KHWAJA FAREED UEIT RAHIM YAR KHAN | Faculty of Information Technology

ARRAY STACKS AND QUEUES

Lab-09

Stacks and Queues

1. **Write java program to implement the following using an Array.**
   a. **Stack ADT**
   b. **Queue ADT**

Stack

A **stack** is a conceptual structure consisting of a set of homogeneous elements and is based on the principle of last in first out (LIFO). It is a commonly used abstract data type with two major operations, namely push and pop.

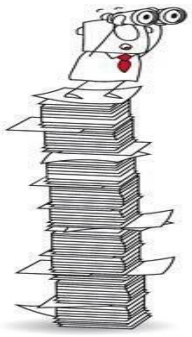

Figure 10: Stack Metaphor

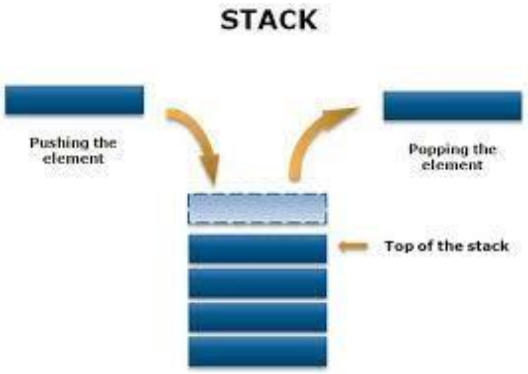

Figure 11: Stack Push and Pop illustration

Stack ADT

Stack as an Abstract Data type supports the following operations:

push(*Obj*): add object at the top of the stack.
Input: Object ; Output: None

Obj **pop**( ): Delete an item from the top of the stack and returns object obj; an error occurs if the stack is empty.
Input: None; Output: Object.

Obj **peek**( ): Returns the top object obj on the stack , without removing it; an error occurs if the stack is empty.
Input: None; Output: Object.

boolean **isEmpty**( ): Returns a boolean indicating if the stack is empty.
Input: None; Output: boolean (true or false).

boolean **isFull**( ): Returns a boolean indicating if the stack is full.
Input: None; Output: boolean (true or false).

int **size**( ): Returns the number of items on the stack.
Input: None; Output: integer.

The push, pop, peek, empty, and size operations are translated directly into specifications for methods named push(), pop(), peek(), isEmpty(), isFull(), and size() respectively. These are conventional names for stack operations. Each method is defined by specifying its return value and any changes that it makes to the object.

```
public class StackUsingArray {

    private int arr[];
    private int size;
    private int index = 0;

    public StackUsingArray(int size) {
        this.size = size;
        arr = new int[size];
    }

    public void push(int element) {

        if (isFull()) {
            System.out.println("Stack is full");
        }

        arr[index] = element;
        index++;
    }

    public int pop() {

        if (isEmpty()) {
            System.out.println("Stack is Empty");
        }
        return arr[--index];

    }
```

```
    public boolean isEmpty() {
        if (index == 0) {
            return true;
        }
        return false;
    }


    public boolean isFull() {
        if (index == size) {
            return true;
        }
        return false;
    }
```

```
public static void main(String[] args) {

        StackUsingArray stack = new StackUsingArray(5);
        stack.push(5);
        stack.push(4);

        stack.push(3);
        stack.push(2);
        stack.push(1);

        System.out.println("1. Size of stack after push operations: " +
stack.size());

        System.out.printf("2. Pop elements from stack : ");
        while (!stack.isEmpty()) {
            System.out.printf(" %d", stack.pop());
        }
```

Queue

A **Queue** is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a **queue** is any **queue** of consumers for a resource where the consumer that came first is served first. The difference between stacks and **queues** is in removing.

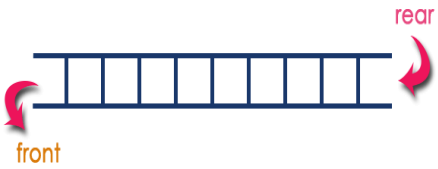

Figure 12: Queue as linear Data structure

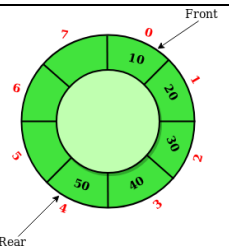

Figure 13: Queue as Circular Data Structure

Queue ADT

The elements in a queue are of generic type Object. The queue elements are linearly ordered from the front to the rear. Elements are inserted at the rear of the queue (*enqueued*) and are removed from the front of the queue (*dequeued*).

A Queue is an Abstract Data Type (ADT) that supports the following methods:

insert(*obj*): Adds object *obj* at the rear of a queue.
Input: Object; *Output*: None.

obj **remove**(): Deletes an item from the front of a queue and returns object *obj*; an error occurs if the queue is empty.
Input: None; *Output*: Object.

obj **peek**(): Returns the object *obj* at the front of a queue , without removing it; an error occurs if the queue is empty.
Input: None; *Output*: Object.

boolean **isEmpty**(): Returns a *boolean* indicating if the queue is empty.
Input: None; *Output*: *boolean* (*true* or *false*).

boolean **isFull**(): Returns a *boolean* indicating if the queue is Full.
Input: None; *Output*: *boolean* (*true* or *false*).

int **size**(): Returns the number of items in the queue.
Input: None; *Output*: *integer*.

Type Object may be any type that can be stored in the queue. The actual type of the object will be provided by the user. The ADT is translated into a Java interface in Program 17(d).

```
public interface Queue {
 public void insert(Object ob);
 public Object remove();
 public Object peek();
 public boolean isEmpty();
 public boolean isFull();
 public int size();
}
```

Note the similarities between these specifications and that of the stack interface. The only real difference, between the names of the operations, is that the queue adds new elements at the opposite end from which they are accessed, while the stack adds them at the same end.

Queue Implementation

The ArrayQueue implementation of queue interface is done by taking an array, que[n] and treating it as if it were circular. The elements are inserted by increasing rear to the next free position. When rear = n-1, the next element is entered at que[0] in case that spot is free. That is, the element que[n-1] follows que[0]. Program 17(e) implements the ArrayQueue class, and Program 17(f) tests this class.

```
class ArrayQueue implements Queue {
 private int maxSize; // maximum queue size
 private Object[] que; // que is an array
 private int front;
 private int rear;
 private int count; // count of items in queue (queue size)
 public ArrayQueue(int s) // constructor
 {
  maxSize = s;
  que = new Object[maxSize];
  front = rear = -1;
  count = 0;
 }
 public void insert(Object item) // add item at rear of queue
 {
  if (count == maxSize) {
   System.out.println("Queue is Full");
   return;
  }
  if (rear == maxSize - 1 || rear == -1) {
   que[0] = item;
   rear = 0;
   if (front == -1) front = 0;
  } else que[++rear] = item;
  count++; // update queue size
 }
 public Object remove() // delete item from front of queue
 {
  if (isEmpty()) {
   System.out.println("Queue is Empty");
   return 0;
  }
  Object tmp = que[front]; // save item to be deleted
  que[front] = null; // make deleted item's cell empty
  if (front == rear)
   rear = front = -1;
  else if (front == maxSize - 1) front = 0;
```

```
  else front++;
  count--; // less one item from the queue size
  return tmp;
 }
 public Object peek() // peek at front of the queue
 {
  return que[front];
 }
 public boolean isEmpty() // true if the queue is empty
 {
  return (count == 0);
 }
 public int size() // current number of items in the queue
 {
  return count;
 }
 public void displayAll() {
  System.out.print("Queue:  ");
  for (int i = 0; i < maxSize; i++)
  System.out.print(que[i] + " ");
  System.out.println();
 }
}
```

```
class QueueDemo {
 public static void main(String[] args) {
  /* queue holds a max of 5 items */
  ArrayQueue q = new ArrayQueue(5);
  Object item;
  q.insert('A');
  q.insert('B');
  q.insert('C');
  q.displayAll();
  item = q.remove(); // delete item
  System.out.println(item + " is deleted");
  item  =  q.remove();
  System.out.println(item + " is deleted");
  q.displayAll();
  q.insert('D'); // insert 3 more items
  q.insert('E');
  q.insert('F');
  q.displayAll();
  item = q.remove();
  System.out.println(item + " is deleted");
  q.displayAll();
  System.out.println("peek(): " + q.peek());
  q.insert('G');
  q.displayAll();
  System.out.println("Queue size: " + q.size());
 }}
```

Output of this program is as follows:

```
Queue: A B C null null
A is deleted
B is deleted
Queue: null null C null null
Queue: F null C D E
C is deleted
Queue: F null null D E
peek(): D
Queue: F G null D E
       Queue size: 4
```