COURSE TECHNOLOGY
CENGAGE Learning™

Data Structures Using C++ 2E

Chapter 7
Stacks

Objectives

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Discover stack applications
- Learn how to use a stack to remove recursion

Stacks

- Data structure
  - Elements added, removed from one end only
  - Last In First Out (LIFO)

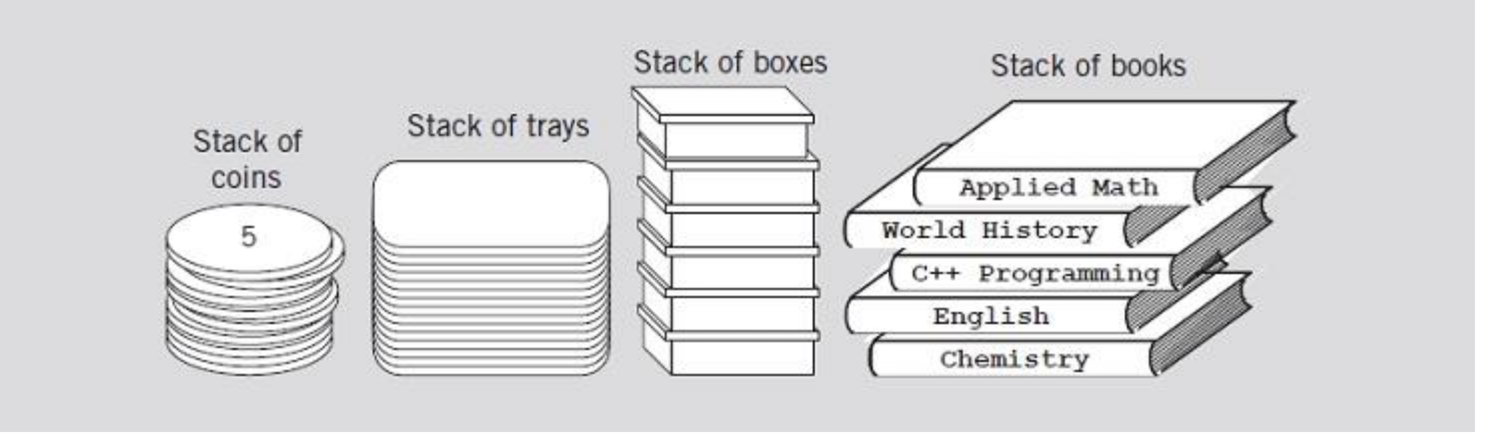


FIGURE 7-1 Various examples of stacks

Stacks (cont'd.)

- `push` operation
  - Add element onto the stack
- `top` operation
  - Retrieve top element of the stack, w/o remove top element
- `pop` operation
  - Remove top element from the stack

Stacks (cont'd.)

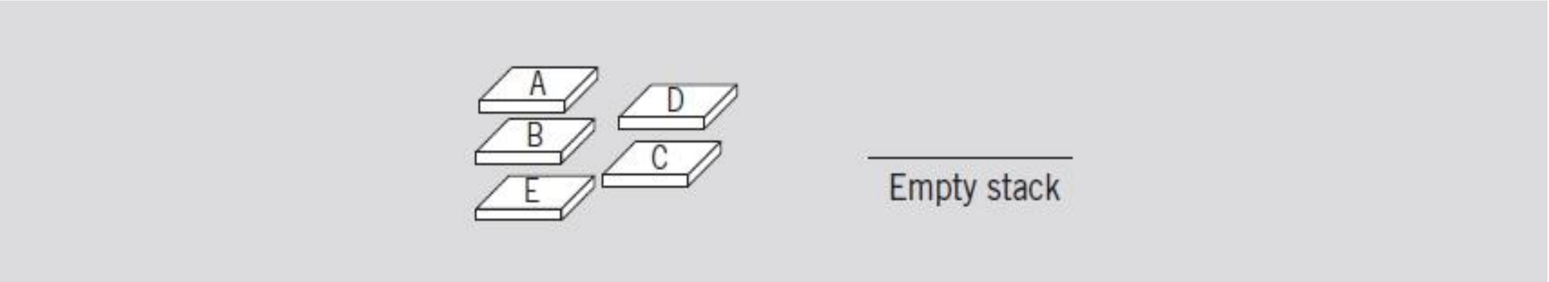


FIGURE 7-2 Empty stack

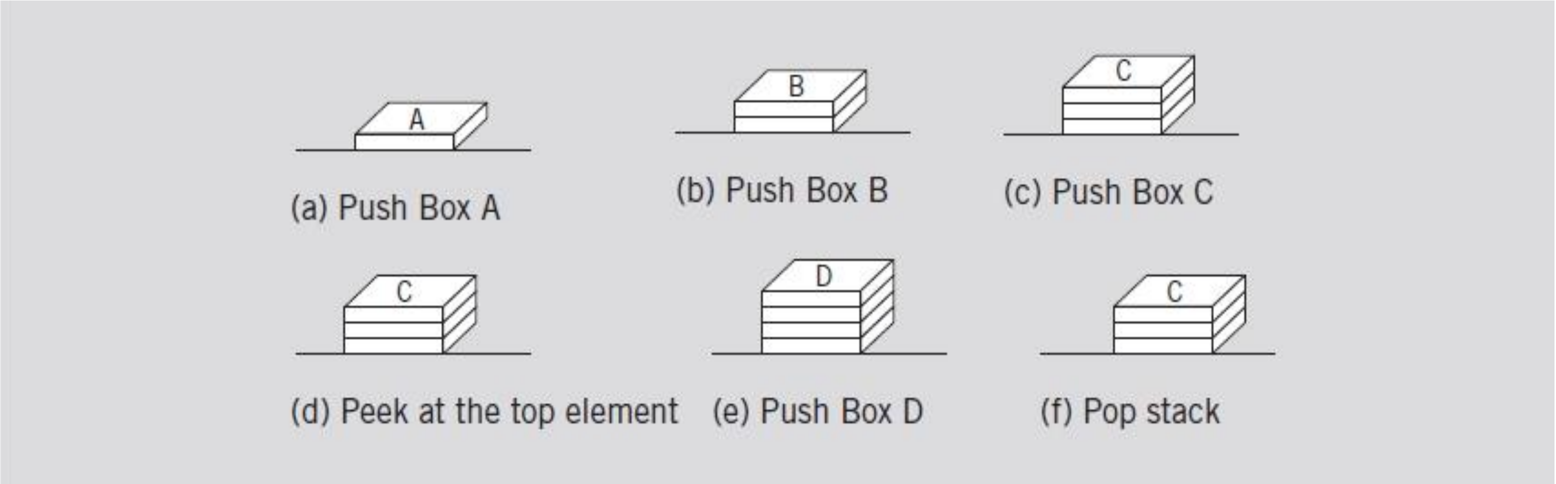


FIGURE 7-3 Stack operations

Stacks (cont'd.)

- Stack element removal
  - Occurs only if something is in the stack
- Stack element added only if room available
- `isFullStack` operation
  - Checks for full stack
- `isEmptyStack` operation
  - Checks for empty stack
- `initializeStack` operation
  - Initializes stack to an empty state

Stacks (cont'd.)

- Review code on page 398
  - Illustrates class specifying basic stack operations

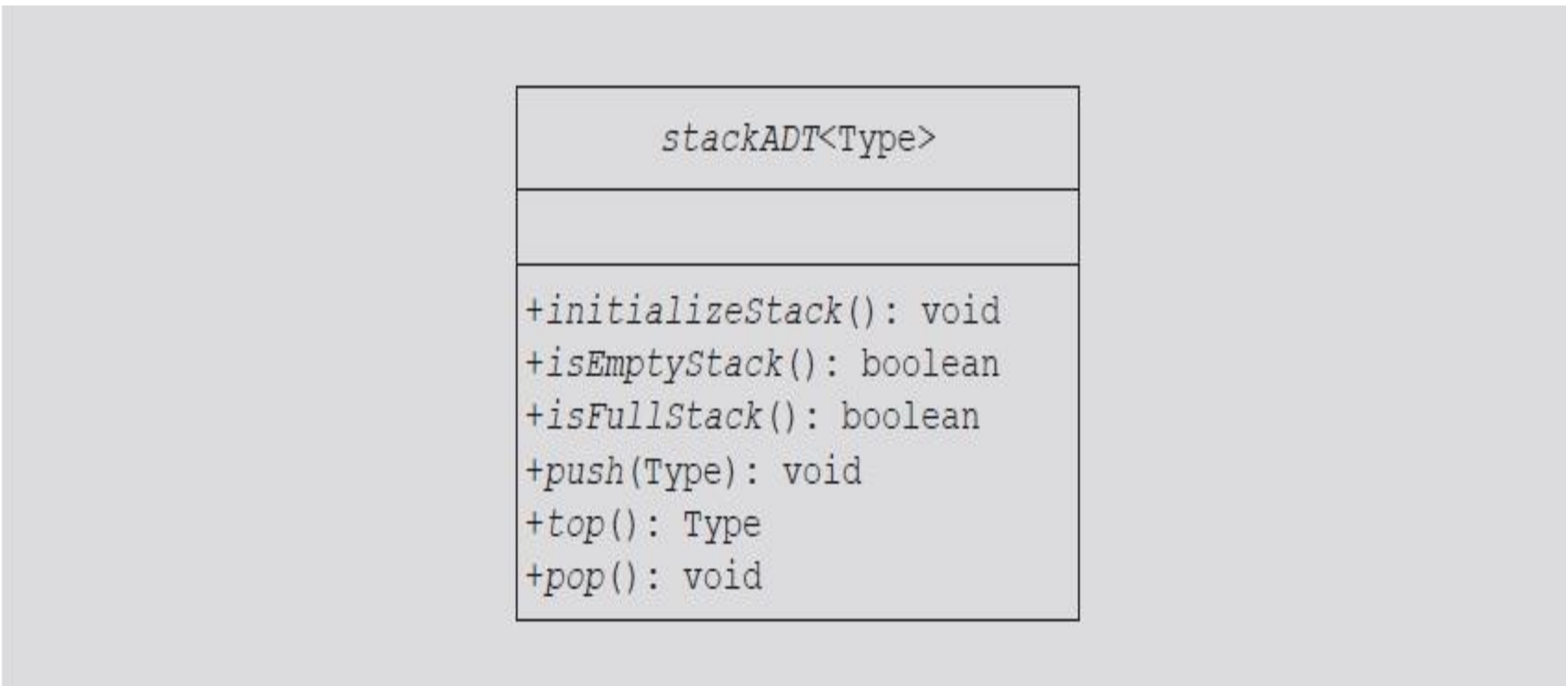


FIGURE 7-4 UML class diagram of the `class stackADT`

```cpp
template <class Type>
class stackADT
{
public:
    virtual void initializeStack() = 0;
       //Method to initialize the stack to an empty state.
       //Postcondition: Stack is empty.

    virtual bool isEmptyStack() const = 0;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty,
      //    otherwise returns false.

    virtual bool isFullStack() const = 0;
      //Function to determine whether the stack is full.
      //Postcondition: Returns true if the stack is full,
      //    otherwise returns false.

    virtual void push(const Type& newItem) = 0;
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem is added
      //    to the top of the stack.

    virtual Type top() const = 0;
      //Function to return the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: If the stack is empty, the program
      //    terminates; otherwise, the top element of the stack
      //    is returned.

    virtual void pop() = 0;
      //Function to remove the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: The stack is changed and the top element
      //    is removed from the stack.
};
```

Implementation of Stacks as Arrays

- (fixed size) array is allocated by constructor
- First stack element
  - Put in first array slot
- Second stack element
  - Put in second array slot, and so on
- Top of stack
  - Index of last element added to stack
- Stack element accessed only through the top
  - Problem: array is a random access data structure
  - Solution: use another variable (`stackTop`)
    - Keeps track of the top position of the array

Implementation of Stacks as Arrays (cont'd.)

- Review code on page 400
  - Illustrates basic operations on a stack as an array

```
stackType<Type>
-----------------------------------------------
-maxStackSize: int
-stackTop: int
-*list: Type
-----------------------------------------------
+operator=(const stackType<Type>&): const stackType<Type>&
+initializeStack(): void
+isEmptyStack() const: bool
+isFullStack() const: bool
+push(const Type&): void
+top() const: Type
+pop(): void
-copyStack(const stackType<Type>&): void
+stackType(int = 100)
+stackType(const stackType<Type>&)
+~stackType()
```

FIGURE 7-5 UML class diagram of the `class stackType`

Implementation of Stacks as Arrays (cont'd.)

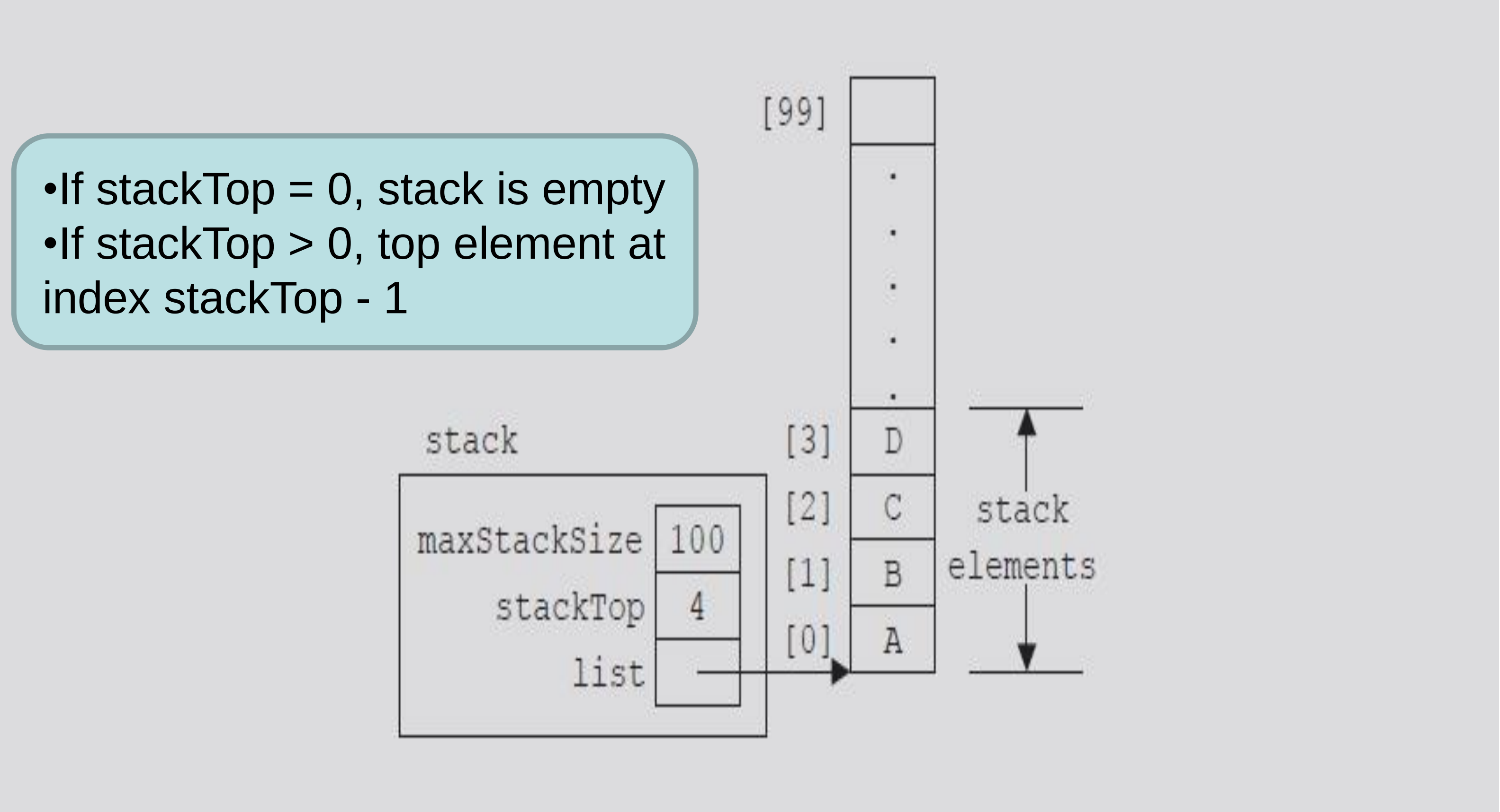


FIGURE 7-6 Example of a stack

Initialize Stack

- Value of `stackTop` if stack empty
  - Set `stackTop` to zero to initialize the stack
- Definition of function `initializeStack`

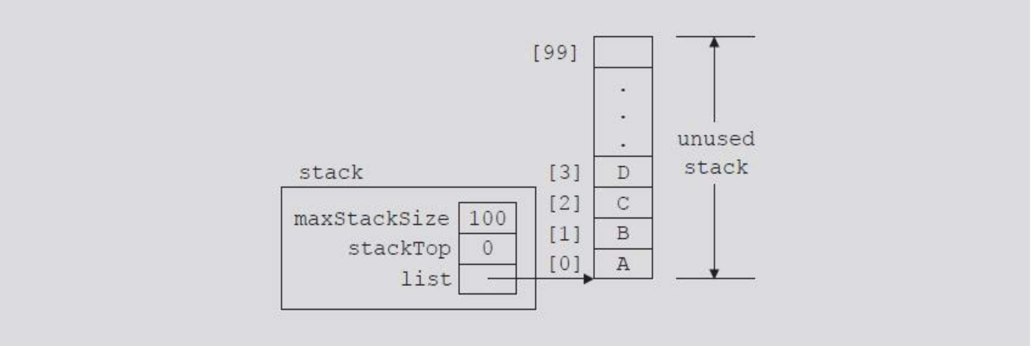


FIGURE 7-7 Empty stack

```
template <class Type>
void stackType<Type>::initializeStack()
{
    stackTop = 0;
}//end initializeStack
```

Empty Stack

- Value of `stackTop` indicates if stack empty
  - If `stackTop` = zero: stack empty
  - Otherwise: stack not empty
- Definition of function `isEmptyStack`

```
template <class Type>
bool stackType<Type>::isEmptyStack() const
{
    return(stackTop == 0);
}//end isEmptyStack
```

Full Stack

- Stack full
  - If `stackTop` is equal to `maxStackSize`
- Definition of function `isFullStack`

```
template <class Type>
bool stackType<Type>::isFullStack() const
{
    return(stackTop == maxStackSize);
} //end isFullStack
```

Push

- Two-step process
  - Store `newItem` in array component indicated by `stackTop`
  - Increment `stackTop`

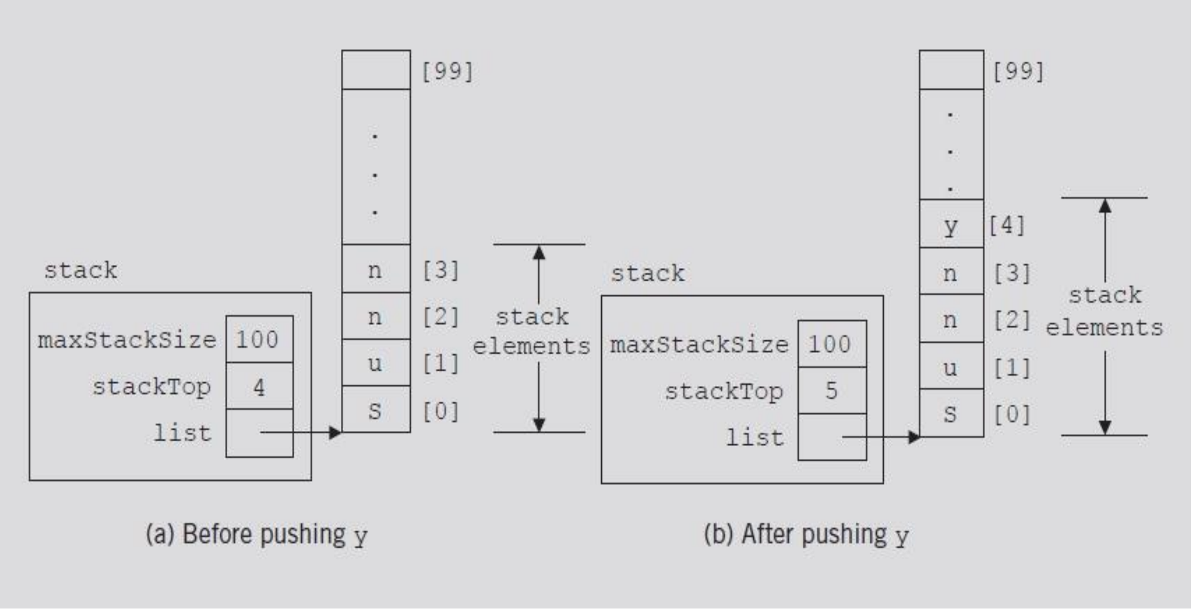


FIGURE 7-8 Stack before and after the `push` operation

Push (cont'd.)

- Definition of `push` operation

```
template <class Type>
void stackType<Type>::push(const Type& newItem)
{
    if (!isFullStack())
    {
        list[stackTop] = newItem;  //add newItem at the top
        stackTop++; //increment stackTop
    }
    else
        cout << "Cannot add to a full stack." << endl;
}//end push
```

Return the Top Element

- Definition of `top` operation

```
template <class Type>
Type stackType<Type>::top() const
{
    assert(stackTop != 0);  //if stack is empty, terminate the
                            //program
    return list[stackTop - 1]; //return the element of the stack
                               //indicated by stackTop - 1
}//end top
```

Pop

- Remove (pop) element from stack
  - Decrement `stackTop` by one

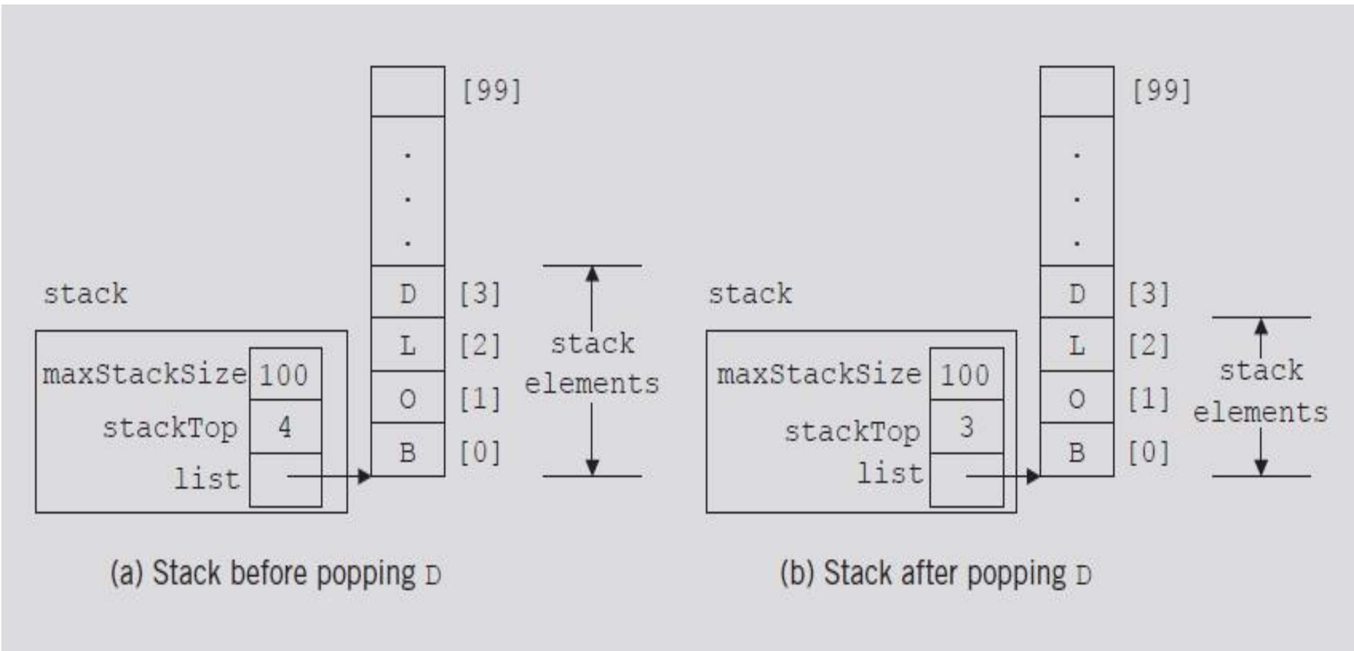


FIGURE 7-9 Stack before and after the `pop` operation

Pop (cont'd.)

- Definition of `pop` operation
- Underflow
  - Removing an item from an empty stack
    - Check within `pop` operation (see below)
    - Check before calling function `pop`

Need "delete" the popped element?

```
template <class Type>
void stackType<Type>::pop()
{
    if (!isEmptyStack())
        stackTop--;      //decrement stackTop
    else
        cout << "Cannot remove from an empty stack." << endl;
}//end pop
```

Copy Stack

- Definition of function `copyStack`

O(n)

What if list is NULL?

```
template <class Type>
void stackType<Type>::copyStack(const stackType<Type>& otherStack)
{
    delete [] list;
    maxStackSize = otherStack.maxStackSize;
    stackTop = otherStack.stackTop;
    list = new Type[maxStackSize];

        //copy otherStack into this stack
    for (int j = 0; j < stackTop; j++)
        list[j] = otherStack.list[j];
} //end copyStack
```

Constructor and Destructor

```
template <class Type>
stackType<Type>::stackType(int stackSize)
{
    if (stackSize <= 0)
    {
        cout << "Size of the array to hold the stack must "
             << "be positive." << endl;
        cout << "Creating an array of size 100." << endl;

        maxStackSize = 100;
    }
    else
        maxStackSize = stackSize;   //set the stack size to
                                    //the value specified by
                                    //the parameter stackSize

    stackTop = 0;                   //set stackTop to 0
    list = new Type[maxStackSize];  //create the array to
                                    //hold the stack elements
}//end constructor

template <class Type>
stackType<Type>::~stackType() //destructor
{
    delete [] list; //deallocate the memory occupied
                    //by the array
}//end destructor
```

Copy Constructor

- Definition of the copy constructor

```
template <class Type>
stackType<Type>::stackType(const stackType<Type>& otherStack)
{
    list = NULL;

    copyStack(otherStack);
}//end copy constructor
```

O(n)

Overloading the Assignment Operator (=)

- Classes with pointer member variables
  - Assignment operator must be explicitly overloaded
    - Why?
- Function definition to overload assignment operator for `class stackType`

O(n)

```
template <class Type>
const stackType<Type>& stackType<Type>::operator=
                         (const stackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
} //end operator=
```

Stack Header File

- `myStack.h`
  - Header file name containing `class stackType` definition

```
//Header file: myStack.h

#ifndef H_StackType
#define H_StackType

#include <iostream>
#include <cassert>

#include "stackADT.h"

using namespace std;

//Place the definition of the class template stackType, as given
//previously in this chapter, here.

//Place the definitions of the member functions as discussed here.
#endif
```

Stack Header File (cont'd.)

- Stack operations analysis
  - Similar to `class arrayListType` operations

TABLE 7-1 Time complexity of the operations of the `class stackType` on a stack with *n* elements

Function	Time complexity
`isEmptyStack`	$O(1)$
`isFullStack`	$O(1)$
`initializeStack`	$O(1)$
constructor	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$
`copyStack`	$O(n)$
destructor	$O(1)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

Linked Implementation of Stacks

- Disadvantage of array (linear) stack representation
  - Fixed number of elements can be pushed onto stack
- Solution
  - Use pointer variables to dynamically allocate, deallocate memory
  - Use linked list to dynamically organize data
- Value of `stackTop`: array (linear) representation
  - Indicates number of elements in the stack
    - Gives index of the array
  - Value of `stackTop` – 1
    - Points to top item in the stack
- Value of `stackTop`: linked representation
  - Locates top element in the stack
    - Gives address (memory location) of the top element of the stack

```
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```

```
template <class Type>
class linkedStackType: public stackADT<Type>
{
public:
    const linkedStackType<Type>& operator=
                              (const linkedStackType<Type>&);

    bool isEmptyStack() const;

    bool isFullStack() const;

    void initializeStack();

    void push(const Type& newItem);

    Type top() const;

    void pop();

    linkedStackType();

    linkedStackType(const linkedStackType<Type>& otherStack);

    ~linkedStackType();

private:
    nodeType<Type> *stackTop; //pointer to the stack

    void copyStack(const linkedStackType<Type>& otherStack);
};
```

Linked Implementation of Stacks (cont'd.)

- Example 7-2
  - Stack: object of type `linkedStackType`

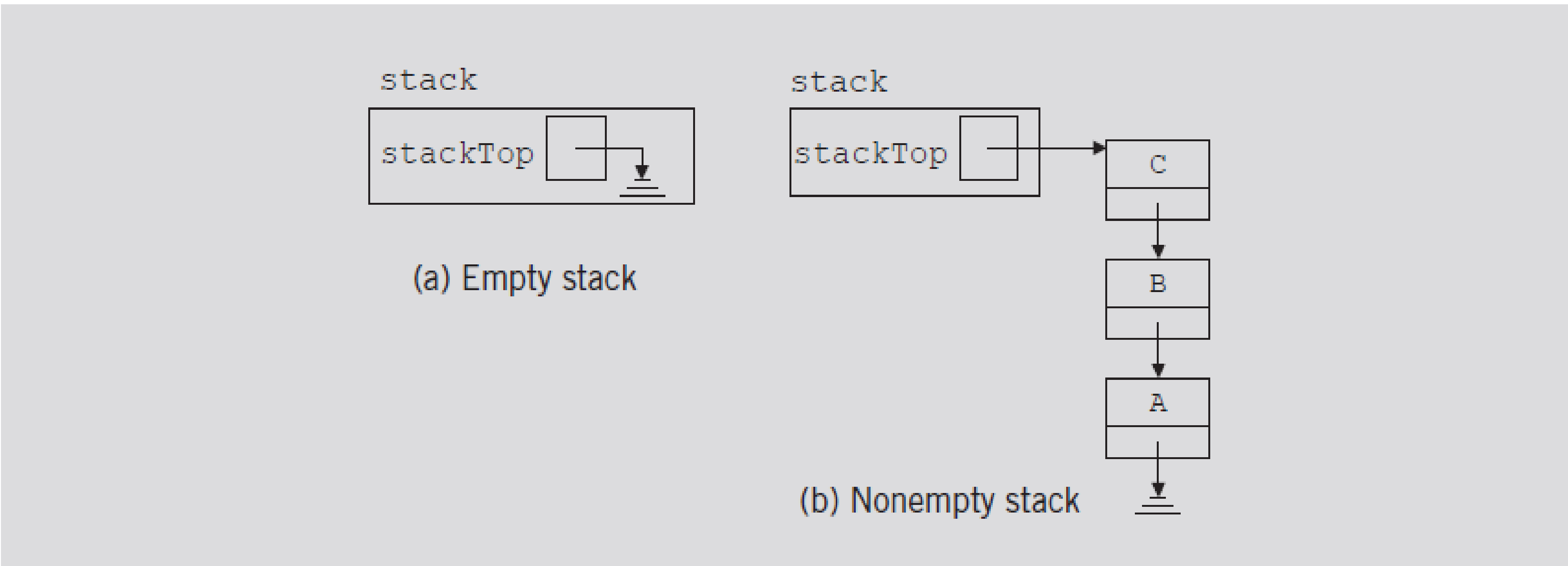


FIGURE 7-10 Empty and nonempty linked stacks

Default Constructor

- When stack object declared
  - Initializes stack to an empty state
  - Sets `stackTop` to `NULL`
- Definition of the default constructor

```
template <class Type>
linkedStackType<Type>::linkedStackType()
{
    stackTop = NULL;
}
```

Empty Stack and Full Stack

- Stack empty if `stackTop` is `NULL`
- Stack never full
  - Element memory allocated/deallocated dynamically
  - Function `isFullStack` always returns false value

```
template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return(stackTop == NULL);
} //end isEmptyStack

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
} //end isFullStack
```

Initialize Stack

- Reinitializes stack to an empty state
  - deallocate memory occupied by the stack elements, set `stackTop` to `NULL`
- Definition of the `initializeStack` function

O(n)

```
template <class Type>
void linkedStackType<Type>:: initializeStack()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (stackTop != NULL)   //while there are elements in
                               //the stack
    {
        temp = stackTop;     //set temp to point to the
                             //current node
        stackTop = stackTop->link;  //advance stackTop to the
                                    //next node
        delete temp;    //deallocate memory occupied by temp
    }
} //end initializeStack
```

Push

- `newElement` added at the beginning of the linked list pointed to by `stackTop`
- Value of pointer `stackTop` updated

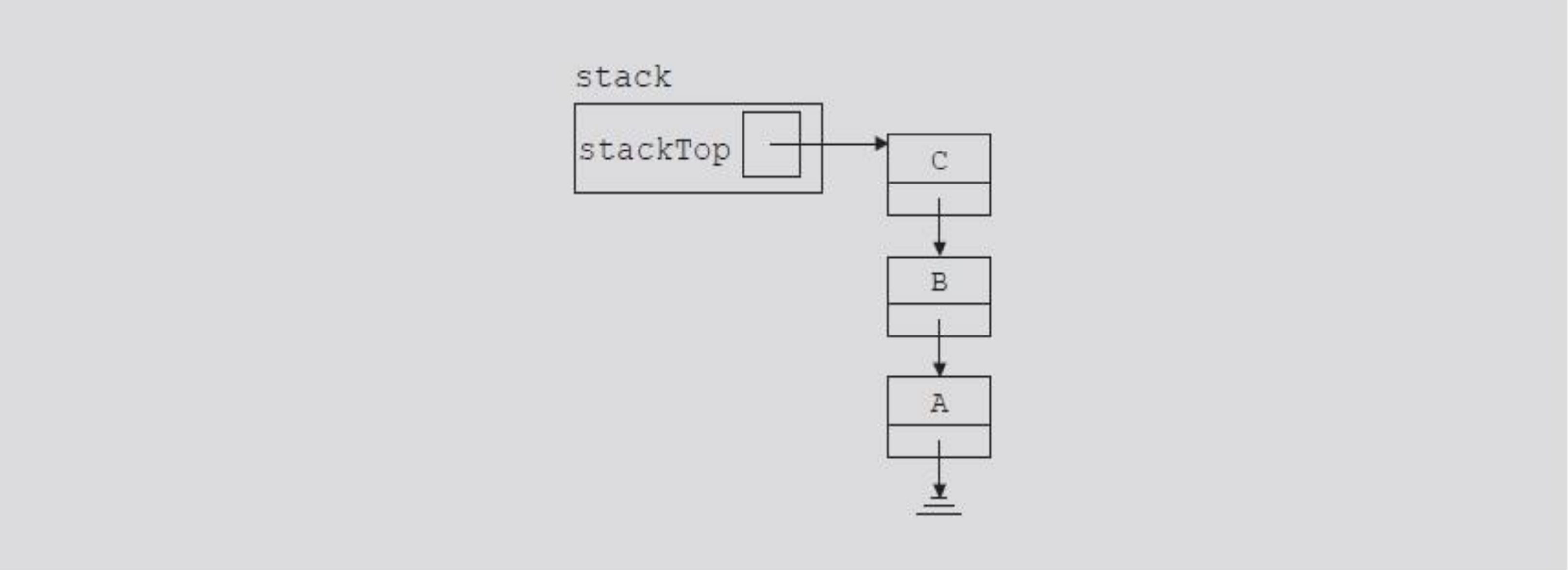


FIGURE 7-11 Stack before the `push` operation

Push (cont'd.)

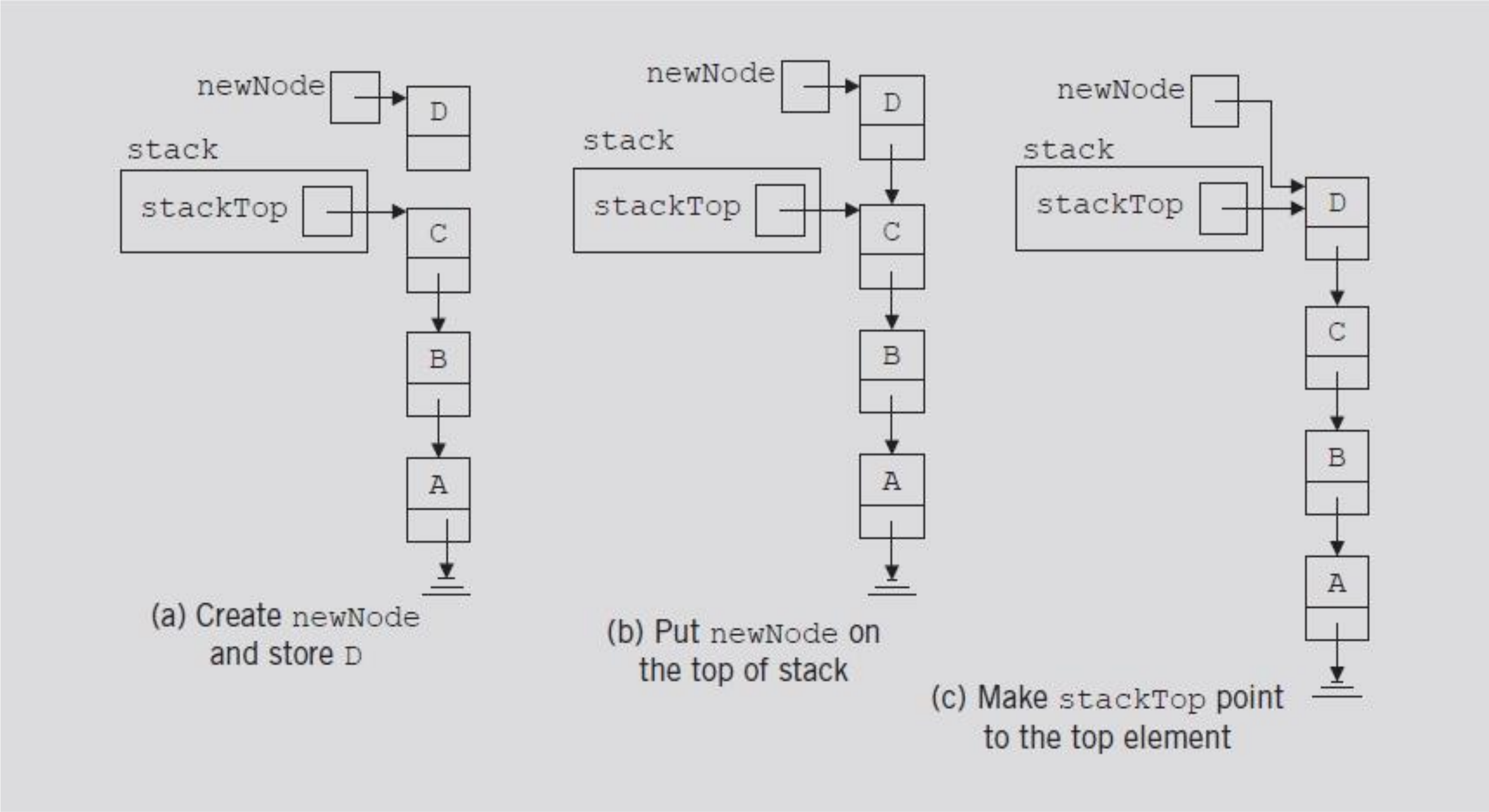


FIGURE 7-12 `Push` operation

Push (cont'd.)

- Definition of the `push` function

```
template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    nodeType<Type> *newNode;  //pointer to create the new node

    newNode = new nodeType<Type>; //create the node

    newNode->info = newElement; //store newElement in the node
    newNode->link = stackTop; //insert newNode before stackTop
    stackTop = newNode;       //set stackTop to point to the
                              //top node
} //end push
```

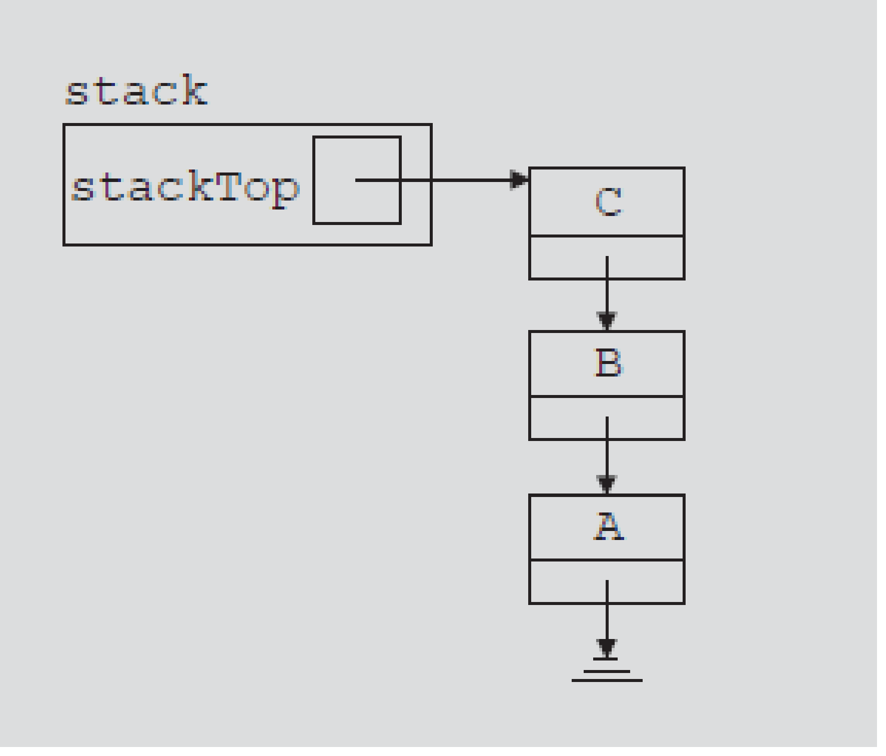
Return the Top Element

- Returns information of the node to which `stackTop` pointing
- Definition of the `top` function

```
template <class Type>
Type linkedStackType<Type>::top() const
{
    assert(stackTop != NULL); //if stack is empty,
                              //terminate the program
    return stackTop->info;    //return the top element
}//end top
```

Pop

- Removes top element of the stack
  - Node pointed to by `stackTop` removed
  - Value of pointer `stackTop` updated



Pop (cont'd.)

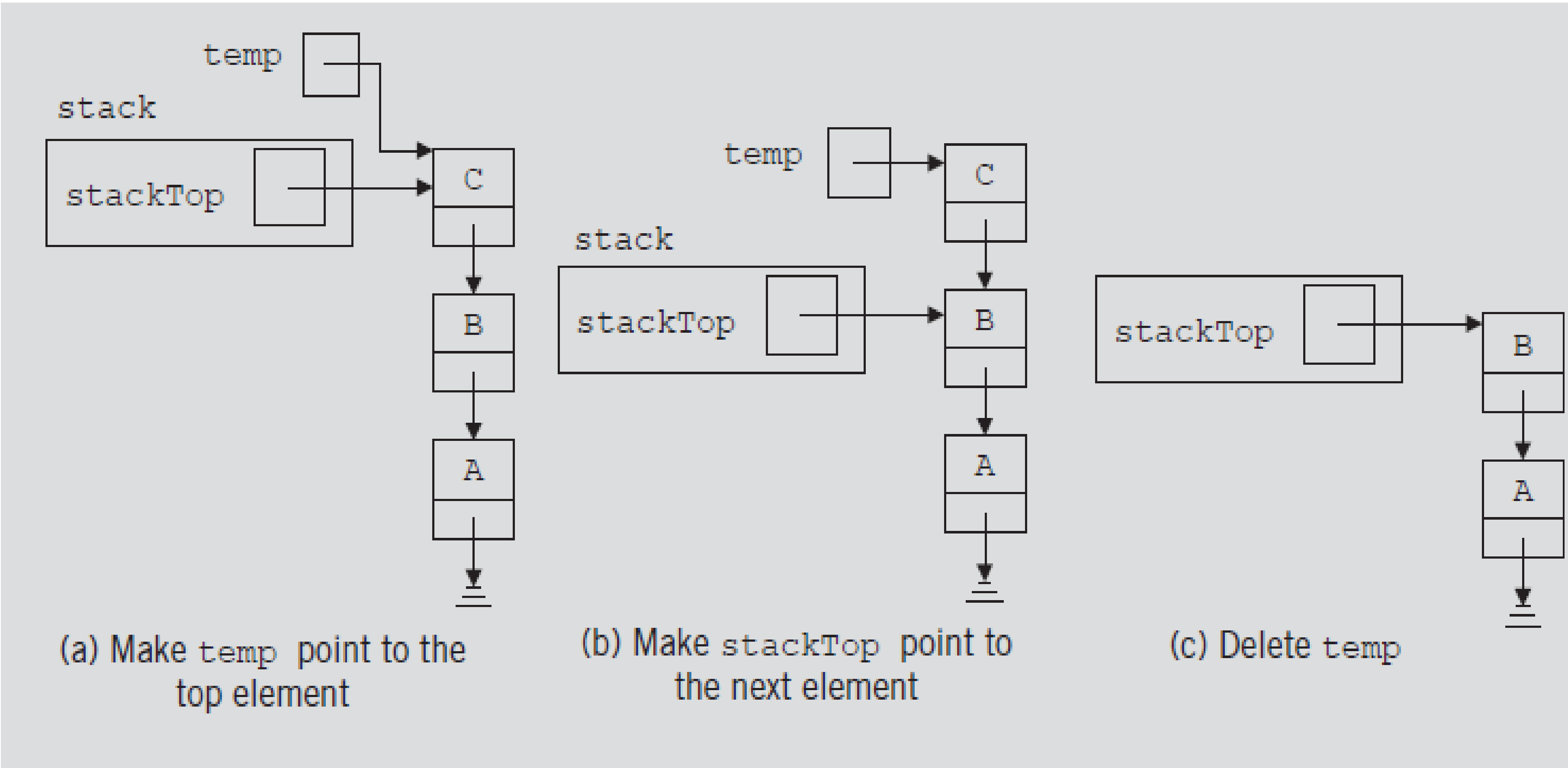


FIGURE 7-14 Pop operation

Pop (cont'd.)

- Definition of the `pop` function

```
template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp;   //pointer to deallocate memory

    if (stackTop != NULL)
    {
        temp = stackTop;  //set temp to point to the top node

        stackTop = stackTop->link;  //advance stackTop to the
                                    //next node
        delete temp;    //delete the top node
    }
    else
        cout << "Cannot remove from an empty stack." << endl;
}//end pop
```

Copy Stack

- Makes an identical copy of a stack
- Definition similar to the definition of copyList for linked lists
- Definition of the `copyStack` function

```
template  <class Type>
void linkedStackType<Type>::copyStack
                     (const linkedStackType<Type>& otherStack)
{
    nodeType<Type> *newNode, *current, *last;

    if (stackTop != NULL) //if stack is nonempty, make it empty
        initializeStack();

    if (otherStack.stackTop == NULL)
        stackTop = NULL;
    else
    {
        current = otherStack.stackTop;  //set current to point
                                   //to the stack to be copied

            //copy the stackTop element of the stack
        stackTop = new nodeType<Type>;  //create the node

        stackTop->info = current->info; //copy the info
        stackTop->link = NULL;  //set the link field to NULL
        last = stackTop;        //set last to point to the node
        current = current->link; //set current to point to the
                                 //next node
            //copy the remaining stack
        while (current != NULL)
        {
            newNode = new nodeType<Type>;

            newNode->info = current->info;
            newNode->link = NULL;
            last->link = newNode;
            last = newNode;
            current = current->link;
        }//end while
    }//end else
} //end copyStack
```

O(n)

Constructors and Destructors

- Definition of the functions to implement the copy constructor and the destructor

O(n)

```
    //copy constructor
template <class Type>
linkedStackType<Type>::linkedStackType(
                      const linkedStackType<Type>& otherStack)
{
    stackTop = NULL;
    copyStack(otherStack);
}//end copy constructor
```

O(n)

```
    //destructor
template  <class Type>
linkedStackType<Type>::~linkedStackType()
{
    initializeStack();
}//end destructor
```

Overloading the Assignment Operator (=)

- Definition of the functions to overload the assignment operator

O(n)

```
template <class Type>
const linkedStackType<Type>& linkedStackType<Type>::operator=
                     (const linkedStackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
}//end operator=
```

Overloading the Assignment Operator (=) (cont'd.)

TABLE 7-2 Time complexity of the operations of the class `linkedStackType` on a stack with *n* elements

Function	Time complexity
`isEmptyStack`	$O(1)$
`isFullStack`	$O(1)$
`initializeStack`	$O(n)$
constructor	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$
`copyStack`	$O(n)$
destructor	$O(n)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

Stack as Derived from the `class unorderedLinkedList`

- Stack `push` function, list `insertFirst` function
  - Similar algorithms
  - `initializeStack` and `initializeList`, `isEmptyList` and `isEmptyStack`, etc.
- `class linkedStackType` can be derived from `class linkedListType`
  - `class linkedListType`: abstract class
- `class linkedStackType` can be derived from `class unorderedLinkedListType`
  - `class unorderedLinkedListType`: derived from `class linkedListType`

```cpp
template<class Type>
class linkedStackType: public unorderedLinkedList<Type>
{
public:
    void initializeStack();
    bool isEmptyStack() const;
    bool isFullStack() const;
    void push(const Type& newItem);
    Type top() const;
    void pop();
};

template<class Type>
void linkedStackType<Type>::initializeStack()
{
    unorderedLinkedList<Type>::initializeList();
}

template<class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return unorderedLinkedList<Type>::isEmptyList();
}

template<class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
}

template<class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    unorderedLinkedList<Type>::insertFirst(newElement);
}

template<class Type>
Type linkedStackType<Type>::top() const
{
    return unorderedLinkedList<Type>::front();
}

template<class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp;

    temp = first;
    first = first->link;
    delete temp;
}
```

Application of Stacks: Postfix Expressions Calculator

- Arithmetic notations
  - Infix notation: operator between operands
  - Prefix (Polish) notation: operator precedes operands
  - Postfix (Reverse Polish) notation: operator follows operands
- Stack use in compliers
  - Translate infix expressions into some form of postfix notation
  - Translate postfix expression into machine code

Infix, Prefix, and Postfix

Infix	Prefix	Postfix
A + B	**+ A B**	**A B +**
A * B + C	**+ * A B C**	**A B * C +**
A * (B + C)	*** A + B C**	**A B C + ***
A - (B - (C - D))	**- A - B - C D**	**A B C D - - -**
A - B - C - D	**- - - A B C D**	**A B - C - D -**

- Prefix and Postfix: no parentheses

Infix and Postfix Expression Conversion

EXAMPLE 7-4

Infix expression	Equivalent postfix expression
$a + b$	$a\ b\ +$
$a + b * c$	$a\ b\ c\ *\ +$
$a * b + c$	$a\ b\ *\ c\ +$
$(a + b) * c$	$a\ b\ +\ c\ *$
$(a - b) * (c + d)$	$a\ b\ -\ c\ d\ +\ *$
$(a + b) * (c - d / e) + f$	$a\ b\ +\ c\ d\ e\ /\ -\ *\ f\ +$

Postfix Algorithm

- Postfix expression can be evaluated using the following algorithm
  - *Scan the expression from left to right*
  - *When an operator is found, back up to get the required number of (preceding) operands, perform the operation*
  - *Repeat until reaching the end of the expression*

Application of Stacks: Postfix Expressions Calculator (cont'd.)

- Postfix expression: 6 3 + 2 * =

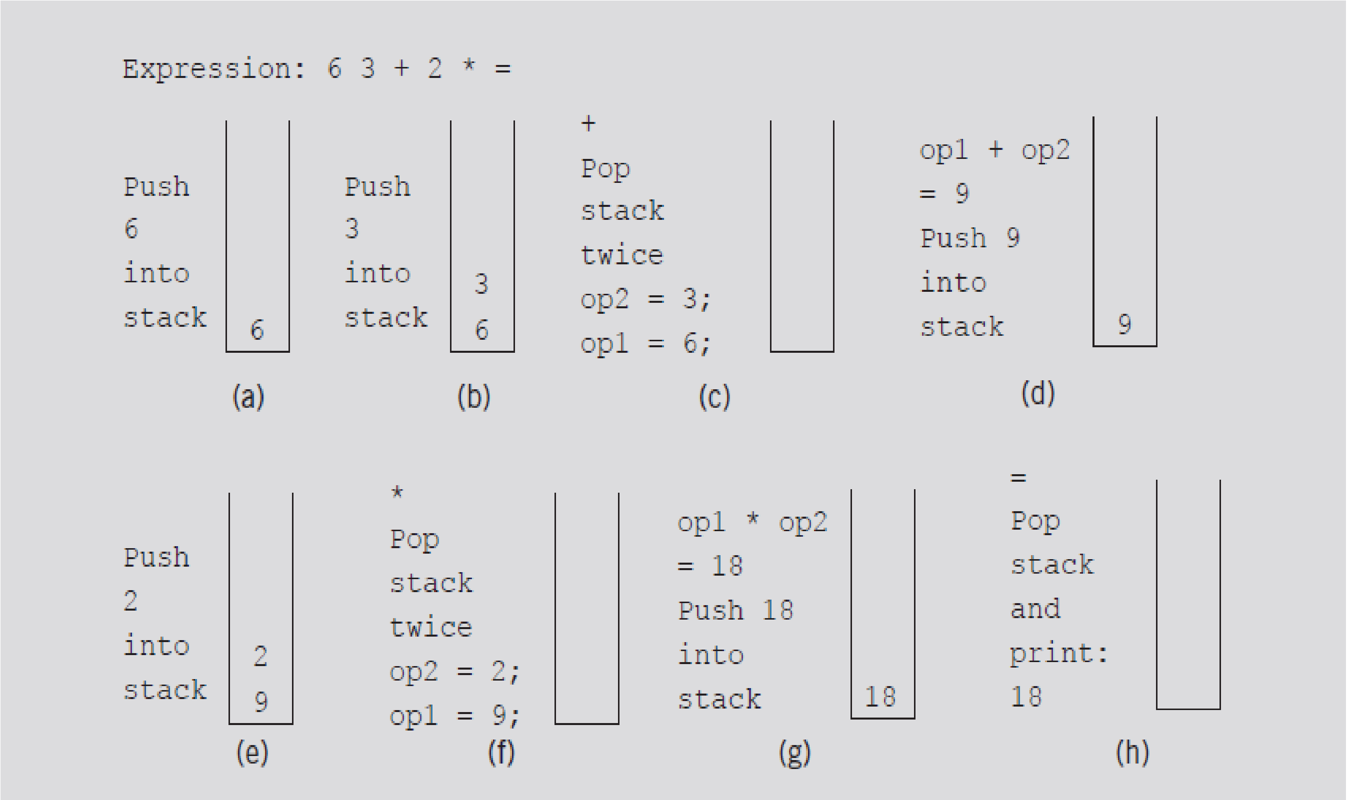


FIGURE 7-15 Evaluating the postfix expression: 6 3 + 2 * =

Postfix Algorithm w/ Stack

- When an operand (number) is encountered in an expression, it is pushed onto the stack.
- When we read a symbol other than a number, the following cases arise
  1. The symbol we read is one of +, -, *, /, or =.
     a) If the symbol is +, -, *, or /, it is an operator and we must evaluate it. Because an operator requires two operands, the stack must have at least two elements; otherwise, the expression has an error
     b) If the symbol is =, the expression ends and we must print the answer. The stack must contain exactly one element, which is the result; otherwise, it has an error.
  2. The symbol we read is something other than +, -, *, /, or =. In this case, the expression contains an illegal operator.

Postfix Expression

- Consider the following expressions
  - 7 6 + 3 ; 6 - =
  - 14 + 2 3 * =
  - 13 2 3 + =

Invalid operator ;

Insufficient operand for +

Too many operands

- To make the input easier to read, we assume the postfix expressions are in the following form:

  #6 #3 + #2 * =

- The symbol # precedes each number in the expression

Application of Stacks: Postfix Expressions Calculator (cont'd.)

- Main algorithm pseudocode

```
Read the first character
while not the end of input data
{
    a. initialize the stack
    b. process the expression
    c. output result
    d. get the next expression
}
```

  – Broken into four functions for simplicity
    - Function `evaluateExpression`
    - Function `evaluateOpr`
    - Function `discardExp`
    - Function `printResult`

```
void evaluateExpression(ifstream& inpF, ofstream& outF,
                        stackType<double>& stack,
                        char& ch, bool& isExpOk)
{
    double num;
    outF << ch;

    while (ch != '=')
    {
        switch (ch)
        {
        case '#':
            inpF >> num;
            outF << num << " ";
            if (!stack.isFullStack())
                stack.push(num);
            else
            {
                cout << "Stack overflow. "
                     << "Program terminates!" << endl;
                exit(0);  //terminate the program
            }

            break;

        default:
            evaluateOpr(outF, stack, ch, isExpOk);
        }//end switch

        if (isExpOk) //if no error
        {
            inpF >> ch;
            outF << ch;

            if (ch != '#')
                outF << " ";
        }
        else
            discardExp(inpF, outF, ch);
    } //end while (!= '=')
} //end evaluateExpression
```

```
void evaluateOpr(ofstream& out, stackType<double>& stack,
                 char& ch, bool& isExpOk)
{
    double op1, op2;

    if (stack.isEmptyStack())
    {
        out << " (Not enough operands)";
        isExpOk = false;
    }
    else
    {
        op2 = stack.top();
        stack.pop();

        if (stack.isEmptyStack())
        {
            out << " (Not enough operands)";
            isExpOk = false;
        }
        else
        {
            op1 = stack.top();
            stack.pop();

            switch (ch)
            {
            case '+':
                stack.push(op1 + op2);
                break;

            case '-':
                stack.push(op1 - op2);
                break;

            case '*':
                stack.push(op1 * op2);
                break;

            case '/':
                if (op2 != 0)
                    stack.push(op1 / op2);
                else
                {
                    out << " (Division by 0)";
                    isExpOk = false;
                }
                break;

            default:
                out << " (Illegal operator)";
                isExpOk = false;
            }//end switch
        } //end else
    } //end else
} //end evaluateOpr
```

```
void discardExp(ifstream& in, ofstream& out, char& ch)
{
    while (ch != '=')
    {
        in.get(ch);
        out << ch;
    }
} //end discardExp

void printResult(ofstream& outF, stackType<double>& stack,
                 bool isExpOk)
{
    double result;

    if (isExpOk) //if no error, print the result
    {
        if (!stack.isEmptyStack())
        {
            result = stack.top();
            stack.pop();

            if (stack.isEmptyStack())
                outF << result << endl;
            else
                outF << " (Error: Too many operands)" << endl;
        } //end if
        else
            outF << " (Error in the expression)" << endl;
    }
    else
        outF << " (Error in the expression)" << endl;

    outF << "_________________________________"
         << endl << endl;
} //end printResult
```

```
int main()
{
    bool expressionOk;
    char ch;
    stackType<double> stack(100);
    ifstream infile;
    ofstream outfile;

    infile.open("RpnData.txt");

    if (!infile)
    {
        cout << "Cannot open the input file. "
             << "Program terminates!" << endl;
        return 1;
    }

    outfile.open("RpnOutput.txt");

    outfile << fixed << showpoint;
    outfile << setprecision(2);

    infile >> ch;
    while (infile)
    {
        stack.initializeStack();
        expressionOk = true;
        outfile << ch;

        evaluateExpression(infile, outfile, stack, ch,
                           expressionOk);
        printResult(outfile, stack, expressionOk);
        infile >> ch; //begin processing the next expression
    } //end while

    infile.close();
    outfile.close();

    return 0;

} //end main
```

fixed: floating-point values are written using fixed-point notation: the value is represented with exactly as many digits in the decimal part as specified by the *precision field* (precision) and with no exponent part.
showpoint: the decimal point is always written for floating point values inserted into the stream (even for those whose decimal part is zero).

Infix to Postfix Conversion

1. **Initialize an empty stack of operators**
2. **While (no error && !end of expression)**
   a) Get next input "token" from infix expression
   b) If token is …
      i. "(" : push onto stack
      ii. ")" : pop and append stack elements until "(" occurs, do not display it
      iii. operator
         if (stack is empty or operator has higher priority than top of stack)
            push token onto stack
         else
            pop and append top of stack to postfix; repeat comparison of token with top of stack
      iv. Operand: append to postfix
3. **When end of infix reached, pop and append stack items to postfix until empty**

NOTE: (in stack has lower priority than operators

- Ex: A * B + C
- Ex: A * (B + C)

Print a Linked List in Reverse Order (Chapter 6)

- Function template to implement previous algorithm and then apply it to a list

```
template <class Type>
void linkedListType<Type>::reversePrint
                              (nodeType<Type> *current) const
{
    if (current != NULL)
    {
        reversePrint(current->link);    //print the tail
        cout << current->info << " ";   //print the node
    }
}
```

Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward

- Stack
  - Used to design nonrecursive algorithm
    - Print a linked list backward
- Use linked implementation of stack

```
current = first;                    //Line 1

while (current != NULL)             //Line 2
{                                   //Line 3
    stack.push(current);            //Line 4
    current = current->link;        //Line 5
}                                   //Line 6
while (!stack.isEmptyStack())        //Line 7
{                                    //Line 8
    current = stack.top();           //Line 9
    stack.pop();                     //Line 10
    cout << current->info << " ";    //Line 11
}                                    //Line 12
```

delete current;

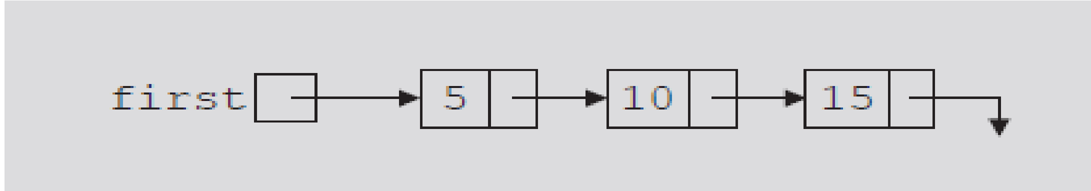


FIGURE 7-16 Linked list

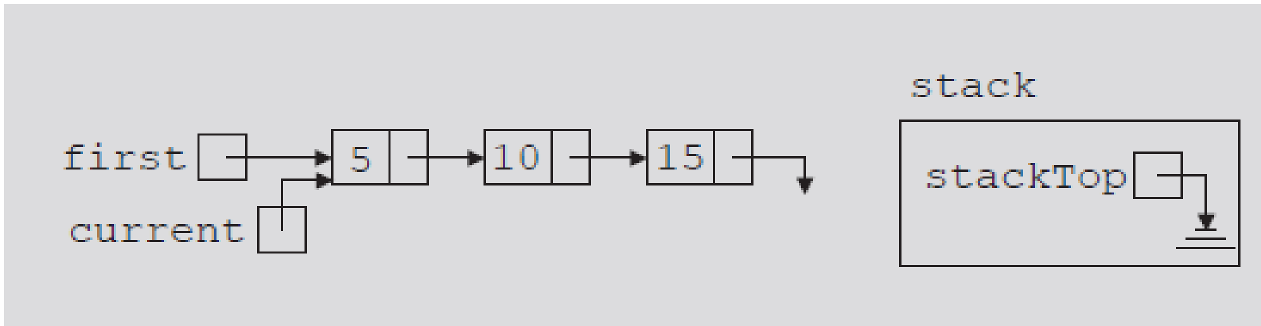


FIGURE 7-17 List after the statement `current = first;` executes

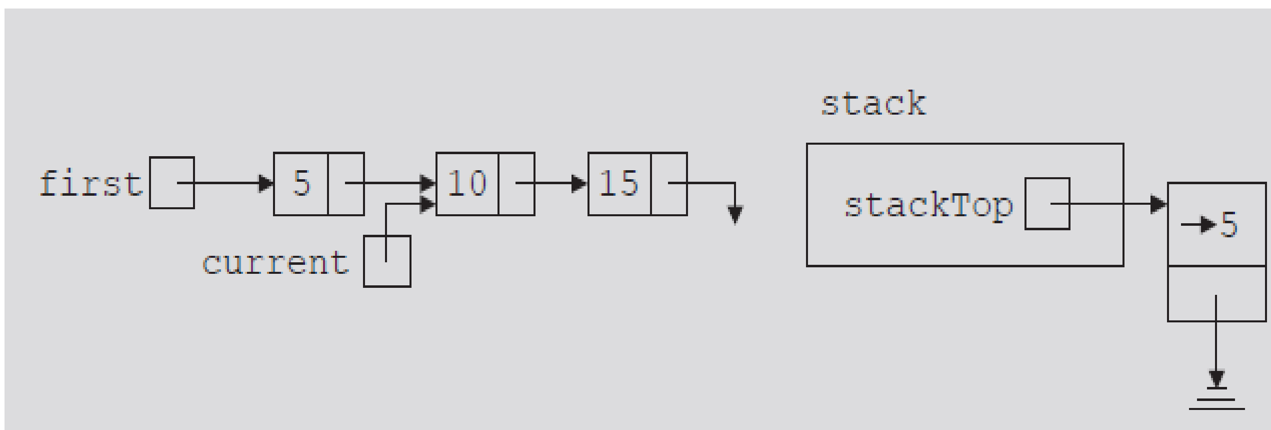


FIGURE 7-18 List and stack after the statements `stack.push(current);` and `current = current->link;` execute

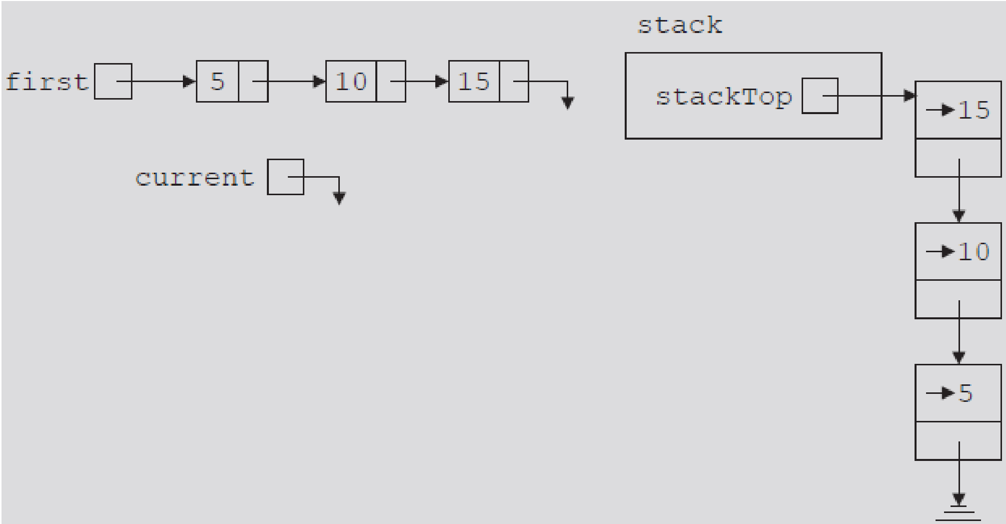


FIGURE 7-19 List and stack after the 1st `while` statement executes

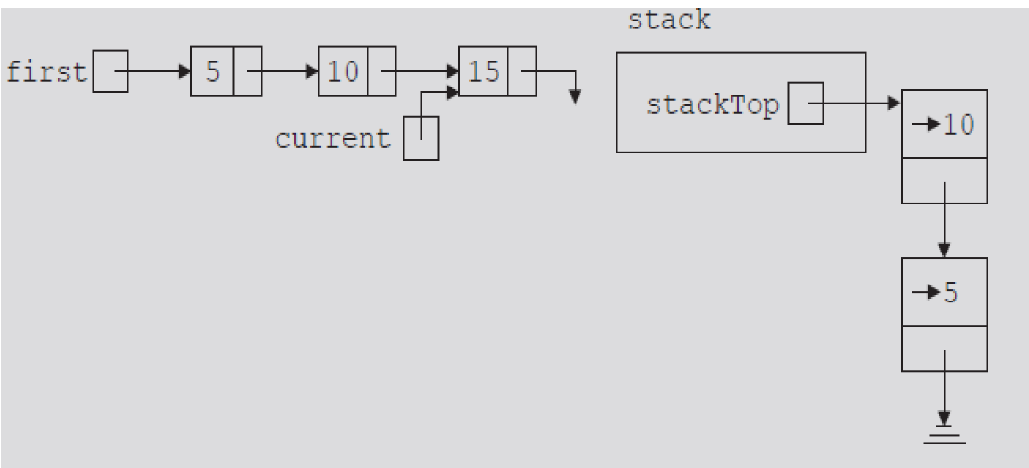


FIGURE 7-20 List and stack after the statements `current = stack.top();` and `stack.pop();` execute

STL `class stack`

- Standard Template Library (STL) library class defining a stack
- Header file containing `class stack` definition
  - `stack`

TABLE 7-3 Operations on a `stack` object

Operation	Effect
`size`	Returns the actual number of elements in the stack.
`empty`	Returns `true` if the stack is empty, and `false` otherwise.
`push(item)`	Inserts a copy of item into the stack.
`top`	Returns the top element of the stack, but does not remove the top element from the stack. This operation is implemented as a value-returning function.
`pop`	Removes the top element of the stack.

Summary

- Stack
  - Last In First Out (LIFO) data structure
  - Implemented as array or linked list
  - Arrays: limited number of elements
  - Linked lists: allow dynamic element addition
- Stack use in compliers
  - Translate infix expressions into some form of postfix notation
  - Translate postfix expression into machine code
- Standard Template Library (STL)
  - Provides a class to implement a stack in a program

Self Exercises

- Programming Exercises: 1, 2, 3, 4, 7, 9