Chapter 21 : Stacks and Queues

Starting Out with Java
From Control Structures through Data Structures

by Tony Gaddis and Godfrey Muganda

Chapter Topics

- Stacks and Their Applications
- Array Implementation of Stacks
- Linked Implementation of Stacks
- Queues and Their Applications
- Array Implementation of Queues
- Linked Implementation of Queues

Stacks

A stack is a collection of items that allows the following operations:

- boolean empty() checks if the stack is empty.
- void push(E o) adds an object o to the stack.
- E pop() removes and returns the item most recently added to the stack.

Last In First Out

A stack is a Last in-First-Out container: the last item added is the first to be removed.

The Stack peek() Operation

Most stacks support a

E peek()

operation: this returns, but does not remove from the stack, the item most recently added to the stack

peek() returns the item that will be removed by the next call to pop()

Applications of Stacks

Many collections of items are naturally regarded as stacks:

- A stack of plates in a cafeteria.
- Cars packed in a narrow driveway.
- Return addresses of method calls in an executing program.

Plates in a Cafeteria

- Plates are stacked in a column.
- A new plate is added to the top of the column of plates.
- Plates are removed from the top of the column of plates, so that the last plate added is the first removed.

Method Return Addresses

- Programs often make chains of method calls.
- The last method called is the first to return.
- The compiler uses a stack to maintain the list of return addresses for executing methods in a stack.

Java Collection Framework Stack Class

The JCF provides a generic implementation of a stack:

- Stack<E>() (constructor)
- E push(E element)
- E pop()
- E peek()
- boolean empty()

The JCF Stack Class

- The JCF stack can be used to create stacks of String, numeric values, or any other types.
- Following example demonstrates the use of the JCF Stack class to work with strings.

The JCF Stack Class

```
import java.util.Stack;
 public class StackDemo
 {
    public static void main(String [] args)
   {
       // Create a stack of strings and add some names
       Stack<String> stack = new Stack<String>();
       String [ ] names = {"Al", "Bob", "Carol"};
       System.out.println("Pushing onto the stack the names:");
       System.out.println("Al Bob Carol");
       for (String s : names)
          stack.push(s);

       // Now pop and print everything on the stack
       String message = "Popping and printing all stack values:";
       System.out.println(message);
       while (! stack.empty() )
          System.out.print( stack.pop() + " " );
    }
}
```

Stacks of Primitive Values

- The JCF Stack class does not support stacks of primitive types.
- To create a stack that can store a primitive type, create a stack of the corresponding wrapper type.

Stack of Primitive Types

These statements will not compile:

```
Stack<int> myIntStack;
Stack<boolean> myBoolStack;
```

Use stacks of the corresponding wrapper class types:

```
Stack<Integer> myIntStack;
Stack<Boolean> myBoolStack;
```

Autoboxing

Once a stack of a wrapper type has been declared and instantiated, you can use the primitive type in all stack methods.

The compiler automatically boxes primitive values passed as parameters to the stack methods to form the required wrapper class types.

Unboxing

The compiler automatically unboxes wrapper types returned as values by the stack methods to form the corresponding

primitive value.

Autoboxing and Unboxing

```
public class StackDemo
 {
   public static void main(String [] args)
   {
      Stack<Integer> intStack = new Stack<Integer>();

      // Push some numbers onto the stack
      for (int k =1 ; k < ; k++)
          intStack.push(k*k);

     // Pop and print all numbers
      while (!intStack.empty())
      {
          int x = intStack.pop();
          System.out.print( x + "  ");
      }
   }
 }
```

Array Implementation of Stacks

Array Implementation of Stacks

A stack can be implemented by using an array to hold the items being stored.

Array Implementation of Stacks

A stack can be implemented by using an array to hold the items being stored.

push can be implemented so it stores items at the end of the array.

Array Implementation of Stacks

A stack can be implemented by using an array to hold the items being stored.

push can be implemented so it stores items at the end of the array.

pop can be implemented so it returns the item at the end of the array.

Array Implementation of Stacks

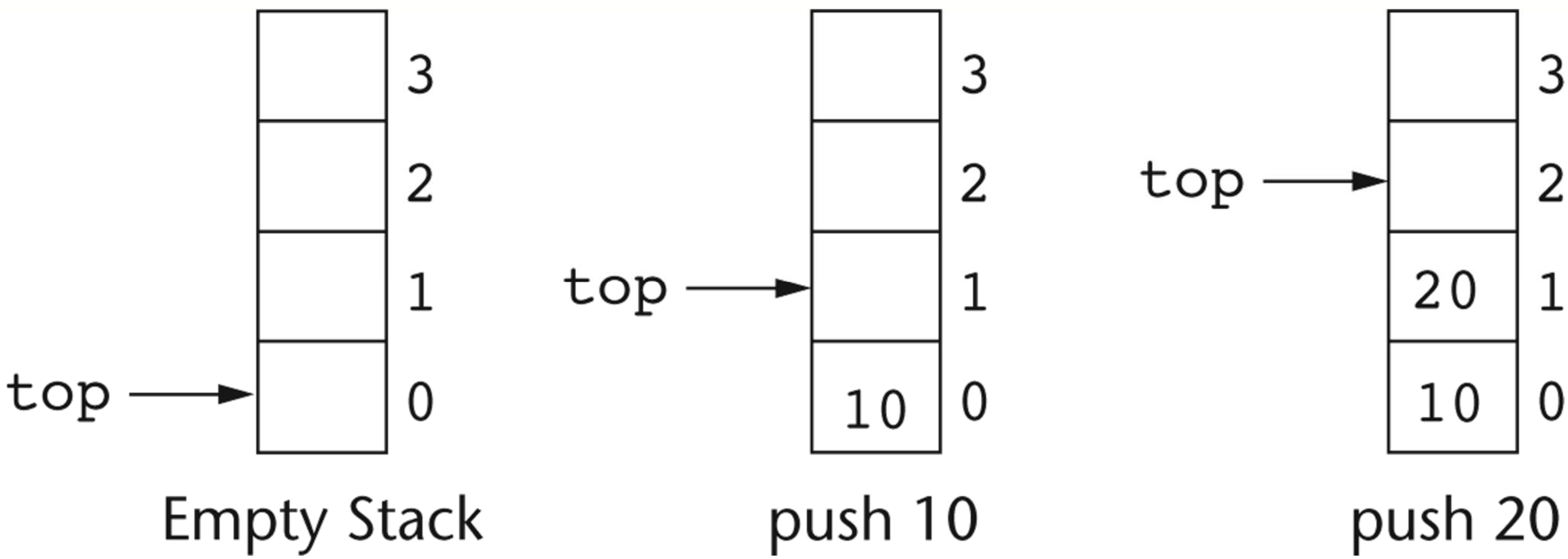
A stack can be implemented by using an array to hold the items being stored.

push can be implemented so it stores items at the end of the array.

pop can be implemented so it returns the item at the end of the array.

A variable top can be used to track the actual end of the array, where the next item will be added.

Pushing Items Onto a Stack



Array Implementation of a Stack

Use a class ArrayStack with private fields:

```
int [ ] s;
int top;   // points to actual end of the stack
```

Constructor:

```
ArrayStack(int capacity)
{
    s = new int[capacity];
    top = 0;
}
```

Array Implementation of a Stack

The method for adding an item to the stack:

```
void push(int x)
{
  if (top == s.length)
     throw new IllegalStateException();
  s[top] = x;
  top ++;
}
```

Array Implementation of a Stack

The method to retrieve an item from the stack:

```
int pop()
{
  if (top == 0)
    then throw new IllegalStateException();
  top --;
  return s[top];
}
```

Stack Overflow and Underflow

The exception thrown when an attempt is made to push a value onto a stack whose array if full is called a stack overflow exception.

Stack Overflow and Underflow

The exception thrown when an attempt is made to push a value onto a stack whose array if full is called a stack overflow exception.

The exception thrown when an attempt is made to pop an empty stack is sometimes called stack underflow.

Checking for an Empty Stack

You check for an empty stack by looking to see if top is 0 :

```
boolean empty()
{
   return top == 0;
}
```

Peek()

To return the item currently at the "top" of the stack:

```
int peek()
{
  if (top ==0 )
    throw new IllegalStateException();
  return s[top-1];
}
```

Array-Based Stacks of Other Types

Implementing stacks of other types is similar to implementing a stack of int.

Main difference: when a value of a reference type is removed from the stack, the array entry must be set to null.

```
s[top-1] = null;
```

Forgetting to set the array entry to null will keep the removed item from being garbage collected.

Popping a Reference Type

```
public String pop()
 {
    if (empty())
       throw new EmptyStackException();
    else
    {
      int retVal = s[top-1];
      s[top-1] = null; // Facilitate garbage collection
      top--;
      return retVal;
    }
 }
```

Linked Implementation of Stacks

A linked list can be used to implement a stack by using the head of the list as the “top” of the stack:

Linked Implementation of Stacks

A linked list can be used to implement a stack by using the head of the list as the “top” of the stack:

push(E o) adds a new item as the new head of the linked list.

Linked Implementation of Stacks

A linked list can be used to implement a stack by using the head of the list as the “top” of the stack:

push(E o) adds a new item as the new head of the linked list.

E pop() removes and returns the head of the linked list.

Implementing a Stack with a Linked List

A reference

```
Node top;    // list head
```

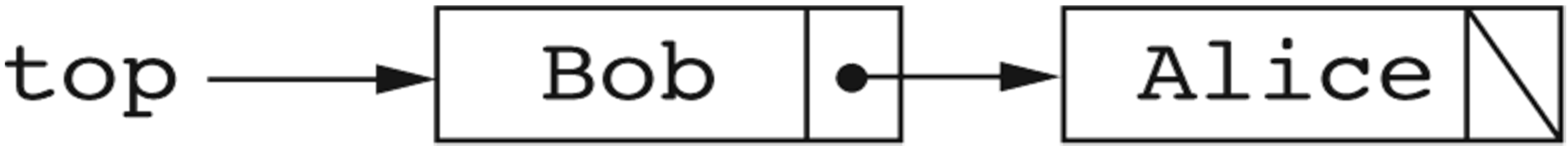
is used to represent the linked list that holds the stack items.

Linked Implementation of a Stack

```
class LinkedStack
{   // Node class used for linked list
    private class Node
    {
       String element;
       Node next;
       Node (String e, Node n)
       {
           element = e;
           next = n;
       }
    }
    private Node top = null;  // head of list is top of stack
}
```

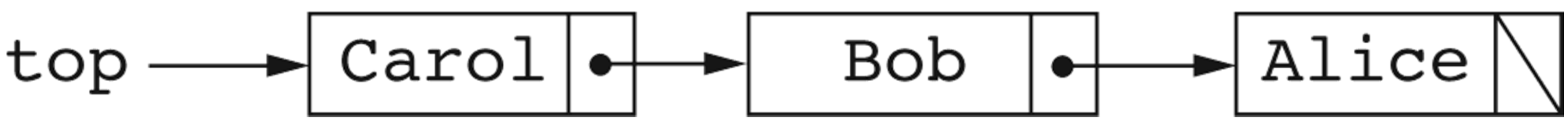
Linked Implementation of a Stack

A linked stack after pushing two items:



Linked Implementation of Stacks

The linked stack of the previous slide after pushing the string Carol.



Linked Implementation of Stack Methods

All of the stack methods have straightforward implementations using linked lists:

- boolean empty()
- String pop()
- void push(String e)
- String peek()

Checking for an Empty Stack

The stack is empty if the head pointer top is null:

```
boolean empty()
{
   return top == null;
}
```

The Stack push Operation

Adds a new item at the beginning of the underlying linked list:

```
void push(String s)
{
    top = new Node(s, top);
}
```

The Stack pop Operation

Removes and returns the element at the head of the stack's linked list:

```
public String pop()
{
    if (empty())
        throw new IllegalStateException();
    else
    {
        String retValue = top.element;
        top = top.next;
        return retValue;
    }
}
```

The Stack peek Operation

Return without removing the element stored in the head of the list.

```
public String peek()
 {
   if (empty())
       throw new IllegalStateException();
    else
      return top.element;
  }
```

Queues

Queues

A Queue is a collection that allows elements to be added and removed in a First-In-First-Out order.

Elements are removed from the queue in the same order in which they are added.

Queue Applications

Many applications have clients that arrive needing service, and depart after being served.

Clients who arrive are added to a queue.

Clients are removed from the queue, receive service, and leave.

Queue Applications

- A print server may service many workstations in a computer network. The print server uses a queue to hold print job requests waiting to be sent to the printer.

- A simulation of cars going through a toll booth would use a queue to hold cars lining up to pay toll.

Queue Operations

- enqueue(E x) : adds an item to the queue.

Queue Operations

- enqueue(E x) : adds an item to the queue.
- E dequeue() : removes and returns an item from the queue.

Queue Operations

- enqueue(E x) : adds an item to the queue.
- E dequeue() : removes and returns an item from the queue.
- E peek() : returns, but does not remove, the item that will be returned by the next call to dequeue().

Queue Operations

- enqueue(E x) : adds an item to the queue
- E dequeue() : remove and return an item from the queue
- E peek() : return, but do not remove, the item that will be returned by the next call to dequeue()
- boolean empty() : check to see if the queue is empty

Implementation of Queues

- An array-based implementation uses an array to hold queue elements.

Implementation of Queues

- An array-based implementation uses an array to hold queue elements.
- A linked list implementation uses a linked list to hold queue elements.

Conceptual View of an Array

A queue can be viewed as a linear sequence of items where

- new items are added at one end (the rear of the queue)

- new items are removed from the other end (the front of the queue)

Array Implementation of a Queue

- Requires an array to hold the queue elements:

```
String [ ] q;   // Queue will hold strings
```

Array Implementation of a Queue

- Requires an array to hold the queue elements:

```
String [ ] q;   // Queue will hold strings
```

- Requires an index to keep track of the array slot that will hold the next item to be added to the queue:

```
int rear;       // Where next item will be added
```

Array Implementation of a Queue

- Requires an array to hold the queue elements:

```
String [ ] q;   // Queue will hold strings
```

- Requires an index to keep track of the array slot that will hold the next item to be added to the queue:

```
int rear;       // Where next item will be added
```

- Requires an index to keep track of the array slot the holds the next item to be removed from the queue:

```
int front;      // Next item to be removed
```

An Invariant for the Array Implementation

- front: index of next item to be dequeued. This slot is normally filled, but is unfilled when the queue is empty.

- rear: index of slot that will receive the next item to be enqueued. This slot is normally unfilled, but is filled when the array is completely filled.

Simplistic Implementation of enqueue

- Initially rear points to the first available unfilled position in the array.
- A new item x is added to the queue at rear as follows:

```
q[rear] = x;
rear ++;
```

As a new item is added, rear is incremented to point to the next unfilled position in the array.

Simplistic Implementation of dequeue

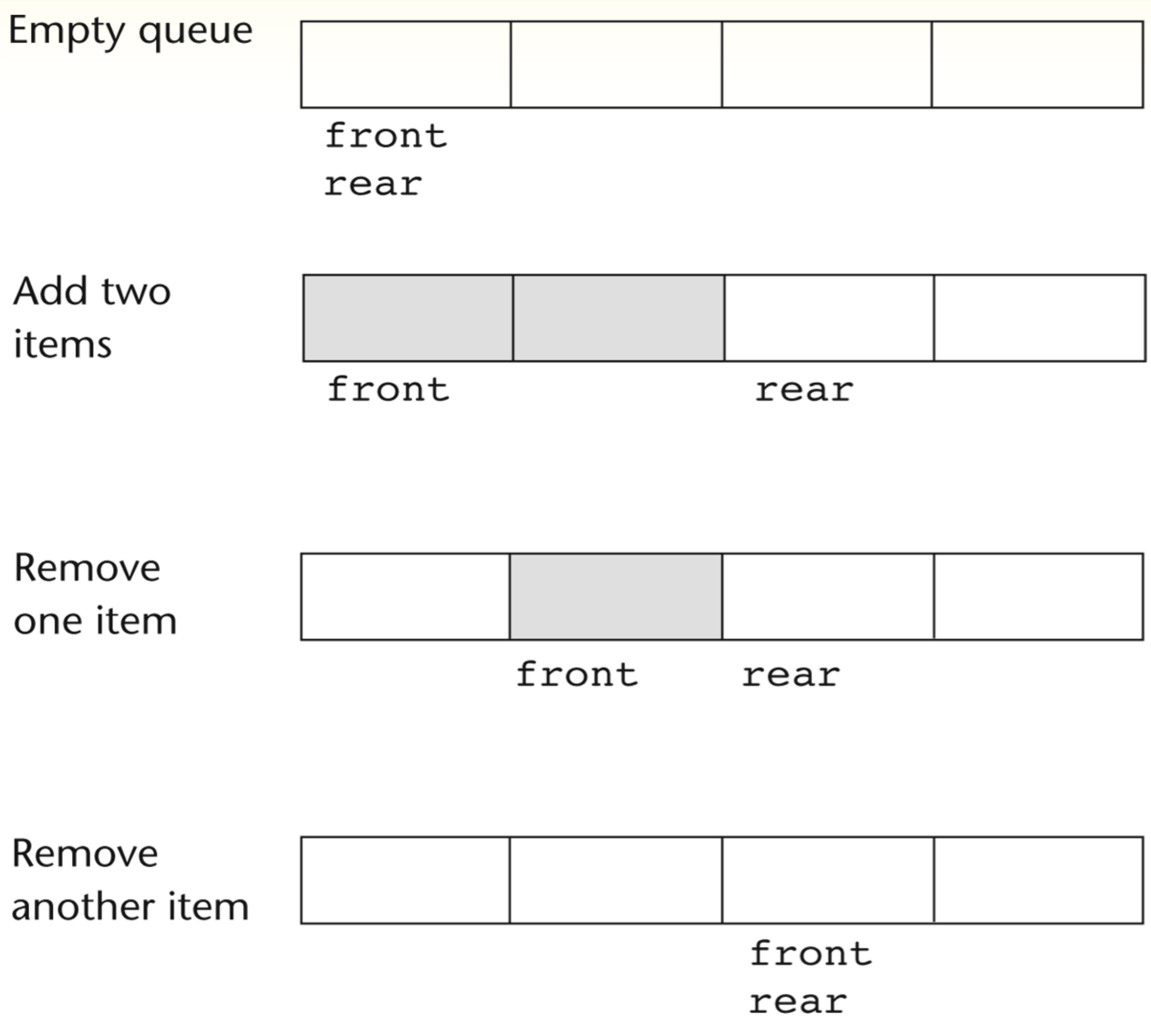
- The item is removed from the queue at the index front.
- The item at front is saved so it can be returned, and front is incremented to point to the next item in the queue:

```
String element = q[front];
q[front] = null;
front ++;
```


Use of the Array as a Circular Buffer

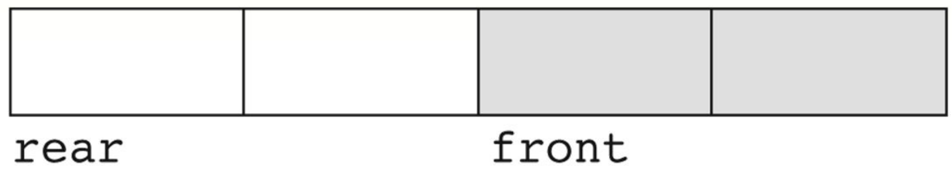
- The simplistic enqueue fills array slots in order of increasing index
- The simplistic dequeue empties array slots behind enqueue, also in order of increasing index.
- By the time enqueue gets to the end of the array, dequeue may have emptied out slots at the beginning, so enqueue should wrap around to the beginning of the array when it gets to the end.

Use of an Array-Based Queue

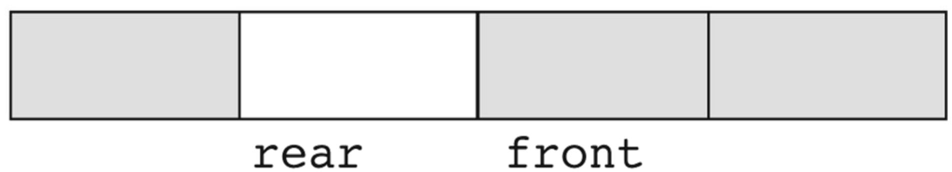


Using an Array as a Circular Buffer

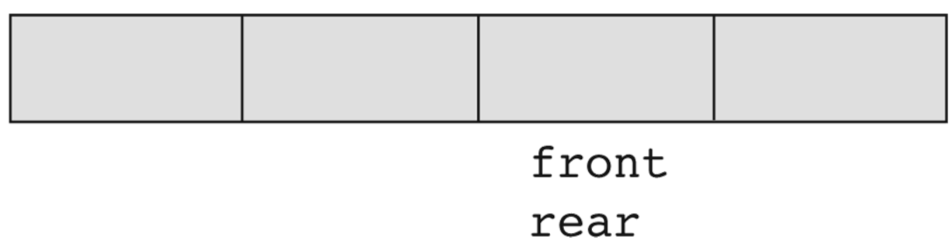
Add two more items: rear wraps around



Add another item



Add another item: queue is full



Tracking the Size of the Queue

- Tracking the number of elements in the queue is useful in determining if the queue is empty, and when the queue is full.
- An integer variable is used to track the size:

```
int size = 0;
```

- The variable size is incremented by enqueue, and decremented by dequeue.

Implementation of enqueue

The correct implementation of enqueue must

- Increment the size variable.
- Add the new item at rear.
- Increment rear and wrap around to 0 when rear reaches the end of the array.

The Array-Based Enqueue Method

```
public void enqueue(String s)
{
   if (size == q.length)
      throw new IllegalStateException();
   else
   {
      // Add to rear
      size ++;
      q[rear] = s;
      rear ++;
      // If at end of array, wrap around
      if (rear == q.length) rear = 0;
   }
}
```

Implementation of dequeue

The correct implementation of dequeue must

- Decrement the size variable.
- Remove an item at front.
- Increment front and wrap around to 0 when front reaches the end of the array.

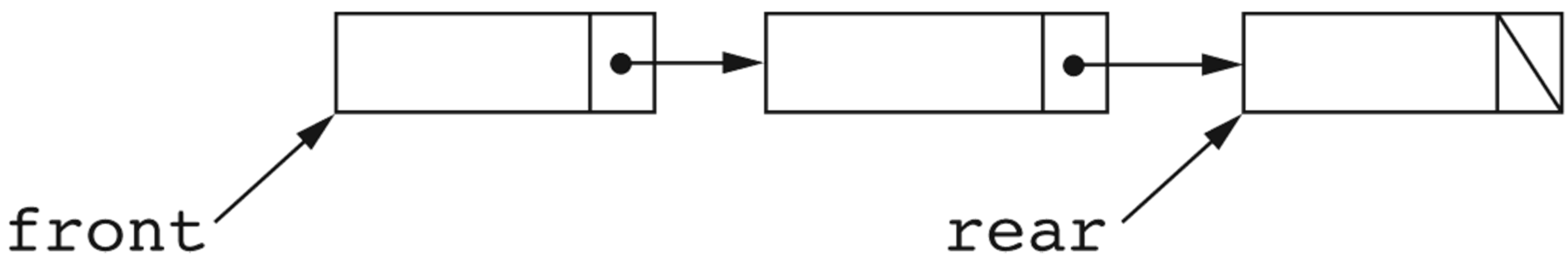
Implementation of dequeue

```
public String dequeue()
 {
     if (empty())
         throw new IllegalStateException();
     else
     {
         size --;
        // Remove from front
        String value = q[front];

        // Facilitate garbage collection
        q[front] = null;

        // Update front
        front++;
        if (front == q.length) front = 0 ;

       return value;
    }
}
```

Linked Implementation of Queues

Linked Implementation of Queues

- A linked list can be used to implement a queue.
- The head of the linked list is used as the front of the queue.
- The end of the linked list is used as the rear of the queue.



Linked Implementation of a Queue

```
class LinkedQueue
{
    private class Node
    {
      String element;
      Node next;
      Node(String e, Node n)
      {
          element = e; next = n;
      }
    }
    Node front = null;    // head of list, where items are removed
    Node rear = null;     // last node in list, where items are added
}
```

Checking if Queue is Empty

This is done by looking to see if there is a node at front, that is, if front is null:

```
boolean empty()
{
    return front = null;
}
```

Linked Implementation of Enqueue

```
public void enqueue(String s)
{
    if (rear != null)
    {
       rear.next = new Node(s, null);
       rear = rear.next;
    }
    else
    {
       rear = new Node(s, null);
       front = rear;
    }
}
```

Linked Implementation of dequeue

```
public String dequeue()
 {
   if (empty())
      throw new IllegalStateException();
   else
  {
      String value = front.element;
      front = front.next;
      if (front == null)
           rear = null;
      return value;
   }
 }
```