STACKS, QUEUES, and DEQUES:-

Stacks:

Defs: A stack is a collection of objects that are inserted and removed according to the LIFO (last in - first out) principle.

we insert objects at the top of the stack to it's bottom by pushing them and we can remove from the top by poping the element and that technique called last in - first out.

java.util.Stack;

The ADT Stack interface is an interface which is implemented by the Stack Class and you can show the interface as following syntax:

Code:

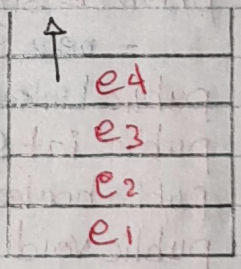
```
public interface Stack <E>{
    int size();  // return stack number of elements
    boolean isEmpty(); // return True if size = 0
    void push (E e); // insert object in stack
    E top(); // return the top element in the stack
    E pop(); // remove and return the top element in the stack
}
```

now we will implement the Stack Data Structure Based on Array as following syntax:

Code:

```
public class ArrayStack<E> implements Stack<E>{
  public static final int CAPACITY} = 1000; //N
O(N) → private ~~int~~ E[] data;
Space:- private int t = -1; // index of The Top element
Complexity public ArrayStack(){ this(CAPACITY);}
  public ArrayStack(int capacity){
    data = (E[]) new Object[capacity]; //safe cast
  }
O(1) → public int size(){ return (t+1); }
O(1) → public boolean isEmpty(){ return (t == -1);}
O(1) → public void push(E e){ throws IllegalStateException {
    if (size() == data.length)
      throw new IllegalStateException("stack is full");
    data[++t] = e;
  }
O(1) → public E top(){
    if (isEmpty()) return null;
    return data[t];
  }
O(1) → public E pop(){
Time Complexity: if (isEmpty()) return null;
    E answer = data[t];
    data[t] = null;
    t--;
    return answer;
  }}
```

(top) → last in element



stack

Of Course, you notice that There is one drawback for This implementation which is based on fixed Capacity So There are Limits of The ultimate size.

now we will use LinkedList DataStructure To Solve The problem of fixedCapacity in Array using Adapter Design Pattern.

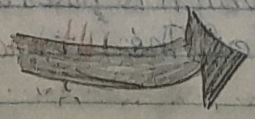
Adapter pattern is to define a new Class in such a way That it Contains an instance of The existing Class as a hidden Field, and Then to implement each method of The new Class using methods of This hidden instance Variable.

~~we Specif~~ we will implement the Stack using SinglyLinkedList Specifically as shown Syntax :

Code:

```
public class LinkedStack <E> implements Stack<E>{
   private SinglyLinkedList <E> list
        = new SinglyLinkedList <>() ;
   public LinkedStack(){}
   public int size(){return list.size();}
   public boolean isEmpty(){return list.isEmpty();}
   public void push(E element){list.addFirst(element);}
   public E top(){return list.First();}
   public E pop(){return list.removeFirst();}
}
```

now we want to use the Stack DataStructure in Something like MatChing Tags in a MarkupLanguage Like HTML and XML.



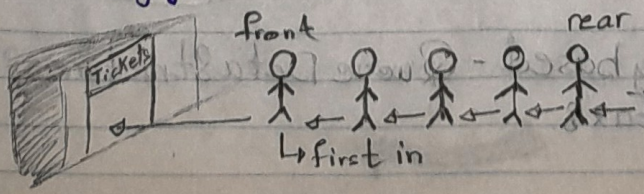
we want To make an Algorithm That Take a string Html Syntax and Test that every tag has a Closing tag or not and we write it as following Syntax :

Code:

```
public static boolean isHtmlMatched (String html) {
    Stack<String> buffer = new LinkedStack<>();
    int j = html.indexOf('<');
    while (j != -1) {
        int k = html.indexOf('>', j+1);
        if (k == -1) return false; // invalid tag
        String tag = html.substring(j+1, k);
        if (!tag.startsWith("/")) buffer.push(tag);   // opening tag
        else {                                        // closing tag
            if (buffer.isEmpty()) return false;
            if (!tag.substring(1).equals(buffer.pop()))   // no tag to match
                return false; // mismatch Tag
        }
        j = html.indexOf('<', k+1);
    }
    return buffer.isEmpty(); // were all opening tags matched?
}
```

Queues :

Def: It is a close "Cousin" of the Stack, but a Queue is a collection of objects that are inserted and removed according to the (first-in-first-out) FIFO principle.



java.util.Queue;

The Queue Class implements ADT Queue interface and This interface is written as following Syntax:

Code:

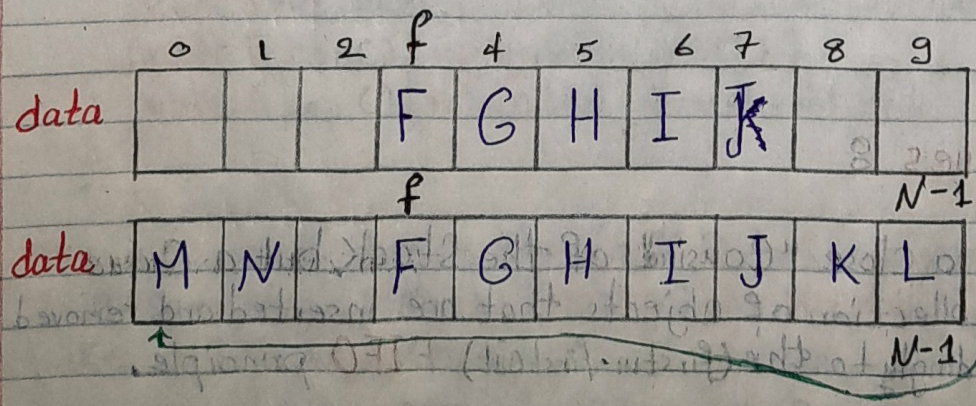
```
public interface Queue <E> {
    int size(); // return number of elements
    boolean isEmpty(); // return true if size = 0
    Void enqueue(E e); // insert element
    E first(); // return first element
    E dequeue(); // return and remove first element
}
```

now we will implement the Queue DataStructure based on fixed Array as a first development step, but we want to use array Circularly. we will use The modulo operator % using The following formula:

$$f = (f + 1) \% N$$

N → length of Array

newPosition ↓ J I ← front index



now Let's Array based - Queue DataStructure as following Syntax:

Code:

O(N) Space complexity

```
public class ArrayQueue<E> implements Queue<E>{
  private E[] data; public static int final N=1000;
  private int f =0; // front index
  private int sz=0 ; // size
  public ArrayQueue(){this(N);}
  public ArrayQueue(int capacity) {
     data = (E[])new Object[capacity] ;
  }
O(1) public int size(){return sz;}
O(1) public boolean isEmpty(){return (sz==0);}
O(1) public void enqueue(E e) throws IllegalStateException {
     if(sz==data.length)
        throw new IllegalStateException("Queue is full");
     int avail=(f+sz)%data.length; //Available index
     data[avail]=e;
     sz++;
  }
O(1) public E first(){
     if(isEmpty()) return null;
     return data[f];
  }
O(1) public E dequeue(){
     if(isEmpty()) return null;
     E answer =data[f];
     data[f]=null;
     f=(f+1)%data.length; //update front index
     sz--;
     return answer;
  }
}
```

Time complexity

now let's implement LinkedQueue class using Adapter Design Pattern by linkeList as following Syntax:

Code:

```
public class LinkedQueue <E> implements Queue <E> {
    private SinglyLinkedList<E> list
            = new SinglyLinkedList <>();
    public LinkedQueue (){}
    public int size(){ return list.size();}
    public boolean isEmpty(){retum list.isEmpty();}
    public void enqueue(E element){list.AddFirst(element);}
    public E first(){ return list.first();}
    public E dequeue(){return list.removeFirst();}
}
```

if we want to make a Circular Queue interface To rotate The elements in the Queue. we need To make CircularQueue extends Queue interface as following Syntax:

Code:

```
public interface CircularQueue <E> extends Queue <E>{
    void rotate();
} //Rotates The front element of The Queue to the back of it
```

OK, Let's use The Queue DataStructure To solve The JosephusProblem which is a childeren's game called Hot Potato That we have n children sitting in a Circular way and each of The pass The potato To his next child. when the leader rings the bell, The child who has The potato at that moment will leave The Game and continue the process up to be 1 remaining child. we want To Know Who is the last Survive when the leader rings bell at Kth child.

Let's solve This problem as following Syntax :

Code:

```
public class Josephus {
    public static <E> E josephus(CircularQueue<E> queue
            , int K){
        if (queue.isEmpty()) return null;
        while (queue.size() > 1){
Rotate by K-1 index  for (int i=0; i < K-1) queue.rotate();
            E e = queue.dequeue();
delete front K-1 element  System.out.println(" " + e + " is out");
        }
        return queue.dequeue(); // winner
    }
    public static <E> CircularQueue<E> buildQueue
            (E a[]){
        CircularQueue<E> queue
            = new LinkedCircularQueue<>();
        for (int i=0; i < a.length; i++)
            queue.enqueue(a[i]);
        return queue;
    }
    public static void main(String[] args){
        String[] a1 = {"Alice", "Bob", "Cindy", "Doug"};
        System.out.println("First winner is " +
                josephus(buildQueue(a1), 3));
    }
}
```

Double Ended - Queues :

Defs it's a datastructure like a Queue That Supports insertion and deletion at both the front and the back of The Queue.

The Deque class implements ADT Deque interface and we can write This interface as following Syntax :

Codes

```
public interface Deque <E> {
    int size();                    O(1)
    boolean isEmpty();             O(1)
    E first();                     O(1)
    E last();                      O(1)
    Void addFirst(E e);            O(1)
    Void addLast(E e);             O(1)
    E removeFirst();               O(1)
    E removeLast();                O(1)
}
```

Time Complexity

now we should make it Circular and use the following 2 formalas :

front Add , $f = (f - 1 + N) \% N$

we can implement The Deque using Doubly LinkedList by Adapter DesignPattern

notes Any method in Deque ADT interface that insert or remove element my throw a NoSuchElementException or accessing an element and return it