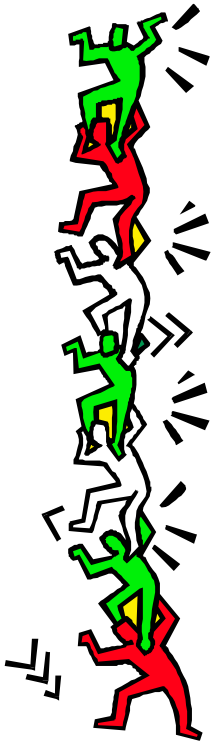


# Assignment 2: Stacks, Queues and Linked Lists

## Summary



In this assignment we created two test programs to use a Stack and a Queue. The Stack test program *reversed a String of characters* while the Queue test program *demonstrated graphically how a queue structure works, using single characters*.

It was found that the test programs we created were able to “pop” or “push” elements to/from a Stack or a Queue irrespective of whether the Stack or Queue was implemented using the standard java class, Arrays or Linked Lists. Apart from a slightly different implementation for linked list Queues, the programs behaved identically in each of these cases.

Here is a breakdown of the **code** created for this assignment:

Stack: 1 test program; 2 implementation classes (*Array, Linked List*)

Queue: 1 test program; 2 implementation classes (*Array, Linked List*)

Linked List: 1 test program and 1 implementation class. This implementation class is used (and modified) when implementing Stacks and Queues using Linked Lists.

In general, when implementing the **Stack**, we needed a “push” method to *insert* an element into the stack and a “pop” method to *remove* an element from the stack. When implementing the Queue, we needed the same methods, except that the pop method removed elements from the beginning of the Queue. It was easier to implement class files for the Stack and the Queue when using *Arrays* rather than using *Linked Lists*.

## Report Research

A number of sources were used to referred to when researching how to write a report like this one. Sources that we used were typically;

*The Internet,*

*Assignment 1: Java Refreshed,*

*and the guide to report writing in the back of engineering lab scripts.*

*Internet sites that were browsed included:*

<http://fbox.vt.edu:10021/eng/mech/writing/>,

<http://www.msue.msu.edu/aee/dissthes/guide.htm>,

<http://www.neltec.com/scifair/report.htm>

## Table of Contents

Page	Description
2	<i>Summary</i> (Gareth), <i>Report Research</i> (Ian)
3	<i>Table of Contents, Why not Exercise 7?</i> (Gareth)
4	<i>Introduction</i> (Gareth)
5-20	<i>Program Design</i> (see pages)
21-31	<i>Testing</i> (Ioan)
32	<i>Conclusions</i> (Gareth), <i>Bibliography</i> (Ioan), <i>References</i> (Ioan)
33-59	<i>Appendix (Source Code, MS-DOS Output)</i> (see pages)

(The names in subscript text denote the *author* of the relevant section.)

### Why not Exercise 7?

Our group decided to tackle exercises *1 to 6* on the assignment sheet rather than exercise *7* (The mini-project involving queues to implement an airport simulation) for the following reasons:

- ❶ To answer exercise *7* properly, the person writing the code needed previous knowledge of implementing queues using linked lists. As no member of our group had previously done this using java, it was thought that doing the preceding *six exercises* would be wise. Only after doing this would we be able to properly answer exercise *7*
- ❷ It was easy to assign different exercises to different members of the group when given a list of exercises to do. As exercise seven involved writing a substantial piece of code for the main program, it was felt that it would be tricky to collaborate on this piece of code.
- ❸ It is much easier to write six short programs than one large one. This is the basis of “top-down” design, where we split up a large piece of work into several pieces.

# Introduction

This report is the result of work done to create several java programs modelling Stacks and Queues. To model a **Stack**, a program was made to reverse a String of characters. The String was provided by the user, entering it through the keyboard. This program used three different stack implementation classes: (1) The standard java class for stacks; (2) A java class implementing a Stack using an Array; (3) A java class implementing a Stack using a Linked List.

To model a **Queue**, a program was made demonstrating graphically how a queue storing single characters works. Again the user provided the input. This program used two Queue implementation classes: (1) A java class implementing a Queue using an Array, (2) A java class implementing a Queue using a Linked List. As an aid to fully understand **Linked Lists**, a simple program was created to implement a linked list structure.

## Stacks & Queues

Stacks use the principle of a **Last In First Out** (LIFO) system. We “push” information to the top of the stack and also “pop” information off the top of the stack. Stacks are often used by Operating Systems (System Stack) and by Compilers (for storing parameters and return addresses when procedures are called).

Queues use the principle of a **First In First Out** (FIFO) system. The system is analogous to a dinner queue. We “push” information to the back of the queue and “pop” information off the front of the queue. The “back” and “front” are often called “*tail*” and “*head*”. An application of the queue is a printer buffer. The computer feeds information into the tail of the queue and the printer takes data from the head of the queue.

One application that uses a queue **and** a stack is when we want to calculate algebraic expressions using a computer. A computer cannot understand normal mathematical expressions e.g.  $A = (B+C)/(D-E)$ . It needs to convert the expression to polish or reverse polish notation in order to process it.

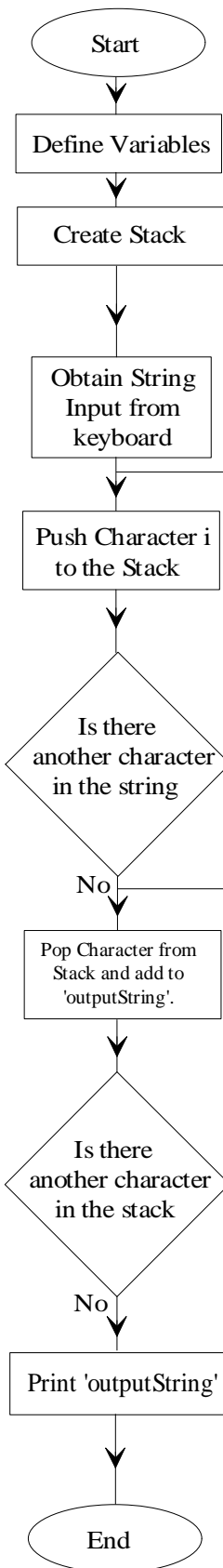
To do this, the elements of the expression are placed in a queue. A series of rules are followed that use a stack as an intermediate storage area so that we can place elements of the original queue in another queue, called the Reverse Polish Notation Queue. The Reverse Polish Form of  $A = (B+C)/(D-E)$  is **ABCD/+E-==**.

Examples of rules used are “If the stack has a ‘=’ or a ‘(’ on top, then push the next element in the queue in the stack” and “At the end of the input queue empty the stack to the Reverse Polish Notation Queue”. Basically, the rules decide whether we should push the next element in the input queue to the stack or to the Reverse Polish Notation Queue.

# Design

## Exercise 1: String Reversion using Stacks

Code: Gareth | Report: Gareth | Flow Chart: Ian



In this program we were trying to write an example program that used the standard java Stack class to reverse a String of characters. The program is also used in *exercises 2 and 5* as the test program there.

It was decided that the String was to be provided by the *user*. Therefore, we imported the BasicIo set of methods that allows simple user interaction, particularly to use the `readString()` method that allows the user to enter a String.

After defining the program's variables (A String for **input**, one for **output** and of course a **Stack**) we asked the user to enter a String. This was stored in the `inputString` variable. We then proceeded to push all the characters in the String into the stack using a *for* loop.

The *for* loop counted from zero up to the amount of characters in the String i.e. `inputString.length()`. For each execution through the loop, we set the internal variable `pushString` to hold the  $i^{\text{th}}$  character in the input String i.e. `inputString.substring(i, i+1)`. We then pushed this character to the stack i.e. `testStack.push(new String(pushString));`

Now that all the characters have been *pushed* to the stack, we need to *pop* them all off again! This is how we **reverse** the String (The First In, First Out principle) Again a loop was constructed counting from zero up to the amount of characters in the input String. This time we defined the internal variable `obj` to hold the next object to be popped off the stack i.e. `Object obj = testStack.pop();`

We printed this character out on screen and then added it (*concatenated it*) to the output String i.e. `outputString = outputString + obj;` All that was now left to do was to print out on the screen the contents of `outputString`, which contains after the loop has finished the input String in *reverse*.

```
System.out.println("Your String after reversion is " + outputString);
```

## Exercise 2: Write a Stack implementation using an Array

Code: Gareth | Report: Gareth

In this exercise, we use the code created in exercise 1 (*A program to reverse a String of characters using the standard java Stack class*) with a class written to implement the Stack class using **arrays**. The only change in the code from exercise one is that we erase the “`import java.util.*;`” line to force java to use the Array implemented Stack, not the default java Stack class.

The class written to implement the Stack class using arrays (`Stack.java`) starts off by defining some variables. For each stack that we create, we want an **array of Objects** (`private Object[] internalStack`), an integer to denote the stack Size (`private int stackSize`) and an integer to denote the position of the next free space in the stack (`private int pointer`).

When we create a stack, it might be the case that we want to define the (static) size of the stack. To cater for the “*maybe*” aspect of the last statement, we need to create two constructors: one for where the user doesn’t specify the stack size, and one for where the user does specify the stack size.

In the former case, we create an array of size which is determined by the constant “`defaultSize`” in the Stack class, set to 100 by the programmer. In the latter case, we introduce an integer parameter when calling Stack, so that if we write `aStack = new Stack(2000)`, the stack constructor will create an array of 2000 objects.

### Case 1: Size *not* specified

Constructor:

```
Stack()  
{  
    SetupStack(defaultSize);  
}
```

### Case 2: Size *is* specified

Constructor:

```
Stack(int userSize)  
{  
    SetupStack(userSize);  
}
```

`SetupStack` is a method that takes an integer input parameter and initialises the variables of the Stack class according to the requirements of the user. Also it initialises the pointer at zero. The purpose of the pointer is to indicate the location of the *next free space* in the stack. The pointer is set at zero in both of the constructor cases.

```
internalStack = new Object[setSize]; // (setSize is the input parameter)  
stackSize = setSize;  
pointer = 0;
```

Now all the variables have been set, we can concentrate on providing the **push** and **pop** methods that are required to *insert* and *delete* elements to/from a stack.

## Push Method

In the push method, we ask for an Object to be given as an input parameter (this is called `obj`). When we push an element to the stack, before we push it, we must make sure there is enough room in the stack to do this. So we must test to see if the location of the next free space in the stack is less than the stack size itself. If it is, then we place 'obj' in the array at the location denoted by "pointer" and then increment the pointer.

```
if (pointer < stackSize)
{
    internalStack[pointer] = obj;
    pointer++;
}
```

In the case that the next free space is more than or equal to the stack size, we print out a message telling the user that the stack is **full**.

## Pop Method

This method returns an Object to the piece of code that called it. Before we return an element of the stack, we must first check to see if the stack contains any elements at all. To do this, we check to see if the value of the pointer is zero (meaning that if it is, the next free space is the first element in the stack i.e. the stack is empty). If this is the case, then we print out a message telling the user that the stack is empty.

```
if (pointer == 0)
{
    return("Stack is empty");
}
```

If the stack is not empty, then we may decrease the pointer by one (meaning that the pointer now points to the last element to be inserted into the stack) and return the data at that location. We would normally then wipe the contents of this location, but because we know that the stack will only be used once in the test program, this code is omitted.

```
else
{
    pointer--;
    return internalStack(pointer);
}
```

## Exercise 3: Write a Queue implementation using an Array

Code: Gareth | Report: Gareth | Flow Charts: Ian

There were *two* parts to this part of the assignment: (a) write a class implementing a queue using an array; (b) write a test program that used the queue abstract data type. Part (a) was an extension of the **Stack** class created for exercise 2, while the test program had to be written from scratch.

### Part (a): Queue.java

As always, we define some **variables** to begin with. Privately, we define the following: an integer that denotes the size of a default sized Queue (that is used if a value is not given when calling the Queue constructor). This variable is called `defaultSize`. We define integers for keeping track of the queue **size**, the location of the **front** and the **back** of the queue (`queueSize`, `frontPointer` and `backPointer` respectively) and a *check* integer `checkEmptyFull` to hold information about whether the queue is **full**, **empty**, or **neither** of the above. (The convention used in the program is that `checkEmptyFull` holds the value *1* if the queue is full; *0* if the queue is empty and *-1* otherwise).

As in the **Stack** class, we again need two different constructors when defining a Queue abstract data type. Again, we set the size of the array depending on whether the user *supplies* a number or whether the user wants to create a *default* sized queue. In the setup method, we create an appropriately sized array to model the queue, and initialise some of the variables above, including setting the pointers to zero.

The **first** major method in queue class is the **Push** method, used to insert elements into the queue. It requires that an object is provided as an input parameter.

In order to ensure that the queue has room enough to insert an element, we first check to see if the `checkEmptyFull` pointer has the value 1, meaning that the queue is full. If it is 1, we print out an error message "*Queue is Full*". If it is not, we may proceed to insert the element into the queue.

```
if (checkEmptyFull == 0)
{
    return("Queue is Empty");
}
else .....
```

In the queue, all elements are inserted at the **back** of the queue. Therefore, we place the parameter object in the location pointed to by the `backPointer` variable. This variable denotes the location of the *next free space* in which we may insert a new element into the queue. After inserting the element into the queue (using the code `internalQueue[backpointer] = obj;`), we *increment* the `backPointer` variable to point to the **next** available space and set `checkEmptyFull` to -1 (if the queue was previously empty, we need to denote that it is *not* now).

To finish off the **Push** method, we need to perform two checks. The first checks whether the `backPointer` variable has overstepped the limit of the boundary of the queue. If it has, we may then reset the `backPointer` to zero, creating the effect of **cycling** the queue to the beginning again. The *second* check looks to see if the queue is now full after an element has been inserted. We know if the queue is full if the `backPointer` and `frontPointer` variables hold the **same** value. If this is the case, we flag using `checkEmptyFull` that the queue is full.

Check 1:

```
if (backPointer > (queueSize-1))
{
    backPointer=0;
}
```

Check 2:

```
if (frontPointer == backPointer)
{
    checkEmptyFull = 1;
}
```

Next, we shall analyse the **Pop** method. Again, this starts off with a check, this time to see whether the queue is *empty*. If it is (if `checkEmptyFull` holds the value zero) then we return an error message “Queue is Empty!”. Otherwise, it is safe for us to delete an element from the queue, and we do so as follows:

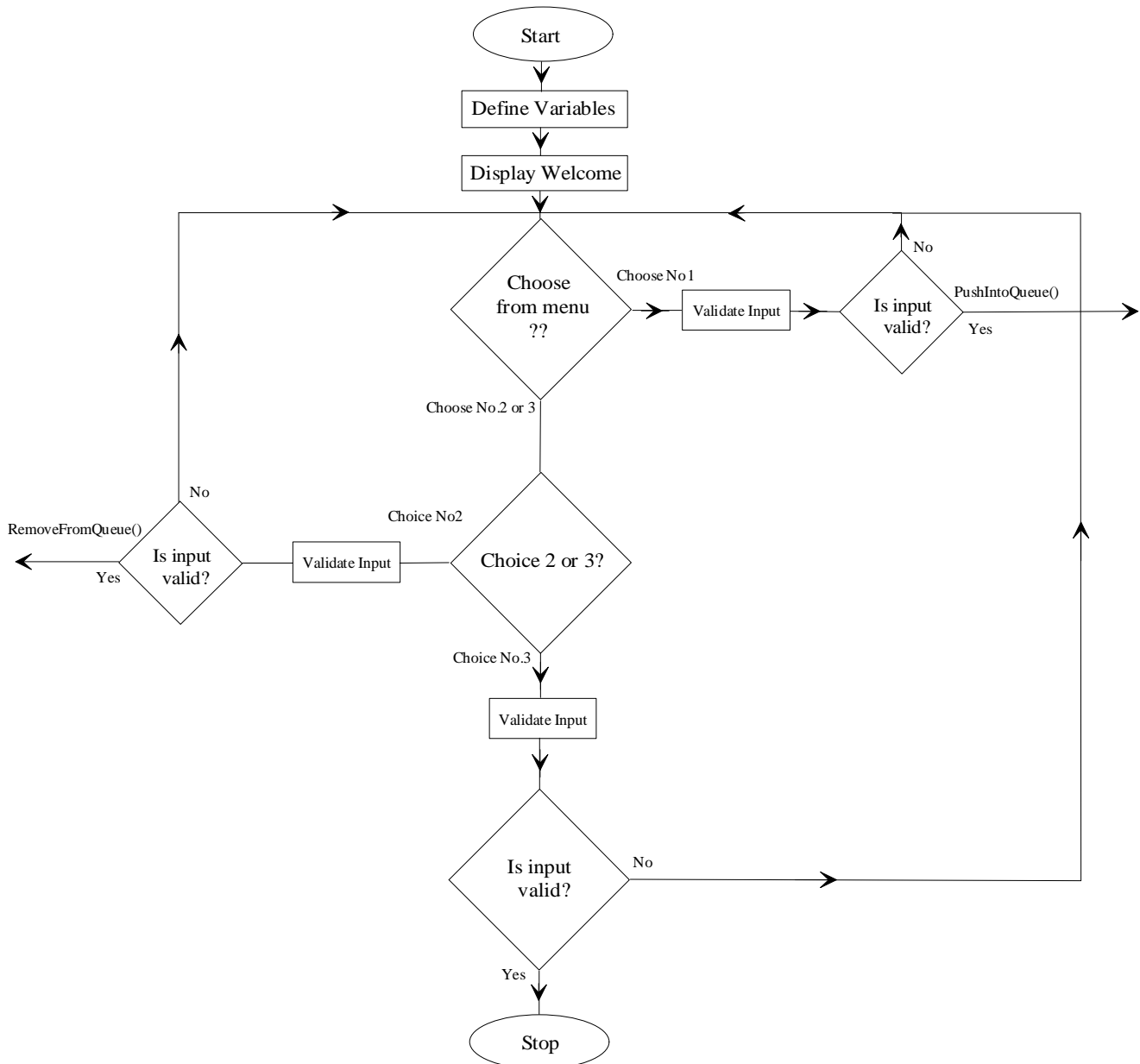
The `frontPointer` variable points to the array element that is at the **front** of our modelled queue. We copy the data at the front of the queue for return **at the end of the method**, and *wipe* the data at that element of the queue (set it to *null*). We then increment the `frontPointer` (so the **next** element in the queue is now at the front of the queue) and set `checkEmptyFull` to -1 (if the queue *was* full, it is not now by virtue of the fact that we have just *deleted* an element from the queue)

We now need the same kind of two checks as were executed *at the end of* the pop method i.e. check if the pointer in question, here `frontPointer`, has exceeded the boundary of the array, in which case we **cycle** it to the beginning; and a check to see if the two pointers now have the *same* values. If this is the case here, we know that the queue is now **empty** and we set `checkEmptyFull` to zero.

Finally, we have a “*Get Element*” method at the end of `Queue.java`. This method is used to display an *element* of the array. Given an integer input parameter, we first check to see if the parameter is in the correct **range** (i.e. greater than or equal to zero and less than the queue Size). If it is not, we return (as the Object requested) an error message “*Incorrect Parameter*”. However, if the parameter is in the correct range, we may return the value of the  $i^{\text{th}}$  element of the array, where  $i$  denotes the value of the *input* parameter element.

```
if ((element >= 0) & (element < queueSize))
{
    return internalQueue[element];
}
else
{
    return("Incorrect Parameter");
}
```

## Part (b): QueueDemo.java



The Queue Test program works by using a **menu** system in which the user may choose to do one of three things. (1) *insert an element into the queue*; (2) *delete an element from the queue*; (3) *exit the program*. The user may only insert **single** characters into the queue, because the program uses a graphical way of showing how the queue develops. This graphical method works much better if we restrict the data to *single* characters.

We will not concern ourselves too much as to how the menu system works. The important principles are that it uses a **switch** statement and is able to catch any **erroneous** data. The menu system is summarised by the flow chart above. We will however concentrate on the methods that the menu system calls, particularly the methods that insert or delete elements to or from the queue.

At the start of the program, we create a Queue abstract data type. We know that we will be modelling a ten element queue, so we create a queue of 10 elements:

```
static Queue testQueue = new Queue(queueLength); // queueLength = 10
```

If the user chooses to enter a character **into** the queue, then the `PushIntoQueue` method is called. In this method we ask the user to enter a character from the keyboard, making sure he or she enters a *single* character. After we obtain some valid input, we then push the input to the queue, using the code

```
....else
{
    testQueue.push(InputString);
    DisplayCharacters();
}
```

Note that the `DisplayCharacters()` method is the method that displays the queue *graphically* on screen. It basically consists of a series of statements printing output to the screen, and uses the `Queue.getElement(int)` method to obtain the values of a particular *element* of the queue.

```
System.out.print(testQueue.getElement(i));
```

If the user chooses to **remove** a character from the queue, then the `RemoveFromQueue` method is called. The code for the method is straightforward, in that we define an object “obj” that gets whatever the *pop* method in the Queue class gives it back.

```
Object obj = testQueue.pop();
```

We then print out what was **removed** from the queue and update the screen with the `DisplayCharacters()` method. Note that if the queue was empty, then trying to remove an element from the queue in this instance would result in “obj” receiving an error message such as “*Queue is Empty!*” or similar.

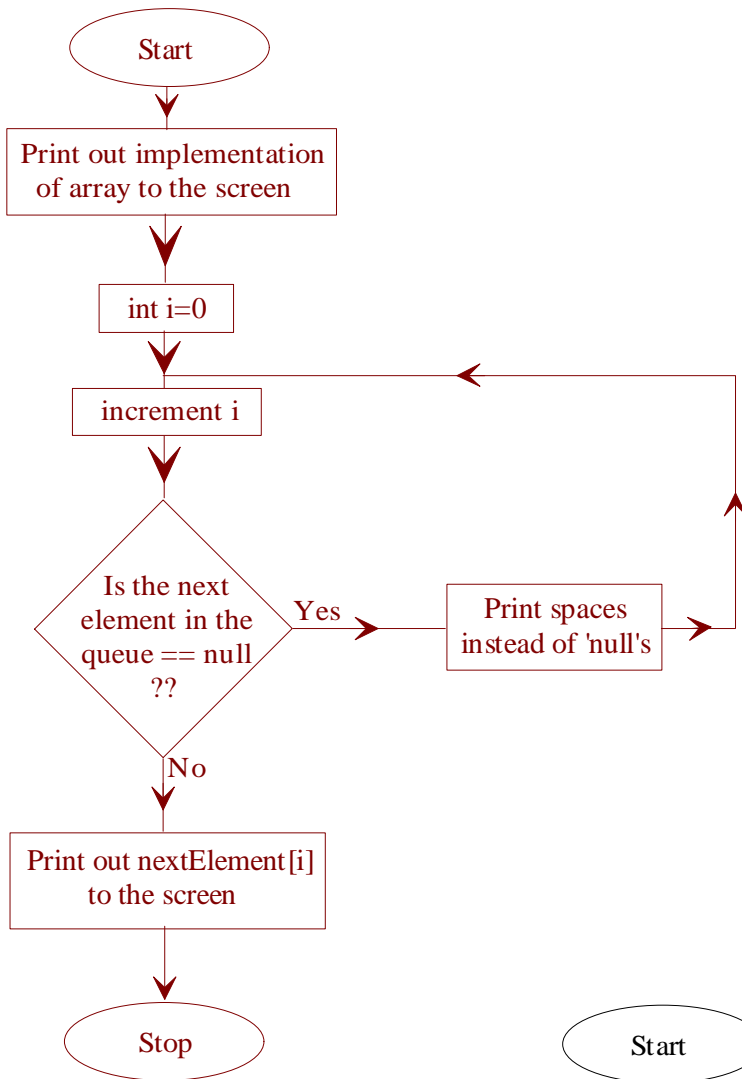
The final choice in the menu system is (3) exit program. It is fairly obvious what this command should do on execution, that is do nothing!

### Note on using QueueDemo.java with different queue classes:

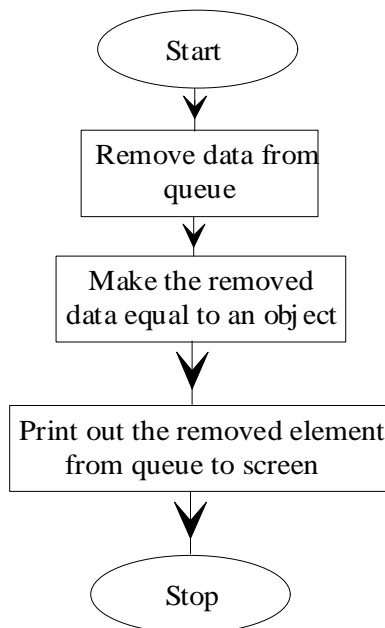
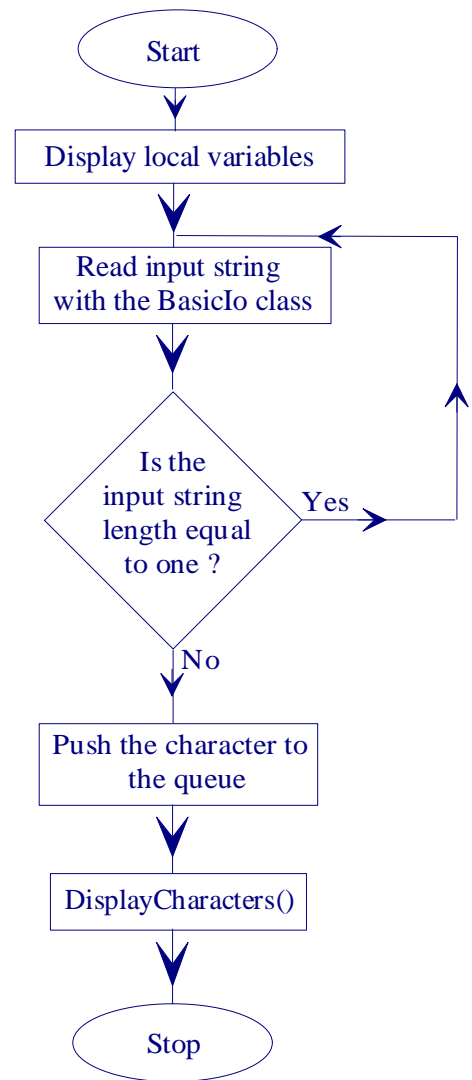
I have tried to write this piece of code so that it can be used with an **array** based queue class and a **linked list** based queue class. Setting a limit to how many elements the queue may contain may not suit a linked list based queue class, but then how do you model the “*cycling*” property used with arrays?. A compromise must be reached. Also, when we use a queue of 10 characters, it is very easy to see on screen how the queue works and therefore easier to understand the concept of a queue.

# Flow Charts for QueueDemo.java.

## (1) The DisplayCharacters() method.



## (2) The insert(object) method.



## 3. The RemoveFromQueue() method

## Exercise 4: Implement a Linked List data structure and associated test program

Code: Ioan | Report: Ioan | Flow Charts: Ian

The task of the program was to implement a linked list data structure and associated test program.

### Part 1 Node.java

Each Node has a data Object and a link (another Node) pointing to the next Node in the list. Firstly the variables are defined: an object, a Node and a private Boolean `deleteLink` which is initialized at first to *false*. Next is a constructor which creates a new node.

The first major method in the class is the *insert* method. In this method we ask for an input parameter, called **obj**. There is an **if** statement that checks if the next element in the list is null (i.e. nothing there, blank). If this condition is true then the input parameter 'obj' will become the current data and a new node is created for the future.

Otherwise, the following code will be executed:

```
else
{
    insert(whatList.nextElement, obj);
}
```

This piece of code will call the *insert* method from within until it reaches the end of the list where we can insert another element.

The second major method in the class is the *delete* method. This uses a Boolean function. If the Boolean is true (i.e. the calling method wants to delete the top element in the list) the delete method will use an **if** statement to check whether the next element in the list is null.

Now two lines of important code will execute, the first will set 'deleteLink' to true and the second line will set the current data in the list to null (i.e. blank it). If the statement was false then the *else* method will call the delete function again to look for a `nextElement` which is equal to null (i.e. blank). After this is completed (the internal loop is broken) there is another **if** statement waiting, an if statement which checks whether 'deleteLink' is equal to zero. If this is true the next code will execute:

```
whatList.nextElement = null;
whatList.data = null
deleteLink = false;
```

The next element will be blanked (set to null), the next element in the list will also be set to null and the `deleteLink` trap will be reset.

If the Boolean was false, then the program will check whether the next element in the list is *null*. Now we have to replace the data in the root node with the data in the second node, and then replace the link from the first node to the second node and then from the second node to the third.

There is one more method called the `displayAll` method. This simply displays all the elements in the list. First of all it starts at the root of the list and prints out the first element, and then the method calls itself from within, this will be recursive until the `nextElement` in the list is null (i.e. until the list has ended)

## Part 2 TestLinkedList.java

The associated test program works by using a menu system, with four options available to the user-

- 1/ Insert item in the list.  
(The user may enter an input string into the list.)
- 2/ Delete item from the top of the list.
- 3/ Delete item from the bottom of the list.
- 4/ Exit the program

The menu has a *switch* statement and is able to catch any exemptions, for example a wrong input at the main menu such as 5 or a string such as “Ioan” (e.g. not 1,2,3 or 4). The menu system will call three methods - **Insert** item, delete from **top** of list and delete from the **bottom** of the list. The menu system is summarized in the *flow* chart. (see next page).

At the start of the program there is a insert method *InsertItem()*, which inserts items into the list. Firstly a new node is created *static Node linkedList = new Node;*. Next the program asks the user to type in a string from the keyboard. Using `BasicIo` to read the string , it is put into the `inputString`, and entered into the list.

```
linkedList.insert(linkedList, inputString);
```

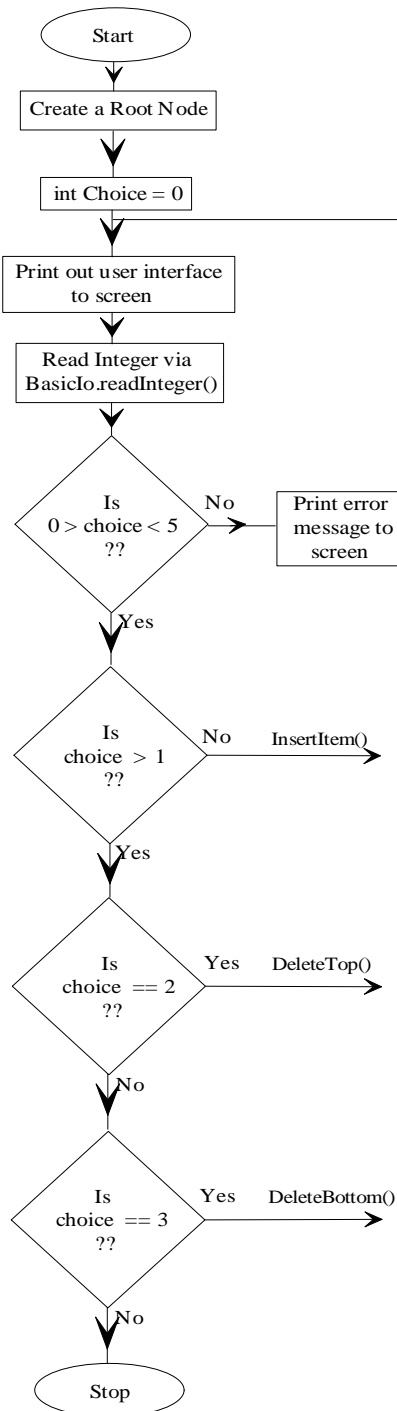
Having entered the string in the list the `displayall` command is used and all the elements in the list so far are displayed on screen.

Next we have *DeleteTop()*, this method deletes elements at the end of the list similar to the theory of a stack LIFO (Last-in First-out). The delete method is called with true Boolean passed as parameter. And the `displayall` method is called, where the root node is passed as a parameter.

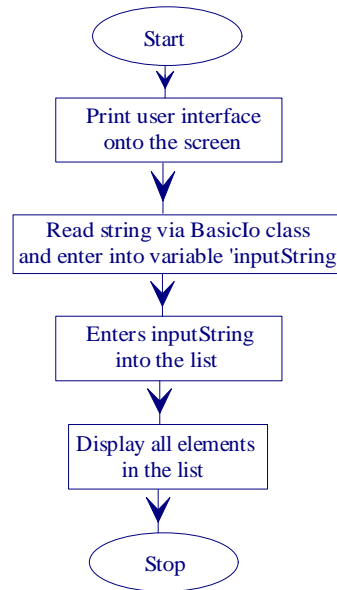
Finally we have *DeleteBottom()*, this method deletes elements from the start of the list similar this time to the theory of a queue FIFO (First-in First-out). The method is called with false Boolean passed as a parameter. And then calling the method `displayall` with the root node `linkedList` as parameter.

# Flow Charts for Exercise 4

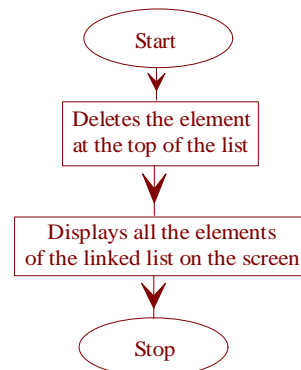
## Main Program



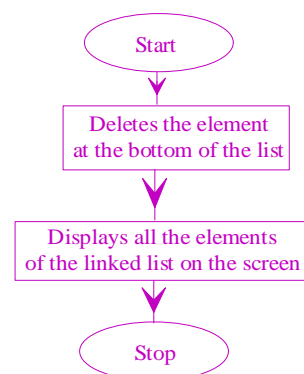
## InsertItem()



## DeleteTop()



## DeleteBottom()



## Exercise 5: Write a Stack implementation using a Linked List

Code: Ian | Report: Ian

This exercise used the Node class in the previous example as a platform for creating the Stack. It also needed a modified 'Stack' class to use the 'Node' class and therefore implementing the Linked Stack. Nodes are instances of the 'Node' class, each of which contain two variables: (1) An object containing data; (2) Another node which is the next element in the linked list.

Firstly there is the Stack class that has to be modified to implement a Stack using a Linked List. At the very start of this class file is a piece of code that defines a root (start) node,

```
private Node linkedList = new Node();
```

The **size** of the stack is not relevant in this case, because Linked Lists are not static data structures. We do not need to define a size, so therefore we do not do anything in the following lines of code (Constructors which were used when implementing array based stacks to define a default sized or user-defined stack size).

```
Stack(int userSize) {}  
Stack() {}
```

The stack Class has **two** methods, one for 'Pushing' the elements onto the stack and the other method for popping elements from the stack. The 'Push' method is quite simple to understand. When the method is called an object parameter is needed, i.e. the data that is to be pushed to the stack, as we can see now,

```
public void push(Object obj)  
{  
    linkedList.insert(linkedList, obj);  
}
```

'linkedList' is the definition of the root node, when the 'Reverse.java' program calls the push method the stack class uses the node class and inserts the element into the stack.

The second method is the **pop** method. This is a bit more complicated than the push method because there is an *if* statement controlling what the method is going to do. No parameters are needed when calling this method from the main program, all this method accomplishes is deleting the last element in the stack. It does this by using the if statement to walk through the linked stack until it reaches the end, it will return a 'Stack is Empty' statement to the calling function.

If this is not true then the program calls the delete method in the **Node** class, passing in the root node of this linked list, and then the delete method deletes the last element in the list, returning the data in the last element before deleting.

The **Node** class is next. At the start we declare two variables:

```
Object data;  
Node nextElement;
```

Each node will have a data object and a link pointing to the next node in the list (the 'link' is actually a node). There are two more variables, this time *private* ones: (returnObject is used in the *delete* method for returning objects, and the severLink variable is obviously used for 'link deletion'.)

```
private Object returnObject = null;  
private int severLink = 0;
```

Now we need a **Constructor** for creating a new node - this is used in the Stack class to create nodes. All this constructor does is set the data object and the nextElement to 'null'.

There are two methods in this 'Node' class, an *insertion* method called insert' and a method for *deleting* called delete. To call the insert method we need to pass **two** parameters. As the following code shows, we need a root node and the object we are pushing into the stack.

```
public void insert(Node whatList, Object obj)
```

Next is an **if** statement that determines if there is *another* element in the list, and if there is none it executes these two lines of code,

```
whatList.data = obj;  
whatList.nextElement = new Node();
```

The first line pushes the input parameter obj as a data portion into the stack, and afterwards a new node is created for the *future*.

If the '**if**' comparison is false then the code calls the method again passing in the *next element*, it will continue to do this until it finds a element which is empty.

The *final* method in the class is the **delete** method. Unlike the insert method, the delete method does not need a data object to be passed as a parameter - all that is needed is the root node as the parameter. Again we observe an **if** statement, and again comparing the next element in the list to *null*, if it's true then it will set severLink to 1, (it was initialised to 0 at the start) and the **trap** will be set. The trap is issued to return the data from the *last* element of the list, and the trap is set in such a way so that it is only executed once, at the end of the list. Had we **not** set the trap, the code would return the data contained in the first element of the list, because of the way recursion is used in the method. Next time the program calls the delete method (after the trap is set), it will go to *else* and execute the code,

```
if(severLink == 1)
```

The program will now return the data that is to be deleted and then *delete* the element.

```
returnObject = whatList.data;  
whatList.nextElement = null;  
severLink = 0;
```

It will also set the trap back to **zero** again. (explanation earlier).

## Exercise 6: Write a Queue implementation using a Linked List

Code: Ian | Report: Ian

This exercise like the previous one uses the **Node** class, but this time it is different - the Node class needs to be modified to model the linked list with a queue, therefore we need an explanation for this revised class file.

At the start like any other class file we need to declare variables and objects, here there are three, one an object to hold the current data called "data" and the other a Node called "nextElement". There is also a private object,

```
private Object returnObject = 0;
```

which is initially set to zero, this is the variable we are using to return the data to the calling method.

Like before there are two methods called *+Insert+* and *+Delete+*, also included this time is a method called *nextNode*, this is simply a method that will return the next node using the following code,

```
return whatList.nextElement;
```

With the *+insert+* method the calling function needs to pass two parameters, the root node and the object that is going to be inserted into the list. This method first needs an **if** statement to check whether there are more elements in the list, this is done by checking if the *+nextElement+* is equal to null (empty). If there are no more elements in the list, (i.e. the *nextElement* is null) the method will set the input parameter as the current data *+whatList.data+* and then create a new node for future use.

If the **if** statement is false then it executes the *+else+* part of the statement, this will simply pass in the *nextElement* of the list as a parameter, it will keep on doing executing this recursive loop until it finds the end of the list (that is until the *nextElement* is null).

The *second* method in this class file is the delete method and again this method is a little different to the previous exercise since we are not implementing the linked list with the stack.

This method sets the current data to *returnObject* ready for returning. We have to enter the data to be returned to a variable because java will not allow us to return it now because we still have to delete the data - an error would be inevitable if we return data and more code followed after.

To delete an element from the list we have to delete the *first* node in the list, this is achieved by copying the link from the second node in the list to the first.

```
whatList.data = whatList.nextElement.data;  
whatList.nextElement = whatList.nextElement.nextElement;
```

Now we can *return* the data that is deleted,

```
return returnObject;
```

if we remember we put the current data `+whatList.data+` equal to `+returnObject+`.

The `Queue.java` class file also has to be modified so that the queue can implement the linked list. At the start of this class file we have to define a root node (which is an empty node) in the linked list, and also other variables which are all `+private+`.

```
private int queueSize;  
private int numberOfNodes;  
private boolean sizeLimit;  
private Node storageNode = new Node();
```

Next there are **two** constructor methods, one for a user size defined queue and the other if the user does not specify, that is we supply a queue of size -1 ( a dummy value to indicate that the queue has no size limit).

The user sized constructor uses the `+queueLength+` variable that is passed by the main program (exercise 3 & 6) to determine the length of the queue. There is also a very important piece of code which is the `sizeLimit` variable, if the main program uses the user defined queue size then the `sizeLimit` is set to boolean **true**, otherwise it is set to false.

Like the previous queue class there are two important methods, a **push** and a **pop** method. The `+push+` method relies only on the object to be inserted into the queue to function, other parameters are required.

This method relies on the private variables defined at the *start* of the class as well, first of all there is an **if** statement, checking that `sizeLimit` is true (that is if it was user size defined) and also if the `numberOfNodes` is less than `queueSize`. If this was true the method would call the insert method from the Node class with the root node and the object to be inserted into the list. `+numberOfNodes+` would be incremented.

If the if statement was *false* (i.e any of the conditions were false) then the else part of the statement would execute. All this would execute would be a statement sent to the screen saying that the queue was full. This code has been specifically written for the task in hand, and if we had wanted to write a more general queue class we would have had to take into account that there is no size restriction to the size of queue in some cases, and therefore we could push elements into the queue regardless of whether there is a size limit. Hence we would need an extra piece of code to deal with this case, e.g.,

```
if(sizeLimit == false)  
    linkedList.insert(linkedList,obj);
```

to cater for the case that there is **no** size limit to the queue.

The next method was pop, that is delete, needs no parameters passed to. Firstly the method has to check whether the queue is *empty* because we can't delete anything from an empty list. If the **if** check is true the method would simply print out to the screen that the queue is empty, otherwise the else part of the statement would execute. To delete an element from the list we first have to decrement the number of nodes by one.....

```
numberOfNodes--;
```

...and then return the data element before deleting:

```
return(linkedList.delete(linkedList));
```

The class is not complete without a method for returning the `nextElement` in the list. First of all we need to set the temporary variable `storageNode` equal to the list in question.

The method is used to return the  $i^{\text{th}}$  element of the queue, where 'i' is a number the user supplies. Basically the method traverses through the list until it finds the  $i^{\text{th}}$  node in the list and returns the data in this node,

```
for(int i=0; i<element; i++)  
{  
    storageNode= storageNode.nextNode(storageNode);  
}
```

# Testing

All Reports: Ioan

Exercise 1: String Reversion using Stacks,  
Exercise 2: Write a Stack implementation using an Array,  
Exercise 5: Write a Stack implementation using a Linked List

**Method of testing: Black Box Testing**

1. When inputting a **NORMAL STRING** the following output was given.

---

```
A Program to reverse a set of characters.  
Please enter a String to reverse...Test
```

```
Item Pushed = T  
Item Pushed = e  
Item Pushed = s  
Item Pushed = t
```

```
Item Popped = t  
Item Popped = s  
Item Popped = e  
Item Popped = T
```

```
Your String after reversion is tseT
```

We clearly see that the correct output was given. The program reversed a sequence of characters, and gave a correct string at the end of the output.

2. When inputting an **EMPTY STRING** the following output was given.

---

```
A Program to reverse a set of characters.  
Please enter a String to reverse...
```

```
Your String after reversion is
```

When inputting an empty string by only pressing return, the program executed and produced no errors.

### 3. When inputting a **NUMERICAL STRING** the following output was given.

---

```
A Program to reverse a set of characters.  
Please enter a String to reverse...12345
```

```
Item Pushed = 1  
Item Pushed = 2  
Item Pushed = 3  
Item Pushed = 4  
Item Pushed = 5
```

```
Item Popped = 5  
Item Popped = 4  
Item Popped = 3  
Item Popped = 2  
Item Popped = 1
```

```
Your String after reversion is 54321
```

Again the correct output was given using numerical values.

### 4. When inputting a **LARGE STRING** the following output was given.

---

```
.....  
.....  
.....  
.....  
Item Popped = 1  
Item Popped = z  
Item Popped = y  
Item Popped = x  
Item Popped = w  
Item Popped = v  
Item Popped = u  
Item Popped = t  
Item Popped = s  
Item Popped = r  
Item Popped = q  
Item Popped = p  
Item Popped = o  
Item Popped = n  
Item Popped = m  
Item Popped = l  
Item Popped = k  
Item Popped = j  
Item Popped = i  
Item Popped = h  
Item Popped = g  
Item Popped = f  
Item Popped = e  
Item Popped = d  
Item Popped = c  
Item Popped = b  
Item Popped = a
```

```
Your String after reversion is zyxwvutsrqponmlkjihgfedcba0987654321zyxwvutsrqpon  
mlkjihgfedcba0987654321zyxwvutsrqppnmlkjihgfedcba0987654321zyxwvutsrqponmlkjihgf  
edcba0987654321syxwvutsrqponmlkjihgfedcba0987654321zyxwvutsrqponmlkjihgfedcba
```

Inputting this extreme value the program dealt with the inputs, reversed the sequence and printed the correct number in the string at the end.

Exercise 3: Write a Queue implementation using an Array  
Exercise 6: Write a Queue implementation using a Linked List  
PART A - TEST THE MENU SYSTEM (same for exercises 3 & 6)

**Method of testing: Black Box Testing**

### 1. Inputting INCORRECT INPUT e.g. **STRING**

---

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

```
1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program
```

```
Test
Sorry, Incorrect Input. Please Try Again.
```

Inputting a string such as “Test” will give the above output. The program will realise that the input was incorrect and produce the output *Sorry, Incorrect Input. Please Try Again.* (The exception is caught).

### 2. INCORRECT NUMBER (i.e. not 1, 2 or 3)

---

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

```
1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program
```

```
5
Sorry, Incorrect Input. Please Try Again.
```

Again we see that the exception is caught

### 3. EMPTY INPUT

---

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

```
1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program
```

```
Sorry, Incorrect Input. Please Try Again.
```

When inputting an empty string by only pressing return, the exception was caught and the correct output was given.

## 4. EXIT FROM PROGRAM

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program

3  
The Program will now close...

The program behaves as expected.

## PART B - TEST THE QUEUE Section 1: Exercise 3 (array implementation)

**Method of testing: Black Box Testing**

### 1. REMOVE AN ITEM FROM AN EMPTY QUEUE

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program

2

Item Removed from the Queue: Queue is Empty!

-----  
1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10  
-----

Firstly when we have an empty queue, the action of removing an item from an empty queue is tested. As seen above the number 2 was inputted and the program produced the output *Queue is Empty!*

## 2. ADD EMPTY ITEM

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1  
Please enter a character to enter into the queue....  
You have entered incorrect input. Please try again...  
Please enter a character to enter into the queue....  
ADD INCORRECT ITEM (i.e. not 1 character)  
Please enter a character to enter into the queue....Test  
You have entered incorrect input. Please try again...

## 3. ADD ITEM TO QUEUE

---

Please enter a character to enter into the queue....G

-----  
1 2 3 4 5 6 7 8 9 10  
G  
1 2 3 4 5 6 7 8 9 10  
-----

After choosing the option *1. Enter a Character into the Queue*, the character G was entered into the queue. Seen above is the output produced by the program.

## 4. REMOVE ITEM FROM QUEUE

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

2  
Item Removed from the Queue: G

-----  
1 2 3 4 5 6 7 8 9 10  
D F E F  
1 2 3 4 5 6 7 8 9 10  
-----

Removing an item from the queue removes the item G. This agrees with the FIFO theory (First-In First-Out) since the item G was the first to be inputted into the queue.

## 5. ADD ITEM TO QUEUE (Test **Cycling** Property)

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1  
Please enter a character to enter into the queue....R

-----

1	2	3	4	5	6	7	8	9	10
		F	E	F	D	T	F	G	R

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1  
Please enter a character to enter into the queue....T

-----

1	2	3	4	5	6	7	8	9	10
T		F	E	F	D	T	F	G	R

1 2 3 4 5 6 7 8 9 10

-----

## 6. ADD ITEM TO A **FULL** QUEUE

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1  
Please enter a character to enter into the queue....F

Queue is Full!

-----

1	2	3	4	5	6	7	8	9	10
T	F	F	E	F	D	T	F	G	R

1 2 3 4 5 6 7 8 9 10

-----

## 7. REMOVE ITEM FROM QUEUE (Test Cycling)

---

MAIN MENU - PLEASE ENTER YOUR CHOICE

-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program

2

Item Removed from the Queue: R

-----  
1 2 3 4 5 6 7 8 9 10

T F

1 2 3 4 5 6 7 8 9 10  
-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program

2

Item Removed from the Queue: T

-----  
1 2 3 4 5 6 7 8 9 10

F

1 2 3 4 5 6 7 8 9 10  
-----

## Section 2: Exercise 6 (linked list implementation)

**Method of testing: Black Box Testing**

**1. REMOVE AN ITEM FROM AN EMPTY QUEUE**

---

**2. ADD EMPTY ITEM**

---

**3. ADD ITEM TO QUEUE**

---

**6. ADD ITEM TO A FULL QUEUE**

---

Tests *exactly* the same as in section 1

## 5. ADD ITEM TO QUEUE (Test **Cycling** Property)

---

## 7. REMOVE ITEM FROM QUEUE (Test **Cycling**)

---

Tests *not needed* (when implementing the queue using a linked list, no cycling is needed)

## 4. REMOVE ITEM FROM QUEUE

---

```
-----  
1 2 3 4 5 6 7 8 9 10  
T e s t  
1 2 3 4 5 6 7 8 9 10  
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE  
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

2

```
Item Removed from the Queue: T
```

```
-----  
1 2 3 4 5 6 7 8 9 10  
e s t  
1 2 3 4 5 6 7 8 9 10  
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE  
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

Removing an item from the queue removes the item T. This agrees with the FIFO theory (First-In First-Out) since the item T was the first to be inputted into the queue. Note that in the *linked list* implementation, the rest of the queue is shifted down to the start when we delete an element from the queue, and this is why no cycling is required.

## Exercise 4: Implement a Linked List data structure and associated test program

**Method of testing: Black Box Testing**

### PART A - TEST THE MENU SYSTEM

#### 1. Inputting the **INCORRECT STRING**

---

```
A program to implement a Linked List data structure
by Ioan Williams, Nov '99
```

```
-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program
test
Sorry, please try again, wrong input
```

#### 2. **INCORRECT NUMBER** (i.e. not 1, 2, 3 or 4)

---

```
-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

7
Sorry, please try again, wrong input
```

#### 3. **EMPTY INPUT**

---

```
-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

Sorry, please try again, wrong input
```

## 4. EXIT FROM PROGRAM

---

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program  
  
4  
The program will now close down
```

## PART B - Linked List data structure

### 1. REMOVE ITEM FROM AN EMPTY LIST

---

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program  
  
2  
List is empty
```

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program  
  
3  
List is empty
```

### 2. ADD ITEMS TO LIST

---

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program  
  
1  
Please enter string to insert in the list.  
Ioan  
Ioan,
```

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

```
1  
Please enter string to insert in the list.  
Gareth  
Ioan, Gareth,
```

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

```
1  
Please enter string to insert in the list.  
Ian  
Ioan, Gareth, Ian,
```

### 3. REMOVE ITEM FROM THE **TOP** OF THE LIST

---

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

```
2  
Ioan, Gareth,
```

### 4. REMOVE ITEM FROM THE **BOTTOM** OF THE LIST

---

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

```
3  
Gareth,
```

## Conclusions

- Using the class **java** provides for implementing a Stack is relatively easy. Java hides how it implements the stack; we just have to provide the correct format of input.
- We can use fixed length **Arrays** to implement Stacks and Queues.
  - **Stacks**: we use a pointer to indicate the location of the next free space in the array. This can increase to a limit, where we signal “*Stack is Full*” if the limit is exceeded. The pointer decreases when we delete (pop) elements from the queue. We cannot delete elements from an empty array, hence the error “*Stack is Empty*”
  - **Queues**: we use *two* array pointers: one to indicate the position of the next free space in the array for *inserting* elements; one to indicate the location of the first element in the queue ready for *deletion*. We need a *third* variable to keep track of whether the stack is empty, full or neither of the above. This variable is needed because complications arise when the two array pointers *meet*.
- We can use **Linked Lists** to implement Stacks and Queues.
  - **Stacks**: We insert elements at the *end* of the list (traverse through the list until we find a null pointer and insert a new node there). We also remove elements from the *end* of the list (traverse through the list until we find a null pointer, return and then delete the data in that node, delete the link to the end node from the *previous* node)
  - **Queues**: We insert elements at the *end* of the list (as above). We remove elements from the *front* of the list (return the data at the root node, copy the *data* from the next node into the root node, copy the *link* from the next node into the root node).
- Implementing Stacks and Queues using *Linked Lists* is more difficult than implementing using *arrays*, because of all the links and nodes within nodes etc. involved, but using Linked Lists is better because we have no **size constraints** to our Stacks and Queues. When using arrays we have to specify the size of our stack or queue when defining it. There is no need to do this when using Linked Lists.

## Bibliography

Roger,G.Mariani, J. (1998) Java: First Contact, 1st ed. Course Technology.

Shaffer, C. (1998) A Practical Introduction To Data Structures And Algorithm Analysis, 1st ed. New Jersey: Prentice-Hall, Inc.

## References

Shaffer, C. (1998) A Practical Introduction To Data Structures And Algorithm Analysis, 1st ed. New Jersey: Prentice-Hall, Inc, pp. 104 (Constructor Methods)

# Appendix: Source Code and MS-DOS Output

---

---

**Exercise 1:** Write a program that reverses a String using the stack class provided with java.

---

---

## Program Code

---

```
/*
 *
 * Gareth Evans
 *
 * Started: 18th October 1999
 * Finished: 18th October 1999 (Revision 1.1)
 *
 * Assignment 2, Exercise 1
 *
 * Write a test program that uses the stack class provided
 * in the java language. The program should reverse
 * a sequence of characters.
 *
 * This code is also used in Exercises 2 and 5.
 * The code is slightly modified for these exercises.
 */

import java.util.*;
import java.bangor.*;

public class Reverse
{
    public static void main(String args[]) throws Exception
    {
        // Define Variables

        String inputString;
        String outputString = new String();
        Stack testStack = new Stack(); // define default sized Stack

        // Display Welcome Information

        System.out.println();
        System.out.println("A Program to reverse a set of characters.");
        System.out.print("Please enter a String to reverse...");
        inputString = BasicIo.readString(); // takes a string from the user
        System.out.println();

        for (int i=0; i < inputString.length(); i++)
        {
            System.out.print("Item Pushed = ");
            System.out.println(inputString.substring(i,i+1));
            String pushString = inputString.substring(i, i+1);

            // takes the ith character in the string (first character is at position 0)
            // and pushes it to the stack

            testStack.push(new String(pushString));
        }

        System.out.println();

        for (int j=0; j < inputString.length(); j++)
        {
            Object obj = testStack.pop();
            System.out.println("Item Popped = " + obj);

            // takes the jth object in the stack and pops it to a concatenated output string

            outputString = outputString + obj;
        }

        System.out.println();
        System.out.println("Your String after reversion is " + outputString);
    }
}
```

## MS-DOS Output

---

```
M:\e2027\Assignment 2>java Reverse
Symantec Java! JustInTime Compiler Version 210.050 for JDK 1.1
Copyright (C) 1996-97 Symantec Corporation
```

```
A Program to reverse a set of characters.
Please enter a String to reverse...TestString
```

```
Item Pushed = T
Item Pushed = e
Item Pushed = s
Item Pushed = t
Item Pushed = S
Item Pushed = t
Item Pushed = r
Item Pushed = i
Item Pushed = n
Item Pushed = g
```

```
Item Popped = g
Item Popped = n
Item Popped = i
Item Popped = r
Item Popped = t
Item Popped = S
Item Popped = t
Item Popped = s
Item Popped = e
Item Popped = T
```

```
Your String after reversion is gnirtStseT
```

```
M:\e2027\Assignment 2>
```

---

## Exercise 2: Write a program that implements a stack structure using an array.

---

### Program Code

---

```
/*
 *
 * Gareth Evans
 *
 * Started: 18th October 1999
 * Finished: 1st November 1999 (Revision 1.3)
 *
 * Constructor methods (lines highlighted // REFERENCED //) taken from "A Practical Introduction to Data
 * Structures And Algorithm Analysis, Java Edition", page 104
 *
 * Assignment 2, Exercise 2
 *
 * Class File for Stack Implementation
 * using an Array
 *
 */

public class Stack
{
    // Constant for stack size
    private static final int defaultSize = 100;

    // Declare Variables
    private Object[] internalStack;
    private int stackSize;
    private int pointer;

    // Constructor for creating a user-defined sized Stack

    Stack(int userSize) // REFERENCED //
    {
        SetupStack(userSize); // make a Stack of the user's size
    }

    // Constructor for creating a default sized Stack

    Stack() // REFERENCED //
    {
        SetupStack(defaultSize); // make a Stack of default size
    }

    // Set up the Stack

    public void SetupStack(int setSize)
    {
        internalStack = new Object[setSize]; // Define an array of size stackSize
        stackSize = setSize;
        pointer = 0; // initialize pointer at start of Array
    }

    // PUSH METHOD

    public void push(Object obj) // the object to be pushed is the parameter
    {
        if (pointer < stackSize) // i.e. is more room left in the stack?
        {
            internalStack[pointer] = obj; // push object to the stack
            pointer++; // increment value of the pointer
        }
        else
        {
            System.out.println("Stack is Full!");
        }
    }

    // POP METHOD

    public Object pop()
    {
        if (pointer==0) // i.e. is the stack empty?
        {
            return ("Stack is Empty!");
        }
        else
        {
            pointer--; // decrement pointer
            return internalStack[pointer]; // return object
        }
    }
}
```

```

/*
 *
 * Gareth Evans
 *
 * Started: 18th October 1999
 * Finished: 25th October 1999 (Revision 1.1a)
 *
 * Assignment 2, Exercise 2
 *
 * Write a program that implements a stack structure using an array.
 * This program is identical to the program in exercise 1 but that it
 * uses the StackArray class to manipulate the stack instead of the
 * standard java Stack class.
 *
 * This is the code used for Exercise 1, except that
 * we do not import java.util.*
 *
 */

import java.bangor.*;

public class Reverse
{
    public static void main(String args[]) throws Exception
    {
        // Define Variables

        String inputString;
        String outputString = new String();
        Stack testStack = new Stack(); // define default sized Stack

        // Display Welcome Information

        System.out.println();
        System.out.println("A Program to reverse a set of characters.");
        System.out.print("Please enter a String to reverse...");
        inputString = BasicIo.readString(); // takes a string from the user
        System.out.println();

        for (int i=0; i < inputString.length(); i++)
        {
            System.out.print("Item Pushed = ");
            System.out.println(inputString.substring(i,i+1));
            String pushString = inputString.substring(i, i+1);

            // takes the ith character in the string (first character is at position 0)
            // and pushes it to the stack

            testStack.push(new String(pushString)); // pushes character to stack
        }

        System.out.println();

        for (int j=0; j < inputString.length(); j++)
        {
            Object obj = testStack.pop();
            System.out.println("Item Popped = " + obj);

            // takes the jth object in the stack and pops it to a concatenated output string

            outputString = outputString + obj; // concatenates Output String
        }

        System.out.println();
        System.out.println("Your String after reversion is " + outputString);
    }
}

```

## MS-DOS Output

---

```
M:\e2027\Assignment 2\ex2>java Reverse  
Symantec Java! JustInTime Compiler Version 210.050 for JDK 1.1  
Copyright (C) 1996-97 Symantec Corporation
```

```
A Program to reverse a set of characters.  
Please enter a String to reverse...123Test
```

```
Item Pushed = 1  
Item Pushed = 2  
Item Pushed = 3  
Item Pushed = T  
Item Pushed = e  
Item Pushed = s  
Item Pushed = t
```

```
Item Popped = t  
Item Popped = s  
Item Popped = e  
Item Popped = T  
Item Popped = 3  
Item Popped = 2  
Item Popped = 1
```

```
Your String after reversion is tseT321
```

```
M:\e2027\Assignment 2\ex2>
```

---

---

**Exercise 3:** Write a program implementing a queue structure using an array. Write a demonstration program using the queue abstract data type.

---

---

## Program Code

---

```
/*
 *
 * Gareth Evans
 *
 * Started: 19th October 1999
 * Finished: 25th October 1999 (Revision 1.2)
 *
 * Assignment 2, Exercise 3
 *
 * Class File for Queue Implementation
 * using an Array
 *
 */

public class Queue
{
    // Constant for queue size
    private static final int defaultSize = 100;

    // Declare Variables
    private Object[] internalQueue;
    private int queueSize; // holds size of Queue
    private int frontPointer;
    private int backPointer;
    private int checkEmptyFull; // if the queue is empty this has value 0;
                                // if full has value 1; otherwise it is -1

    public void SetupQueue(int setSize) // Initialize queue
    {
        internalQueue = new Object[setSize]; // Define an array of size "size"
        queueSize = setSize; // set size of Queue
        frontPointer = 0; // initialise Front Pointer at start of Array
        backPointer = 0; // initialise Back Pointer at start of Array
        checkEmptyFull = 0; // i.e. Queue is empty to begin with
    }

    // Constructor for defining a default sized Queue

    Queue()
    {
        SetupQueue(defaultSize);
    }

    // Constructor for defining a user-defined size Queue

    Queue(int userSize)
    {
        SetupQueue(userSize);
    }
}
```

```
// PUSH METHOD
```

```
public void push(Object obj)          // the object to be pushed is the parameter
{
    if (checkEmptyFull == 1)          // i.e. the queue is empty
    {
        System.out.println("Queue is Full!");
    }
    else
    {
        internalQueue[backPointer] = obj; // push object to the stack
        backPointer++;                    // increment value of the pointer
        checkEmptyFull = -1;              // if the queue was empty it is not now

        if (backPointer > (queueSize-1)) // i.e. we need to recycle to the beginning
        {
            backPointer = 0;
        }

        if (backPointer == frontPointer) // i.e. queue has reached maximum capacity
        {
            checkEmptyFull = 1;
        }
    }
}
```

```
// POP METHOD
```

```
public Object pop()
{
    if (checkEmptyFull == 0) // i.e. is queue empty?
    {
        return ("Queue is Empty!");
    }
    else
    {
        Object frontPointerObj = internalQueue[frontPointer];

        // use frontPointerObj to remember the contents of the location of frontPointer
        // so we can return the contents later.

        internalQueue[frontPointer] = null; // erase array location
        frontPointer++;                       // increment front pointer
        checkEmptyFull = -1;                  // i.e. if the queue was full it is not now
        if (frontPointer > (queueSize-1))    // i.e. we need to recycle to the beginning
        {
            frontPointer = 0;
        }

        if (frontPointer == backPointer)      // i.e. the queue is now empty
        {
            checkEmptyFull = 0;
        }

        return frontPointerObj;
    }
}
```

```
// GET ELEMENT METHOD - used to display an element of the array
```

```
public Object getElement(int element)
{
    if ((element >= 0) & (element < queueSize))
        // i.e. is the parameter in the correct range?
    {
        return internalQueue[element];
    }
    else
    {
        return("Incorrect Parameter");
    }
}
```

```

/*
 *
 * Gareth Evans
 *
 * Started: 19th October 1999
 * Finished: 1st November 1999 (Revision 1.3)
 *
 * Assignment 2, Exercise 3
 *
 * Write a program that implements a queue structure using an array.
 * Write a demonstration program that uses the queue abstract data type.
 *
 * This program uses a queue to store single characters. It shows the
 * structure of the queue on screen.
 */

import java.bangor.*;

public class QueueDemo
{
    // Define Global Variables

    static final int queueLength = 10;
    static Queue testQueue = new Queue(queueLength); // define user-defined size Queue

    // Method for displaying the elements of the Queue on screen

    public static void DisplayCharacters()
    {
        System.out.println();
        System.out.println("-----");
        System.out.println("1  2  3  4  5  6  7  8  9  10");
        System.out.println();
        for( int i=0; i < queueLength; i++ )
        {
            if (testQueue.getElement(i) == null)
            {
                System.out.print("  "); // we need this to save printing out "null"
            }
            else
            {
                System.out.print(testQueue.getElement(i));
                System.out.print(" ");
            }
        }
        System.out.println();
        System.out.println();
        System.out.println("1  2  3  4  5  6  7  8  9  10");
        System.out.println("-----");
        System.out.println();
    }

    // Method for pushing elements into the Queue. Allows the user to enter a character
    // to enter into the Queue. Rejects input if it is not a single character

    public static void PushIntoQueue() throws Exception
    {
        // declare local variables
        String InputString = new String();

        System.out.print("Please enter a character to enter into the queue....");
        InputString = BasicIo.readString();
        System.out.println();

        if ( InputString.length() != 1 ) // error trapping
        {
            System.out.println("You have entered incorrect input. Please try again...");
            System.out.println();
            PushIntoQueue(); // calls itself until user enter correct information.
        }
        else // i.e. input correct
        {
            testQueue.push(InputString); // push character into Queue
            DisplayCharacters();
        }
    }
}

```

```

public static void RemoveFromQueue()
{
    Object obj = testQueue.pop(); // Here we are just removing data. Usually the
                                   // data will be used for some purpose
    System.out.println();
    System.out.println("Item Removed from the Queue: " + obj);
    System.out.println();
    DisplayCharacters();
}

public static void main(String args[]) throws Exception
{
    // Define Variables

    int Choice;

    // Display Welcome Information

    System.out.println();
    System.out.println("A Program to model a set of single characters.");
    System.out.println("Please enter SINGLE characters only.");
    System.out.println();

    // MENU PART OF PROGRAM

    do // repeat this code until the user wants to exit
    {
        System.out.println();
        System.out.println("MAIN MENU - PLEASE ENTER YOUR CHOICE");
        System.out.println("-----");
        System.out.println();
        System.out.println("1. Enter a Character into the Queue");
        System.out.println("2. Remove a Character from the Queue");
        System.out.println("3: Exit Program");
        System.out.println();

        try
        {
            Choice = BasicIo.readInteger();
        }
        catch (Exception exp) { } // Throws an exception if a String is given

        switch(Choice)
        {
            case 1: PushIntoQueue();
                    break;

            case 2: RemoveFromQueue();
                    break;

            case 3: System.out.println("The Program will now close...");
                    break;

            default: System.out.println("Sorry, Wrong Number. Please Try Again. ");
        }
    }
    while (Choice != 3);
}
}

```

# MS-DOS Output

---

```
M:\e2027\Assignment 2\ex3>java QueueDemo
Symantec Java! JustInTime Compiler Version 210.050
for JDK 1.1
Copyright (C) 1996-97 Symantec Corporation
```

```
A Program to model a set of single characters.
Please enter SINGLE characters only.
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

2

```
Item Removed from the Queue: Queue is Empty!
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

1

```
Please enter a character to enter into the
queue....T
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

T

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

1

```
Please enter a character to enter into the
queue....e
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

T e

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

1

```
Please enter a character to enter into the
queue....s
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

T e s

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

1

```
Please enter a character to enter into the
queue....t
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

T e s t

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

2

```
Item Removed from the Queue: T
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

e s t

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

1

```
Please enter a character to enter into the
queue....i
```

```
-----
1 2 3 4 5 6 7 8 9 10
```

e s t i

```
-----
1 2 3 4 5 6 7 8 9 10
-----
```

```
MAIN MENU - PLEASE ENTER YOUR CHOICE
-----
```

1. Enter a Character into the Queue
2. Remove a Character from the Queue
- 3: Exit Program

1

```
Please enter a character to enter into the
queue....m
```

```
-----
1 2 3 4 5 6 7 8 9 10
   e s t i m
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....a

```
-----
1 2 3 4 5 6 7 8 9 10
   e s t i m a
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....t

```
-----
1 2 3 4 5 6 7 8 9 10
   e s t i m a t
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....e

```
-----
1 2 3 4 5 6 7 8 9 10
   e s t i m a t e
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: e

```
-----
1 2 3 4 5 6 7 8 9 10
       s t i m a t e
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: s

```
-----
1 2 3 4 5 6 7 8 9 10
       t i m a t e
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....1

Cycle!

```
-----
1 2 3 4 5 6 7 8 9 10
       t i m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....2

```
-----
1 2 3 4 5 6 7 8 9 10
2       t i m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....3

```
-----
1 2 3 4 5 6 7 8 9 10
2 3   t i m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....4

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4 t i m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

1
Please enter a character to enter into the
queue....G

Queue is Full!

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4 t i m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: t

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4   i m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: i

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4           m a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: m

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4           a t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: a

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4           t e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: t

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4           e 1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2
Item Removed from the Queue: e

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4                1
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2

Item Removed from the Queue: 1

```
-----
1 2 3 4 5 6 7 8 9 10
2 3 4
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2

Item Removed from the Queue: 2

```
-----
1 2 3 4 5 6 7 8 9 10
3 4
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2

Item Removed from the Queue: 3

```
-----
1 2 3 4 5 6 7 8 9 10
4
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2

Item Removed from the Queue: 4

```
-----
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

2

Item Removed from the Queue: Queue is Empty!

```
-----
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 1. Enter a Character into the Queue
2. Remove a Character from the Queue
3: Exit Program

3

The Program will now close...

M:\e2027\Assignment 2\ex3>

---

## Exercise 4: Implement a linked list data structure and associated test program.

---

### Program Code

---

```
/*
 * Ioan Williams
 *
 * Started: 24th October 1999
 * Finished: 1st November 1999 (version 1.4)
 *
 * Assignment 2, Exercise 4
 * Implement a linked list data structure and associated test program
 *
 */

public class Node
{
    Object data;
    Node nextElement;

    private boolean deleteLink = false;

    Node(Object obj)
    {
        data = obj;
        nextElement = new Node();
    }

    // Creating a new node
    Node()
    {
        data = null;
        nextElement = null;
    }

    // Insert method
    public void insert(Node whatList, Object obj)
    {
        if (whatList.nextElement == null) // Checks next element in list
        {
            whatList.data = obj; // Add element to list
            whatList.nextElement = new Node(); // Create a fresh new node
        }
        else
        {
            insert(whatList.nextElement, obj); // calls the method again, within itself
        }
    }
}
```

```

public void delete(Node whatList, boolean atEnd)
{
    if (atEnd == true) // If true method deletes last element in list
    {
        if (whatList.nextElement == null) // Checks if next element in list is null
        {
            deleteLink = true; // Sets deleteLink trap to boolean true
            whatList.data = null; // Deletes current data (null)
        }
        else
        {
            delete(whatList.nextElement, true); // Calls delete method again, with true
            if (deleteLink == true)
            {
                whatList.nextElement = null; // Sets next element in list to null
                whatList.data = null; // Set current data to null
                deleteLink = false; // Resets the dataLink trap
            }
        }
    }
    else
    {
        if (whatList.data != null)
        {
            whatList.data = whatList.nextElement.data;
            // replace data in the root node with the data in the second node
            whatList.nextElement = whatList.nextElement.nextElement;
            // replace the link from the first node to the second node with the link
            // from the second node to the third node.
        }
    }
}

public void displayall(Node whatList)
{
    if (whatList.data != null) // If current data is not equal to
    null
    {
        System.out.print(whatList.data + ", "); // Print out current data
        displayall(whatList.nextElement); // Calls the method again with the
        // nextElement as the parameter
    }
}
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*
 * Ioan Williams
 *
 * Started: 24th October 1999
 * Finished: 1st November 1999 (version 1.4)
 *
 * Assignment 2, Exercise 4
 *
 * Test Program
 */

```

```

import java.bangor.*;

public class TestLinkedList
{
    static Node linkedList = new Node(); // Creates a new node
    public static void InsertItem() throws Exception
    {
        System.out.println("Please enter string to insert in the list.");
        String inputString = BasicIo.readString(); // Put the read string into input string

        linkedList.insert(linkedList, inputString); // Enters the input string into list
        linkedList.displayall(linkedList); // Displays all elements in list so far
    }
}

```

```

public static void DeleteTop() throws Exception // Method deletes elements at end of
list
{
    linkedList.delete(linkedList, true);
    // Calls delete method with true boolean passed as parameter
    linkedList.displayall(linkedList);
    // Calls method displayAll, root node passed as parameter
}

public static void DeleteBottom() throws Exception
// Method deletes element at start of list
{
    linkedList.delete(linkedList, false);
    // Done by calling delete with false boolean as parameter
    linkedList.displayall(linkedList);
    // Calls method displayAll with the root node linkedList as parameter
}

public static void main(String args[]) throws Exception // main method
{
    int Choice = 0;

    System.out.println();
    System.out.println("A program to implement a Linked List data structure");
    System.out.println("by Ioan Williams, Nov '99");
    System.out.println();

    // main menu
    do //Repeat menu until the user chooses to exit
    {
        System.out.println();
        System.out.println("-----");
        System.out.println("-----MAIN MENU-----");
        System.out.println("-----");
        System.out.println("1/ Insert item in the list.");
        System.out.println("2/ Delete item from the top of the list.");
        System.out.println("3/ Delete item from the bottom of the list.");
        System.out.println("4/ Exit the program");
        System.out.println();

        try
        {
            Choice = BasicIo.readInteger();
        }
        catch (Exception exp) { }

        switch(Choice)
        {
            case 1: InsertItem();
                    break;

            case 2: DeleteTop();
                    break;

            case 3: DeleteBottom();
                    break;

            case 4: System.out.println("The program will now close down");
                    break;

            default: System.out.println("Sorry, please try again, wrong input");
        }
    }
    while (Choice != 4);
}
}

```

## MS-DOS Output

---

D:\Assignment 2\ex4>java TestLinkedList

A program to implement a Linked List data structure  
by Ioan Williams, Nov '99

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

2

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

3

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

1  
Please enter string to insert in the list.  
Ioan  
Ioan,

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

1  
Please enter string to insert in the list.  
Gareth  
Ioan, Gareth,

```
-----  
-----MAIN MENU-----  
-----  
1/ Insert item in the list.  
2/ Delete item from the top of the list.  
3/ Delete item from the bottom of the list.  
4/ Exit the program
```

```
1
Please enter string to insert in the list.
Ian
Ioan, Gareth, Ian,
-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

2
Ioan, Gareth,
-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

3
Gareth,
-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

2

-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

2

-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

3

-----
-----MAIN MENU-----
-----
1/ Insert item in the list.
2/ Delete item from the top of the list.
3/ Delete item from the bottom of the list.
4/ Exit the program

4
The program will now close down

D:\Assignment 2\ex4>
```

---

---

**Exercise 5:** Use a linked list to implement a stack structure. Demonstrate that the stack test program in (1) works without any alteration.

---

---

## Program Code

---

**Note:** This exercise uses `Reverse.java` from Exercise 2. It is exactly the same code as in exercise two, but that in the comment at the top we state “Assignment 2, Exercise 5” not “Assignment 2, Exercise 2”.

```
/*
 *
 * Written:      Ian Roberts
 *
 * Date:        1st November 1999
 * Description:  Assignment 2, Exercise 5
 *              Class File for Node
 *              for use in the Linked List (Stack Version)
 *
 *
 *
 */

public class Node
{
    // Each Node has a data Object and a link (another Node) pointing
    // to the next Node in the list.

    Object data;
    Node nextElement;

    private Object returnObject = null; // used in the delete method for returning objects
    private int severLink = 0;         // used in the delete method for link deletion

    // CONSTRUCTOR for creating a new node. It doesn't point to anything

    Node()
    {
        data = null;
        nextElement = null;
    }

    // INSERT METHOD

    public void insert(Node whatList, Object obj) // input root Node; Object we want to insert
    {
        if (whatList.nextElement == null)      // is there another element in the list?
        {
            whatList.data = obj;                // if NOT, then set the
            whatList.nextElement = new Node();  // input parameter obj as the data portion,
                                                // create a new (linked) Node for use later.
        }
        else                                     // i.e. there is another element in the list
        {
            insert(whatList.nextElement, obj); // call method again, with the next element
        }
    }
}
```

```

// DELETE METHOD

public Object delete(Node whatList)          // pass in root note of the list
{
    if (whatList.nextElement == null)      // i.e. is there another element in the list?
    {
        severLink = 1;                      // set the "trap" to return the data in the previous element
        return returnObject;                // we need the return or java will complain.
    }
    else                                    // i.e. there is another element in the list
    {
        delete(whatList.nextElement);      // Call method again, with next element of list
        if (severLink==1)                   // i.e. has the trap been set for this element?
        {
            returnObject = whatList.data;   // return the data at this node
            whatList.nextElement = null;    // erase the link to the next element
            severLink = 0;                  // reset the trap
        }
        return returnObject;                // after going through all the recursion, what we
    }                                       // actually return (after several returns) is what
}                                           // was assigned in the "if" statement above.
}
}

```

```

////////////////////////////////////
/*
 *
 * Written:          Ian Roberts
 *
 * Date:            1st November 1999
 * Description:     Assignment 2, Exercise 5
 *                  Class File for Stack
 *
 *
 *
 */

```

```

public class Stack
{
    // Define root node in the linked list (an empty one)
    private Node linkedList = new Node();

    // In a linked list the initial size is always zero so we cannot define it. So
    // it doesn't matter if stack is called with a size parameter or not. No special
    // code for creating different sized stacks is needed.

    Stack(int userSize) { }
    Stack() { }

    // PUSH METHOD

    public void push(Object obj) // the object to be pushed is the parameter
    {
        // Call the insert method in Node.java, passing in the root node of this linked
        // list and the object to be inserted in the list.

        linkedList.insert(linkedList, obj);
    }

    // POP METHOD

    public Object pop()
    {
        if (linkedList.data == null) // i.e. is the stack empty - occurs when list is empty
        {
            return ("Stack is Empty!");
        }
        else
        {
            // Cal the delete method in Node.java, passing in the root node of this linked
            // list. The delete method deletes the last element in the list, returning the
            // data in that last element before deleting it.

            return(linkedList.delete(linkedList));
        }
    }
}

```

## MS-DOS Output

---

```
M:\e2027\Assignment 2\ex5>java Reverse
Symantec Java! JustInTime Compiler Version 210.050 for JDK 1.1
Copyright (C) 1996-97 Symantec Corporation
```

```
A Program to reverse a set of characters.
Please enter a String to reverse...Exercise 5
```

```
Item Pushed = E
Item Pushed = x
Item Pushed = e
Item Pushed = r
Item Pushed = c
Item Pushed = i
Item Pushed = s
Item Pushed = e
Item Pushed =
Item Pushed = 5
```

```
Item Popped = 5
Item Popped =
Item Popped = e
Item Popped = s
Item Popped = i
Item Popped = c
Item Popped = r
Item Popped = e
Item Popped = x
Item Popped = E
```

```
Your String after reversion is 5 esicrexE
```

```
M:\e2027\Assignment 2\ex5>
```

---

---

## Exercise 6: Use a linked list to implement a queue. Demonstrate that the queue test program in (3) works without any alteration.

---

---

### Program Code

---

```
/*
 *
 * Written:      Ian Roberts
 *
 * Date:        2nd November 1999
 * Description: Assignment 2, Exercise 6
 *              Class File for Node
 *              for use in the Linked List (Queue Version)
 *
 *
 */

public class Node
{
    // Each Node has a data Object and a link (another Node) pointing
    // to the next Node in the list.

    Object data;
    Node nextElement;

    private Object returnObject = null;    // used in the delete method for returning objects
    // CONSTRUCTOR for creating a new node. It doesn't point to anything

    Node()
    {
        data = null;
        nextElement = null;
    }

    // INSERT METHOD

    public void insert(Node whatList, Object obj) // input root Node, Object we want to insert
    {
        if (whatList.nextElement == null)        // i.e. is there another element in list?
        {
            // if NOT, then
            whatList.data = obj;                  // set obj as the data portion,
            whatList.nextElement = new Node();    // create a new (linked) Node for use later
        }
        else // i.e. there is another element in the list
        {
            insert(whatList.nextElement, obj);
            // call this method again, passing in the next element in the list
        }
    }

    // DELETE METHOD

    public Object delete(Node whatList)           // pass in root note of the list
    {
        returnObject = whatList.data;             // get the data ready for return

        // We now want to delete the first node in the list. To do this, we copy the data
        // from the 2nd node into the 1st node and copy the link from the 2nd node into
        // the 1st node

        whatList.data = whatList.nextElement.data;
        whatList.nextElement = whatList.nextElement.nextElement;

        return returnObject;                     // return data
    }

    // NEXT METHOD

    public Node nextNode(Node whatList) // pass in a node, want to return the next node
    {
        return whatList.nextElement;
    }
}
```

```

/*
 *
 * Written:          Ian Roberts
 *
 * Date:            2nd November 1999
 * Description:     Assignment 2, Exercise 6
 *                 Class File for Queue
 *
 *
 *
 */

public class Queue
{
    // Define root node in the linked list (an empty one)
    private Node linkedList = new Node();

    private int queueSize;
    private int numberOfNodes;
    private boolean sizeLimit;
    private Node storageNode = new Node();

    Queue(int userSize)
    {
        queueSize = userSize;
        numberOfNodes = 0;
        sizeLimit = true;
    }

    Queue()
    {
        queueSize = -1;
        numberOfNodes = 0;
        sizeLimit = false;
    }

    // PUSH METHOD

    public void push(Object obj) // the object to be pushed is the parameter
    {
        if ((sizeLimit = true) & (numberOfNodes < queueSize))
        {
            // Call the insert method in Node.java, passing in the root node of this linked
            // list and the object to be inserted in the list.

            linkedList.insert(linkedList, obj);
            numberOfNodes++;
        }
        else
        {
            System.out.println("Queue is Full");
        }

        if (sizeLimit = false)
        {
            linkedList.insert(linkedList, obj);
        }
    }
}

```

```

// POP METHOD

public Object pop()
{
    if (numberOfNodes == 0)           // i.e. list is empty
    {
        return ("Queue is Empty!");
    }
    else
    {
        // Call the delete method in Node.java, passing in the root node of this linked
        // list. The delete method deletes the first element in the list, returning the
        // data in that first element before deleting it.

        numberOfNodes--;
        return(linkedList.delete(linkedList));
    }
}

// GET ELEMENT METHOD

public Object getElement(int element)
{
    storageNode = linkedList;

    if (numberOfNodes < element)
    {
        return null;
    }
    else
    {
        for (int i=0; i < element; i++)
        {
            storageNode = storageNode.nextNode(storageNode);
        }
        return storageNode.data;
    }
}
}

```

## MS-DOS Output

---

D:\Assignment 2\ex6>java QueueDemo

A Program to model a set of single characters.  
Please enter SINGLE characters only.

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

2

Item Removed from the Queue: Queue is Empty!

-----

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1

Please enter a character to enter into the queue....A

-----

1 2 3 4 5 6 7 8 9 10

A

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1

Please enter a character to enter into the queue....B

-----

1 2 3 4 5 6 7 8 9 10

A B

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

2

Item Removed from the Queue: A

-----

1 2 3 4 5 6 7 8 9 10

B

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1

Please enter a character to enter into the queue....C

-----

1 2 3 4 5 6 7 8 9 10

B C

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

1

Please enter a character to enter into the queue....

-----

1 2 3 4 5 6 7 8 9 10

B C D

1 2 3 4 5 6 7 8 9 10

-----

MAIN MENU - PLEASE ENTER YOUR CHOICE

- 
1. Enter a Character into the Queue
  2. Remove a Character from the Queue
  - 3: Exit Program

(CONTINUE TO ADD ELEMENTS TO THE QUEUE)

1  
Please enter a character to enter into the queue....J

```
-----  
1 2 3 4 5 6 7 8 9 10  
B C D E F G H I J  
1 2 3 4 5 6 7 8 9 10  
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- ```
-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program
```

1  
Please enter a character to enter into the queue....K

```
-----  
1 2 3 4 5 6 7 8 9 10  
B C D E F G H I J K  
1 2 3 4 5 6 7 8 9 10  
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- ```
-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program
```

1  
Please enter a character to enter into the queue....M

Queue is Full

```
-----  
1 2 3 4 5 6 7 8 9 10  
B C D E F G H I J K  
1 2 3 4 5 6 7 8 9 10  
-----
```

MAIN MENU - PLEASE ENTER YOUR CHOICE

- ```
-----  
1. Enter a Character into the Queue  
2. Remove a Character from the Queue  
3: Exit Program
```

3  
The Program will now close...

D:\Assignment 2\ex6>

# End of Assignment 2



Ioan Williams | Ian Roberts | Gareth Evans  
Stacks, Queues and Linked Lists  
October/November 1999