Lesson 10
Stacks and Queues

Stacks

- Stack: a data structure in which elements are added and removed from one end only
  - Addition/deletion occur only at the top of the stack
  - Last in first out (LIFO) data structure
- Operations:
  - Push: to add an element onto the stack
  - Pop: to remove an element from the stack

Stacks (cont'd.)

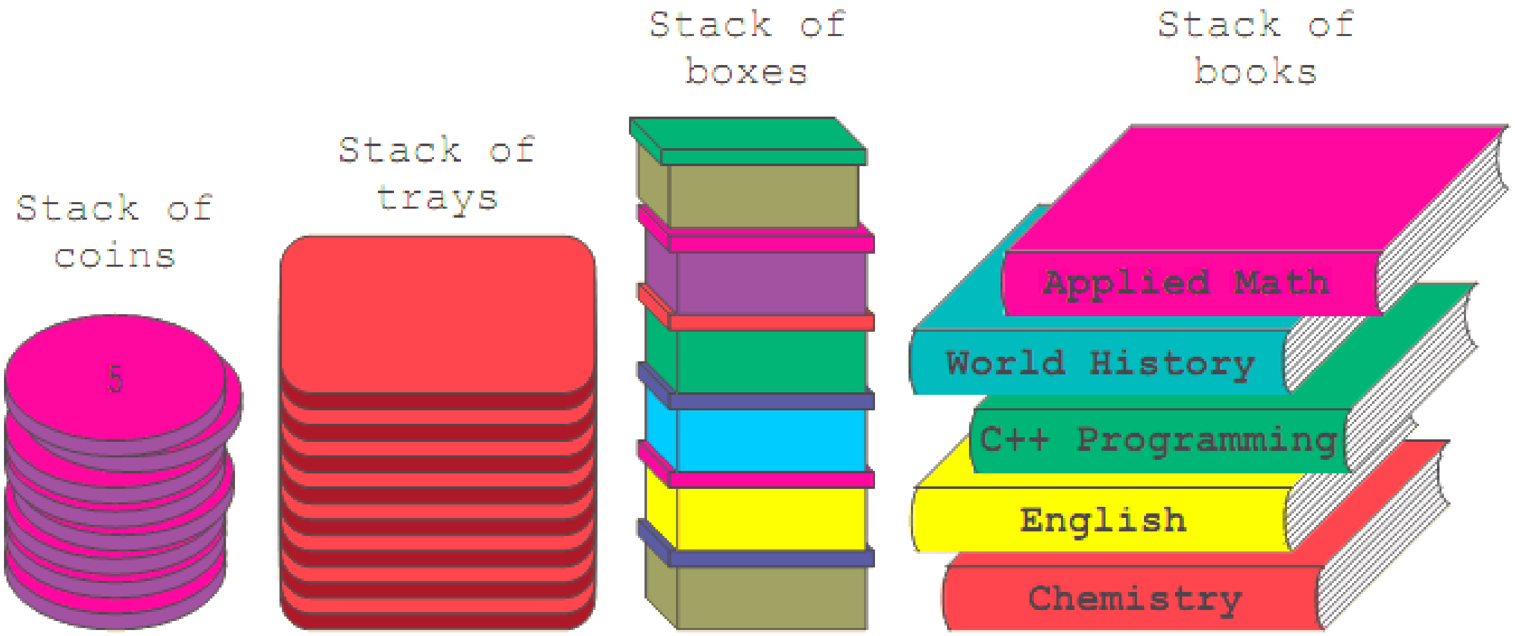

FIGURE 17-1 Various types of stacks

Stacks (cont'd.)

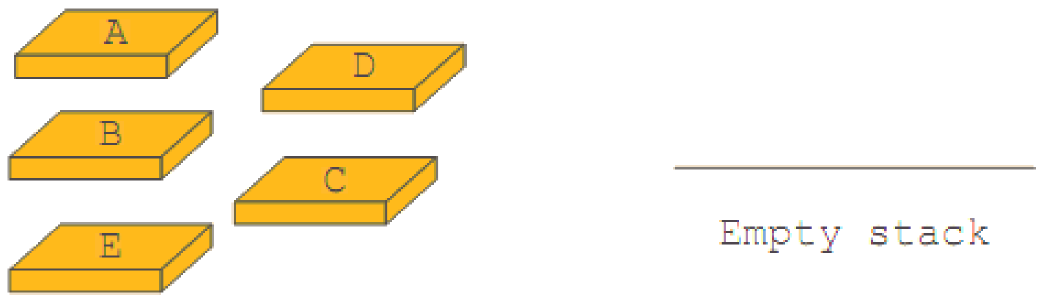

FIGURE 17-2 Empty stack

Stacks (cont'd.)

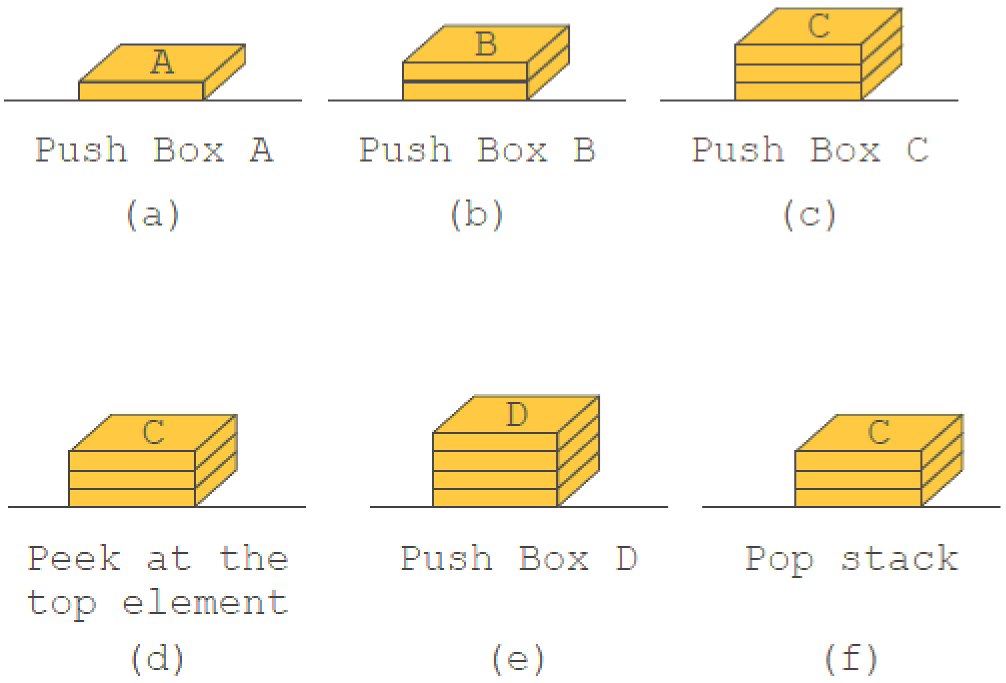

FIGURE 17-3 Stack operations

Stack Operations

- In the abstract `class stackADT`:
  - `initializeStack`
  - `isEmptyStack`
  - `isFullStack`
  - `push`
  - `top`
  - `pop`

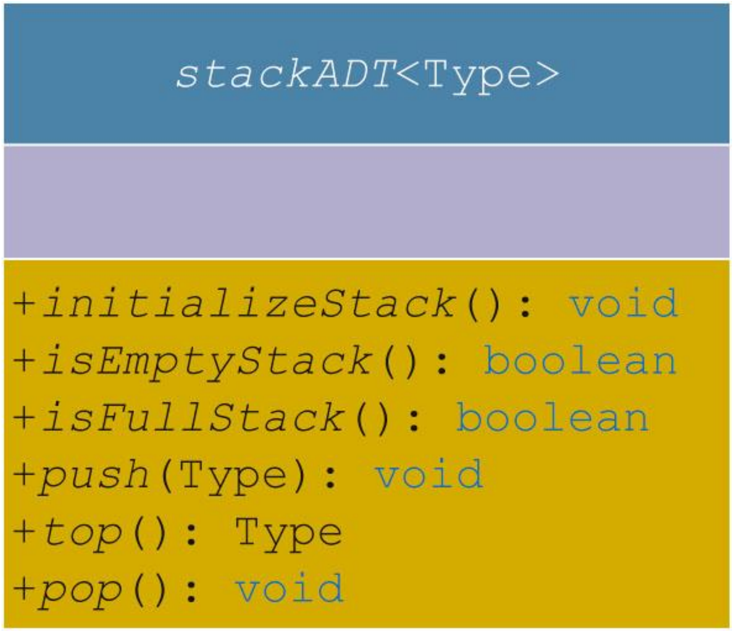

FIGURE 17-4 UML class diagram of the **class** stackADT

Implementation of Stacks as Arrays

- First element goes in first array position, second in the second position, etc.
- Top of the stack is index of the last element added to the stack
- Stack elements are stored in an array, which is a random access data structure
  - Stack element is accessed only through top
- To track the top position, use a variable called `stackTop`

Implementation of Stacks as Arrays (cont'd.)

- Can dynamically allocate array
  - Enables user to specify size of the array
- `class stackType` implements the functions of the abstract `class stackADT`

Implementation of Stacks as Arrays (cont'd.)

```
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 17-5 UML class diagram of the **class** stackType

Implementation of Stacks as Arrays (cont'd.)

- C++ arrays begin with the index 0
  - Must distinguish between:
    - Value of `stackTop`
    - Array position indicated by `stackTop`
- If `stackTop` is 0, stack is empty
- If `stackTop` is nonzero, stack is not empty
  - Top element is given by `stackTop - 1`

Implementation of Stacks as Arrays (cont'd.)

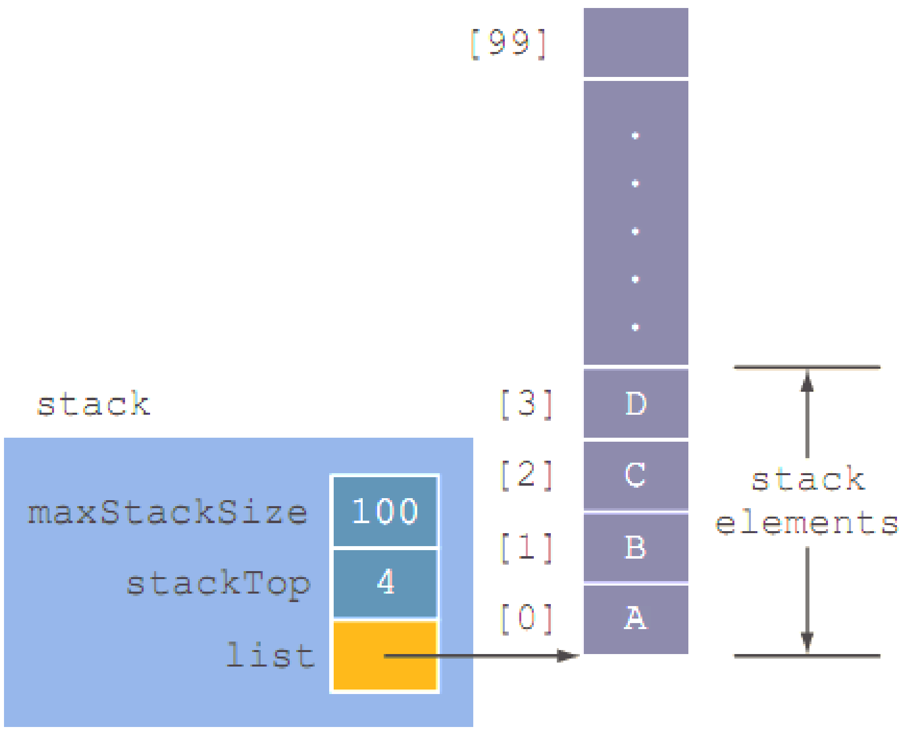

FIGURE 17-6 Example of a stack

Initialize Stack

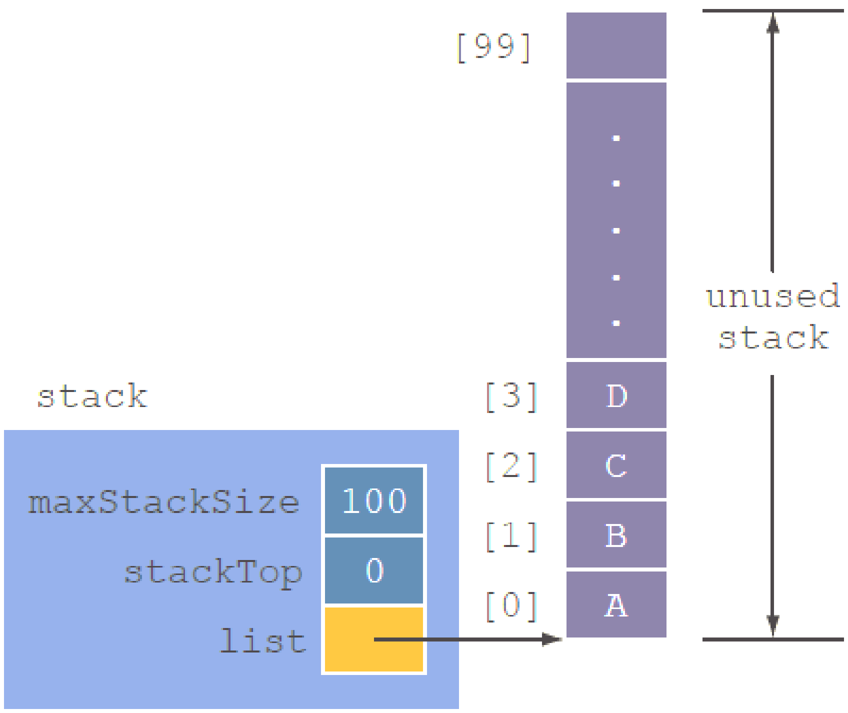

FIGURE 17-7 Empty stack

Empty Stack/Full Stack

- Stack is empty if `stackTop` = 0

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

- Stack is full if `stackTop` = `maxStackSize`

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

Push

- Store the `newItem` in the array component indicated by `stackTop`
- Increment `stackTop`
- <u>Overflow</u> occurs if we try to add a new item to a full stack

Push (cont'd.)

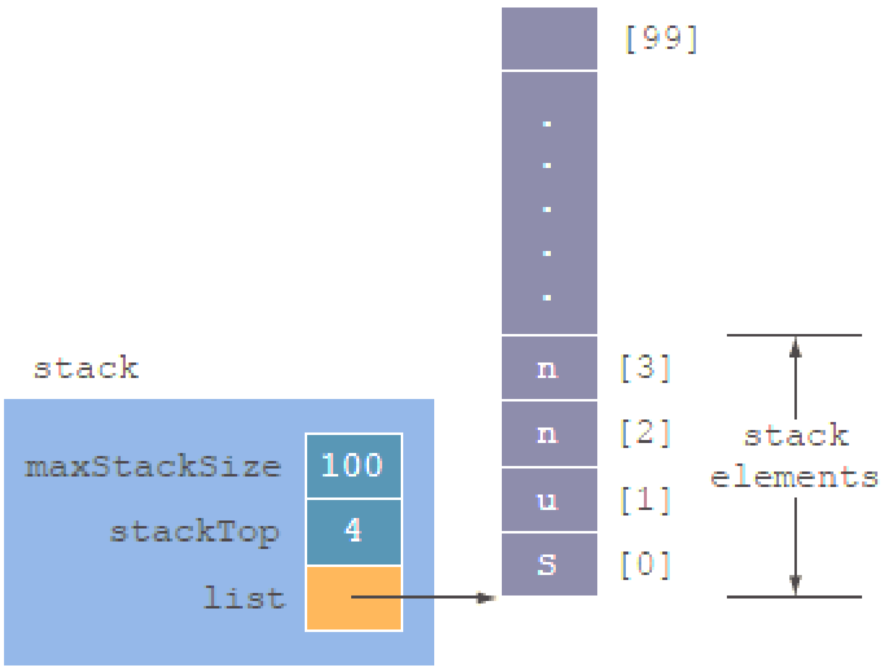

FIGURE 17-8 Stack before pushing y

Push (cont'd.)

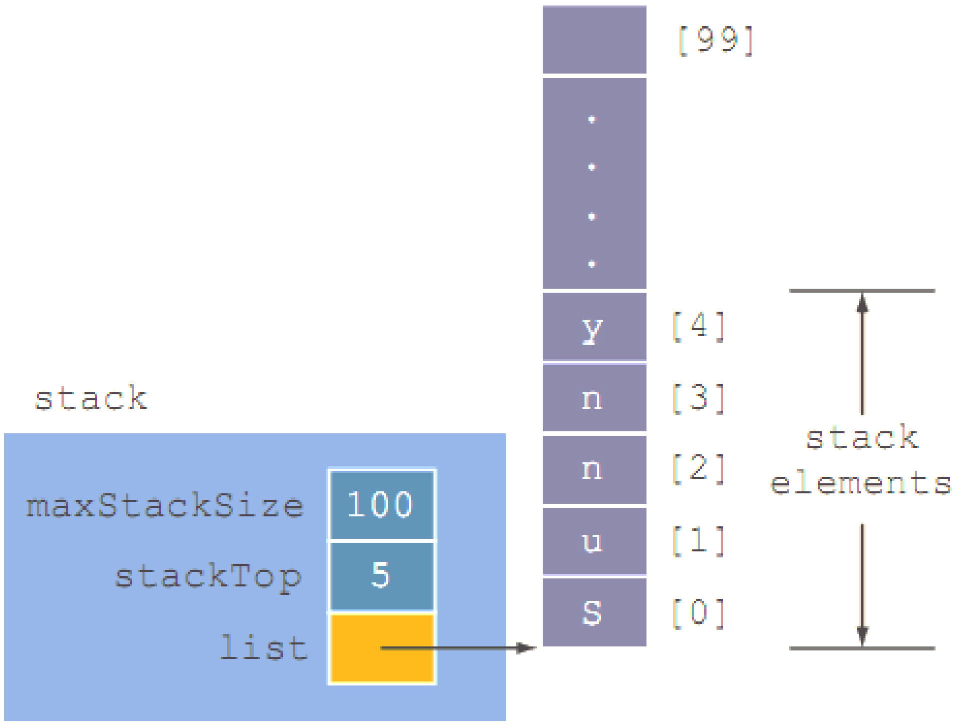

FIGURE 17-9 Stack after pushing y

Return the Top Element

- `top` operation:
  - Returns the top element of the stack

```
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

- To remove an element from the stack, decrement `stackTop` by `1`
- Underflow condition: trying to remove an item from an empty stack

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

Pop (cont'd.)

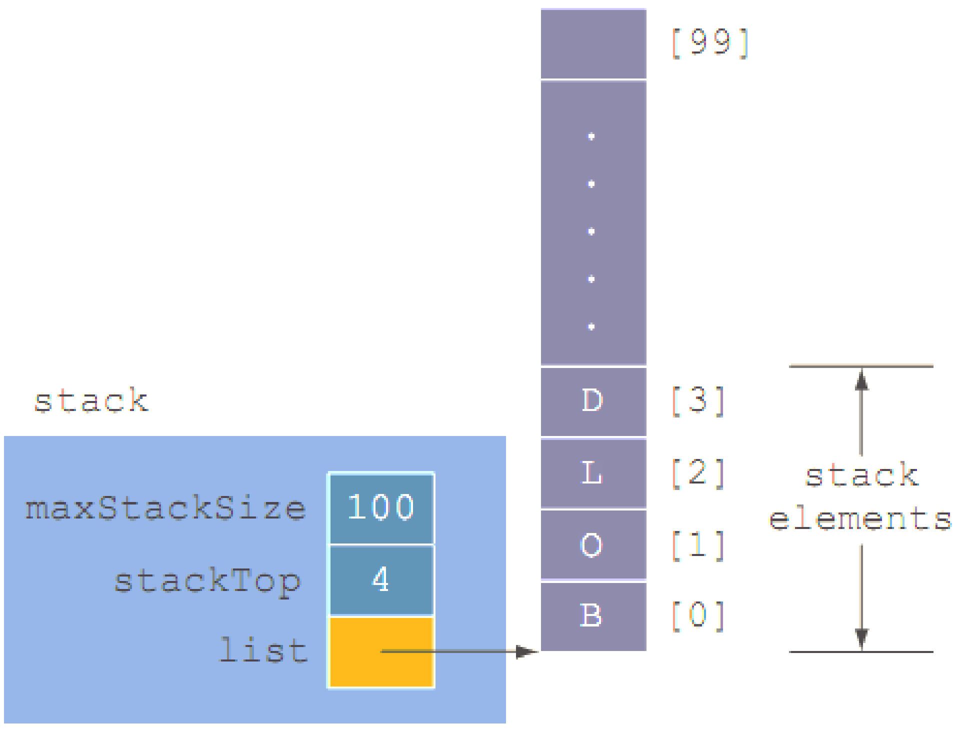

FIGURE 17-10 Stack before popping D

Pop (cont'd.)

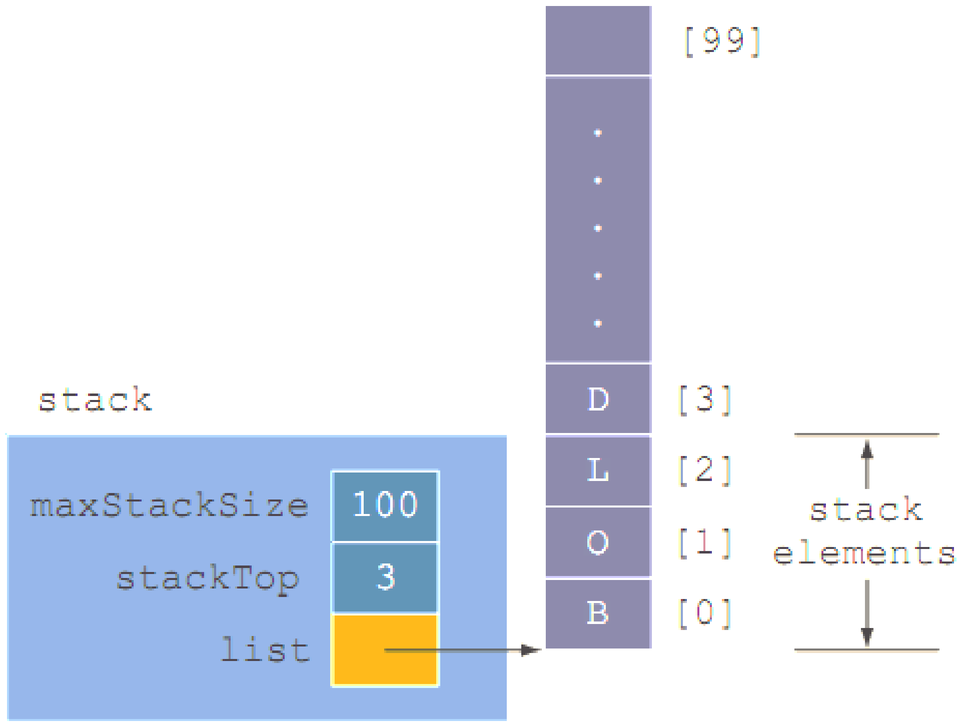

FIGURE 17-11 Stack after popping D

Copy Stack

- `copyStack` function: copies a stack

```
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

- Constructor:
  - Sets stack size to parameter value (or default value if not specified)
  - Sets `stackTop` to 0
  - Creates array to store stack elements
- Destructor:
  - Deallocates memory occupied by the array
  - Sets `stackTop` to 0

Copy Constructor

- Copy constructor:
  - Called when a stack object is passed as a (value) parameter to a function
  - Copies values of member variables from actual parameter to formal parameter

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

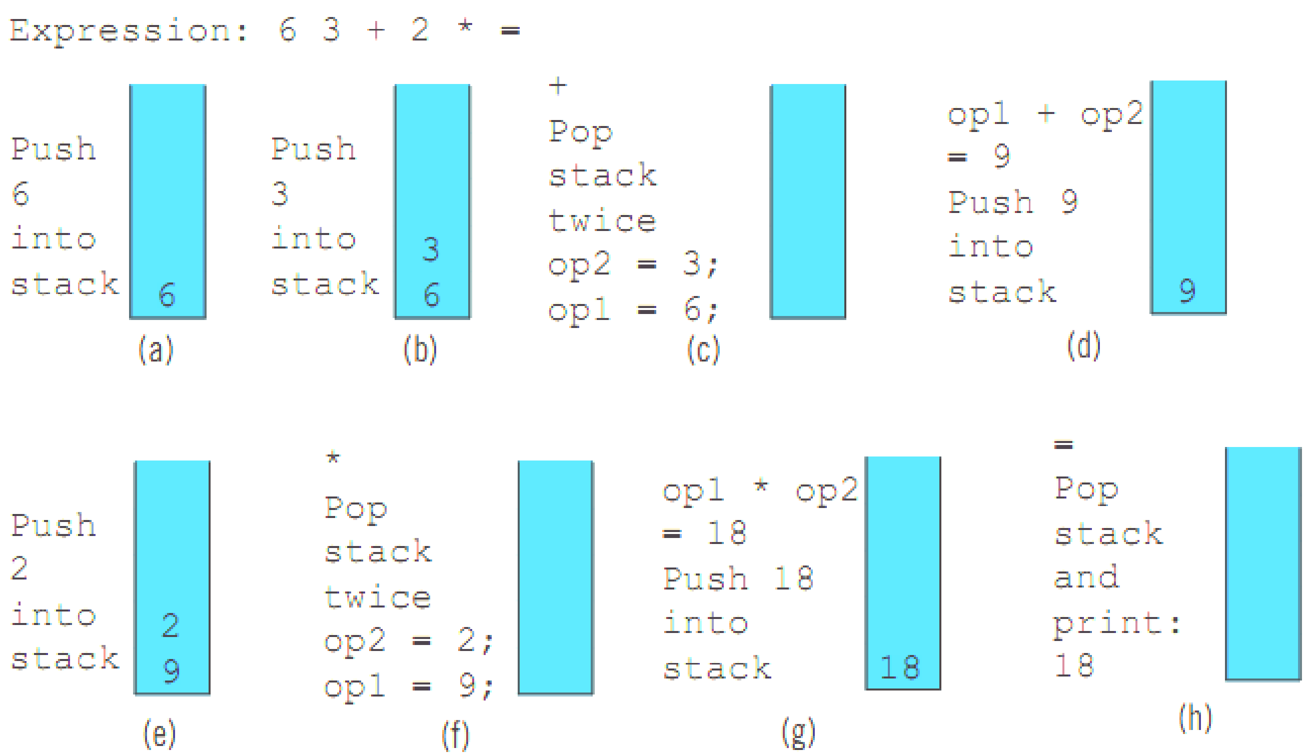

FIGURE 17-17 Evaluating the postfix expression: 6 3 + 2 * =

Queues

- <u>Queue</u>: set of elements of the same type
- Elements are:
  - Added at one end (the <u>back</u> or <u>rear</u>)
  - Deleted from the other end (the <u>front</u>)
- First In First Out (FIFO) data structure
  - Middle elements are inaccessible
- Example:
  - Waiting line in a bank

Queue Operations

- Queue operations include:
  - `initializeQueue`
  - `isEmptyQueue`
  - `isFullQueue`
  - `front`
  - `back`
  - `addQueue`
  - `deleteQueue`
- Abstract `class queueADT` defines these operations

Implementation of Queues as Arrays

- Need at least four (member) variables:
  - Array to store queue elements
  - `queueFront` and `queueRear`
    - To track first and last elements
  - `maxQueueSize`
    - To specify maximum size of the queue

Implementation of Queues as Arrays (cont'd.)

- To add an element to the queue:
  - Advance `queueRear` to next array position
  - Add element to position pointed by `queueRear`

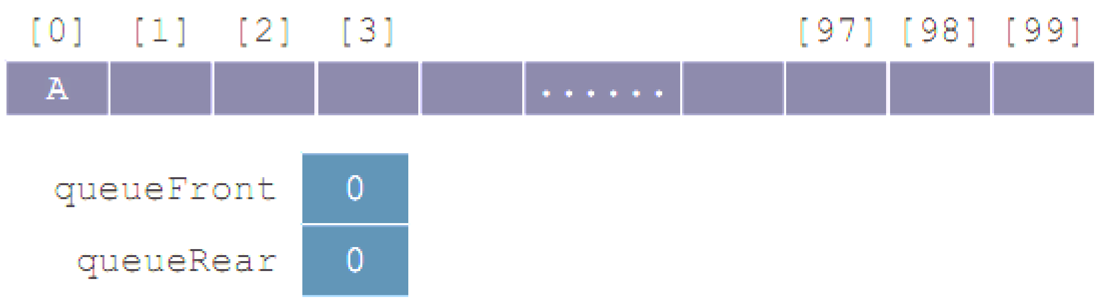

FIGURE 17-26 Queue after the first `addQueue` operation

Implementation of Queues as Arrays (cont'd.)

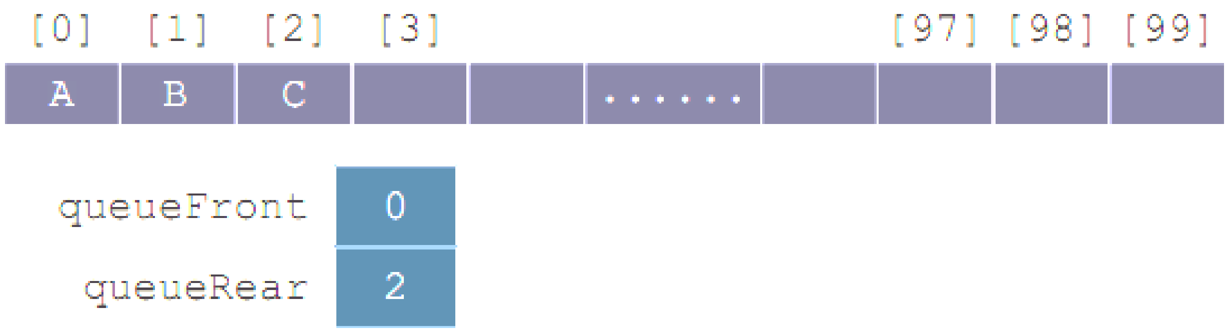

FIGURE 17-27 Queue after two more `addQueue` operations

Implementation of Queues as Arrays (cont'd.)

- To delete an element from the queue:
  - Retrieve element pointed to by `queueFront`
  - Advance `queueFront` to next queue element

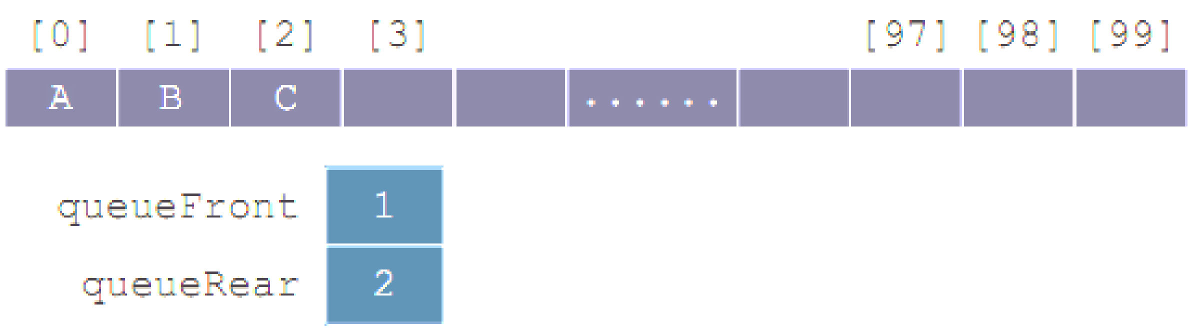

FIGURE 17-28 Queue after the `deleteQueue` operation

Implementation of Queues as Arrays (cont'd.)

- Will this queue design work?
  - Let A represent adding an element to the queue
  - Let D represent deleting an element from the queue
  - Consider the following sequence of operations:
    - `AAADADADADADADADA...`

Implementation of Queues as Arrays (cont'd.)

- This would eventually set `queueRear` to point to the last array position
  - Giving the impression that the queue is full

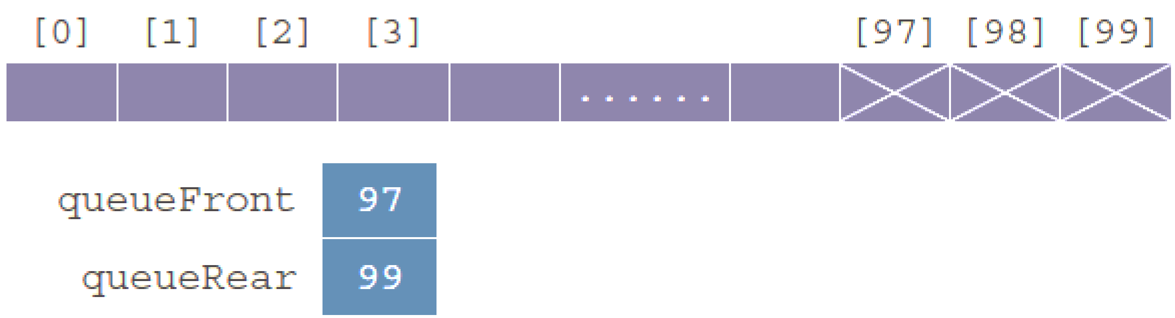

FIGURE 17-29 Queue after the sequence of operations AAADADADADADA. . .

Implementation of Queues as Arrays (cont'd.)

- Solution 1: When queue overflows at rear (`queueRear` points to the last array position):
  - Check value of `queueFront`
  - If `queueFront` indicates there is room at front of array, slide all queue elements toward the first array position
- Problem: too slow for large queues
- Solution 2: Assume that the array is circular

Implementation of Queues as Arrays (cont'd.)

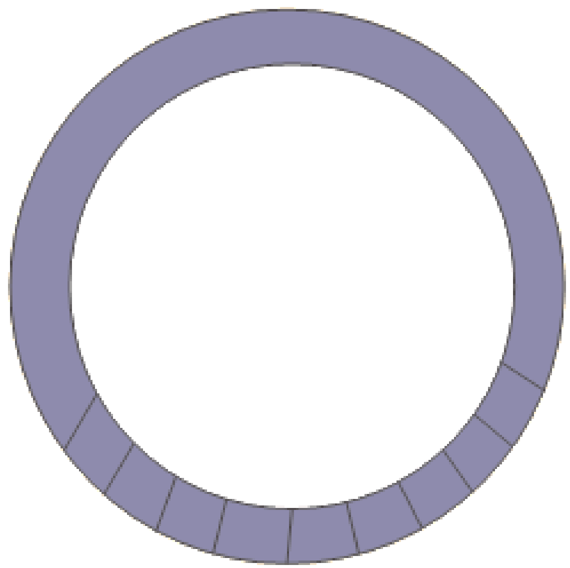

FIGURE 17-30 Circular queue

Implementation of Queues as Arrays (cont'd.)

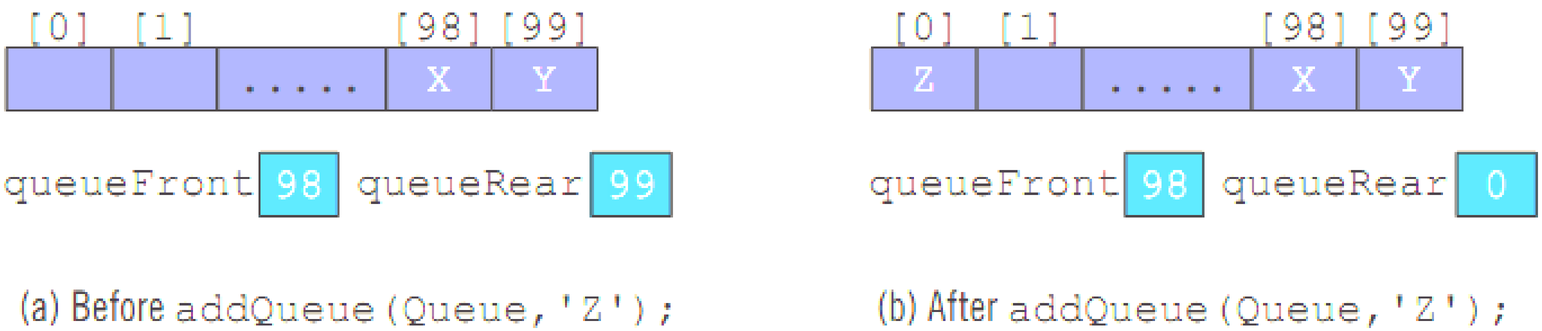

FIGURE 17-31 Queue before and after the add operation

Implementation of Queues as Arrays (cont'd.)

- Deletion Case 1:

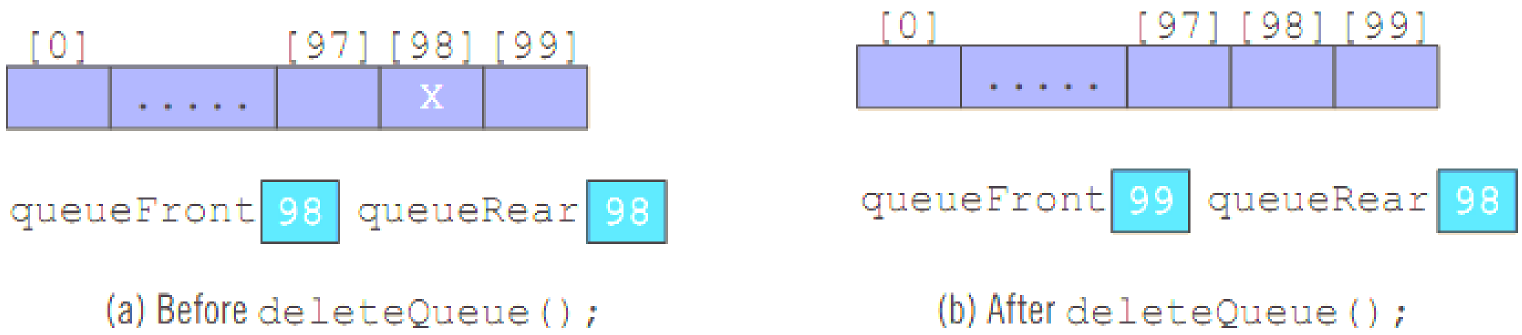

FIGURE 17-32 Queue before and after the delete operation

Implementation of Queues as Arrays (cont'd.)

- Deletion Case 2:

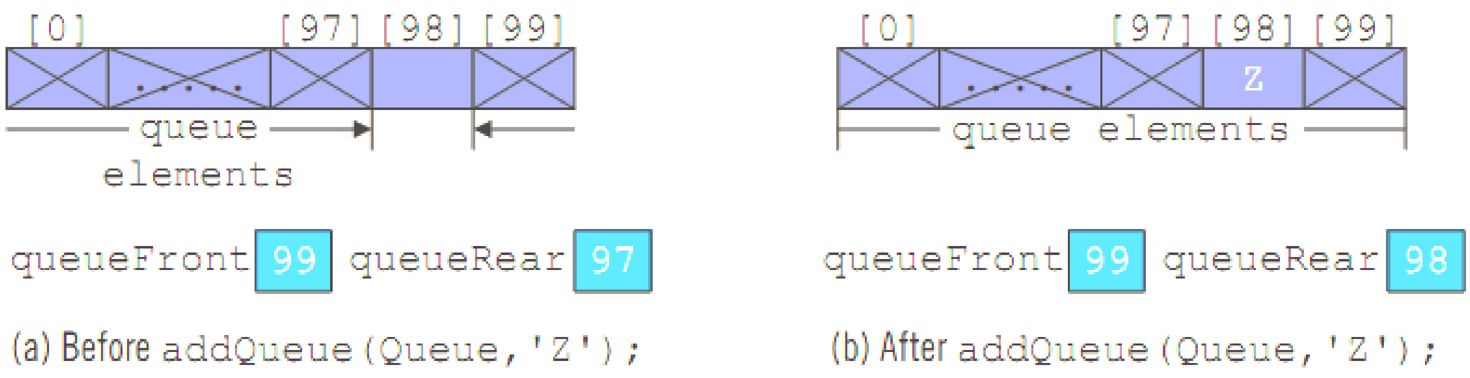

FIGURE 17-33 Queue before and after the add operation