Data Structures and Algorithms

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)

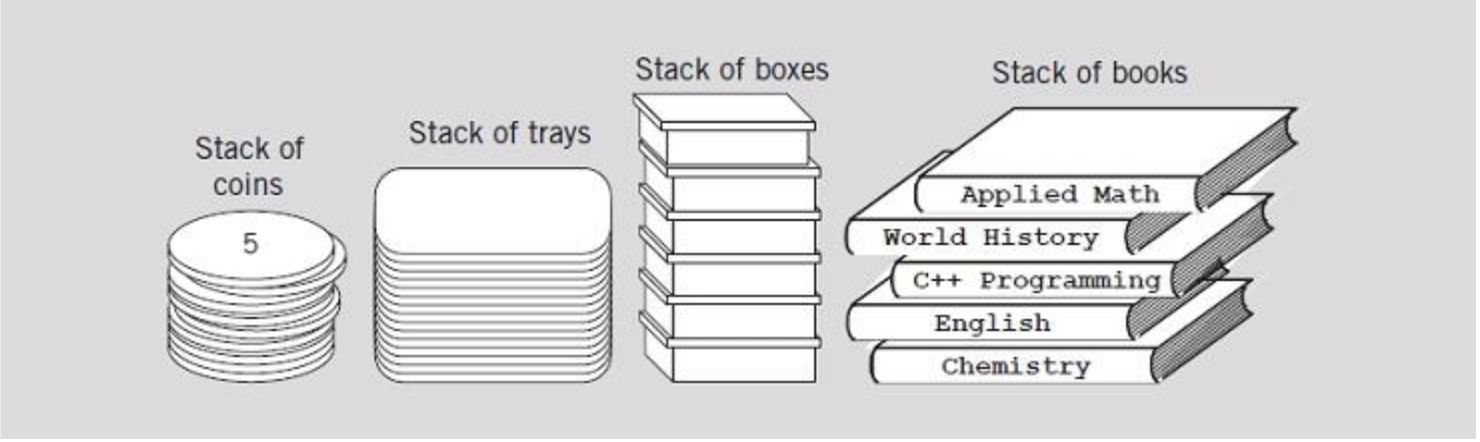


FIGURE 7-1 Various examples of stacks

Stacks (cont'd.)

- `push` operation
  - Add element onto the stack
- `top` operation
  - Retrieve top element of the stack
- `pop` operation
  - Remove top element from the stack

Stacks (cont'd.)

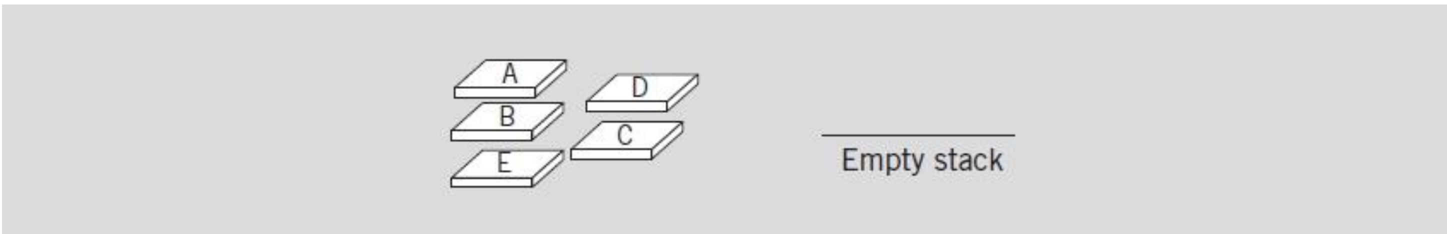


FIGURE 7-2 Empty stack

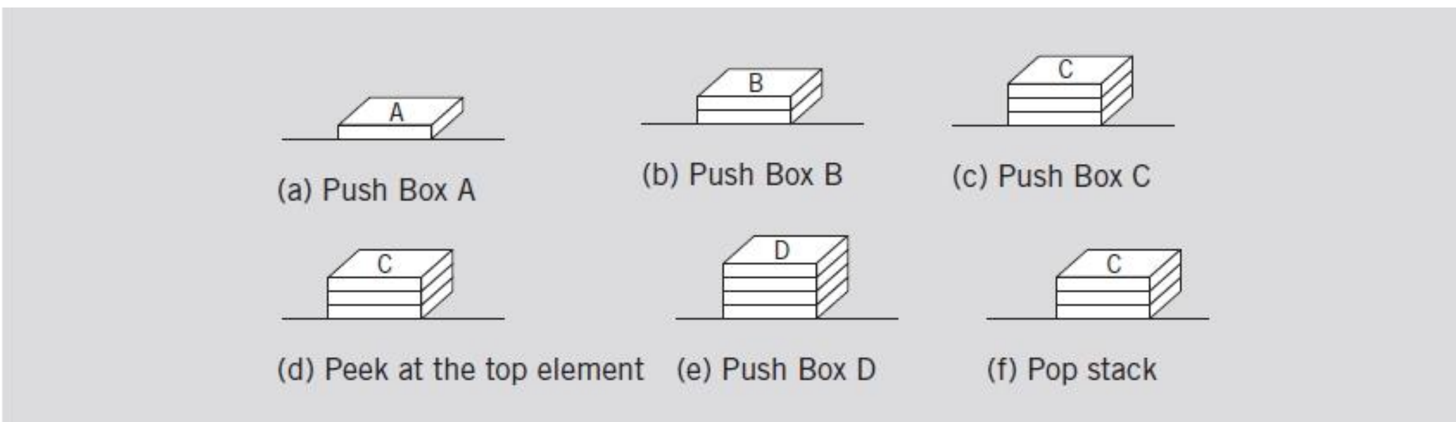


FIGURE 7-3 Stack operations

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

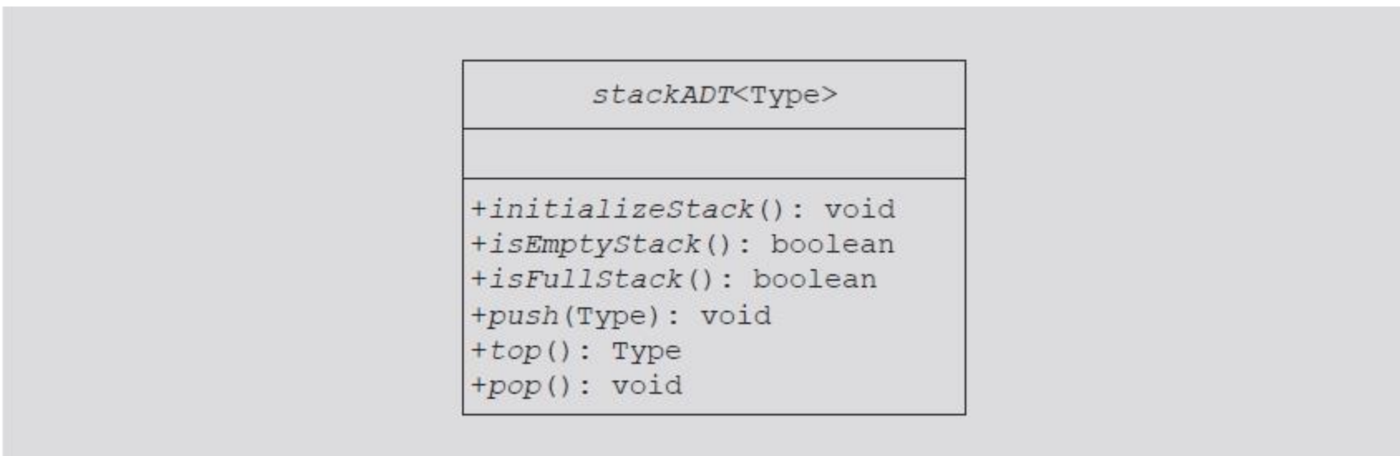


FIGURE 7-4 UML class diagram of the `class stackADT`

Implementation of Stacks as Arrays

- 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

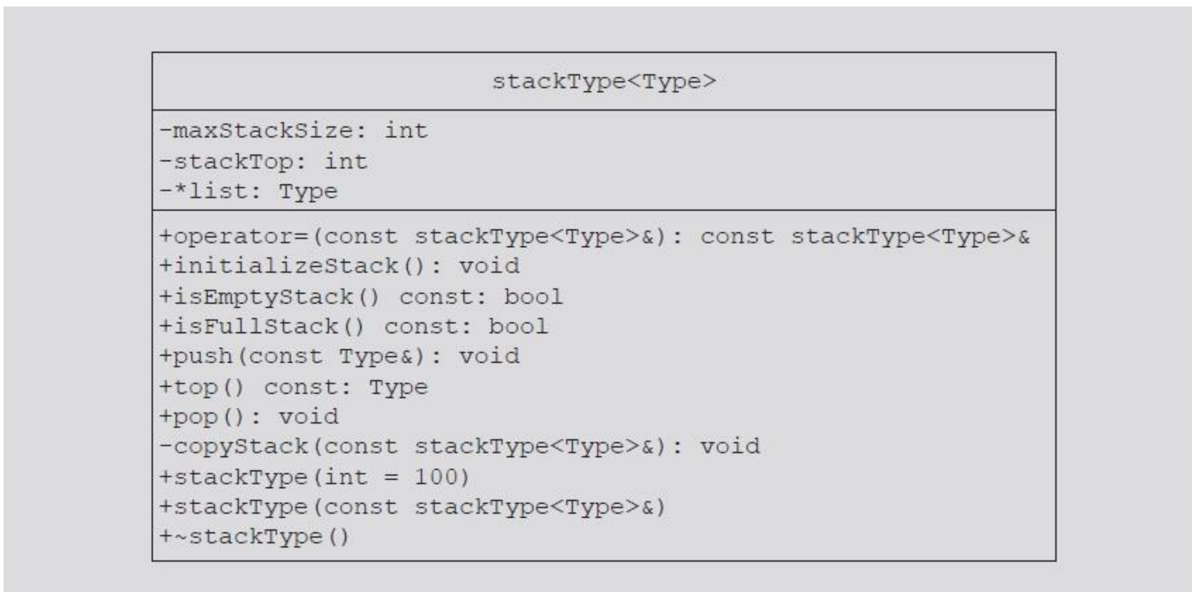


FIGURE 7-5 UML class diagram of the `class stackType`

Implementation of Stacks as Arrays (cont'd.)

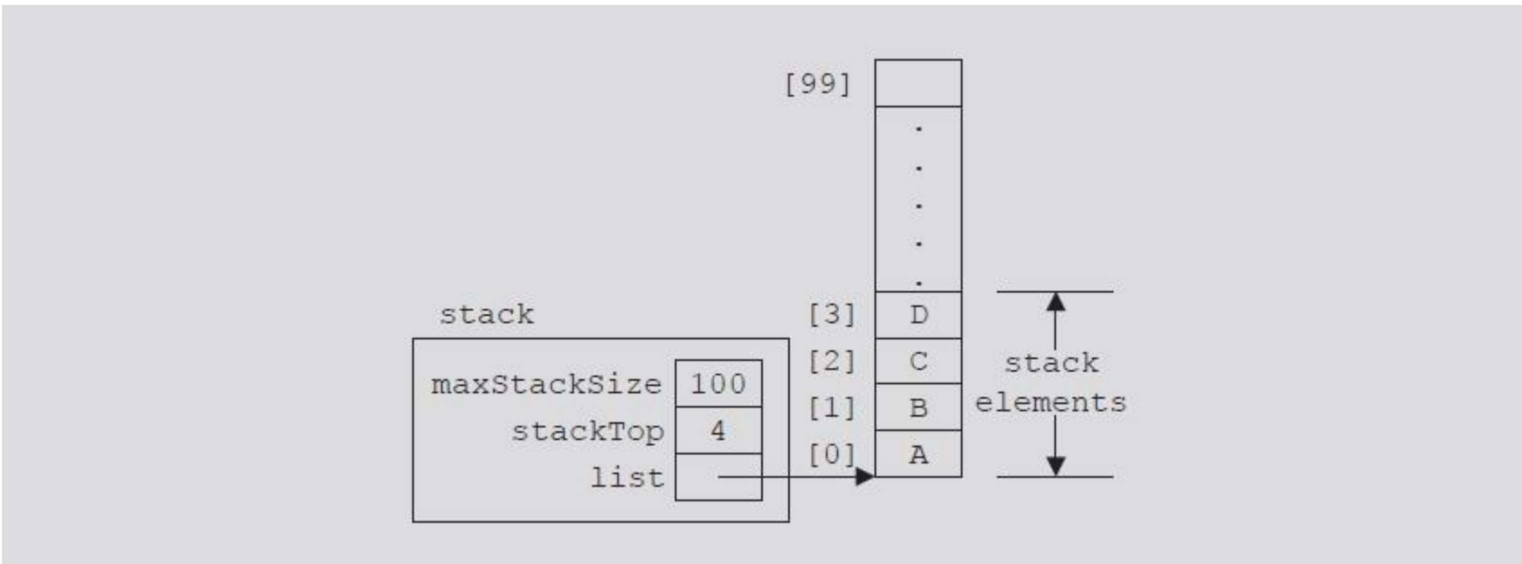


FIGURE 7-6 Example of a stack

Initialize Stack

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

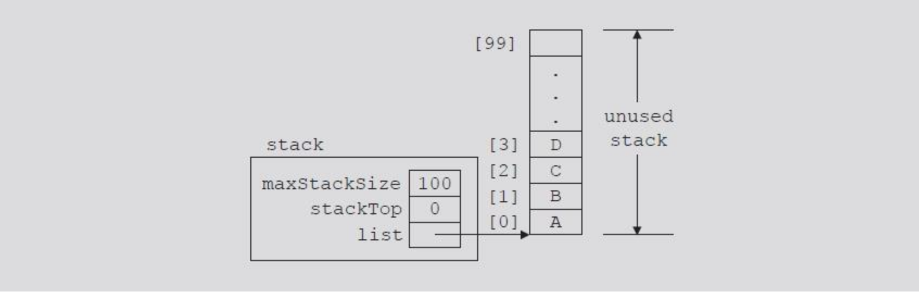


FIGURE 7-7 Empty stack

```
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`

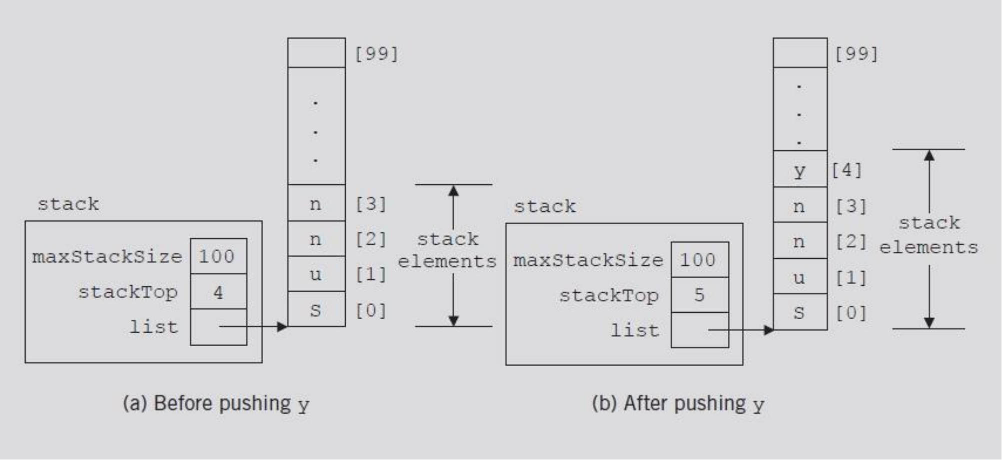


FIGURE 7-8 Stack before and after the `push` operation

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

```
ItemType StackType::Top()
{
      if (IsEmpty())
        throw EmptyStack();
  return list[top];
}
```

Pop

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

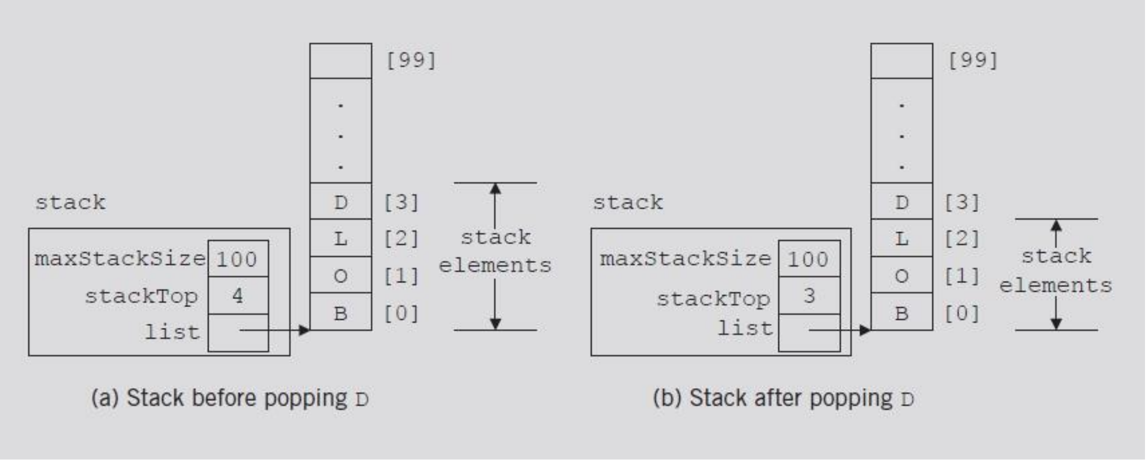


FIGURE 7-9 Stack before and after the `pop` operation

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`

```
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`

```
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
```

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`: 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

Linked Implementation of Stacks (cont'd.)

- Value of `stackTop`: linked representation
  - Locates top element in the stack
    - Gives address (memory location) of the top element of the stack
- Review program on page 415
  - Class specifying basic operation on a stack as a linked list

Linked Implementation of Stacks (cont'd.)

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

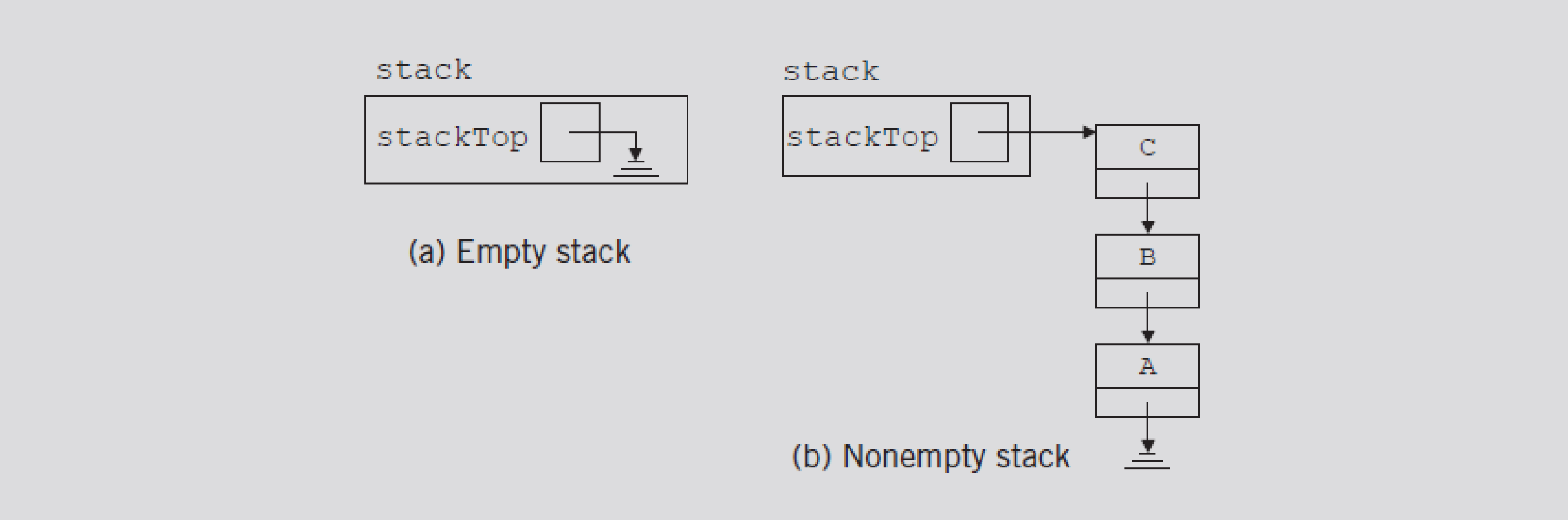


FIGURE 7-10 Empty and nonempty linked stacks

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
- Because stack might contain elements and you are using a linked implementation of a stack
  – Must deallocate memory occupied by the stack elements, set `stackTop` to `NULL`
- Definition of the `initializeStack` function

Initialize Stack (cont'd.)

```
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

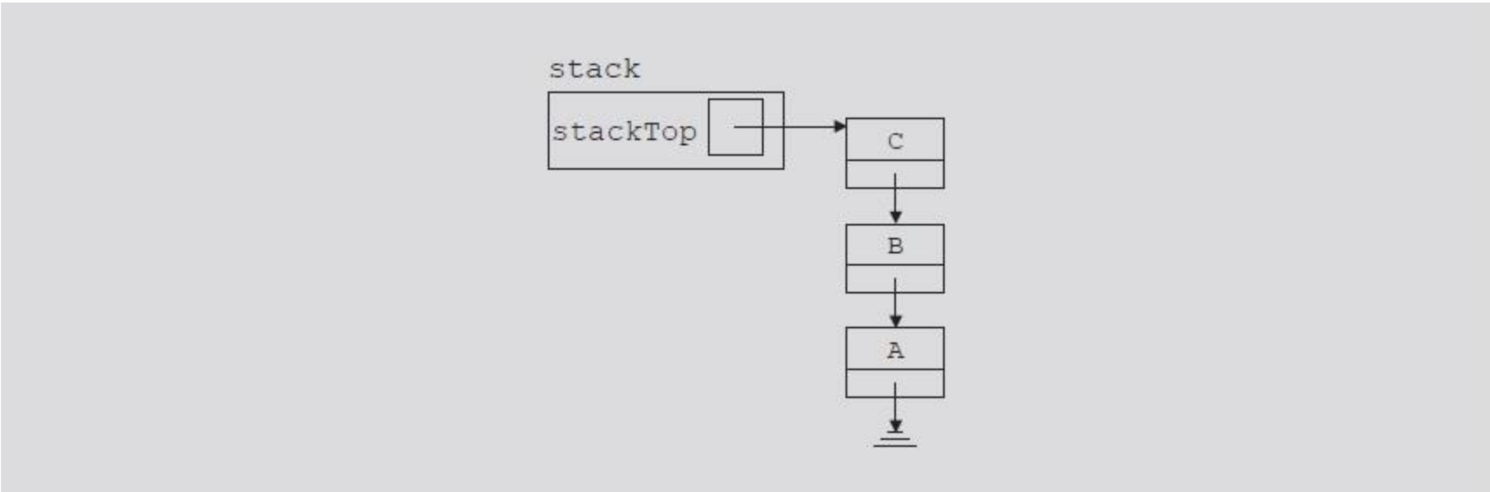


FIGURE 7-11 Stack before the `push` operation

Push (cont'd.)

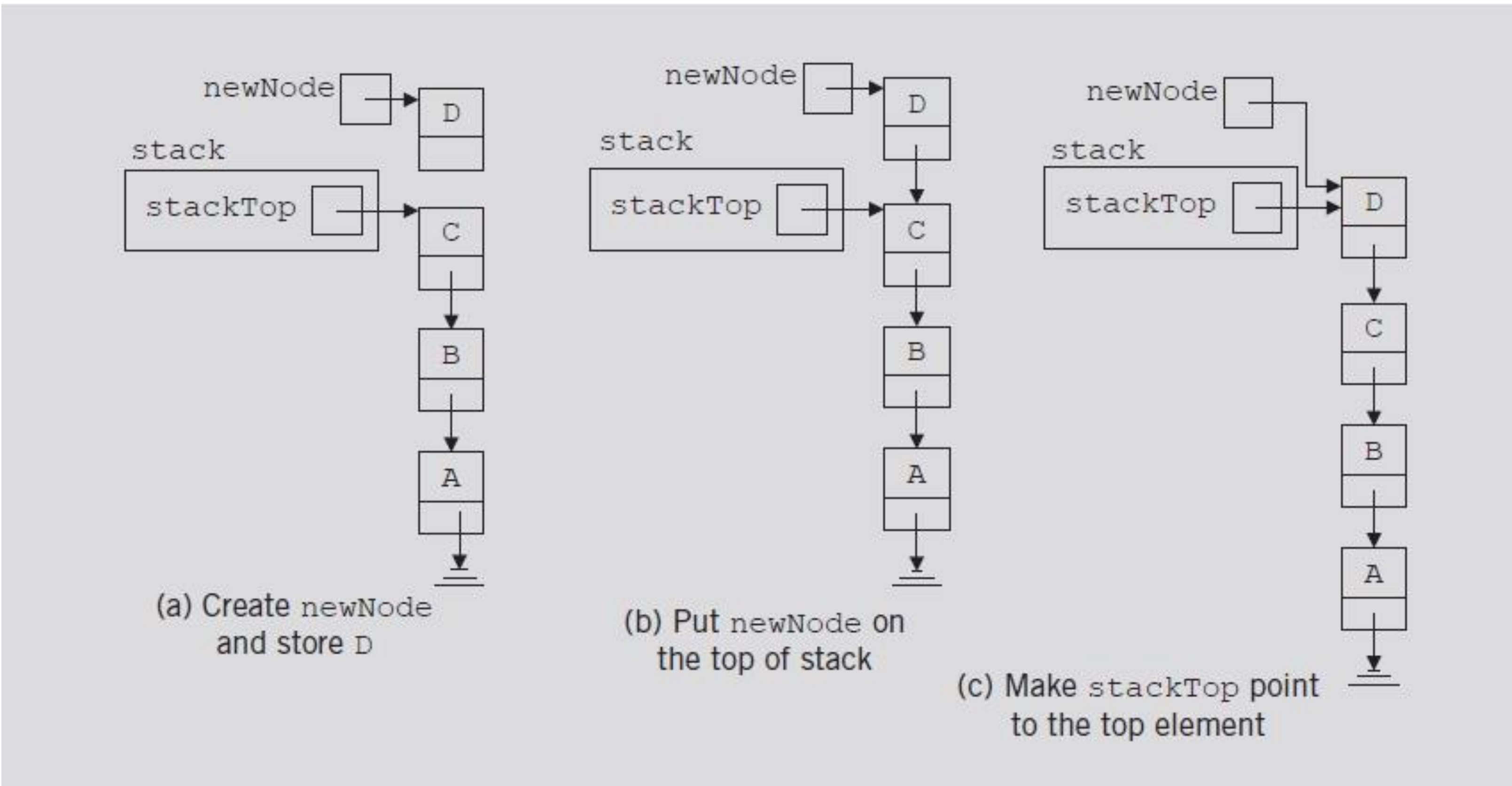


FIGURE 7-12 `Push` operation

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
```

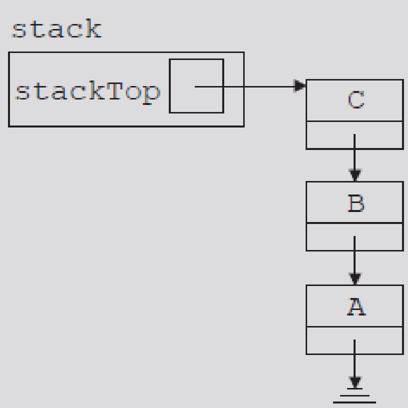
Return the Top Element

- 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.)

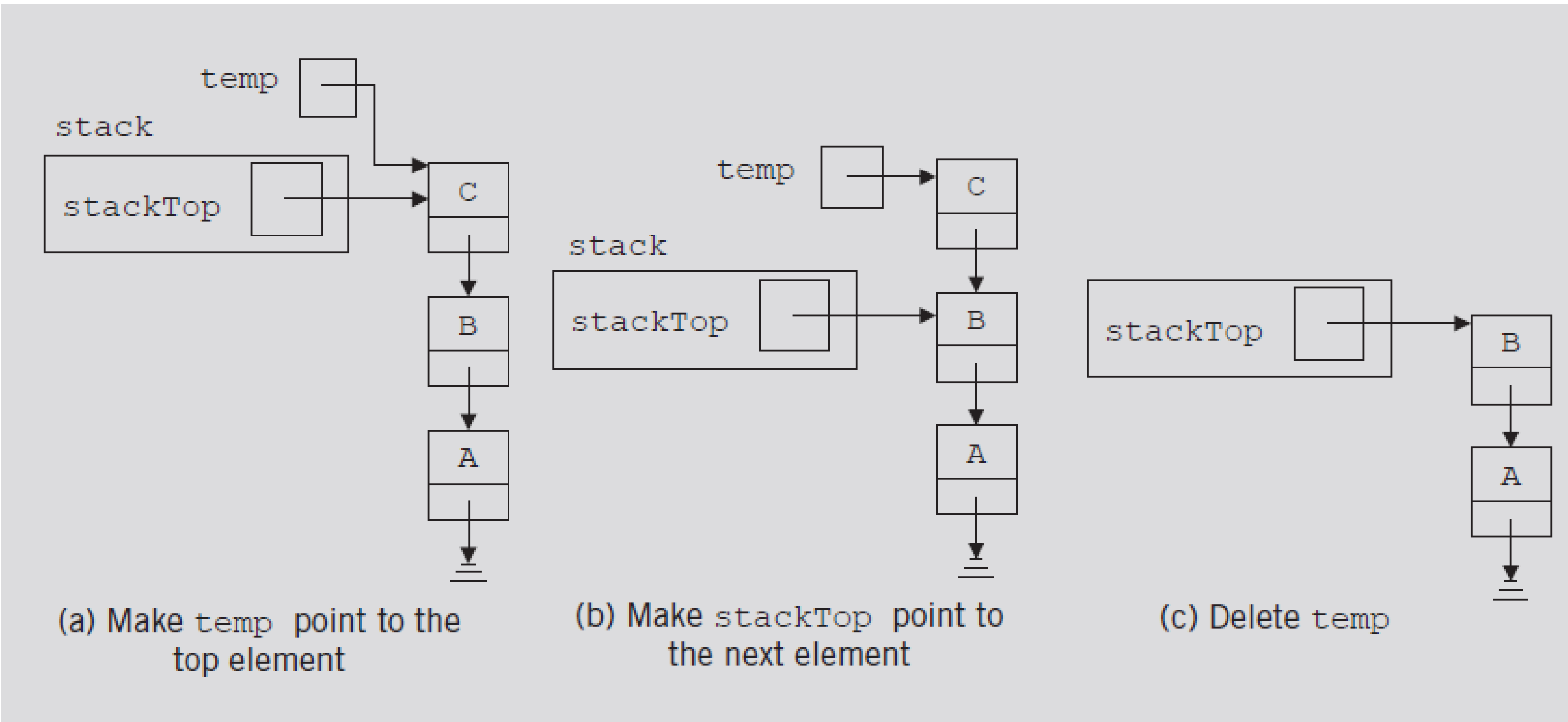


FIGURE 7-14 `Pop` operation

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
```

Constructors and Destructors

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

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

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

Overloading the Assignment Operator (=)

- Definition of the functions to overload the assignment operator

```
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	$O(n)$

Application of Stacks: Postfix Expressions Calculator

- Arithmetic notations
  - Infix notation: operator between operands
  - Prefix (Polish) notation: operator precedes operands
  - 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

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

- Postfix expression: 6 3 + 2 * =

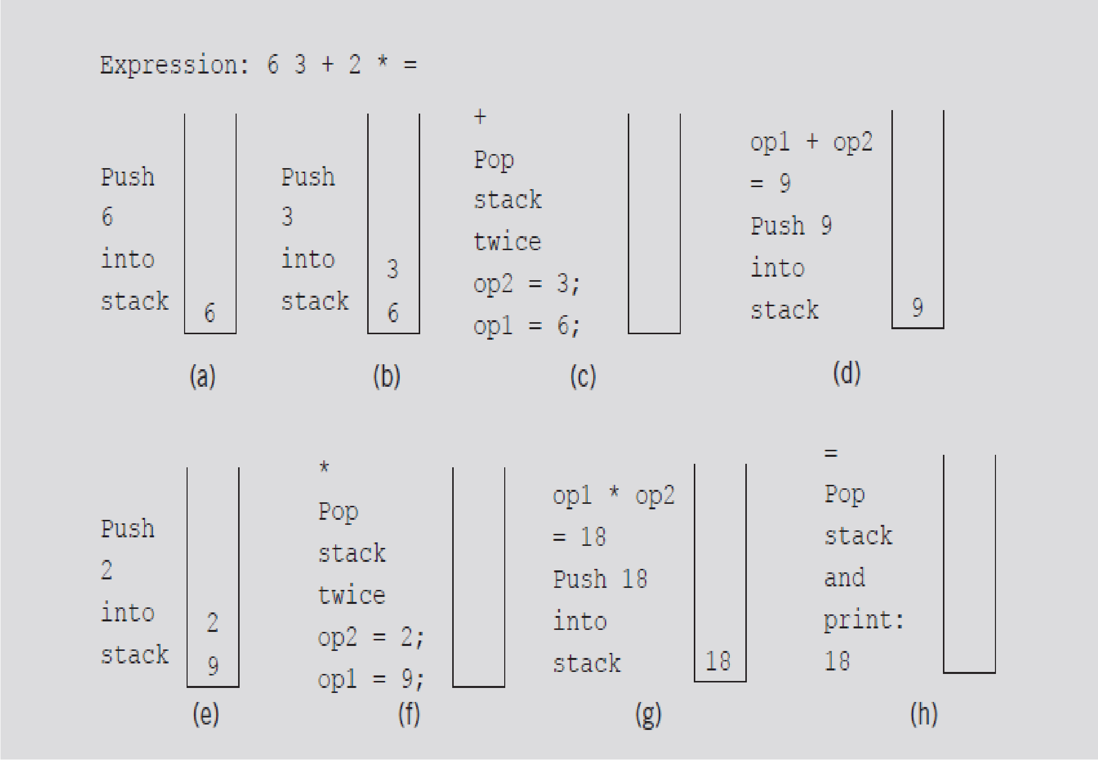


FIGURE 7-15 Evaluating the postfix expression: 6 3 + 2 * =

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