Stacks and Queues

Chris Kiekintveld

CS 2401 (Fall 2010)

Elementary Data Structures and Algorithms

Two New ADTs

- Define two new abstract data types
  - Both are restricted lists
  - Can be implemented using arrays or linked lists
- Stacks
  - "Last In First Out" (LIFO)
- Queues
  - "First In First Out" (FIFO)

Stacks

- List of the same kind of elements
  - Addition and deletion of elements occur only at one end, called the top of the stack
- Computers use stacks to implement method calls
- Stacks are also used to convert recursive algorithms into nonrecursive algorithms

Conceptual Stacks

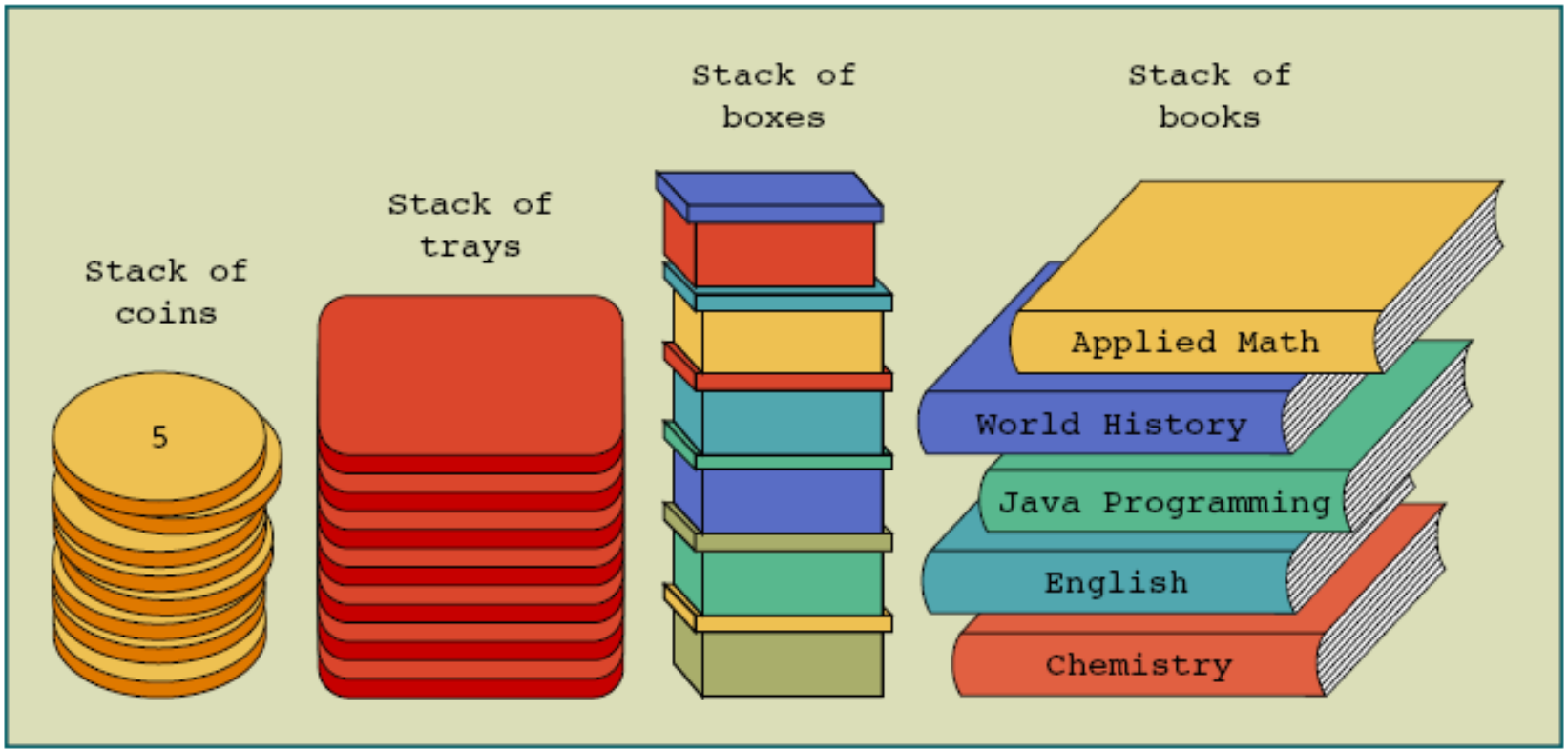

Figure 17-1 Various types of stacks

Stacks (continued)

- Stacks are also called Last Input First Output (LIFO) data structures
- Operations performed on stacks
  - Push: adds an element to the stack
  - Pop: removes an element from the stack
  - Peek: looks at the top element of the stack

Stacks (continued)

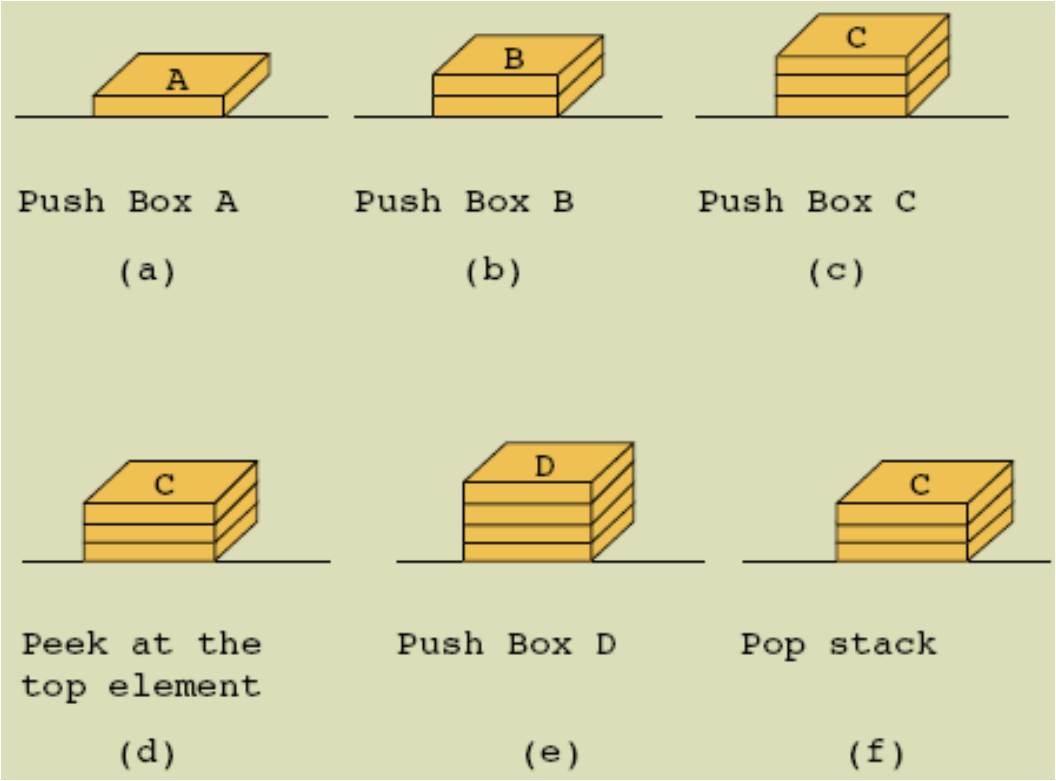

Figure 17-3 Stack operations

Stacks (continued)

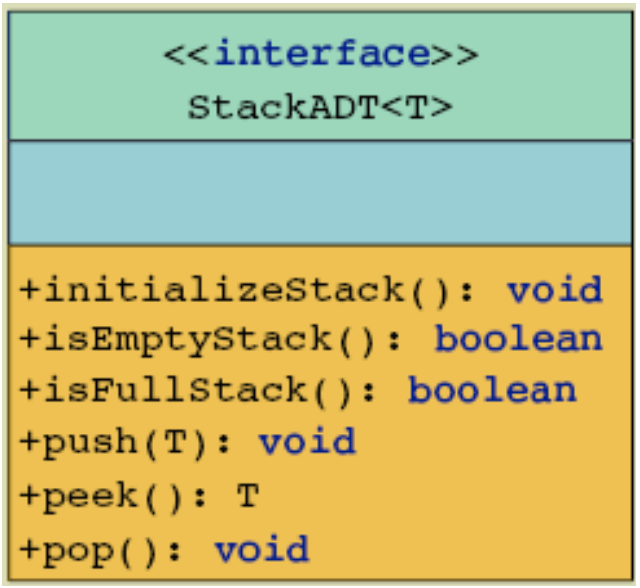

Figure 17-4 UML diagram of the `interface StackADT`

`StackException` Class

- Adding an element to a full stack and removing an element from an empty stack would generate errors or exceptions
  - Stack overflow exception
  - Stack underflow exception
- Classes that handle these exceptions
  - `StackException extends RunTimeException`
  - `StackOverflowException extends StackException`
  - `StackUnderflowException extends StackException`

Implementation of Stacks as Arrays

- The array implementing a stack is an array of reference variables
- Each element of the stack can be assigned to an array slot
- The top of the stack is the index of the last element added to the stack
- To keep track of the top position, declare a variable called `stackTop`

Implementation of Stacks as Arrays (continued)

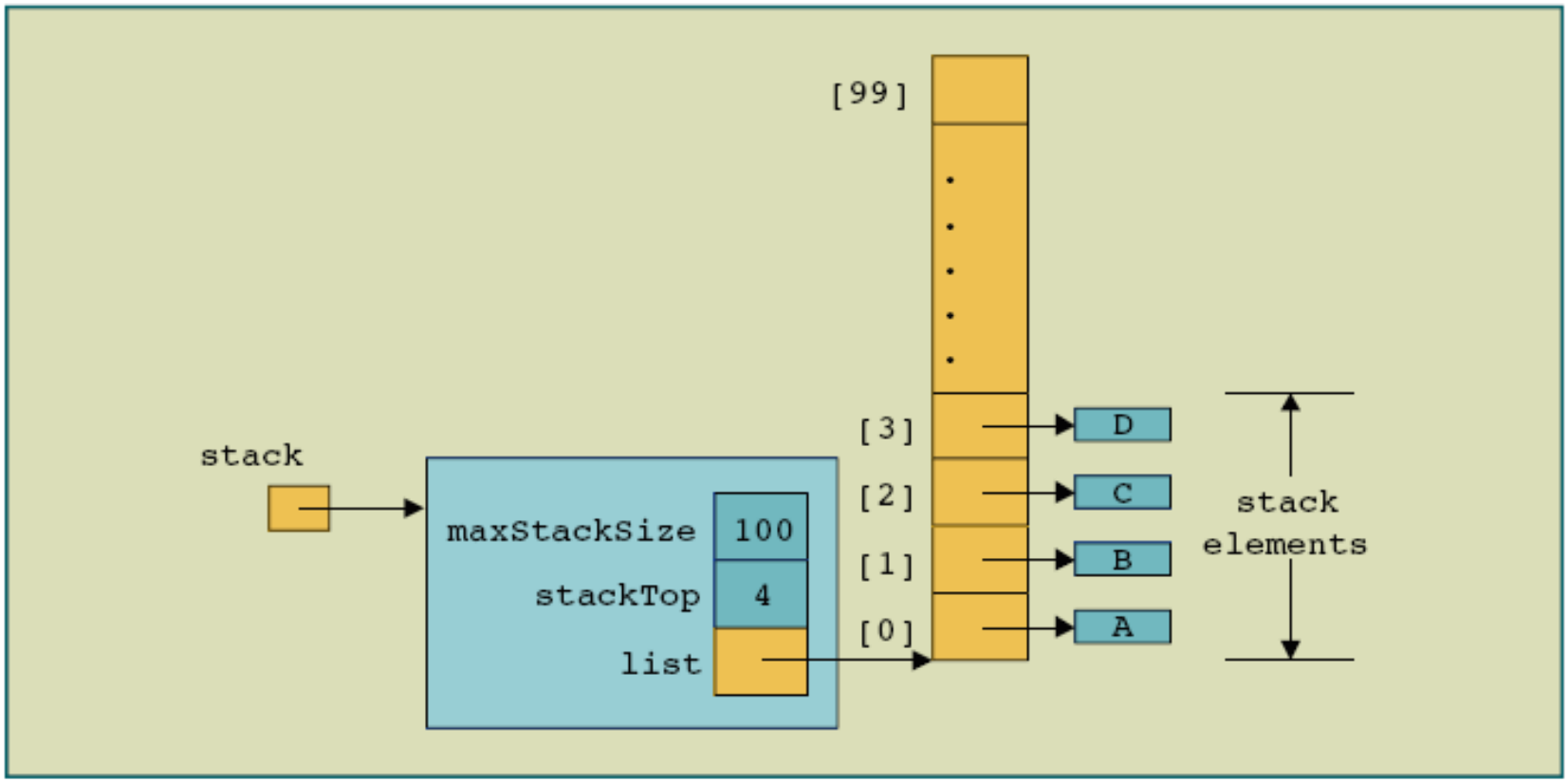

Figure 17-6 Example of a stack

Constructors

- Default constructor

```
public StackClass()
{
    maxStackSize = 100;
    stackTop = 0;          //set stackTop to 0
    list = (T[]) new Object[maxStackSize]; //create the array
}//end default constructor
```

Initialize Stack

- Method `initializeStack`

```
public void initializeStack()
{
    for (int i = 0; i < stackTop; i++)
        list[i] = null;
    stackTop = 0;
}//end initializeStack
```

Empty Stack

- Method `isEmptyStack`

```
public boolean isEmptyStack()
{
    return (stackTop == 0);
}//end isEmptyStack
```

Full Stack

- Method `isFullStack`

```
public boolean isFullStack()
{
    return (stackTop == maxStackSize);
}//end isFullStack
```

Push

- Method `push`

```
public void push(T newItem) throws StackOverflowException
{
    if (isFullStack())
        throw new StackOverflowException();
    list[stackTop] = newItem; //add newItem at the
                              //top of the stack
    stackTop++;               //increment stackTop
}//end push
```

Peek

- Method `peek`

```
public T peek() throws StackUnderflowException
{
    if (isEmptyStack())
        throw new StackUnderflowException();
    return (T) list[stackTop - 1];
}//end peek
```

Pop

- Method `pop`

```
public void pop() throws StackUnderflowException
{
    if (isEmptyStack())
       throw new StackUnderflowException();
    stackTop--;        //decrement stackTop
    list[stackTop] = null;
}//end pop
```

Linked Implementation of Stacks

- Arrays have fixed sizes
  - Only a fixed number of elements can be pushed onto the stack
- Dynamically allocate memory using reference variables
  - Implement a stack dynamically
- Similar to the array representation, `stackTop` is used to locate the top element
  - `stackTop` is now a reference variable

Linked Implementation of Stacks (continued)

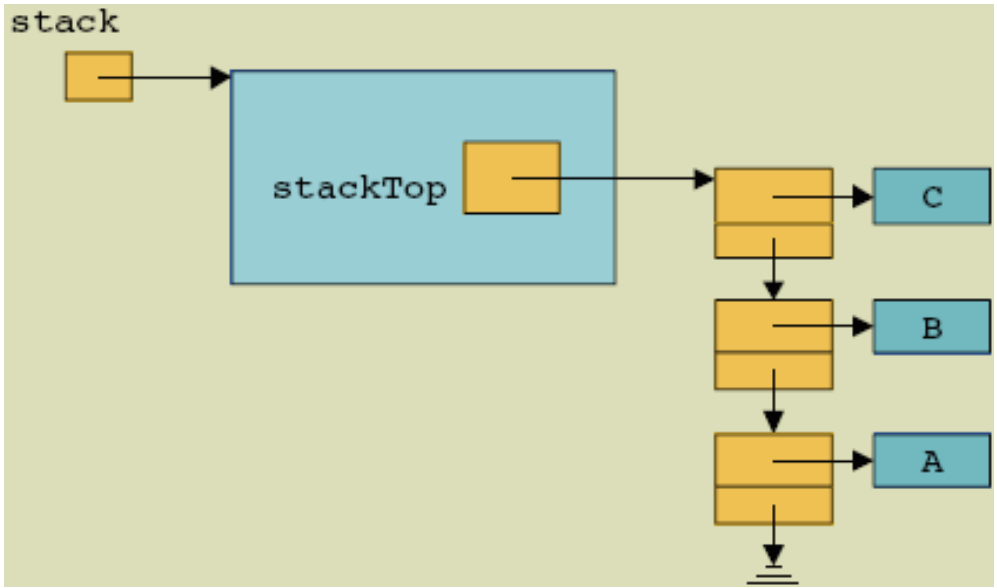

Figure 17-13 Nonempty linked stack

Default Constructor

- Default constructor

```
public LinkedStackClass()
{
    stackTop = null;
}//end constructor
```

Initialize Stack

- Method `initializeStack`

```
public void initializeStack()
{
    stackTop = null;
}//end initializeStack
```

Empty Stack and Full Stack

- Methods `isEmptyStack` and `isFullStack`

```
public boolean isEmptyStack()
{
    return (stackTop == null);
}//end isEmptyStack

public boolean isFullStack()
{
    return false;
}//end isFullStack
```

Push

- Method `push`

```
public void push(T newElement)
{
    StackNode<T> newNode; //reference variable to create
                          //the new node
    newNode = new
              StackNode<T>(newElement, stackTop); //create
                                                  //newNode and insert
                                                  //before stackTop
    stackTop = newNode;      //set stackTop to point to
                             //the top element
} //end push
```

Peek

- Method `peek`

```
public T peek() throws StackUnderflowException
{
    if (stackTop == null)
        throw new StackUnderflowException();
    return stackTop.info;
}//end top
```

Pop

- Method `pop`

```
public void pop() throws StackUnderflowException
{
    if (stackTop == null)
        throw new StackUnderflowException();
    stackTop = stackTop.link;  //advance stackTop to the
                               //next node
}//end pop
```

Applications of Stacks: Postfix Expression Calculator

- Infix notation
  - The operator is written between the operands
- Prefix or Polish notation
  - Operators are written before the operands
  - Does not require parentheses
- Reverse Polish or postfix notation
  - Operators follow the operands
  - Has the advantage that the operators appear in the order required for computation

Applications of Stacks: Postfix Expression Calculator (continued)

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 +

Table 17-1 Infix expressions and their equivalent postfix expressions

Applications of Stacks: Postfix Expression Calculator (continued)

- Algorithm to evaluate postfix expressions
  - Scan the expression from left to right
  - When an operator is found, back up to get the required number of operands
  - Perform the operation
  - Continue processing the expression

Main Algorithm

- Main algorithm in pseudocode for processing a postfix expression

```
Get the next expression
while more data to process
{
    a. initialize the stack
    b. process the expression
    c. output result
    d. get the next expression
}
```

Method `evaluateExpression`

- General algorithm for `evaluateExpression`

```
get the next token
while (token != '=')
{
    if (token is a number)
    {
       output number
       push number into stack
    }
    else
    {
       token is an operation
       call method evaluateOpr to evaluate the operation
    }
    if (no error in the expression)
        get next token
    else
        discard the expression
}
```

Method `evaluateOpr`

- This method evaluates an operation
- Two operands are needed to evaluate an operation
- Operands are stored in the stack
  - The stack must contain at least two operands
  - Otherwise, the operation cannot be evaluated

Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward

- Naïve approach
  - Get the last node of the list and print it
    - Traverse the link starting at the first node
  - Repeat this process for every node
    - Traverse the link starting at the first node until you reach the desired node
  - Very inefficient

Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward (continued)

```
current = first;                    //Line 1
while (current != null)             //Line 2
{
    stack.push(current);            //Line 3
    current = current.link;         //Line 4
}
```

Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward (continued)

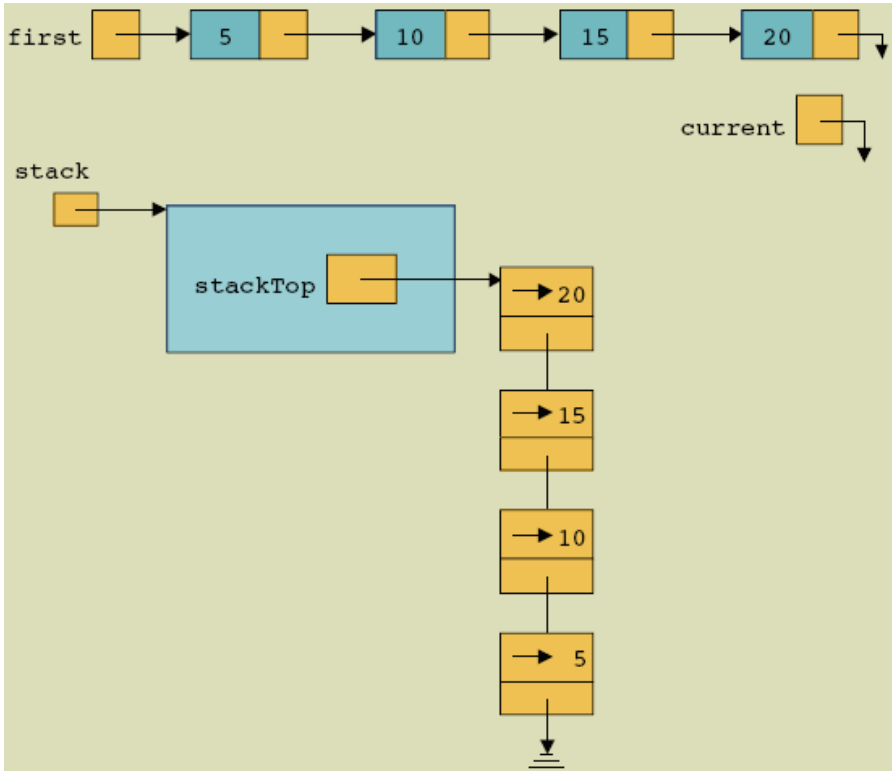

Figure 17-36 List and stack after the staments `stack.push(current);` and `current = current.link;` executes

The `class Stack`

```
public Stack()
//Creates an empty Stack object.
```

```
public boolean empty()
//Returns true is the stack is empty; otherwise returns false.
```

```
public Object peek() throws EmptyStackException
//Returns the top element of the stack. It does not remove the
//top element.
```

```
public Object pop() throws EmptyStackException
//Returns and removes the top element of the stack.
```

```
public Object push(Object obj)
//Pushes the item specified by obj onto the stack.
```

```
public int search(Object obj)
//Returns the relative position of the item specified by obj
//from the top of the stack. If the item is not in the stack,
//it returns -1.
```

Table 17-2 Members of the `class Stack`

Queues

- Data structure in which the elements are added at one end, called the rear, and deleted from the other end, called the front
- A queue is a First In First Out data structure
- As in a stack, the middle elements of the queue are inaccessible

Queue Operations

- Queue operations
  - `initializeQueue`
  - `isEmptyQueue`
  - `isFullQueue`
  - `front`
  - `back`
  - `addQueue`
  - `deleteQueue`

`QueueException` Class

- Adding an element to a full queue and removing an element from an empty queue would generate errors or exceptions
  - Queue overflow exception
  - Queue underflow exception
- Classes that handle these exceptions
  - `QueueException extends RunTimeException`
  - `QueueOverflowException extends QueueException`
  - `QueueUnderflowException extends QueueException`

Implementation of Queues as Arrays

- Instance variables
  - An array to store the queue elements
  - `queueFront`: keeps track of the first element
  - `queueRear`: keeps track of the last element
  - `maxQueueSize`: specifies the maximum size of the queues

Implementation of Queues as Arrays (continued)

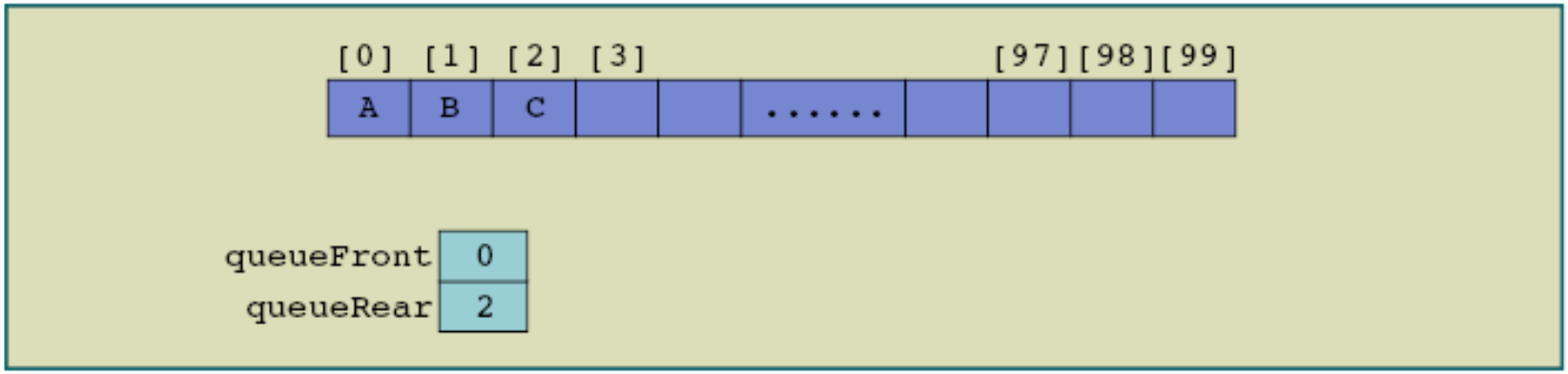

Figure 17-42 Queue after two more `addQueue` operations

Implementation of Queues as Arrays (continued)

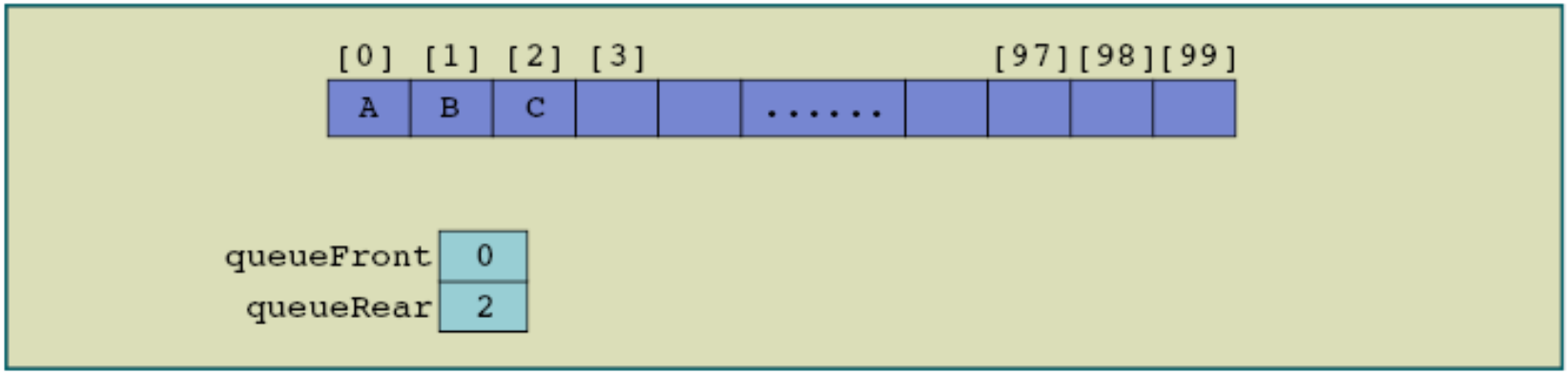

Figure 17-43 Queue after the `deleteQueue` operation

Implementation of Queues as Arrays (continued)

- Problems with this implementation
  - Arrays have fixed sizes
  - After various insertion and deletion operations, `queueRear` will point to the last array position
    - Giving the impression that the queue is full
- Solutions
  - Slide all of the queue elements toward the first array position
  - Use a circular array

Implementation of Queues as Arrays (continued)

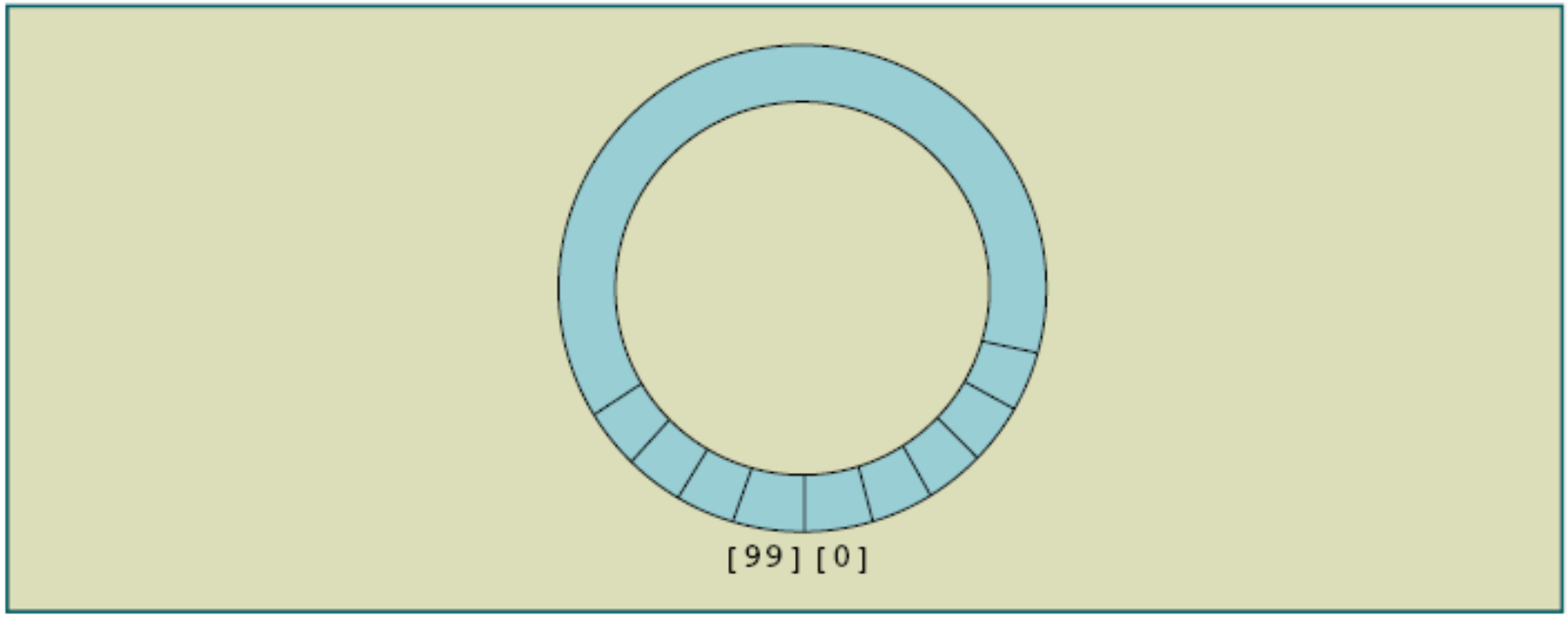

Figure 17-45 Circular queue

Implementation of Queues as Arrays (continued)

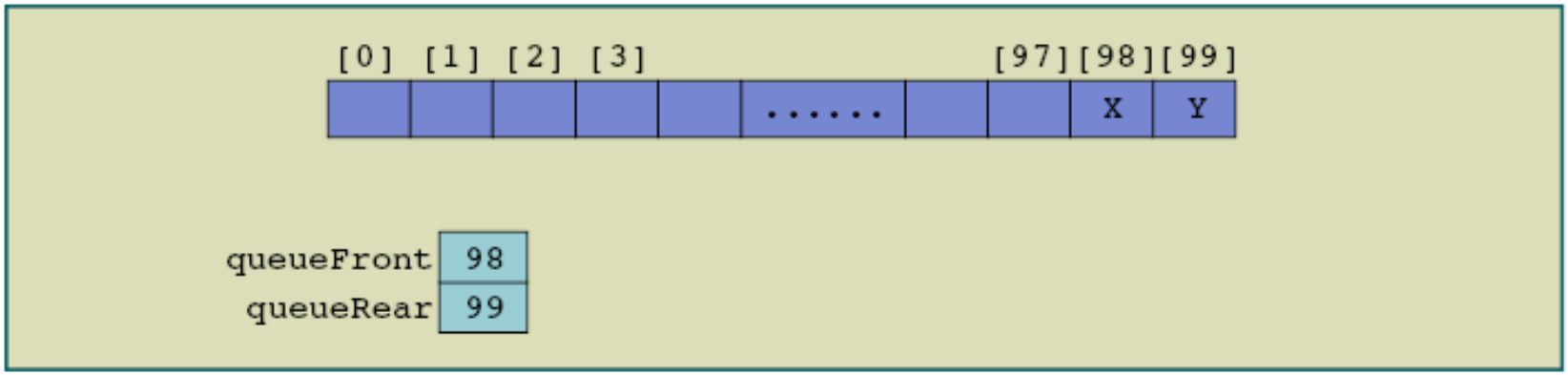

Figure 17-46 Queue with two elements at positions `98` and `99`

Implementation of Queues as Arrays (continued)

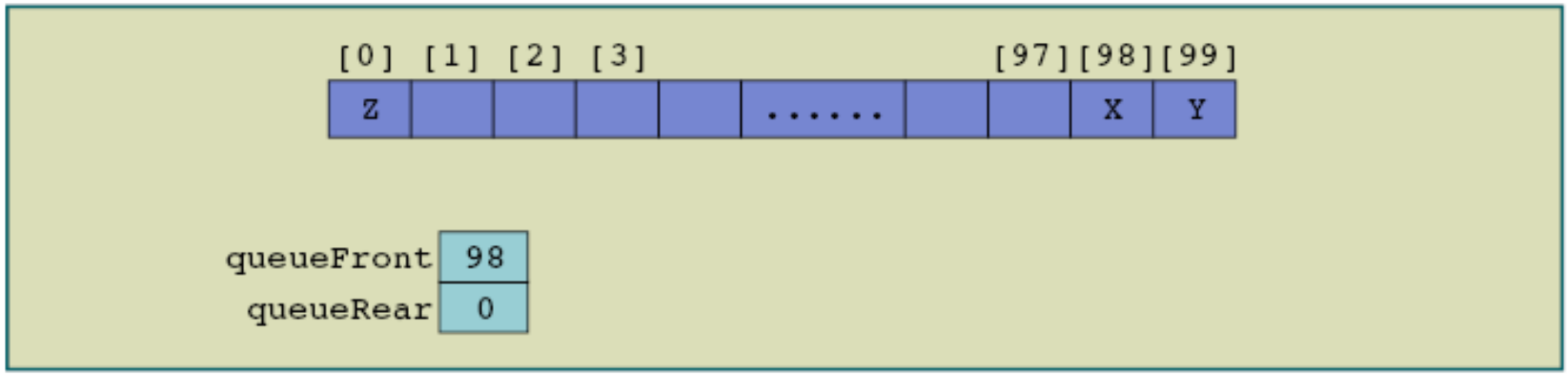

Figure 17-47 Queue after one more `addQueue` operation

Constructors

- Default constructor

```
//Default constructor
public QueueClass()
{
    maxQueueSize = 100;
    queueFront = 0;                  //initialize queueFront
    queueRear = maxQueueSize - 1;    //initialize queueRear
    count = 0;
    list = (T[]) new Object[maxQueueSize];  //create the
                                 //array to implement the queue
}
```

initializeQueue

- Method initilializeQueue

```
public void initializeQueue()
{
    for (int i = queueFront; i < queueRear;
                             i = (i + 1) % maxQueueSize)
        list[i] = null;
    queueFront = 0;
    queueRear = maxQueueSize - 1;
    count = 0;
}
```

Empty Queue and Full Queue

- Methods `isEmptyQueue` and `isFullQueue`

```
public boolean isEmptyQueue()
{
    return (count == 0);
}

public boolean isFullQueue()
{
    return (count == maxQueueSize);
}
```

Front

- Method `front`

```
public T front() throws QueueUnderflowException
{
    if (isEmptyQueue())
        throw new QueueUnderflowException();
    return (T) list[queueFront];
}
```

Back

- Method `back`

```
public T back() throws QueueUnderflowException
{
    if (isEmptyQueue())
        throw new QueueUnderflowException();
    return (T) list[queueRear];
}
```

addQueue

- Method `addQueue`

```
public void addQueue(T queueElement)
                        throws QueueOverflowException
{
    if (isFullQueue())
        throw new QueueOverflowException();
    queueRear = (queueRear + 1) % maxQueueSize; //use the
                         //mod operator to advance queueRear
                         //because the array is circular
    count++;
    list[queueRear] = queueElement;
}
```

deleteQueue

- Method `deleteQueue`

```
public void deleteQueue() throws QueueUnderflowException
{
    if (isEmptyQueue())
        throw new QueueUnderflowException();
    count--;
    list[queueFront] = null;
    queueFront = (queueFront + 1) % maxQueueSize; //use the
                        //mod operator to advance queueFront
                        //because the array is circular
}
```

Linked Implementation of Queues

- Simplifies many of the special cases of the array implementation
- Because the memory to store a queue element is allocated dynamically, the queue is never full
- Class `LinkedQueueClass` implements a queue as a linked data structure
  - It uses nodes of type `QueueNode`

Linked Implementation of Queues (continued)

- Method `initializeQueue`

```
public void initializeQueue()
{
    queueFront = null;
    queueRear = null;
}
```

Linked Implementation of Queues (continued)

- Methods `isEmptyQueue` and `isFullQueue`

```
public boolean isEmptyQueue()
{
     return (queueFront == null);
}

public boolean isFullQueue()
{
     return false;
}
```

addQueue

- Method `addQueue`

```
public void addQueue(T newElement)
{
    QueueNode<T> newNode;
    newNode = new QueueNode<T>(newElement, null); //create
                               //newNode and assign newElement to newNode
    if (queueFront == null) //if initially the queue is empty
    {
        queueFront = newNode;
        queueRear = newNode;
    }
    else   //add newNode at the end
    {
        queueRear.link = newNode;
        queueRear = queueRear.link;
    }
}//end addQueue
```

front and back

- Methods front and back

```
public T front() throws QueueUnderflowException
{
    if (isEmptyQueue())
        throw new QueueUnderflowException();
    return queueFront.info;
}

public T back() throws QueueUnderflowException
{
    if (isEmptyQueue())
        throw new QueueUnderflowException();
    return queueRear.info;
}
```

deleteQueue

- Method `deleteQueue`

```
public void deleteQueue() throws QueueUnderflowException
{
    if (isEmptyQueue())
        throw new QueueUnderflowException();
    queueFront = queueFront.link; //advance queueFront
    if (queueFront == null)  //if after deletion the queue
        queueRear = null;  //is empty, set queueRear to null
} //end deleteQueue
```