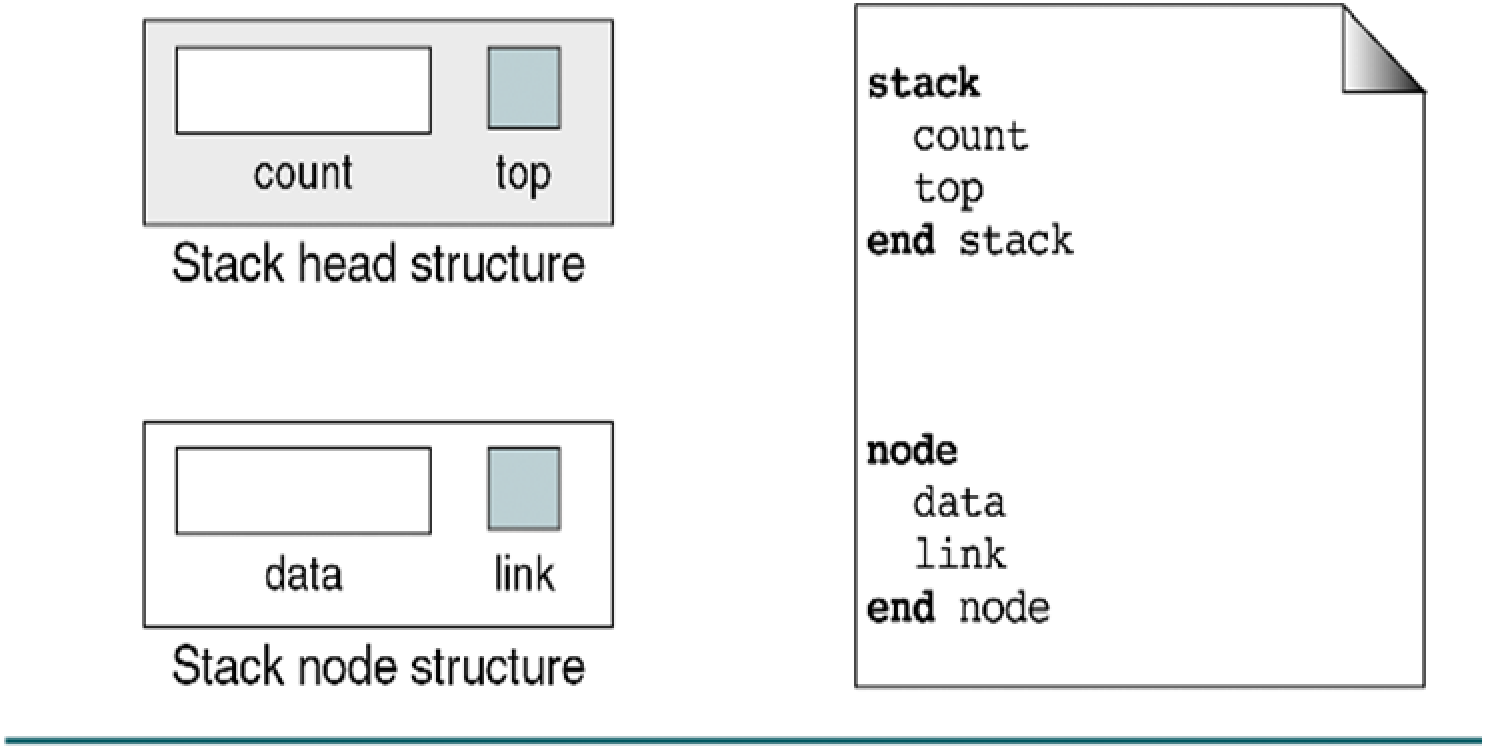
Stack Example

Stack Linked List Implementation



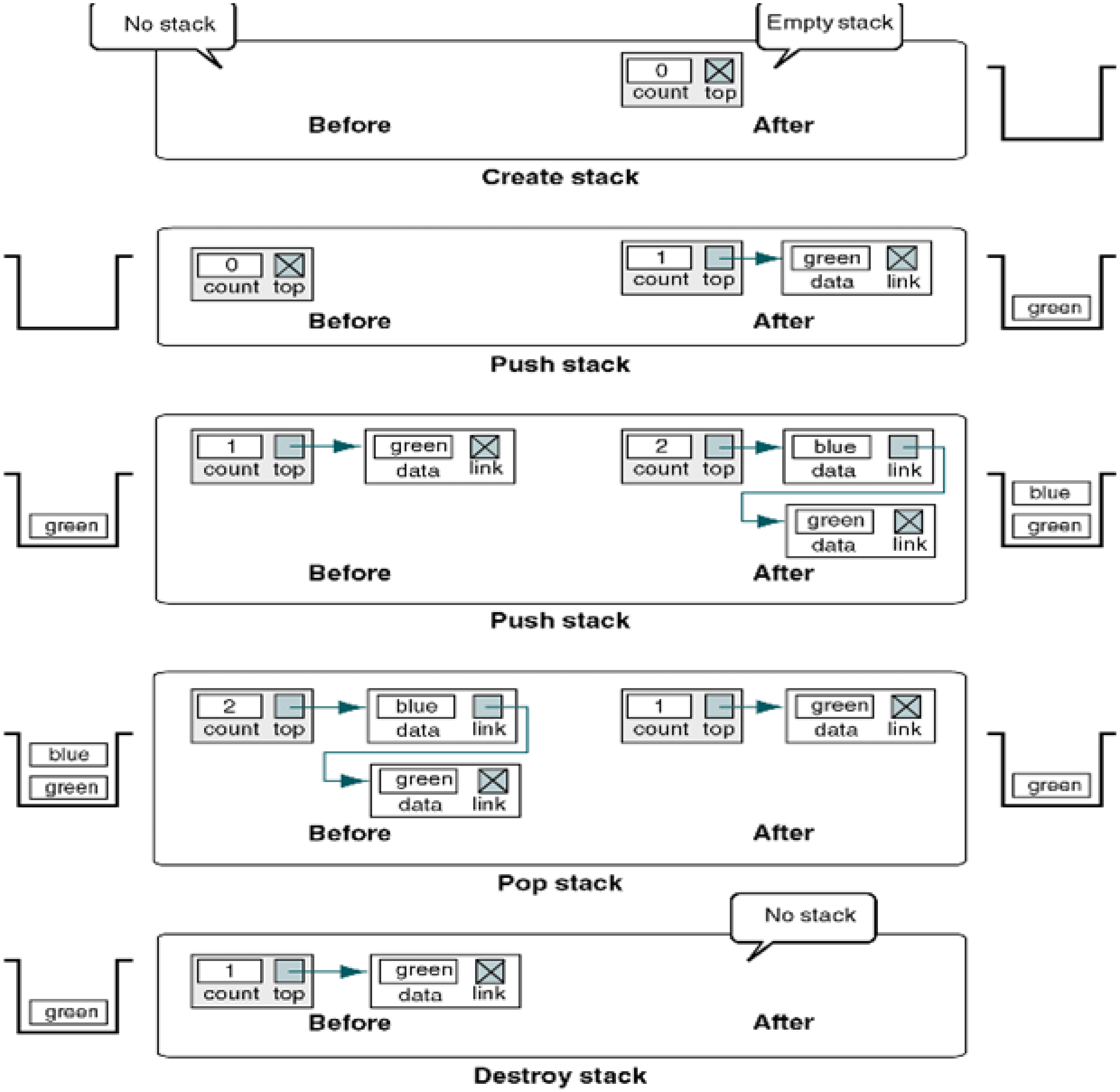
Conceptual and Physical Stack Implementations

Stack Linked List Implementation

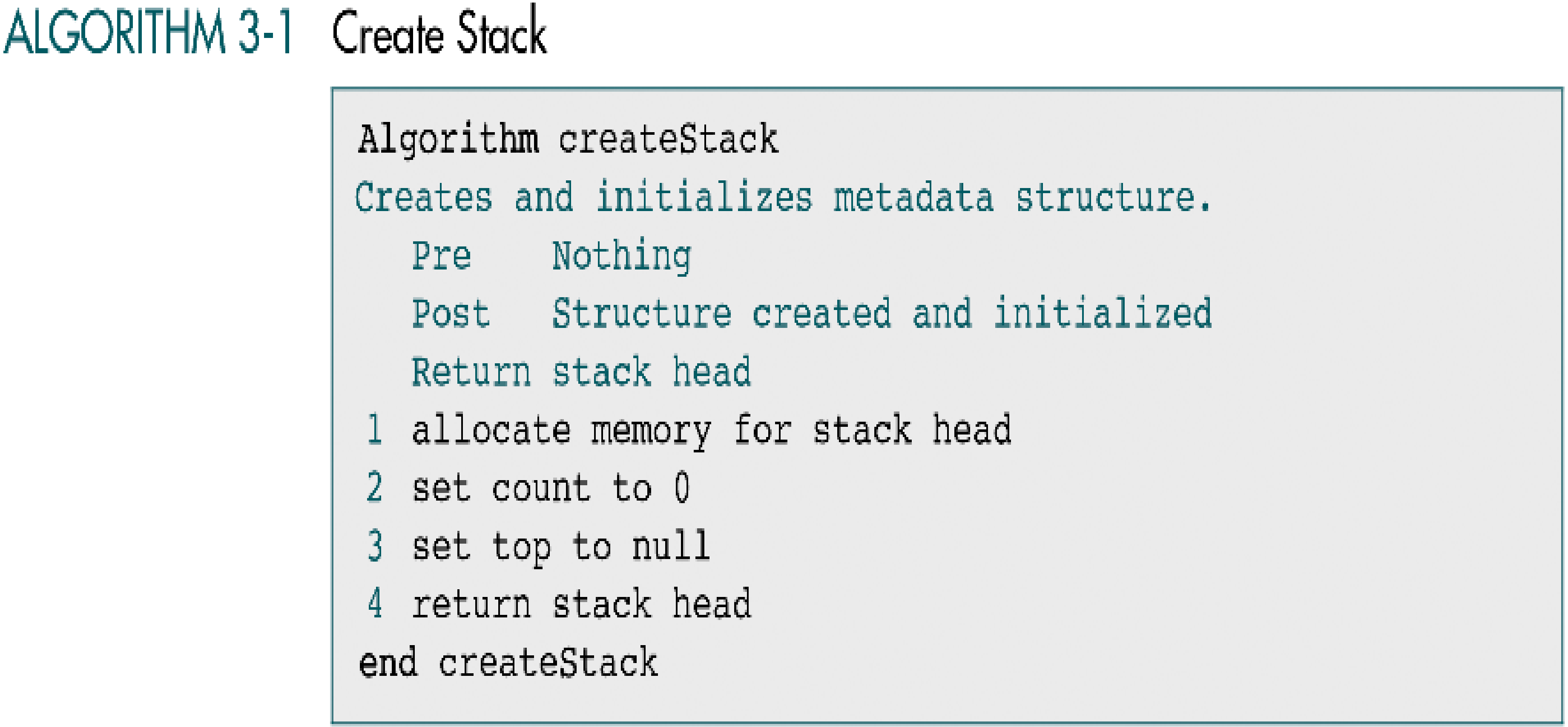


```
stack
  count
  top
end stack


node
  data
  link
end node
```

Stack Data Structure



Stack Operations



ALGORITHM 3-1 Create Stack

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

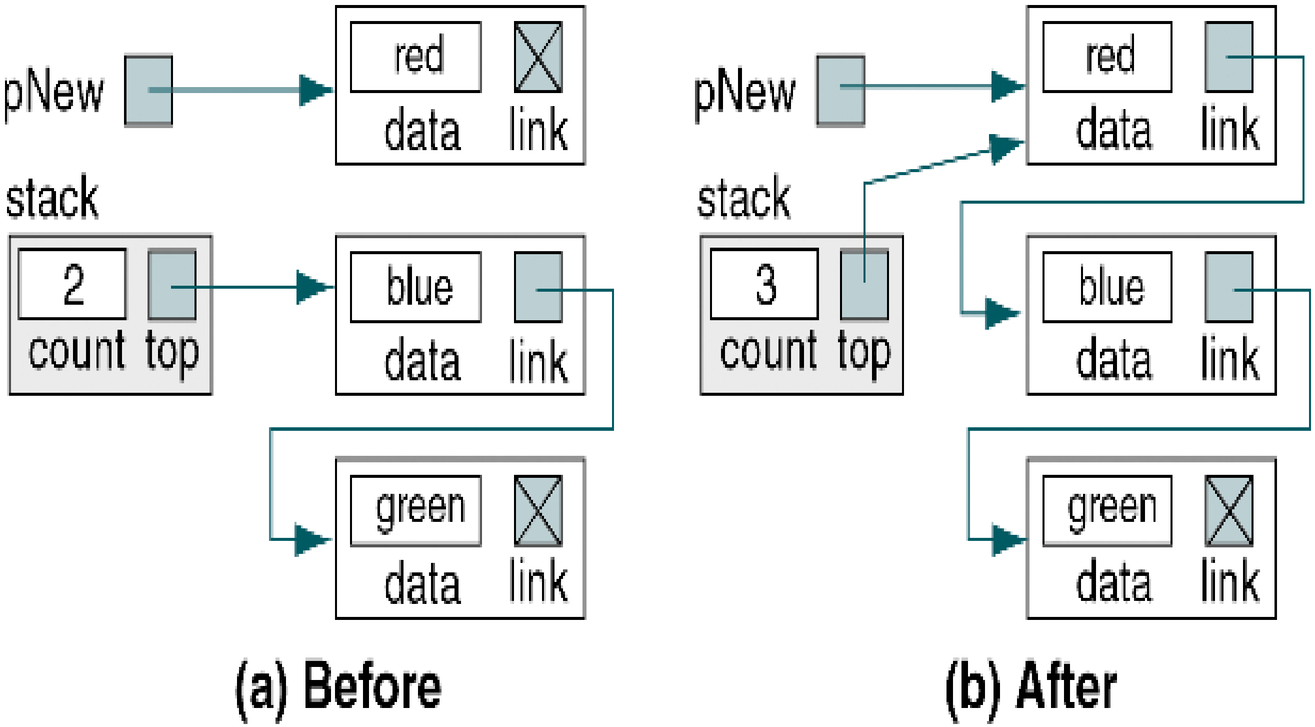
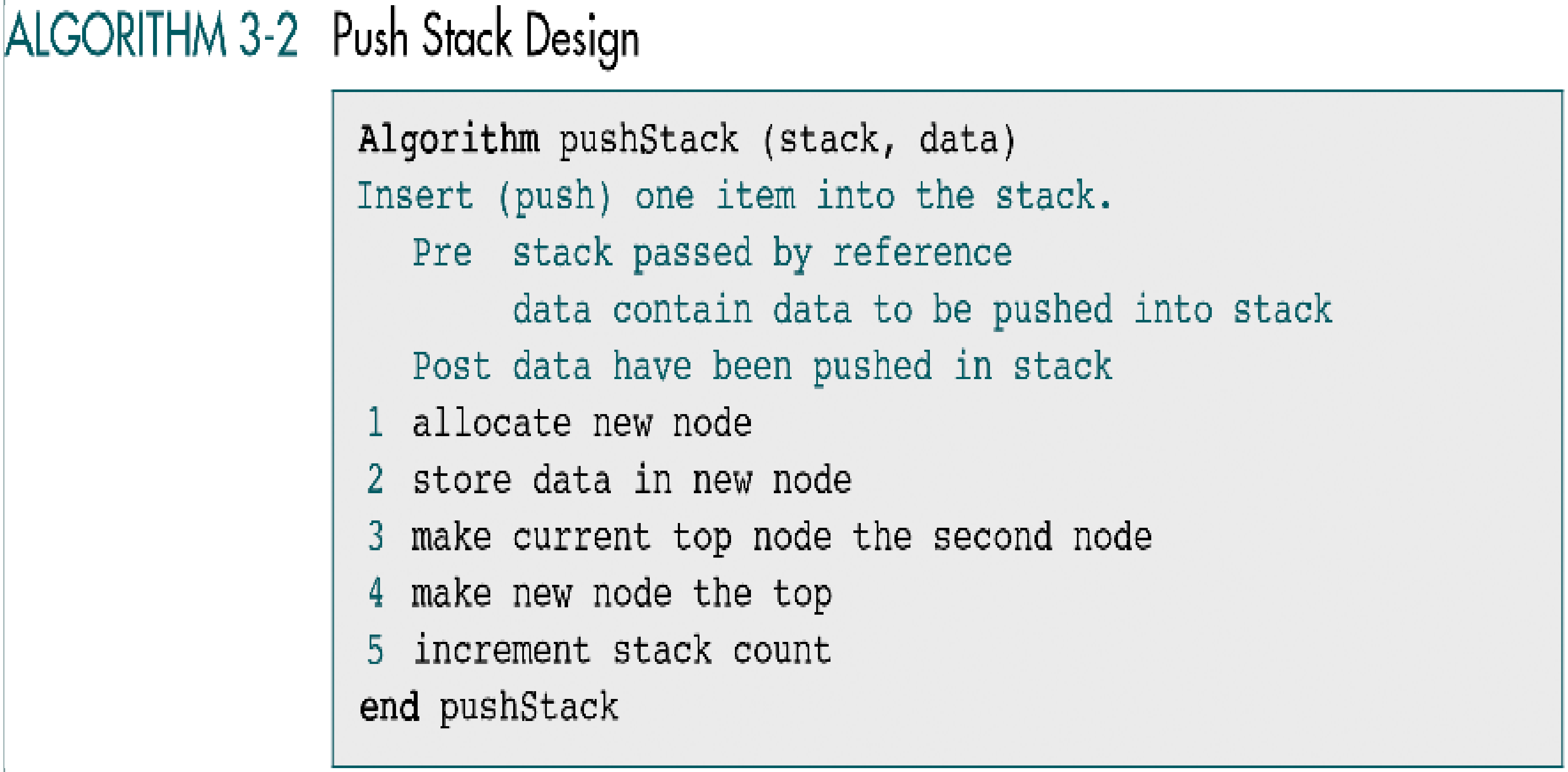


FIGURE 3-9 Push Stack Example



ALGORITHM 3-2 Push Stack Design

```
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
```

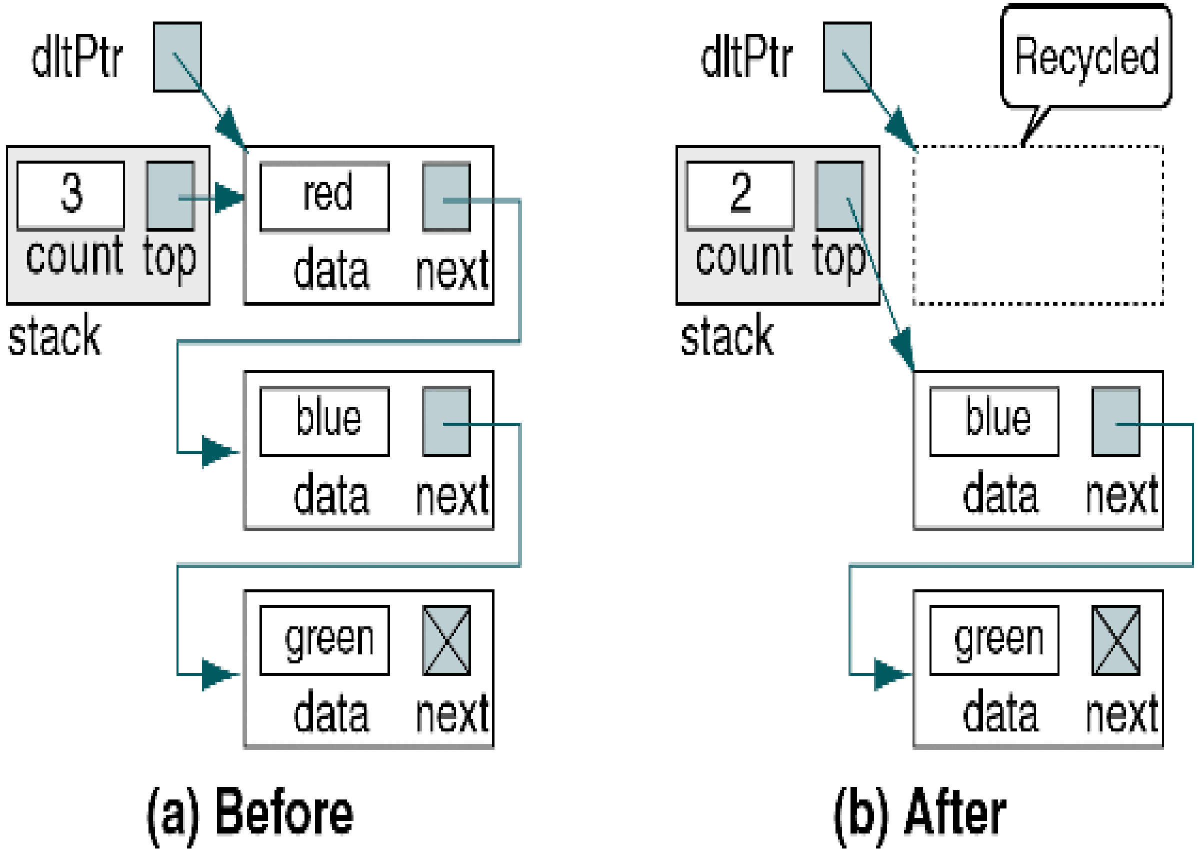
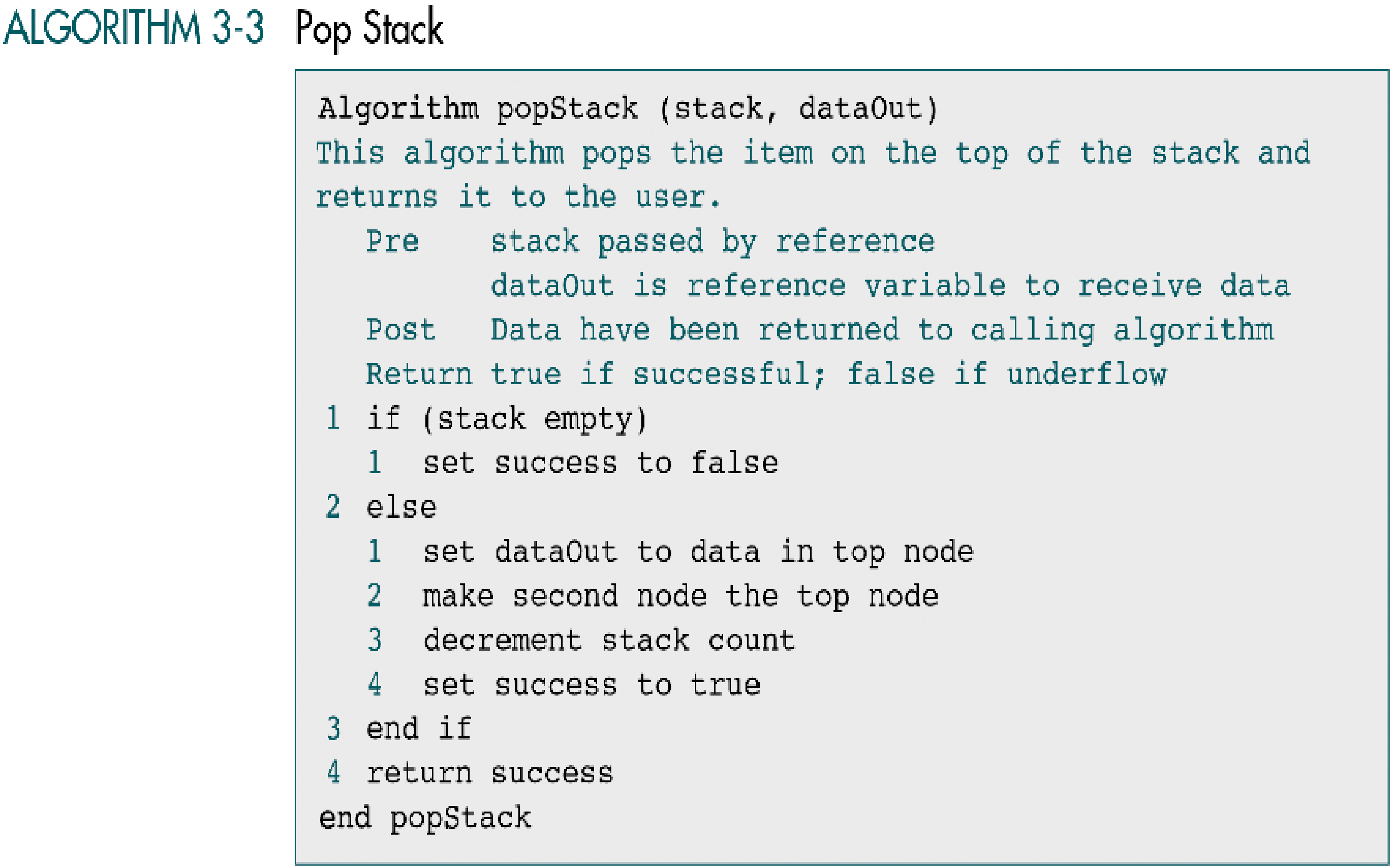
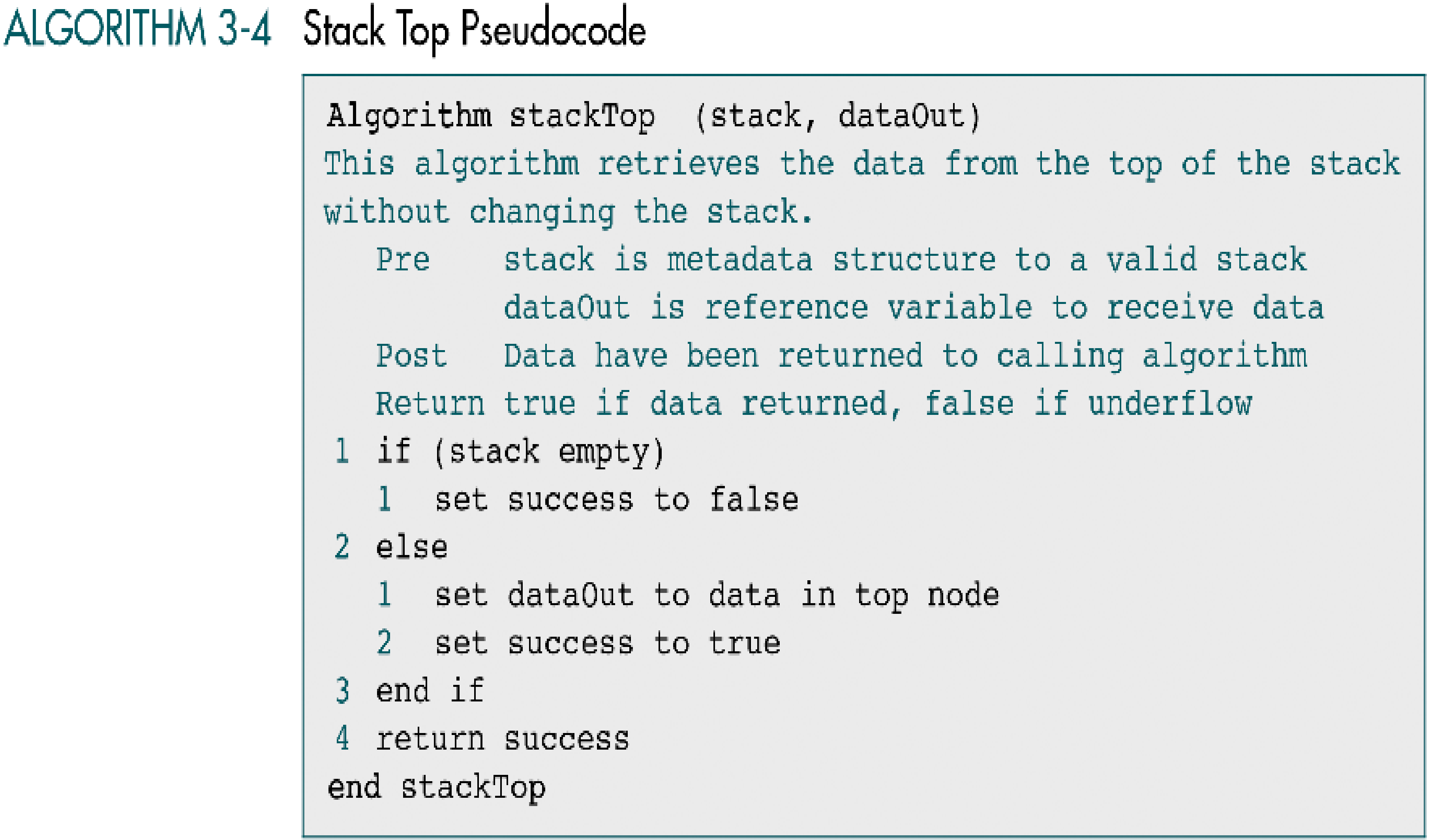


FIGURE 3-10 Pop Stack Example



ALGORITHM 3-3 Pop Stack

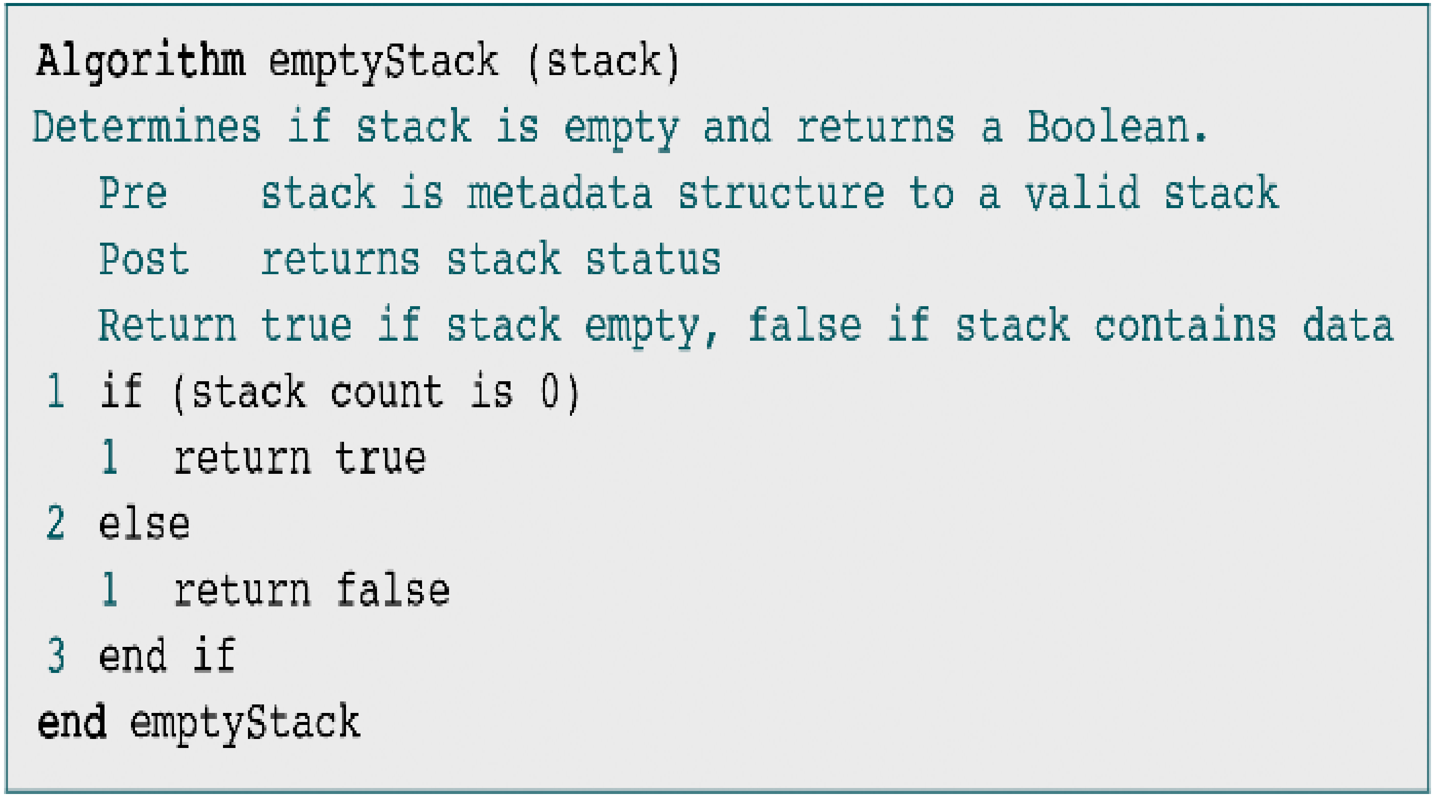
```
Algorithm popStack (stack, dataOut)
This algorithm pops the item on the top of the stack and
returns it to the user.
   Pre    stack passed by reference
          dataOut is reference variable to receive data
   Post   Data have been returned to calling algorithm
   Return true if successful; false 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
```



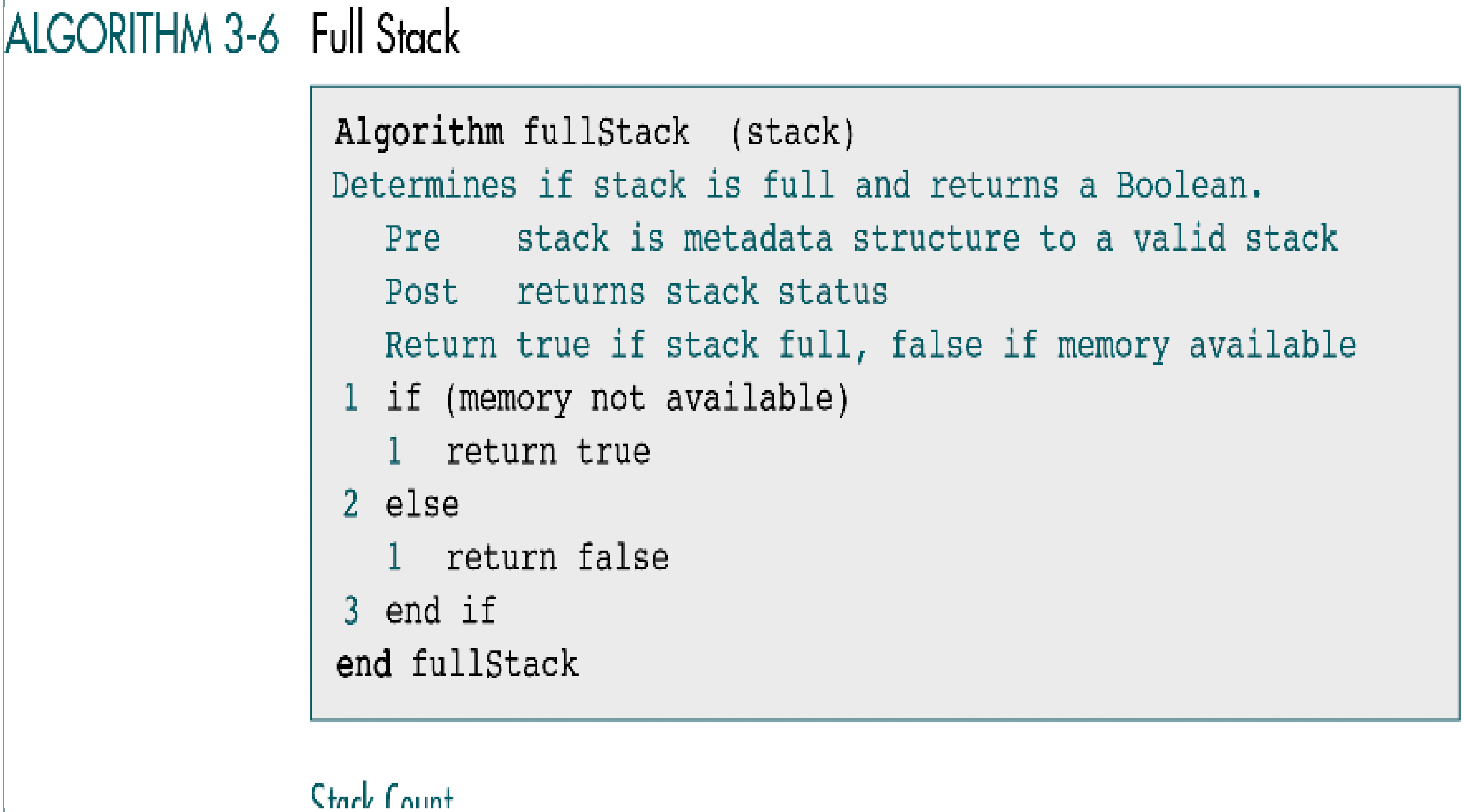
ALGORITHM 3-4 Stack Top Pseudocode

```
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
```

ALGORITHM 3-5 Empty Stack



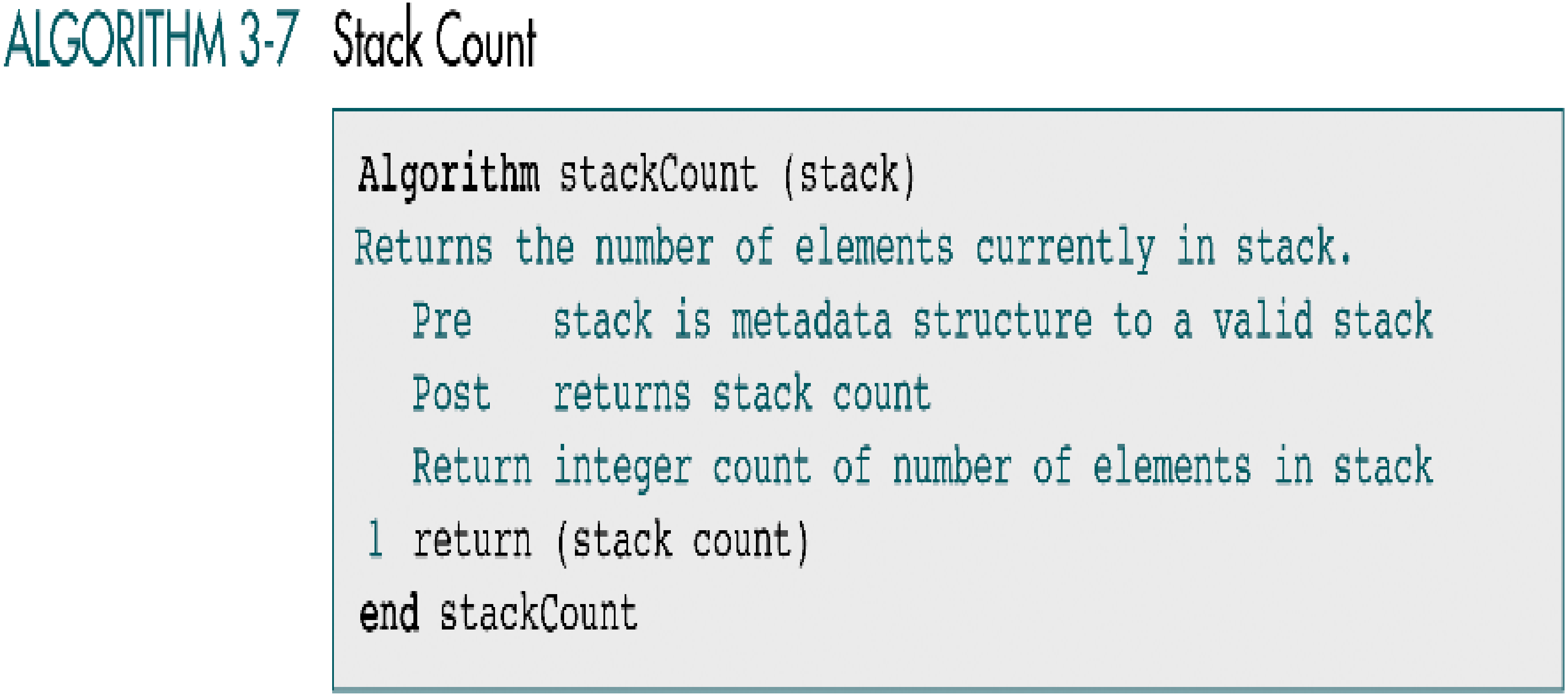
```
Algorithm emptyStack (stack)
Determines if stack is empty and returns a Boolean.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack status
   Return true if stack empty, false if stack contains data
1 if (stack count is 0)
   1 return true
2 else
   1 return false
3 end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



ALGORITHM 3-6 Full Stack

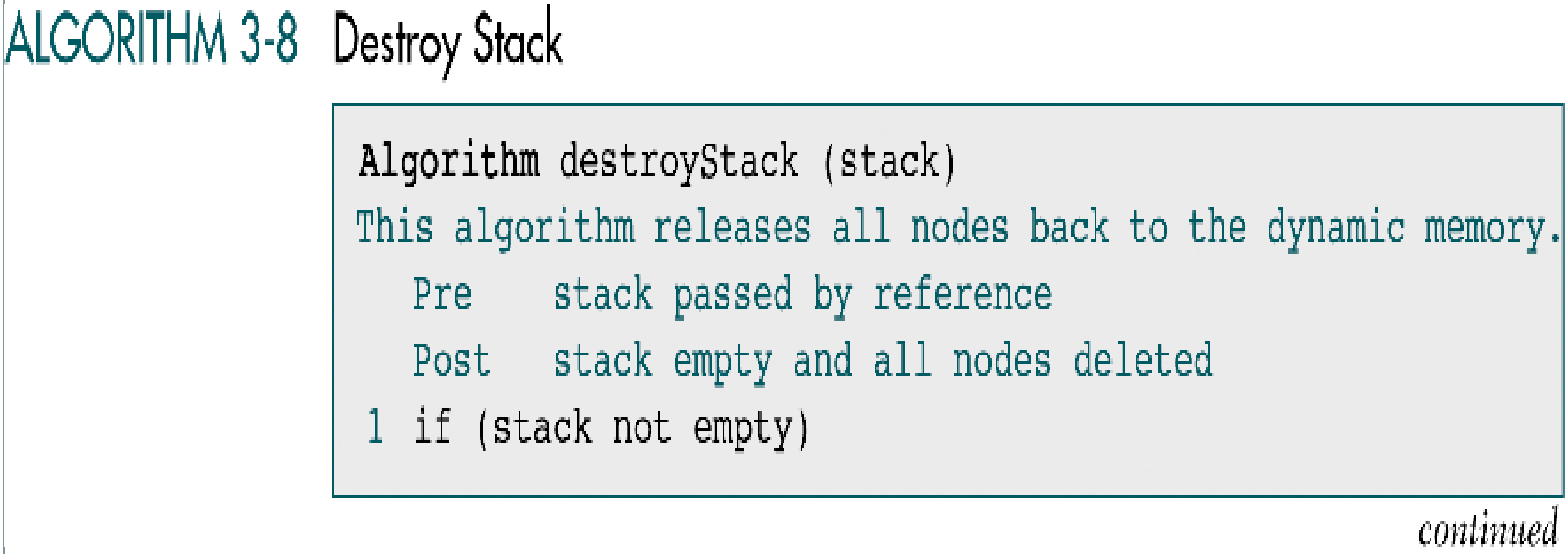
```
Algorithm fullStack  (stack)
Determines if stack is full and returns a Boolean.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack status
   Return true if stack full, false if memory available
1 if (memory not available)
   1 return true
2 else
   1 return false
3 end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

Stack Count



ALGORITHM 3-7 Stack Count

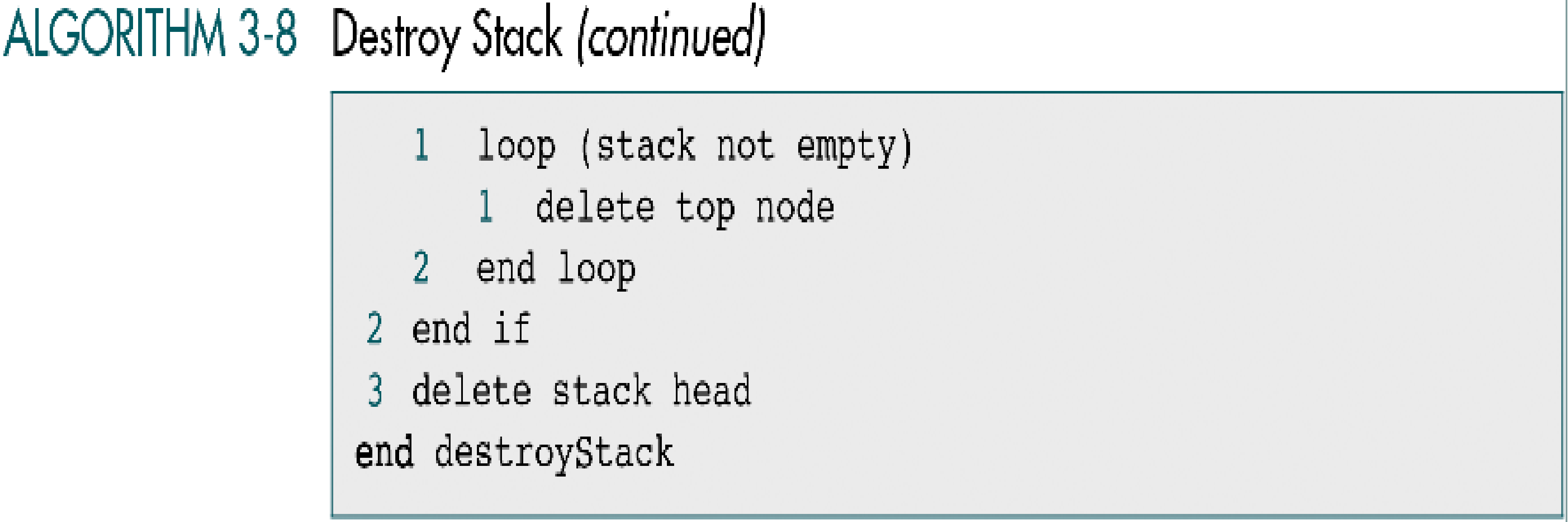
```
Algorithm stackCount (stack)
Returns the number of elements currently in stack.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack count
   Return integer count of number of elements in stack
   return (stack count)
end stackCount
```



ALGORITHM 3-8 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
if (stack not empty)
```

continued



ALGORITHM 3-8 Destroy Stack *(continued)*

```
      1  loop (stack not empty)
         1  delete top node
      2  end loop
   2  end if
   3  delete stack head
end destroyStack
```

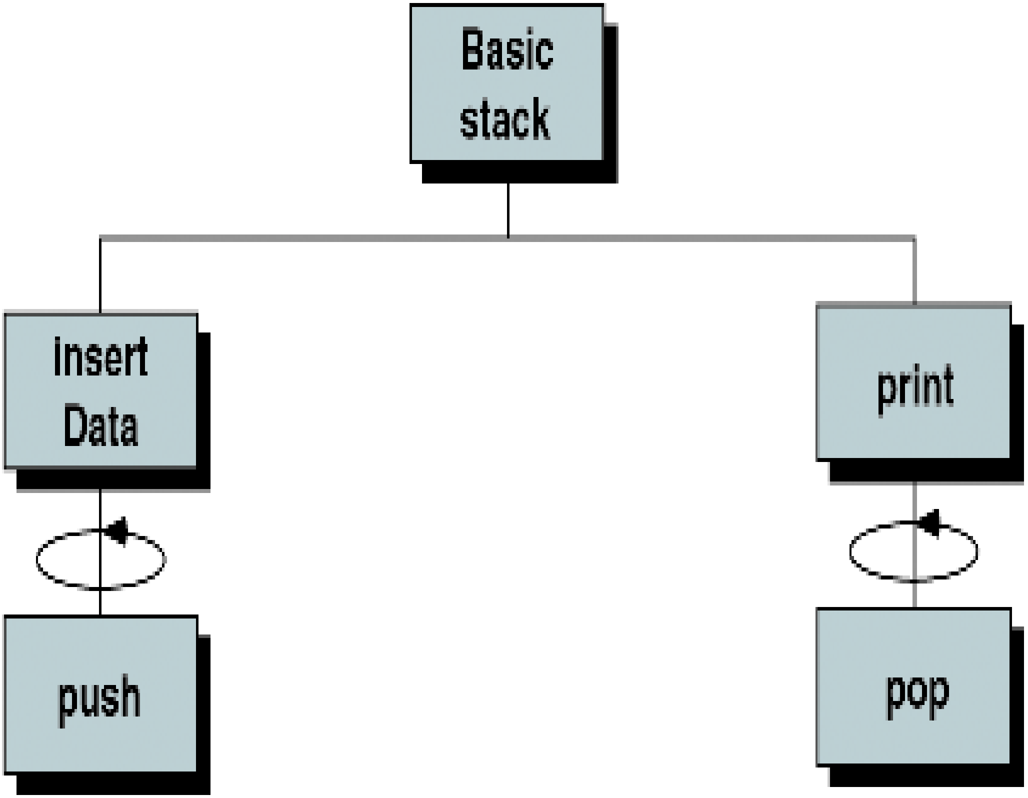
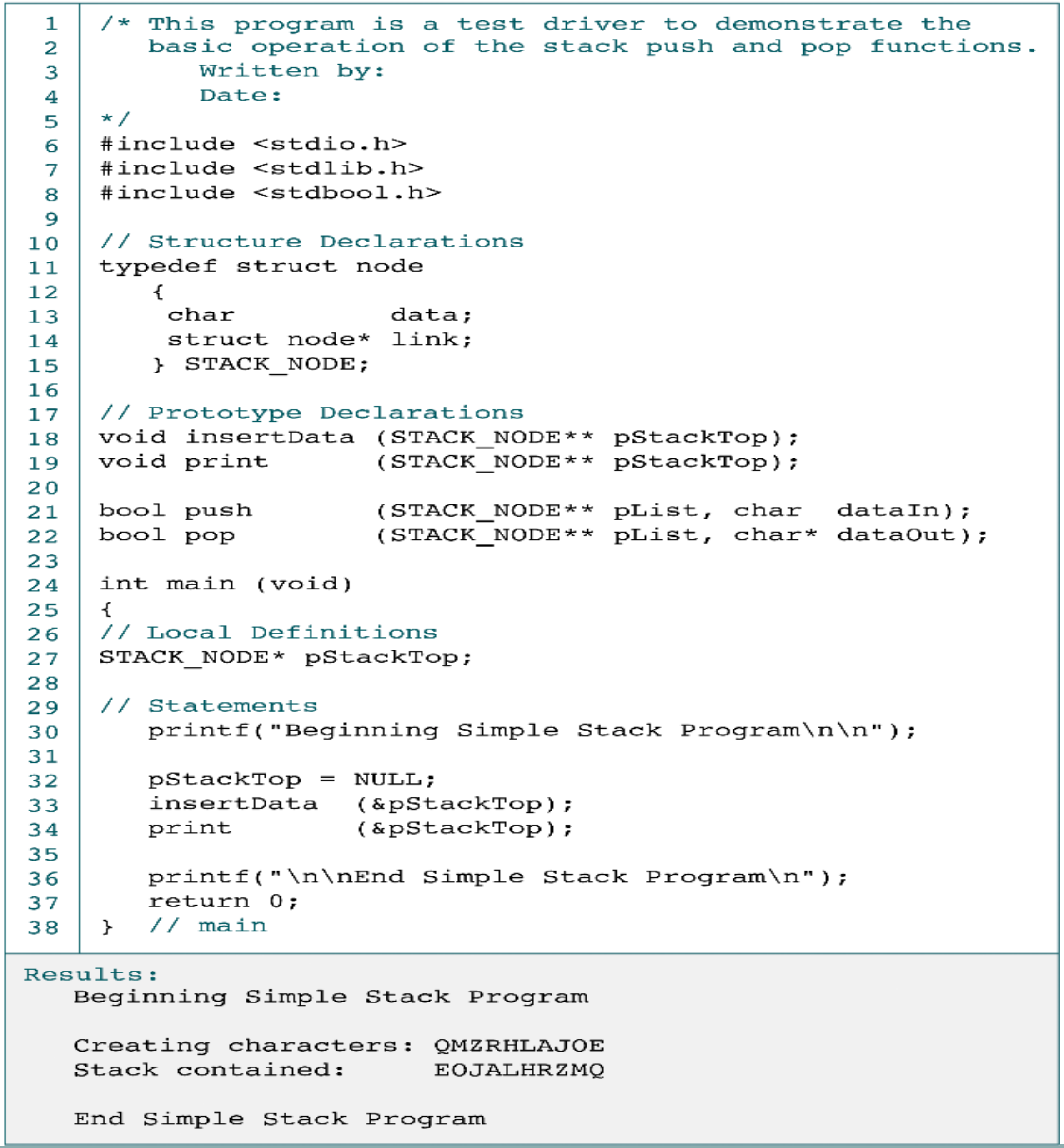


FIGURE 3-11 Design for Basic Stack Program

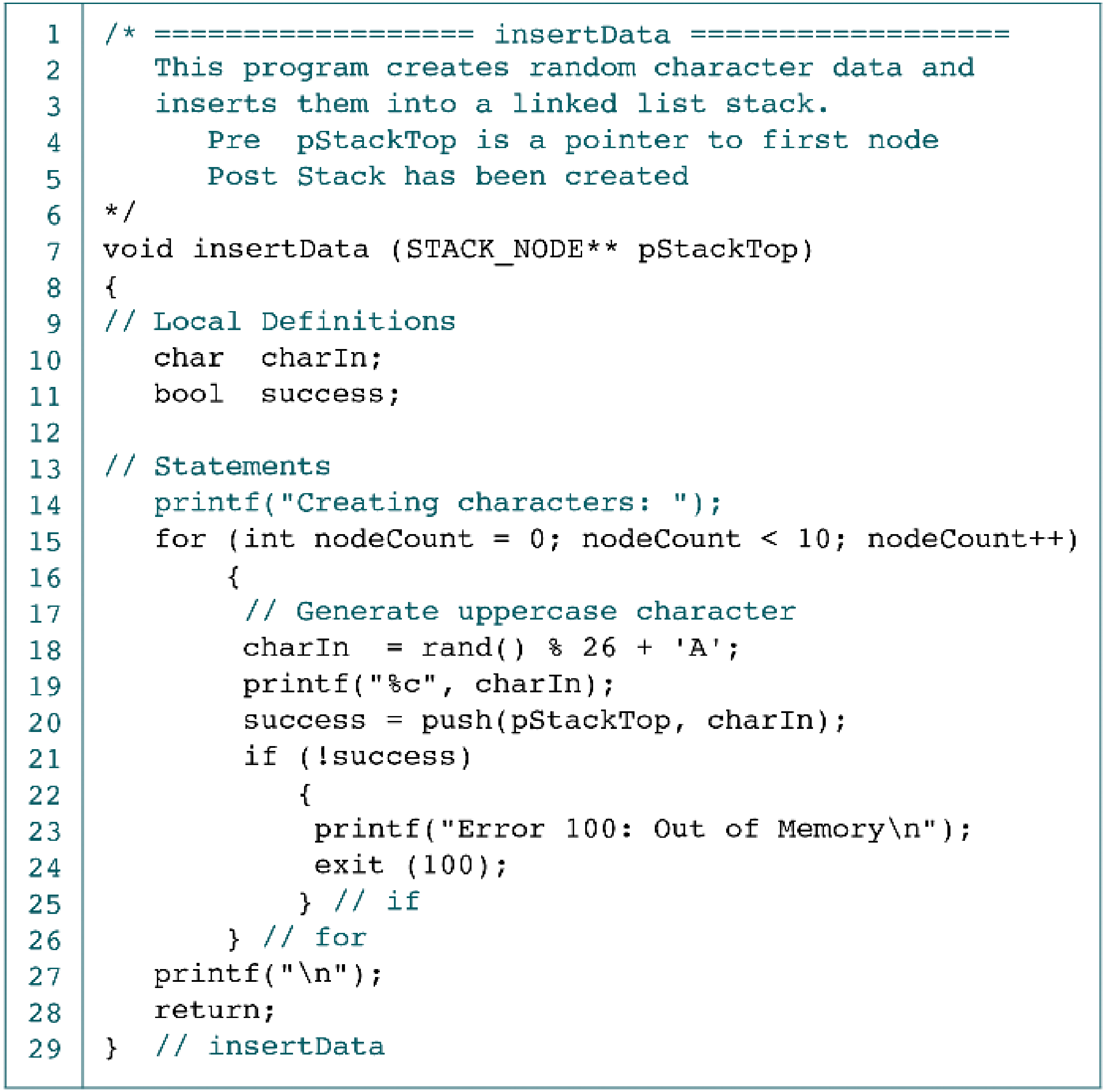
PROGRAM 3-1 Simple Stack Application Program



```c
/* This program is a test driver to demonstrate the
   basic operation of the stack push and pop functions.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Structure Declarations
typedef struct node
   {
    char          data;
    struct node*  link;
   } STACK_NODE;

// Prototype Declarations
void insertData (STACK_NODE** pStackTop);
void print      (STACK_NODE** pStackTop);

bool push       (STACK_NODE** pList, char  dataIn);
bool pop        (STACK_NODE** pList, char* dataOut);

int main (void)
{
// Local Definitions
STACK_NODE* pStackTop;

// Statements
   printf("Beginning Simple Stack Program\n\n");

   pStackTop = NULL;
   insertData  (&pStackTop);
   print       (&pStackTop);

   printf("\n\nEnd Simple Stack Program\n");
   return 0;
}  // main
```

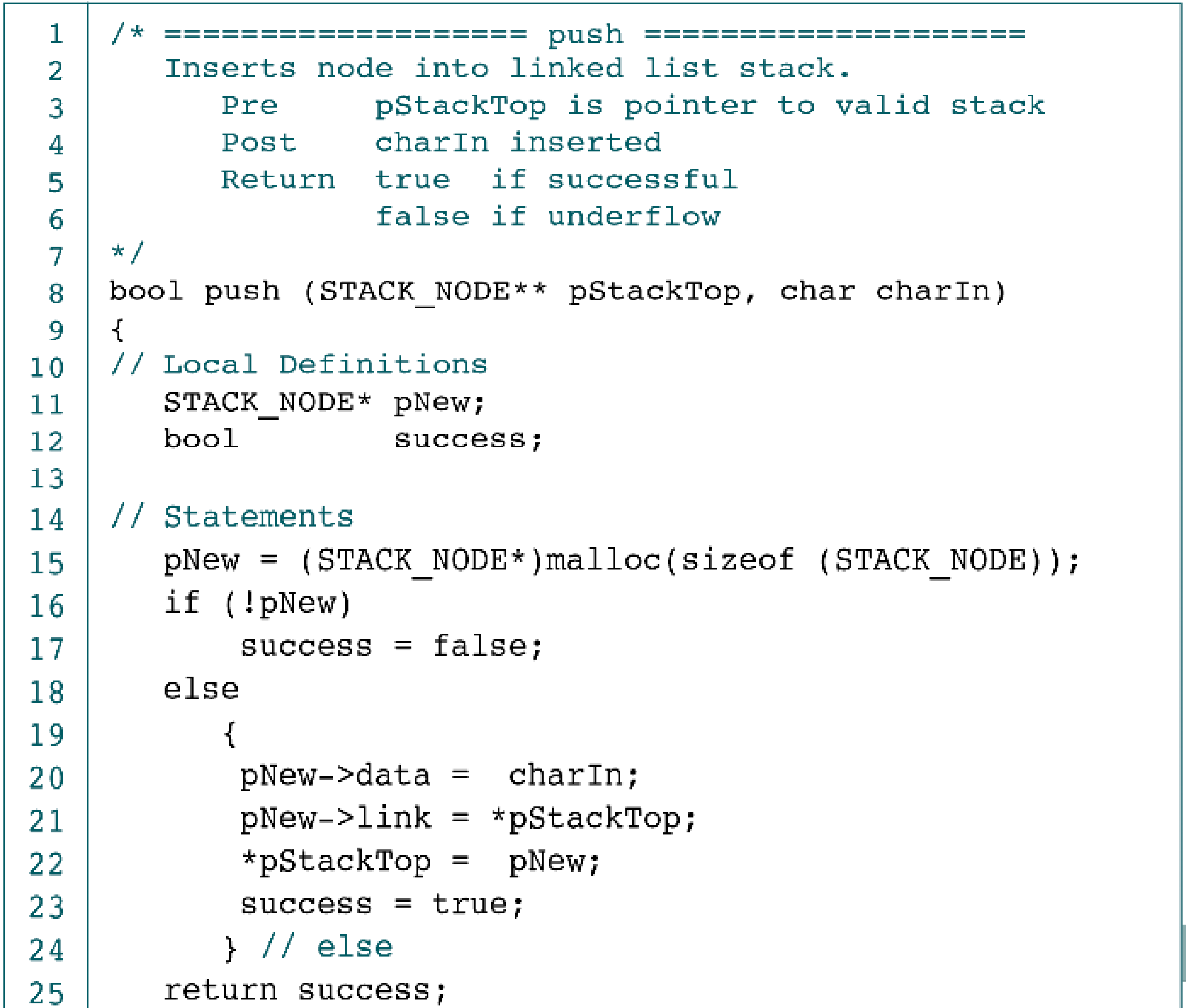
```
Results:
   Beginning Simple Stack Program

   Creating characters: QMZRHLAJOE
   Stack contained:     EOJALHRZMQ

   End Simple Stack Program
```

PROGRAM 3-2 Insert Data

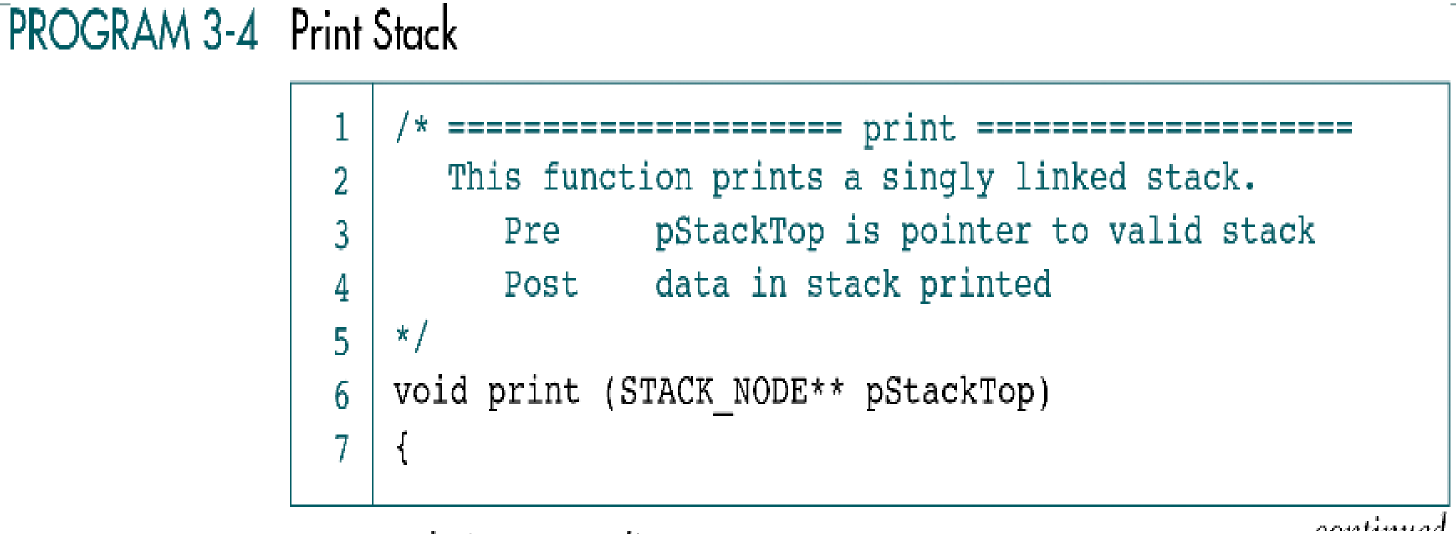


```c
/* ================== insertData ==================
   This program creates random character data and
   inserts them into a linked list stack.
      Pre  pStackTop is a pointer to first node
      Post Stack has been created
*/
void insertData (STACK_NODE** pStackTop)
{
// Local Definitions
   char   charIn;
   bool   success;

// Statements
   printf("Creating characters: ");
   for (int nodeCount = 0; nodeCount < 10; nodeCount++)
       {
        // Generate uppercase character
        charIn  = rand() % 26 + 'A';
        printf("%c", charIn);
        success = push(pStackTop, charIn);
        if (!success)
           {
            printf("Error 100: Out of Memory\n");
            exit (100);
           } // if
       } // for
   printf("\n");
   return;
}  // insertData
```

PROGRAM 3-3 Push Stack



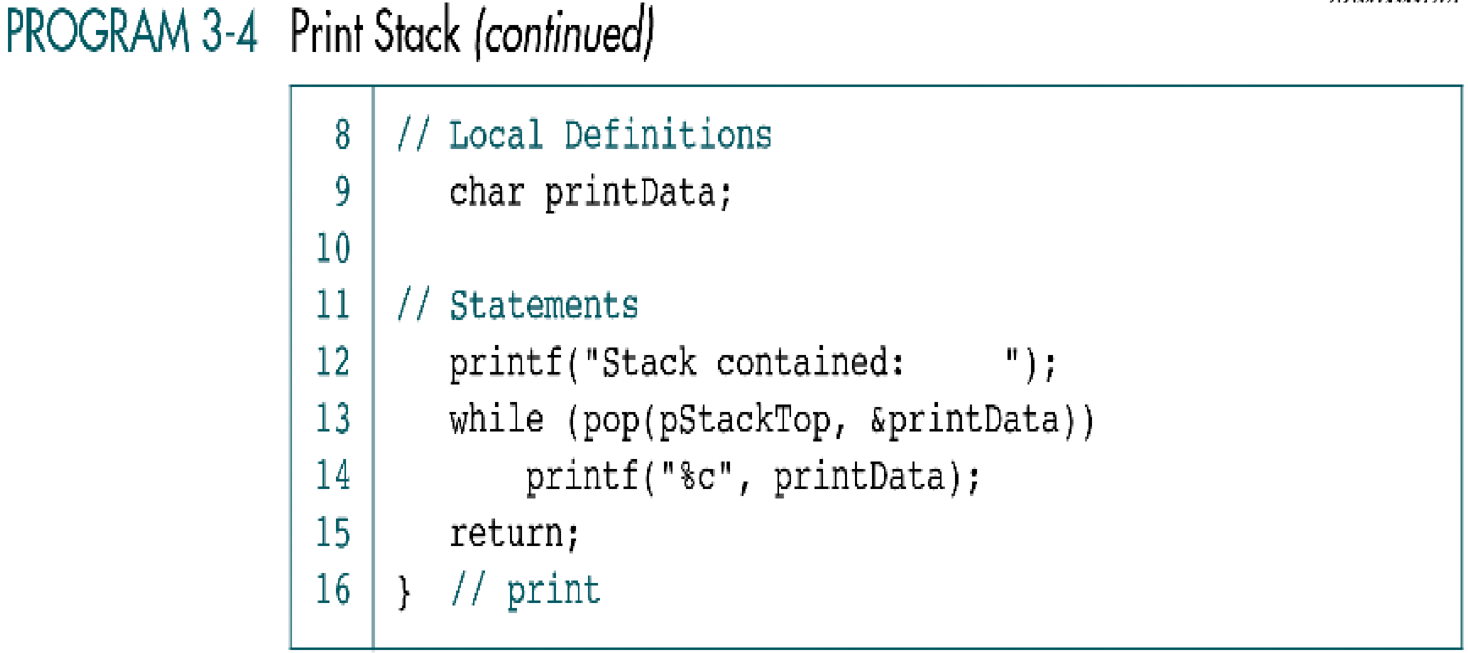
```
/* =================== push ====================
   Inserts node into linked list stack.
      Pre     pStackTop is pointer to valid stack
      Post    charIn inserted
      Return  true  if successful
              false if underflow
*/
bool push (STACK_NODE** pStackTop, char charIn)
{
// Local Definitions
   STACK_NODE* pNew;
   bool        success;

// Statements
   pNew = (STACK_NODE*)malloc(sizeof (STACK_NODE));
   if (!pNew)
       success = false;
   else
      {
       pNew->data =  charIn;
       pNew->link = *pStackTop;
       *pStackTop =  pNew;
       success = true;
      } // else
   return success;
```



PROGRAM 3-4 Print Stack

```
/* ===================== print =====================
   This function prints a singly linked stack.
      Pre      pStackTop is pointer to valid stack
      Post     data in stack printed
*/
void print (STACK_NODE** pStackTop)
{
```

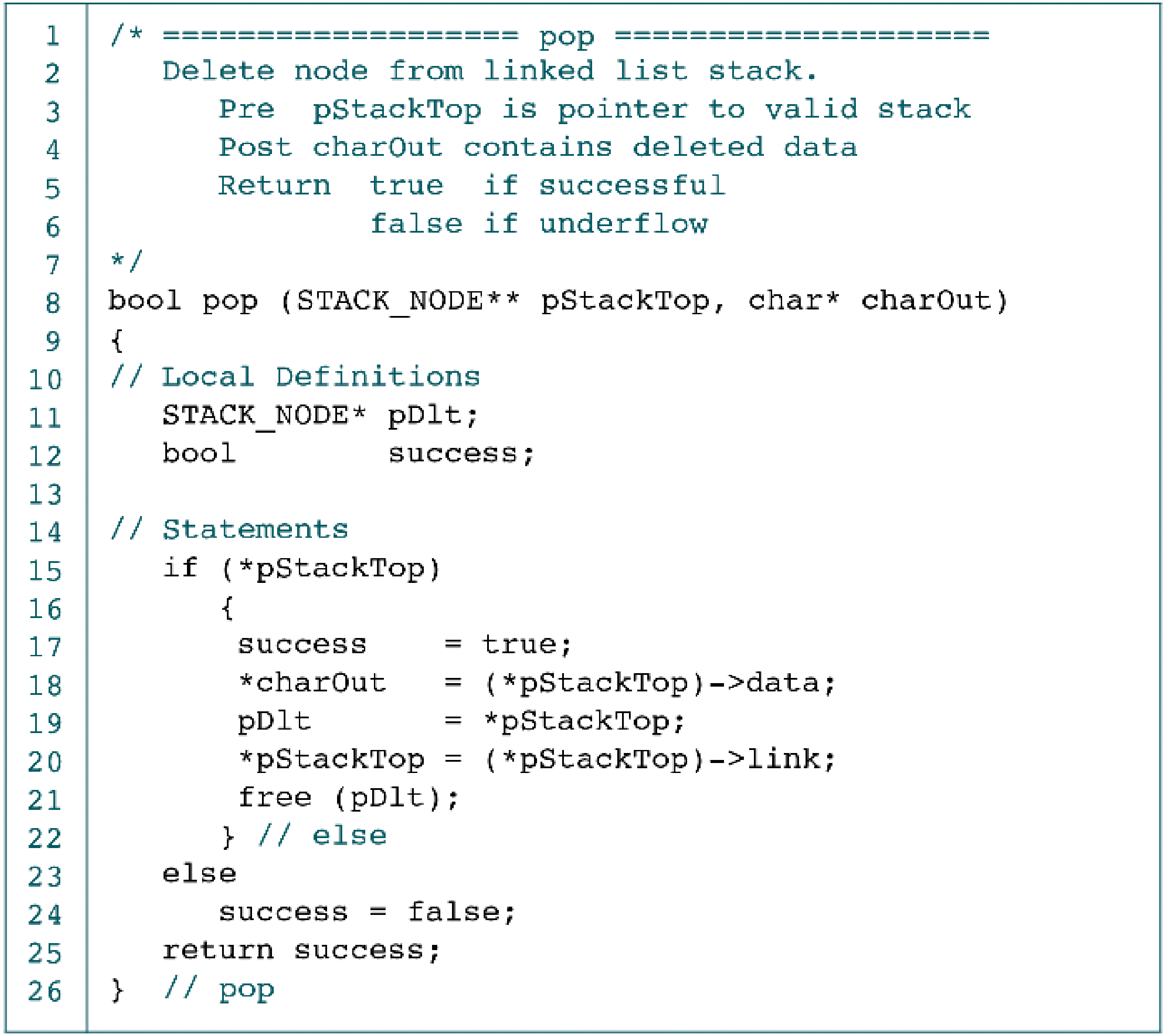
continued



PROGRAM 3-4 Print Stack *(continued)*

```
// Local Definitions
   char printData;

// Statements
   printf("Stack contained:     ");
   while (pop(pStackTop, &printData))
       printf("%c", printData);
   return;
}  // print
```

PROGRAM 3-5 Pop Stack



```
/* =================== pop ====================
   Delete node from linked list stack.
      Pre  pStackTop is pointer to valid stack
      Post charOut contains deleted data
      Return  true  if successful
              false if underflow
*/
bool pop (STACK_NODE** pStackTop, char* charOut)
{
// Local Definitions
   STACK_NODE* pDlt;
   bool        success;

// Statements
   if (*pStackTop)
      {
       success    = true;
       *charOut   = (*pStackTop)->data;
       pDlt       = *pStackTop;
       *pStackTop = (*pStackTop)->link;
       free (pDlt);
      } // else
   else
      success = false;
   return success;
}  // pop
```

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 Implemenation**

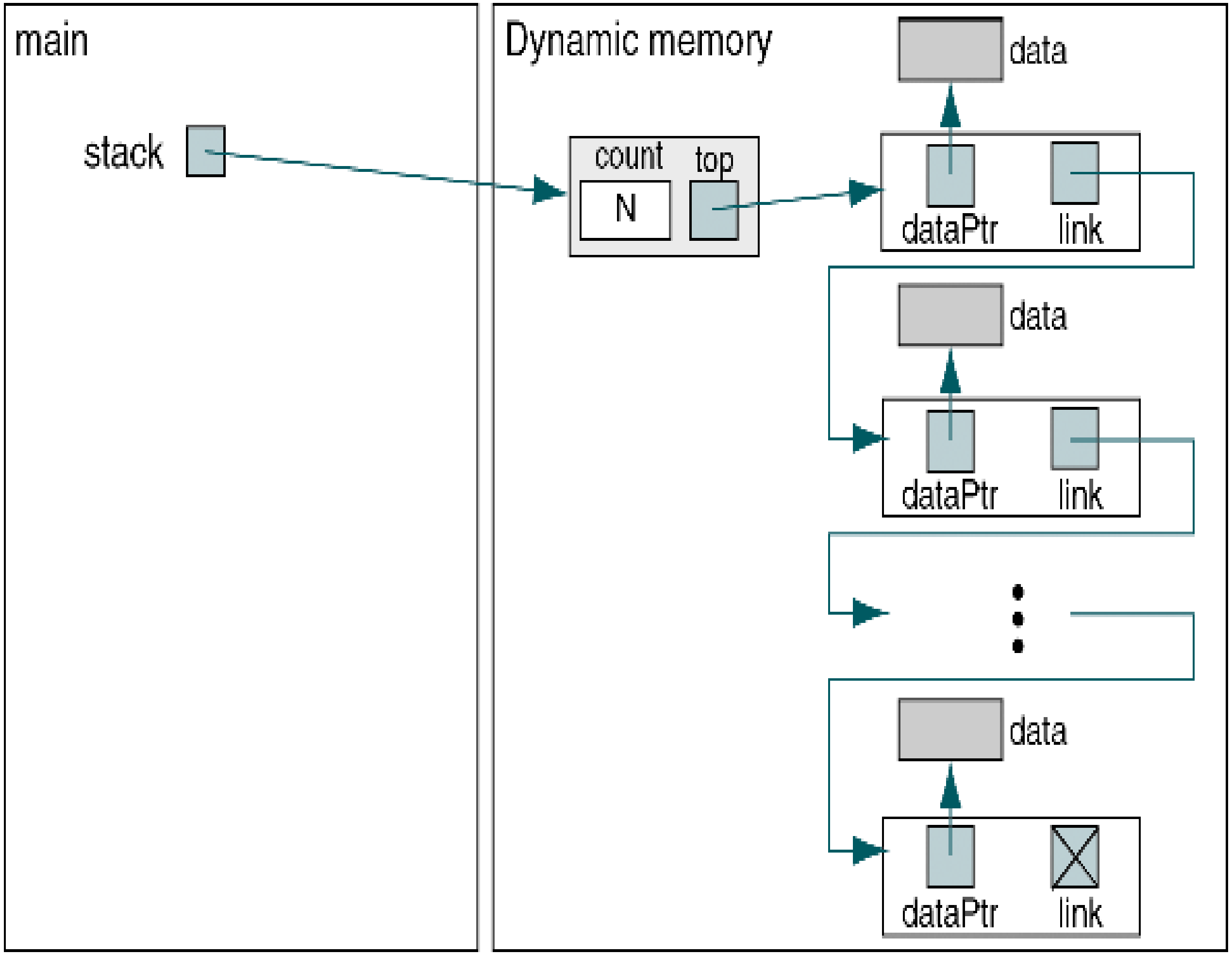
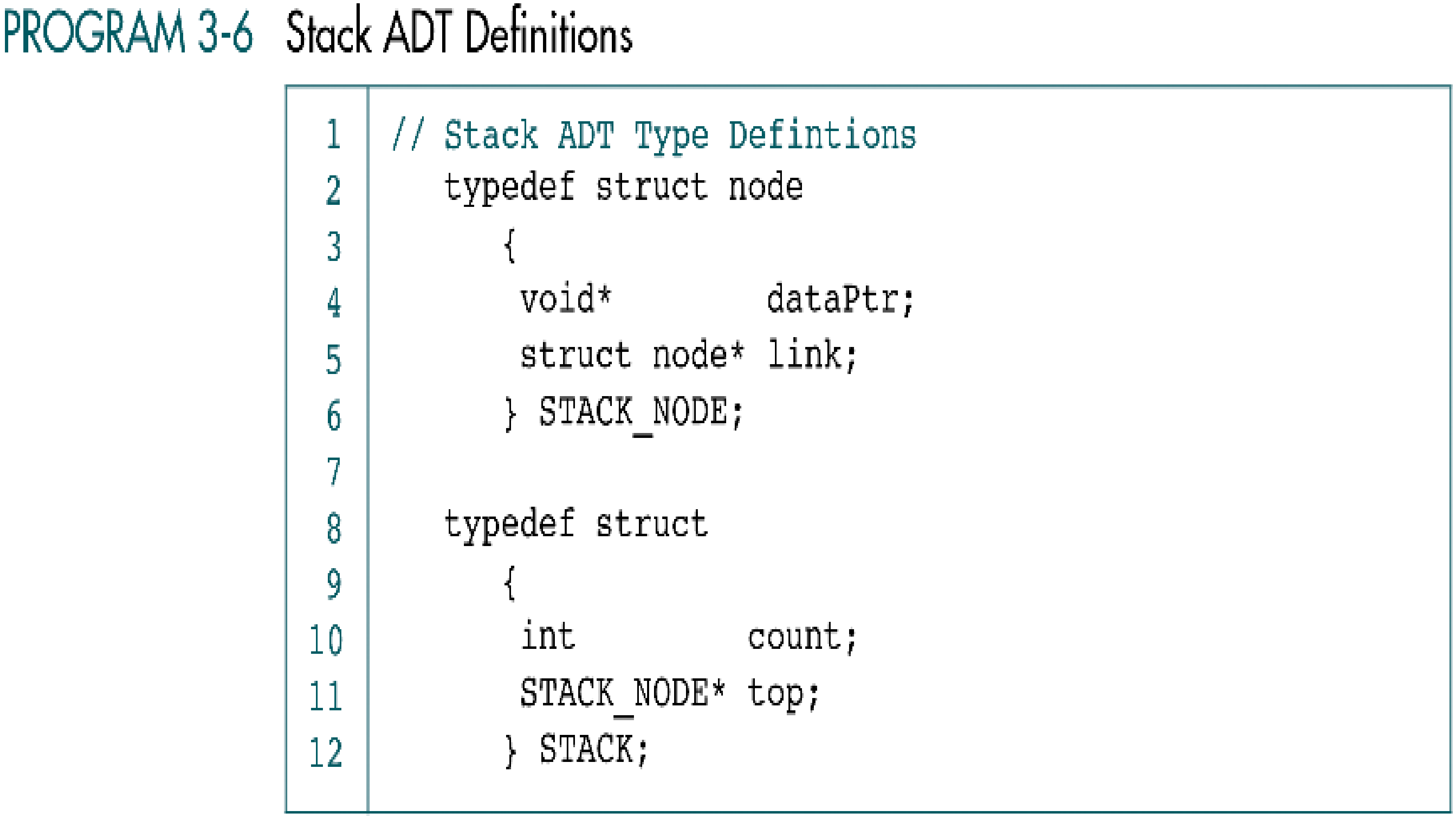
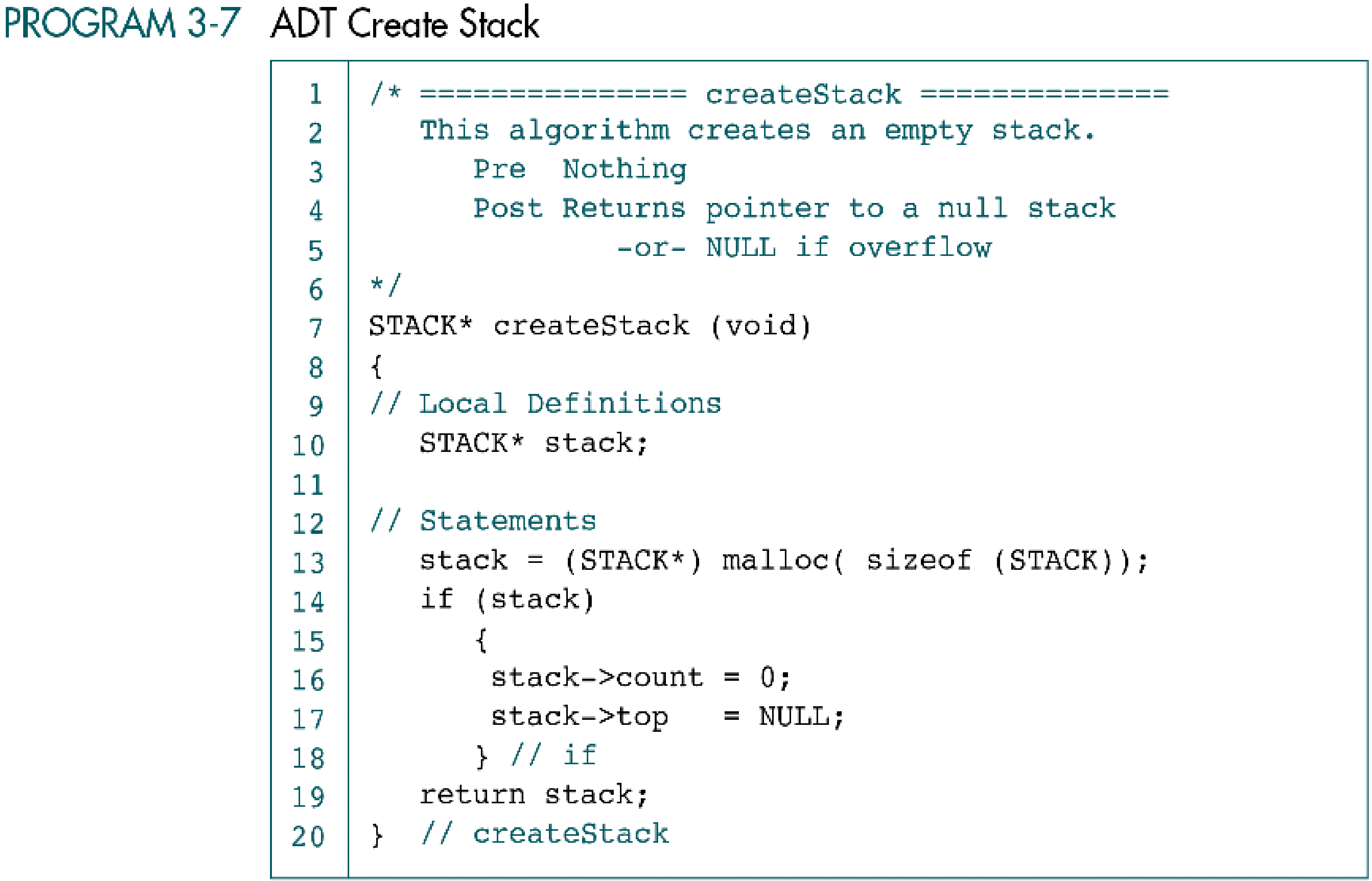


FIGURE 3-12 Stack ADT Structural Concepts



PROGRAM 3-6 Stack ADT Definitions

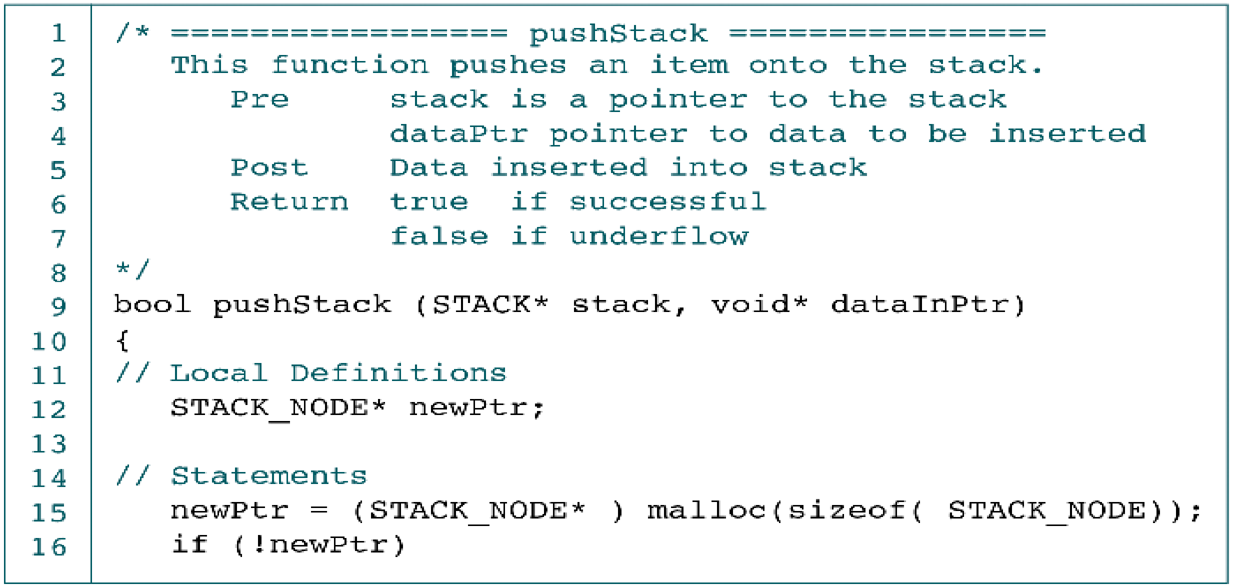
```
// Stack ADT Type Defintions
   typedef struct node
      {
       void*        dataPtr;
       struct node* link;
      } STACK_NODE;

   typedef struct
      {
       int         count;
       STACK_NODE* top;
      } STACK;
```



PROGRAM 3-7 ADT Create Stack

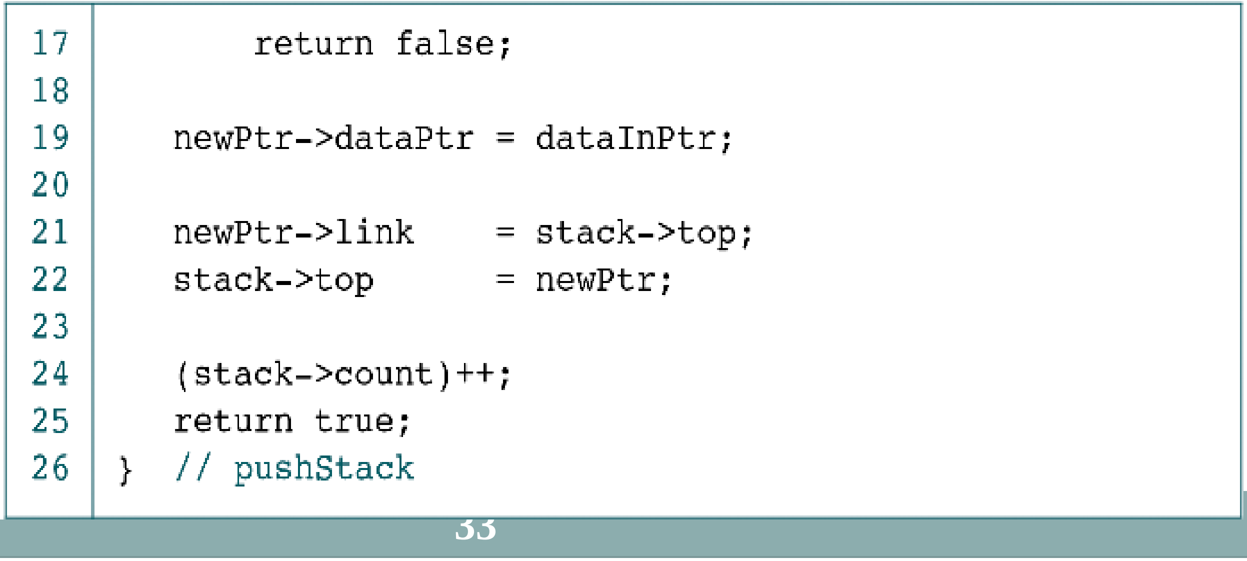
```
/* =============== createStack ==============
   This algorithm creates an empty stack.
      Pre  Nothing
      Post Returns pointer to a null stack
              -or- NULL if overflow
*/
STACK* createStack (void)
{
// Local Definitions
   STACK* stack;

// Statements
   stack = (STACK*) malloc( sizeof (STACK));
   if (stack)
      {
       stack->count = 0;
       stack->top   = NULL;
      } // if
   return stack;
}  // createStack
```

PROGRAM 3-8 Push Stack



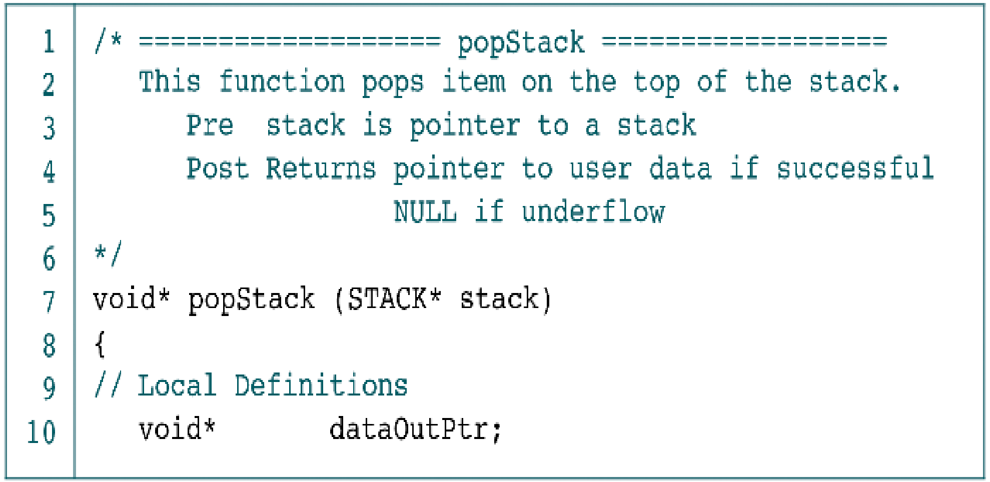
```
/* ================= pushStack =================
   This function pushes an item onto the stack.
      Pre      stack is a pointer to the stack
               dataPtr pointer to data to be inserted
      Post     Data inserted into stack
      Return   true  if successful
               false if underflow
*/
bool pushStack (STACK* stack, void* dataInPtr)
{
// Local Definitions
   STACK_NODE* newPtr;

// Statements
   newPtr = (STACK_NODE* ) malloc(sizeof( STACK_NODE));
   if (!newPtr)
```

PROGRAM 3-8 Push Stack *(continued)*



```
       return false;

   newPtr->dataPtr = dataInPtr;

   newPtr->link    = stack->top;
   stack->top      = newPtr;

   (stack->count)++;
   return true;
}  // pushStack
```

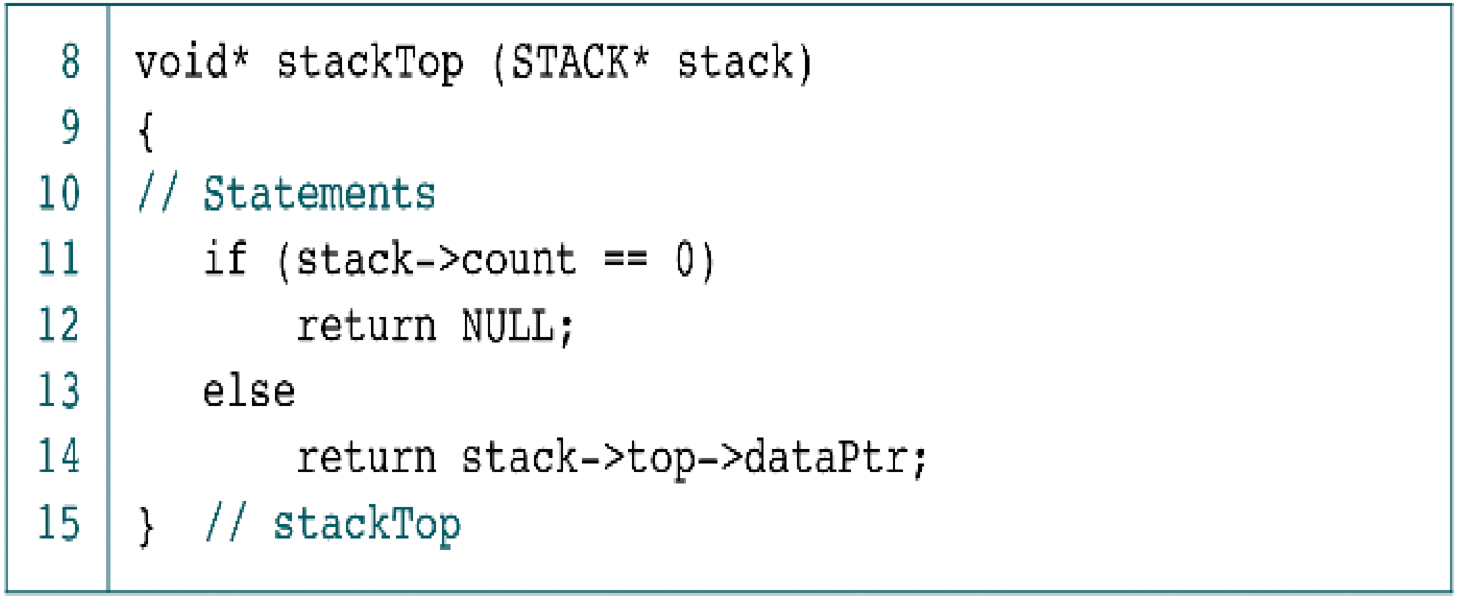
PROGRAM 3-9 ADT Pop Stack



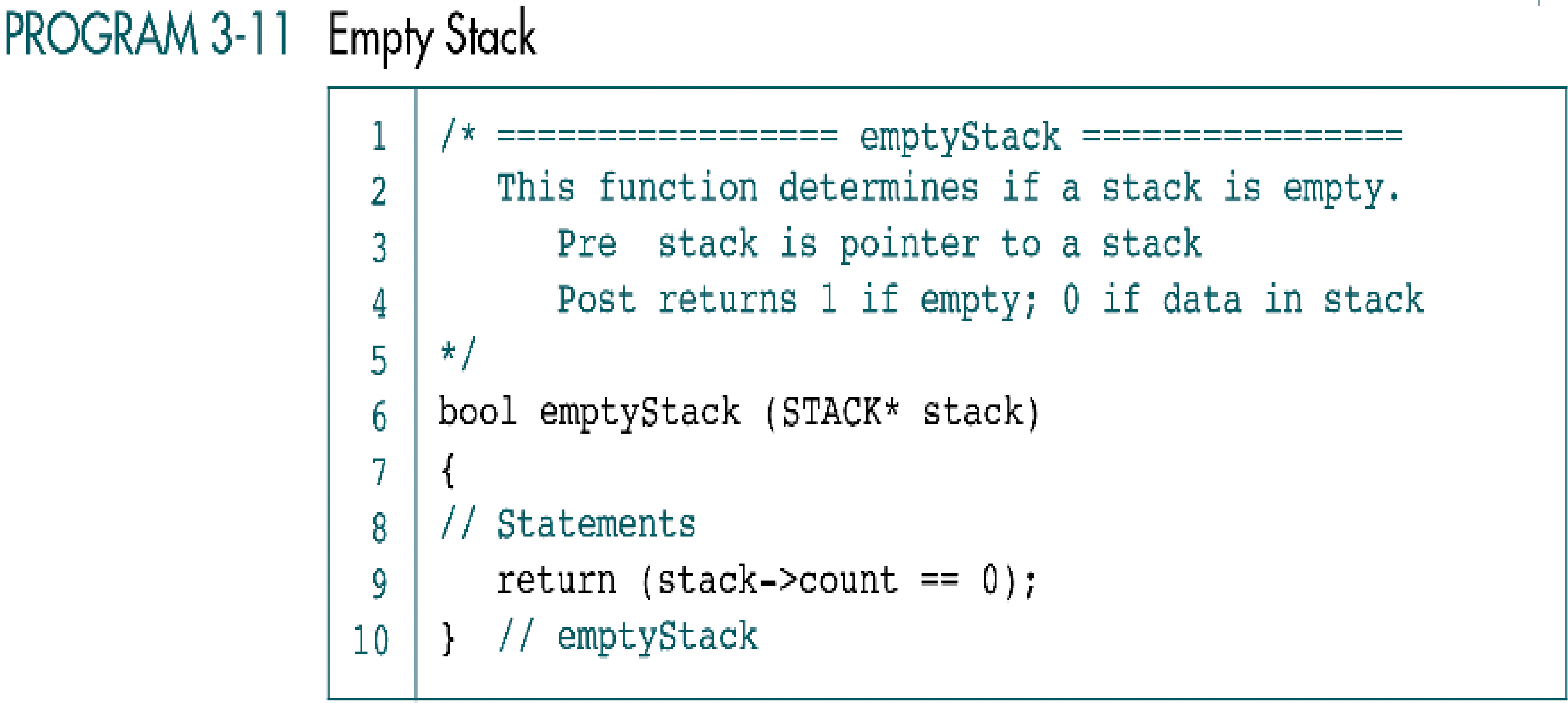
```
/* =================== popStack ==================
   This function pops item on the top of the stack.
      Pre  stack is pointer to a stack
      Post Returns pointer to user data if successful
                   NULL if underflow
*/
void* popStack (STACK* stack)
{
// Local Definitions
   void*       dataOutPtr;
```

continued

PROGRAM 3-10 Retrieve Stack Top *(continued)*



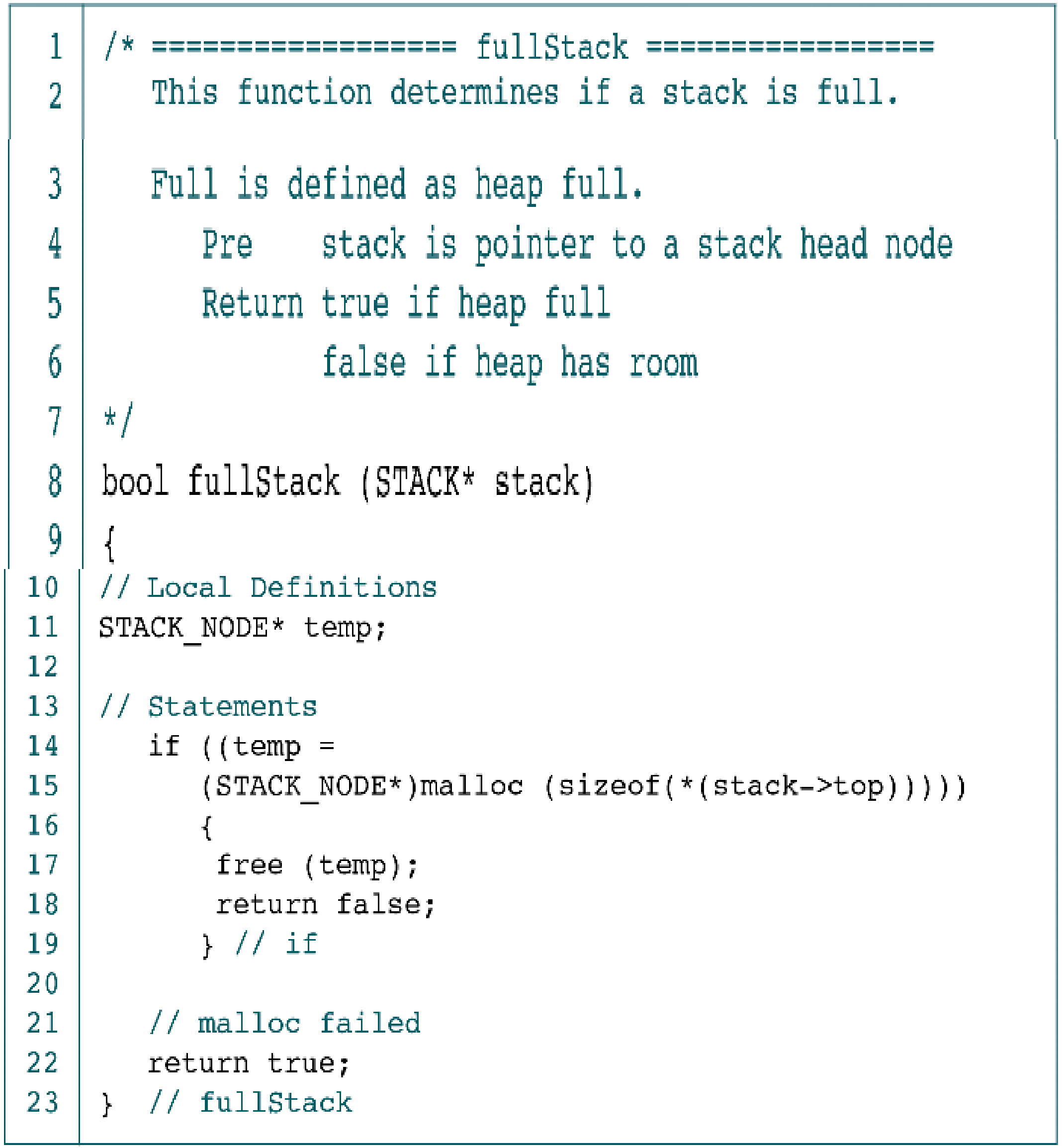
```
void* stackTop (STACK* stack)
{
// Statements
   if (stack->count == 0)
       return NULL;
   else
       return stack->top->dataPtr;
}  // stackTop
```



PROGRAM 3-11 Empty Stack

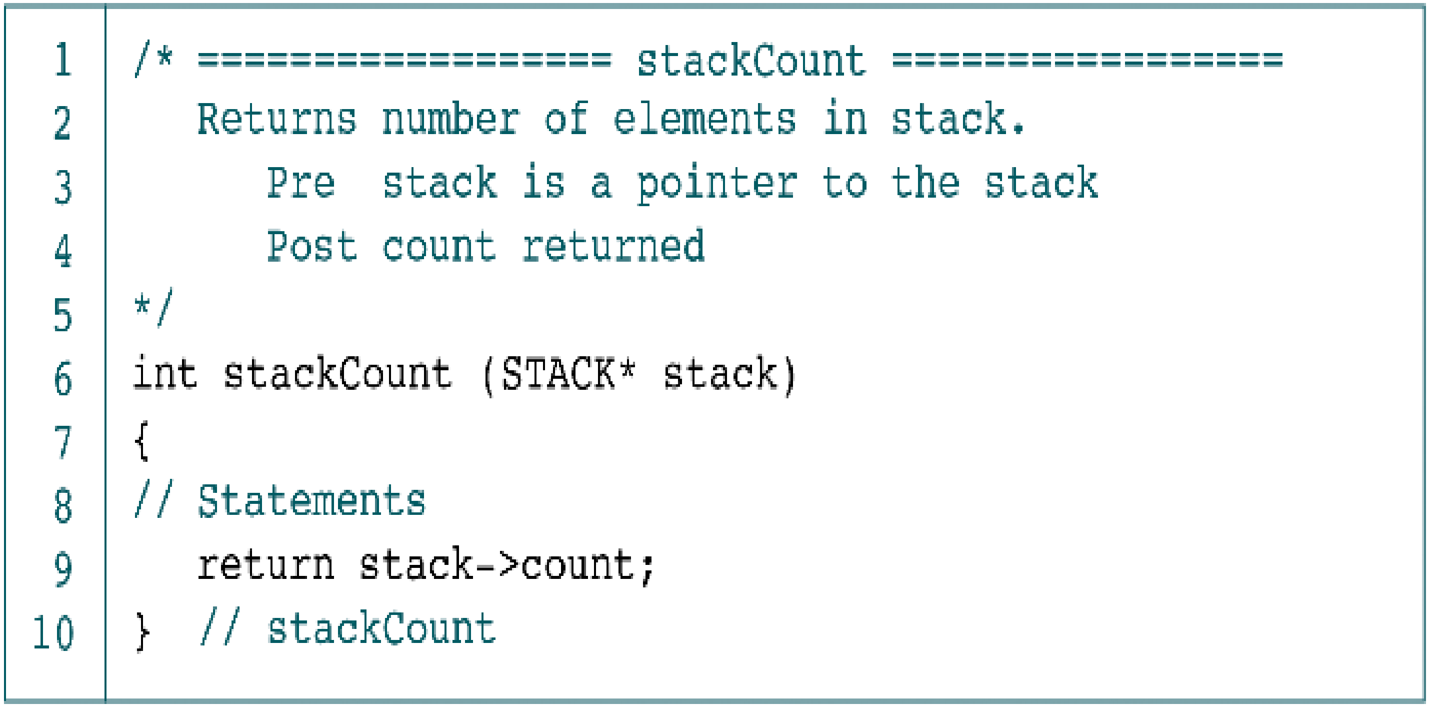
```
/* ================= emptyStack ================
   This function determines if a stack is empty.
      Pre  stack is pointer to a stack
      Post returns 1 if empty; 0 if data in stack
*/
bool emptyStack (STACK* stack)
{
// Statements
   return (stack->count == 0);
}  // emptyStack
```

PROGRAM 3-12 Full Stack



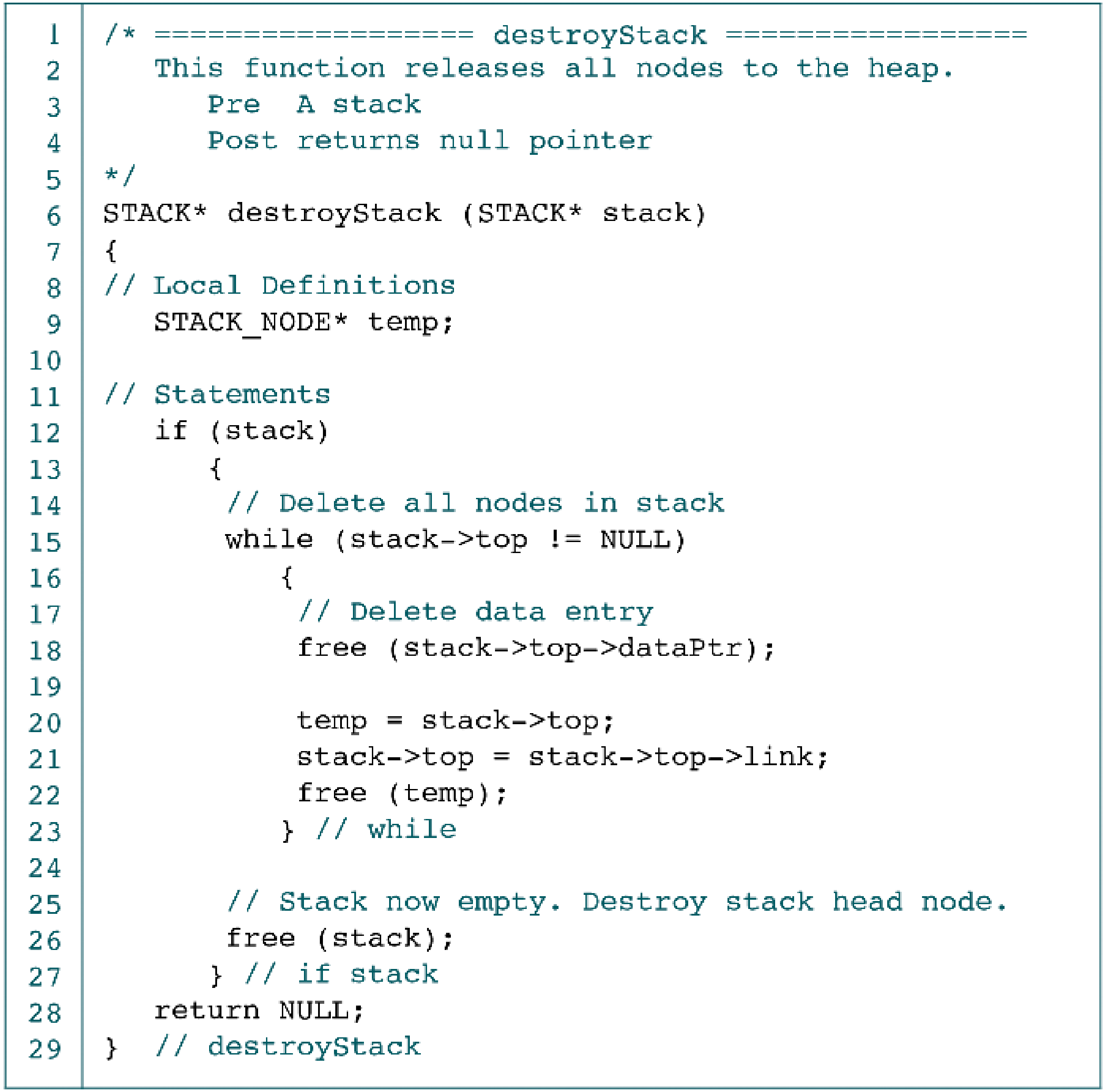
```
/* ================= fullStack =================
   This function determines if a stack is full.
   Full is defined as heap full.
      Pre    stack is pointer to a stack head node
      Return true if heap full
             false if heap has room
*/
bool fullStack (STACK* stack)
{
// Local Definitions
STACK_NODE* temp;

// Statements
   if ((temp =
      (STACK_NODE*)malloc (sizeof(*(stack->top)))))
      {
       free (temp);
       return false;
      } // if

   // malloc failed
   return true;
}  // fullStack
```

PROGRAM 3-13 Stack Count



```
/* ================== stackCount ==================
   Returns number of elements in stack.
      Pre  stack is a pointer to the stack
      Post count returned
*/
int stackCount (STACK* stack)
{
// Statements
   return stack->count;
}  // stackCount
```

PROGRAM 3-14 Destroy Stack



```c
/* ================== destroyStack ==================
   This function releases all nodes to the heap.
      Pre  A stack
      Post returns null pointer
*/
STACK* destroyStack (STACK* stack)
{
// Local Definitions
   STACK_NODE* temp;

// Statements
   if (stack)
      {
       // Delete all nodes in stack
       while (stack->top != NULL)
          {
           // Delete data entry
           free (stack->top->dataPtr);

           temp = stack->top;
           stack->top = stack->top->link;
           free (temp);
          } // while

       // Stack now empty. Destroy stack head node.
       free (stack);
      } // if stack
   return NULL;
}  // destroyStack
```