Unit 3: Stacks and Queues

Stacks

A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles piles of trays in a cafeteria. New trays are put on the top of the pile and top tray is the first tray removed from the stack. For this reason, a stack is called an LIFO structure: last in /first out.

The operations on stack are as follows:

- element in the Clear (): Clear the stack
- isEmpty() : Check to see if the stack is empty.
- Push (el): Put the element el on the top of stack.
- Pop(): Take the topmost element from the stack.
- topEl() : Return the topmost stack without removing it.

Algorithm for stack implementation using arrays:

This algorithm pushes an ITEM into a stack; MAXSTK is the maximum size of the stack.

```
1. Is Stack full? If TOP = MAXSTK then PRINT "Overflow" and return.
2. Set TOP = TOP+1
3. Set STACK [TOP] = ITEM.
4. Return
```

Method to push an item onto a stack:

```
void push (int stack[], int top, int maxstack, item)
{
        if(top == maxstack – 1)
                System.out.println("Overflow");
        else
        {
                top = top + 1;
                stack[top] = item;
        }
}
```

This algorithm pop an ITEM from a stack:

```
1. If TOP =0 then PRINT "Underflow" and return.
2.  Set ITEM = STACK[TOP].
3. Set TOP = TOP-1.
4. Return
```

Method to pop an item from a stack:

```
void pop (int stack[], int top, item)
{
        if(top == – 1)
                System.out.println("Underflow");
        else
        {
                item = stack[top];
                top = top - 1;
         }
}
```

The stack can be implemented in Java using array

```
public class MyStack
{
        private int maxSize;
        private int[] stackArray;
        private int top;
        public MyStack(int s)
        {
                maxSize = s;
                stackArray = new int[maxSize];
                top = -1;
        }
        public void push(int j)
        {
                stackArray[++top] = j;
        }
        public int pop()
        {
                return stackArray[top--];
        }
        public int peek()
        {
                return stackArray[top];
        }
        public boolean isEmpty()
        {
                return (top == -1);
        }
        public boolean isFull()
        {
                return (top == maxSize - 1);
        }
        public static void main(String[] args)
        {
                MyStack ob = new MyStack(5);
                ob.push(10);
                ob.push(20);
                ob.push(30);
                ob.push(40);
                ob.push(50);

                while (!ob.isEmpty())
                {
                int value = ob.pop();
                System.out.print(value+" ");
                }
                System.out.println();
        }
}
```
The output of the above program is: 50 40 30 20 10

Linked List Implementation of Stack

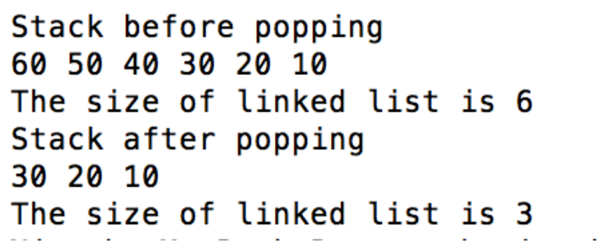
A linked list of nodes could be used to implement a stack. Linked list representation of stack is nothing but a liked list of nodes where each node is element of the stack. To point to the top most node we have a pointer top most node we have a pointer **top** that points to node recently added. Every new node is added to the beginning of the liked list and top point to it. The main advantage of this representation is that size of stack is not fixed so there is least chance of stack overflow. The structure of the node is:

```
class Node{
   protected int data;
   protected Node next;
   public Node(){
      next = null;
      data = 0;
   }
   public Node(int d){
      data = d;
      next = null;
   }
}
```

The stack can be implemented in using linked list as follows:

```
class Node
{
        protected int data;
        protected Node next;
        public Node()
        {
        next = null;
        data = 0;
        }
        public Node(int d)
        {
        data = d;
        next = null;
        }
}
class linkedStack
{
        protected Node top ;
        protected int size ;
        public linkedStack()
        {
        top = null;
        size = 0;
        }
        public boolean isEmpty()
        {
        return top == null;
        }
        public int getSize()
        {
        return size;
        }
        public void push(int data)
        {
        Node newNode = new Node (data);
        if (top == null)
                top = newNode;
        else
        {
                newNode.next = top;
                top = newNode;
        }
        size++ ;
        }
        public int pop()
        {
        if (isEmpty())
                return 0;
        Node newNode = top;
        top = newNode.next;
        size-- ;
        return newNode.data;
        }
        public void display()
        {
        if (size == 0)
        {
                System.out.print("Empty\n");
                return ;
        }
        Node newNode = top;
        while (newNode != null)
        {
                System.out.print(newNode.data+" ");
                newNode = newNode.next;
        }
        System.out.println();
        }
}
class LinkedStackImplement
{
        public static void main(String[] args)
        {
        linkedStack ls = new linkedStack();
        ls.push(10);
        ls.push(20);
        ls.push(30);
        ls.push(40);
```

```
          ls.push(50);
          ls.push(60);
          System.out.println("Stack before popping");
          ls.display();
          System.out.println("The size of linked list is "+ls.getSize());
          System.out.println("Stack after popping");
          ls.pop();
          ls.pop();
          ls.pop();
          ls.display();
          System.out.println("The size of linked list is "+ls.getSize());
          }
}
```
Output:



```
Stack before popping
60 50 40 30 20 10
The size of linked list is 6
Stack after popping
30 20 10
The size of linked list is 3
```

Queue

The queue is a linear data structure where operations of insertion and deletions are performed at separate ends known as front and rear. Whenever a new item is added to the queue, rear pointer is used. During insertion operation rear is incremented by 1 and data is stored in the queue at that location indicated by rear. Insertion of queue always takes place using rear and this operation of adding new to the end of the queue is known as enque.
The front pointer is used when an item is deleted from the queue, whenever an item is deleted from the queue, front pointer is decremented by 1 and the deleted item is returned.
Some common operations are:

- Clear(): Clear the queue.
- isEmpty() : Check to see if the queue is empty.
- Enqueue() : Put the element el at the end of the queue.
- Dequeue() : Take the first element from the queue.
- firstEL() : Return the first element in the queue without removing it.

<u>Method to insert (enqueue) an element in a queue:</u>

```
void enqueue(int queue[], int rear, int size, int data)
{
        if(isFull())
                System.out.println(“Overflow”);
        else
        {
                rear = rear + 1;
                queue[rear] = data;
                size++;
        }
}
```

<u>Method to delete (dequeue) an element from a queue:</u>

```
void dequeue(int queue[], int front, int size, int data)
{
        if(isEmpty())
                System.out.println(“Underflow”);
        else
        {
                front = front + 1;
                data = queue[front];
                size--;
        }
}
```

Queue can also be implemented in two ways.

Array Implementation

In the array implementation of linear queue, we take an array of fixed size and two variables front and rear. These two variables contain the current index at which insertion and deletion can be performed.

We use one dimensional array. Hence the size is static. The queue is empty if the front == rear and full if front == 0 and rear == n.

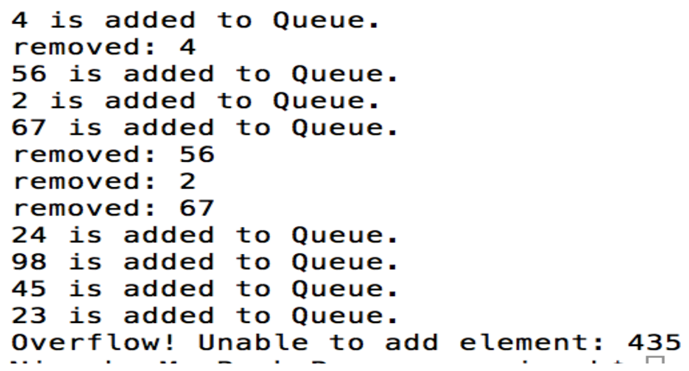
Deletion from the empty queue causes an underflow, while insertion onto a full queue causes an overflow. The following figure shows status of queue.

Front (deletion) rear (Insertion)

X	X	X	X	X	X	X	X		

```
class ArrayQueue
{
        private int capacity;
        int queueArr[];
        int front;
        int rear;
        int currentSize;
        public ArrayQueue(int queueSize)
        {
                this.capacity = queueSize;
                queueArr = new int[this.capacity];
                front =0;
                rear = -1;
                currentSize = 0;
        }
        public boolean isQueueFull()
        {
                return currentSize==capacity;
        }
        public boolean isQueueEmpty()
        {
                return currentSize==0;
        }
        public void enqueue(int item)
        {
                if (isQueueFull())
                {
                        System.out.println("Overflow! Unable to add element: "+item);
                }
                else
                {
                        rear++;
                        if(rear == capacity-1)
                        {
                                rear = 0;
                        }
                        queueArr[rear] = item;
                        currentSize++;
                        System.out.println(item+ " is added to Queue.");
                }
        }
        public void dequeue()
        {
                if (isQueueEmpty())
                {
                        System.out.println("Underflow ! Unable to remove element from Queue");
                }
                else
                {
                        front++;
                        if(front == capacity-1)
                        {
                                System.out.println("removed: "+queueArr[front-1]);
                                front = 0;
                        }
                        else
                        {
                                System.out.println("removed: "+queueArr[front-1]);
                        }
                        currentSize--;
                }
        }
        public static void main(String a[])
        {
                ArrayQueue queue = new ArrayQueue(4);
                queue.enqueue(4);  queue.dequeue();
                queue.enqueue(56); queue.enqueue(2);
                queue.enqueue(67); queue.dequeue();
                queue.dequeue(); queue.dequeue();
                queue.enqueue(24); queue.enqueue(98);
                queue.enqueue(45); queue.enqueue(23);
                queue.enqueue(435);
        }
}
```

Output:



```
4 is added to Queue.
removed: 4
56 is added to Queue.
2 is added to Queue.
67 is added to Queue.
removed: 56
removed: 2
removed: 67
24 is added to Queue.
98 is added to Queue.
45 is added to Queue.
23 is added to Queue.
Overflow! Unable to add element: 435
```

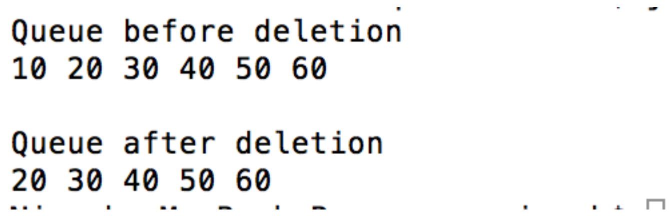
Linked List Implementation of Queue

A linear queue can be implemented using linked list. The structure of the node remains same as that of stack but we have two pointers front and rear. The front pointer points to the stating of the queue (first item inserted) and rear points to the end of the queue (last item inserted). The structure of the node is as given below:

```
class QNode
{
	public int info;
	public QNode next;
	public QNode()
	{
		info = 0;
		next = null;
	}
	public QNode(int el)
	{
		info = el;
		next = null;
	}
}
class Queue
{
	private QNode front;
	private QNode rear;
	public Queue()
	{
		front = null;
		rear = null;
	}
	public boolean isEmpty()
	{
		return front==null;
	}
	public void enqueue(int el)
	{
		QNode temp = new QNode(el);
		if(rear==null)
		{
			rear = temp;
			front = rear;
		}
		else
		{
			rear.next = temp;
			rear = temp;
		}
	}
```

```
    public void dequeue()
    {
        QNode temp;
        if(front==null)
        {
            System.out.println("Queue is empty");
            front = rear = null;
        }
        else
        {
            front = front.next;
        }
    }
    public void display()
    {
        QNode temp;
        for(temp=front; temp !=rear; temp = temp.next)
            System.out.print(temp.info+" ");
        System.out.print(temp.info);
        System.out.println();
    }
}
class LinkedListQueue
{
    public static void main(String[] args)
    {
        Queue q = new Queue();
        q.enqueue(10);   q.enqueue(20);
        q.enqueue(30);   q.enqueue(40);
        q.enqueue(50);   q.enqueue(60);
        System.out.println("Queue before deletion");
        q.display();
        System.out.println("\nQueue after deletion");
        q.dequeue();
        q.display();
    }
}
```

The output of the above program is:



```
Queue before deletion
10 20 30 40 50 60

Queue after deletion
20 30 40 50 60
```

Priority Queues
In many situations, simple queues are inadequate, as when first in /first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clear is available, a handicapped person is served instead of someone from the front of the queue.
In a sequence of processes, process p2 may need to be executed before process p1 for the proper functioning of the system, even though p1 was put on the queue of waiting processes before p2. In situations like these, a modified queue or priority queue is needed. In priority queues, elements are dequeued according to their priority and current position.
The problem with a priority queue is in finding an efficient implementation that allows relatively fast enquiring. Because elements may arrive randomly to the queue, there is no guarantee that the front element will be most likely to be dequeued and that the elements put at the end will be the most likely to be dequeued. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases such as frequency of use, birthday, salary, position, status, and others.
Priority queues can be represented by two variations of linked lists. In one type of linked list, all elements are entry ordered, and in another, order is maintained by putting a new element in its proper position according to its priority. In both cases, the total operational times are O(n) because, for an unordered list, adding an element is immediate but searching is O(n), and in a sorted list, taking an element is immediate but adding an element is O(n).

Exiting a Maze

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze. The mouse hopes to escape from the maze by systematically trying all the routes. If it reaches a dead end, it retraces its steps to the last position and begins at least one more untried path. For each position, the mouse can go in one of four directions: left, right, up, down. Regardless of how close it is to the exit, it always tries the open paths in this order, which may lead to some unnecessary detours. By retaining information that allows for resuming the search after a dead end is reached, the mouse uses a method called backtracking.

The maze is implemented as a two dimensional character array in which passages are marked with 0s and walls by 1s, exit position by the letter e, and the initial positions of the mouse by the letter m. In this program, the maze problem is slightly generalized by allowing the exit to be in any position of the maze and allowing passages to be on the borderline. To protect itself from falling off the array by trying to continue its path when an open cell is reached on one of the borderlines, the mouse also has to constantly check whether it is in such a borderline position or not. To avoid it, the program automatically puts a frame of 1s around the maze entered by the user.

The program uses two stacks: one to initialize the maze and another to implement backtracking.

The user enters a maze one line at a time. The maze entered by the user can have any number of rows and any number of columns. The only assumption the program makes is that all rows are of the same length and that it uses only these characters: any number of 1s, any number of 0s, one e, and one m. The rows are pushed on the stack mazeRows in the order they are entered after attaching one 1 at the beginning and one 1 at the end. After all rows are entered, the size of the array store can be determined and then the rows from the stack are transferred to the array.

A second stack, mazeStack, is used in the process of escaping the maze. To remember untried paths for subsequent tries, the positions of the untried neighbors of the current position (if any) are stored on a stack and always in the same order, first upper neighbor, then lower, then left, and finally right.

Here is a pseudo code of an algorithm for escaping a maze

```
exitMaze()
    initialize stack, exitCell, entryCell, currentCell = entryCell;
    while currentCell isnot exitCell
        mark currentCell as visited;
        push onto the stack the unvisited neighbours of currentCell;
        if stack is empty
            failure;
        else pop off a cell from the stack and make it currentCell;
    success;
```