Module 2 | Linear Data Structures

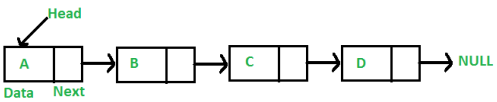
Importance	★★
Syllabus	Linear Data Structures: Arrays, Stacks, Applications of stacks, Queues, Applications of Queues, Deques, Lists, Doubly Linked Lists, Circular Linked Lists, Applications of lists.

Linked Lists

A linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

Single Linked List

Single linked list is a sequence of elements in which every element has link to its next element in the sequence. In any single linked list, the individual element is called as "Node".



Node Declaration (SLL)

```
class Node{
    int data;
    Node next;
    Node(int x){
        data=x;
        next=null;
    }
}
```

Creating Node (SLL)

`Node head = new Node(x);`

`curr.next` gives the access to next Node
`curr.data` gives the access to data in that Node

Printing (SLL)

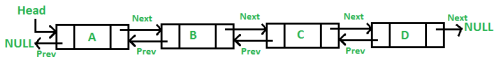
```
static void printlist(Node head){
        Node curr = head;
        while(curr!=null){
            System.out.print(curr.data+" ");
            curr = curr.next;
        }
        System.out.println();
    }
```

Insert before Head (SLL)

```
static Node insertBegin(Node head, int x){
        Node temp=new Node(x);
        temp.next=head;
        return temp;
    }

    public static void main(String args[])
    {
      Node head=null;
      head=insertBegin(head,30);
```

Doubly Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.



Node Declaration (DLL)

```
class Node{
    int data;
    Node next;
    Node prev;
    Node(int x){
        data = x;
        next = null;
        prev = null;
    }
}
```

Creating Node (DLL)

`Node head = new Node(x);`

`curr.next` gives the access to next Node
`curr.prev` gives the access to previous Node
`curr.data` gives the access to data in that Node

Printing (DLL)

```
static void printlist(Node head){
        Node curr = head;
        while(curr!=null){
            System.out.print(curr.data+" ");
            curr = curr.next;
        }
        System.out.println();
    }
//Same as SLL
```

Insert before Head (DLL)

```
static Node inserthead(Node head, int x){
        Node temp = new Node(x);
        if(head==null)return temp;
        temp.next = head;
        head.prev = temp;
        return temp;
    }
public static void main(String[] args) {
```

```
      head=insertBegin(head,20);
      head=insertBegin(head,10);
      printlist(head); // 10 20 30 " "

    }
```

Insert after Tail (SLL)

```
static Node insertEnd(Node head, int x){
        Node temp=new Node(x);
        if(head==null)return temp;//If there's no Node it makes x as Node head
        Node curr=head;
        while(curr.next!=null){ //Takes curr to last node
            curr=curr.next;
        }
        curr.next=temp;
        return head;
    }

//O(n)
```

Insert at a Position (SLL)

```
static Node insertat(Node head,int pos, int x){
        Node temp = new Node(x);
        if(head==null)return temp;
        Node curr = head;
        int i=1;
        while(i<pos-1){ //curr stops before pos
            curr = curr.next;
            i++;
        }
        temp.next = curr.next;
        curr.next = temp;
        return head;
    }
```

Search for position of X (SLL)

```
static int search(Node head, int x){
        int pos=1;
        Node curr=head;
        while(curr!=null){
            if(curr.data==x)
                return pos;
            else{
                pos++;
                curr=curr.next;
            }
        }
        return -1; //If not found
    }
```

Search for position of X I Using Recursion (SLL)

```
static int search(Node head, int x){
        if(head==null)return -1;//corner cases
        if(head.data==x)return 1;//corner cases\\head
        else{
```

```
          Node head=null;
        head=inserthead(head,30);
        head=inserthead(head,20);
        head=inserthead(head,10);
        printlist(head); // 10 20 30
    }
```

Insert after Tail (DLL)

```
static Node insertEnd(Node head,int x){
        Node temp=new Node(x);
        if(head==null)return temp;//null > temp(only e
        Node curr=head;
        while(curr.next!=null){// tail position
            curr=curr.next;//continue traversing
        }
        curr.next=temp;
        temp.prev=curr;
        return head;
    }
```

Insert at a Position (DLL)

```
static Node insertat(Node head,int pos,int x){
        Node temp = new Node(x);
        if(head==null)return temp;
        Node curr = head;
        int i=1;
        while(i<pos-1){ //curr stops before pos
            curr = curr.next;
            i++;
        }
        temp.next = curr.next;
        temp.next.prev = temp;
        curr.next = temp;
        temp.prev = curr;
        return head;
    }
```

Search for position of X (DLL)

```
//same as SLL
static int search(Node head, int x){
        int pos=1;
        Node curr=head;
        while(curr!=null){
            if(curr.data==x)
                return pos;
            else{
                pos++;
                curr=curr.next;
            }
        }
        return -1; //If not found
    }
```

Search for position of X | Using Recursion (DLL)

```
//same as SLL
static int search(Node head, int x){
```

```
            int res=search(head.next,x);
            if(res==-1)return -1;
            else return res+1;
        }
    }
```

Delete Last Node (SLL)

```
static Node deltail(Node head){
        if(head == null || head.next == null) return null;
        Node curr = head;
        while(curr.next.next!=null)//Takes curr to last 2nd Node
            curr = curr.next;
        curr.next=null;
        return head;
    }
```

Delete First Node | Head Node (SLL)

```
public static Node delhead(Node head,int n){
        if(head==null)return null;
        return head.next;
    }
```

Delete at a Position (SLL)

```
static Node delat(Node head,int pos){
        if(head == null || head.next == null) return null;
        Node curr = head;
        int i=1;
        while(i<pos-1){ //curr stops before pos
            curr = curr.next;
            i++;
        }
        curr.next=curr.next.next;
        return head;
    }
```

```
        if(head==null)return -1;//corner cases
        if(head.data==x)return 1;//corner cases\\head
        else{
            int res=search(head.next,x);
            if(res==-1)return -1;
            else return res+1;
        }
    }
```

Delete Last Node (DLL)

```
static Node deltail(Node head){
        if(head == null || head.next == null) return r
        Node curr = head;
        while(curr.next.next!=null)//Takes curr to las
            curr = curr.next;
        curr.next=null;
        return head;
    }
```

Delete First Node (DLL)

```
static Node delHead(Node head){
        if(head==null || head.next==null)return null;
        Node temp=head.next;
        temp.prev = null;
        head.next = null;
        return temp;
    }
```

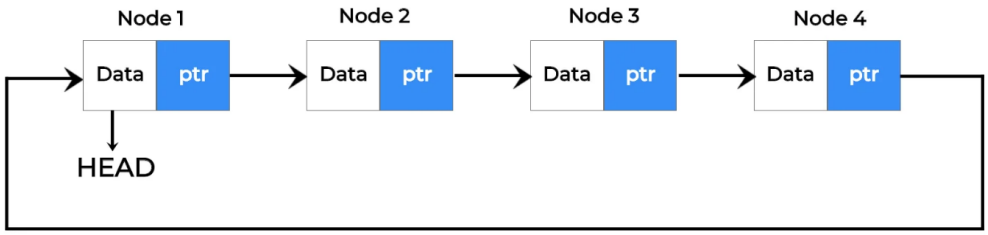
Delete at a Position (DLL)

```
static Node delat(Node head,int pos){
        if(head == null || head.next == null) return r
        Node curr = head;
        int i=1;
        while(i<pos-1){ //curr stops before pos
            curr = curr.next;
            i++;
        }
        curr.next=curr.next.next;
        curr.next.prev = curr;
        return head;
    }
```

> [Single Linked Lists]
> [Doubly Linked Lists]

Circular Linked Lists

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Printing (CLL)

```
public static void printlist(Node head){
        if(head == null) return;
        Node curr = head;
        do{
            System.out.print(curr.data+" ");
            curr=curr.next;
        }while(curr!=head);
        System.out.println();
    }
```

Insert at Front (CLL)

```
static Node insertBegin(Node head,int x){
        Node temp=new Node(x);
        if(head==null){
            temp.next=temp;
            return temp;
        }
        else{
            temp.next=head.next;
            head.next=temp;
            int t=head.data;
            head.data=temp.data;
            temp.data=t;
            return head;
        }
    }
```

One of efficient way to add an element at head is, adding node next to head and then swapping the values of temp and head nodes.

Insert at End (CLL)

```
static Node insertEnd(Node head,int x){
        Node temp=new Node(x);
        if(head==null){
            temp.next=temp;
            return temp;
        }
        else{
            temp.next=head.next;
            head.next=temp;
            int t=head.data;
            head.data=temp.data;
            temp.data=t;
            return temp;
        }
    }
```

Delete Head (CLL)

```
Node delhead(Node head)
{
    if (head==null)
    return null;
    if (head.next==head)
    return null;
    Node curr = head;
    while(curr.next!=head)
      curr=curr.next;
    curr.next=head.next;
    return curr.next;//head in main()

    //efficient
    head.data = head.next.data;
    head.next = head.next.next;//O(1)
}
```

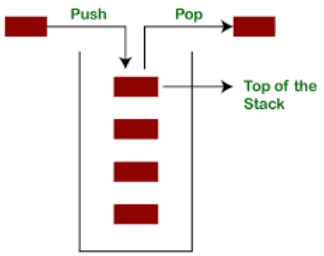
Deleting Kth Node (CLL)

```
static Node deleteKth(Node head,int k){
        if(head==null)return head;
        if(k==1)return deleteHead(head);//base case
        Node curr=head;
        for(int i=0;i<k-2;i++)
            curr=curr.next;
        curr.next=curr.next.next;
        return head;
    }
```

> Practice Problems on CLL

Stacks

A stack (an abstract data type - ADT) is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the top. The fundamental operations on a stack are push, which is equivalent to an insert, and pop, which deletes the most recently inserted element.



Basic Operations of Stack

- push(x) - Add an element to the top of a stack
- pop() - Remove an element from the top of a stack
- Peek() - Get the value of the top element without removing it
- isEmpty() - Check if the stack is empty
- isFull() - Check if the stack is Full

Stacks using Arrays

Stack Declaration

`Stack s = new Stack(capacity);`

push(x)

```
void push(int x){
        if(top == cap-1){System.out.println("Stack is full");return;}
        arr[++top] = x;
    }
```

pop()

```
int pop(){
        if(isEmpty()){System.out.println("Stack is Empty");}
        int res = arr[top];
        top--;
         return res;
    }
```

peek()

```
int peek(){
        if(isEmpty()){System.out.println("Stack is Empty");}
        return arr[top];
    }
```

isEmpty()

```
boolean isEmpty(){
        return top==-1;
    }
```

isFull()

```
boolean isFull(){
        return top==(cap-1);
    }
```

printStack()

```
//Print the stack from bottom
    void printStack(){
        for(int i=0;i<=top;i++)
            System.out.print(arr[i]+" ");
    }
```

middle_element()

```
int middle_element(){
        return arr[top/2];
    }
```

Stack Class

```
class Stack{
    int arr[];
    int cap;
    int top;
    Stack(int c){
        top = -1;
        cap = c;
        arr = new int[cap];
    }
    //Push an Element
    void push(int x){
        if(isFull()){System.out.println("Stack is full");return;}
        arr[++top] = x;
    }
    //Pop top Element
    int pop(){
        if(isEmpty()){System.out.println("Stack is Empty");}
        int res = arr[top];
        top--;
         return res;
    }
    //Peek at top Element
    int peek(){
        if(isEmpty()){System.out.println("Stack is Empty");}
        return arr[top];
    }
    //Check if stack is Empty
    boolean isEmpty(){
        return top==-1;
    }
    //Check if stack is Full
    boolean isFull(){
        return top==(cap-1);
    }
    //Print the stack from bottom
    void printStack(){
        for(int i=0;i<=top;i++)
            System.out.print(arr[i]+" ");
    }
}
```

Stacks using Linked Lists

Unlike arrays, linked lists can increase its capacity. Thus stack never gets Full.

Stack Declaration

`Stack s = new Stack();`

Node Class

```
class Node
{
    int data;
    Node next;
    Node(int d){
```

```
        data=d;
        next=null;
    }
}
```

push(x)

```
void push(int x){
        Node temp = new Node(x);
        temp.next = head;
        head = temp;
        sz++;
    }
```

pop()

```
int pop(){
        if(head==null){System.out.println("Stack is Empty");}
        int res = head.data;
        Node temp = head;
        head = head.next;
        sz--;
        return res;
    }
```

peek()

```
int peek(){
        if(head==null){System.out.println("Stack is Empty");}
        return head.data;
    }
    boolean isEmpty(){
        return head==null;
    }
```

size()

```
int size(){
        return sz;
    }
```

printStack()

```
//Prints from top to bottom
    void printStack(){
        for(Node curr = head;curr!=null;curr=curr.next)
            System.out.print(curr.data+" ");
        System.out.println();
    }
```

Stack Class

```
class Node{
    int data;
    Node next;
```

```
    Node(int x){
        data = x;
        next = null;
    }
}
class Stack{
    Node head;
    int sz;
    Stack(){
        head = null;
        sz = 0;
    }
    //Push an Element
    void push(int x){
        Node temp = new Node(x);
        temp.next = head;
        head = temp;
        sz++;
    }
    //Pop top Element
    int pop(){
        if(head==null){System.out.println("Stack is Empty");}
        int res = head.data;
        Node temp = head;
        head = head.next;
        sz--;
        return res;
    }
    //peek at top Element
    int peek(){
        if(head==null){System.out.println("Stack is Empty");}
        return head.data;
    }
    boolean isEmpty(){
        return head==null;
    }
    //Prints from top to bottom
    void printStack(){
        for(Node curr = head;curr!=null;curr=curr.next)
            System.out.print(curr.data+" ");
        System.out.println();
    }
    //Get size of Stack
    int size(){
        return sz;
    }
}
```

> Stacks

Balanced Parenthesis Checker using Stack

- When any open symbol i.e, (, {, [is encountered, it will be pushed in the stack.
- If any close symbol i.e,), },] is encountered, any of the three can happen
- The TOS (Top Of Stack) is checked, if the encountered close symbol matches with its open symbol, then open symbol which is at TOS is popped out.(OR)
- The TOS (Top Of Stack) is checked, if the encountered close symbol does not match with its open symbol, then -1 is returned as there is no matching symbol. (OR)
- The TOS (Top Of Stack) is checked,if the stack is empty, then -1 is returned as there is no open symbol in the stack.

Infix to Postfix using Stacks

Postfix notation is a notation for writing arithmetic expressions in which the operands appear before their operators. There are no precedence rules to learn, and parentheses are never needed. Because of this simplicity.

1. Create an empty Stack
2. Do the following operations for every character x from left to right.
3. If x is:
    1. Operand(a, b, c..): Output it
    2. Left parenthesis : Push to Stack
    3. Right parenthesis : Pop from Stack until left parenthesis is found. Output popped operators
    4. Operator : If Stack is empty, push x to Stack else compare with top of Stack
        1. Higher precedence than top of Stack push to Stack
        2. Lower precedence pop at top and output until a higher precedence operator is found. Push x to Stack
        3. Equal precedence, use associativity
4. Pop and output everything from Stack

The infix expression is

$(P/(Q-R)*S+T)$

Symbol	Stack	Expression
(	(	—
P	(	P
/	(/	P
(	(/(	P
Q	(/(	PQ
–	(/(–	PQ
R	(/(–	PQR
)	(/	PQR–
*	(*	PQR–/
S	(*	PQR–/S
+	(+	PQR–/S*
T	(+	PQR–/S*T
)		PQR–/ST*+

So, the postfix expression is *PQR–/ST+**.

Infix to Prefix using Stacks

Prefix notation is a notation for writing arithmetic expressions in which the operands appear after their operators. Let's assume the below

1. Reverse the infix expression
2. Scan the characters one by one
3. If the character is an operand, copy it to the prefix notation output
4. If the character is a closing parenthesis, then push it to the stack

5. if the character is an opening parenthesis, pop the elements in the stack until we find the corresponding closing parenthesis
6. If the character scanned is an operator:
    1. If the operator has precedence greater than or equal to the top of the stack, push the operator to the stack
    2. If the operator has precedence lesser than the top of the stack, pop the operator and output it to prefix notation output and then check the above condition again with the new top of the stack
7. After all the characters are scanned, reverse the prefix notation output

(a + (b * c) / (d – e)) = +a/*bc-de

NOTE: scan the infix string in reverse order.

SYMBOL	PREFIX	OPSTACK
)	Empty	)
)	Empty	))
e	e	))
-	e	))-
d	de	))-
(	-de	)
/	-de	)/
)	-de	)/)
c	c-de	)/)
*	c-de	)/)*
b	bc-de	)/)*
(	*bc-de	)/
+	/*bc-de	)+
a	a/*bc-de	)+
(	+a/*bc-de	Empty

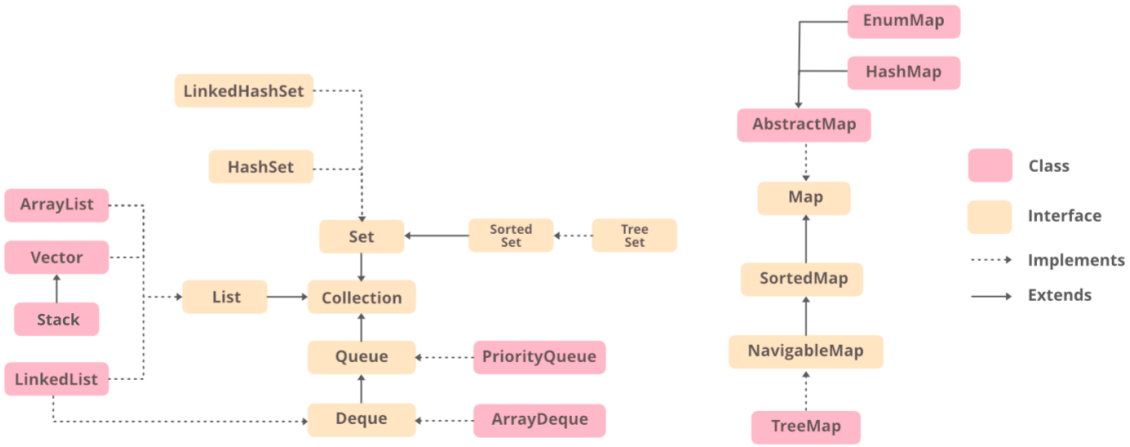
> Balanced Parenthesis Checker
> Infix to postfix
> Infix to prefix

Stacks using Collections

Collections

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



Stacks using ArrayList

Declaration `ArrayList<Integer> name = new ArrayList<>();`

Basic Operations

1. Adding Elements: `al.add(object)` or `al.add(index,object)`
2. Get an Element: `al.get(index)`
3. Removing Elements: `al.remove(index)` or `al.remove(object)`
4. Changing Elements: `al.set(index,object)`
5. Sorting Elements: `Collections.sort(al)`

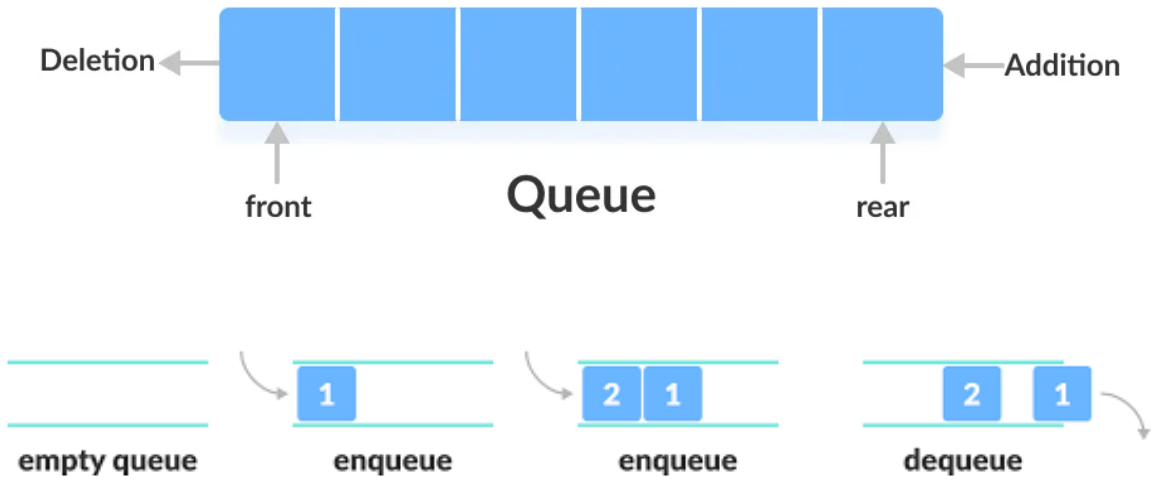
Stack class code using ArrayList

```
import java.util.*; //java.util.Collection
class MyStack{

    ArrayList<Integer> al=new ArrayList<>();
    void push(int x){
        al.add(x);
    }

    int pop(){
        int res=al.get(al.size()-1);//get gets any element in any index
        al.remove(al.size()-1);
        return res;
    }

    int peek(){
        return al.get(al.size()-1);
    }

    int size(){
```

```
        return al.size();
    }

    boolean isEmpty(){
        return al.isEmpty();
    }
}
```

> [stacks using arraylist - javapoint]

Queues



A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Basic Operations in Queues

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Queues can be implemented using:

- Arrays
- Circular Arrays
- Linked List

- Stacks

Queues using Arrays

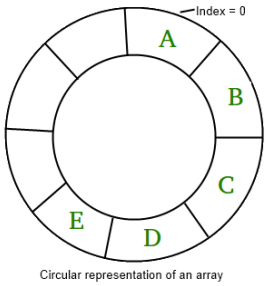
```
class q1 {
    int size,cap;
    int[] arr;
    q1(int c){
        cap = c;
        size = 0;
        arr = new int[cap];
    }
    boolean isFull(){
        return (size == cap);
    }
    boolean isEmpty(){
        return (size == 0);
    }
    void enqueue(int x){
        if(isFull())
        return;
        arr[size++] = x;
    }
    int dequeue(){
        if(isEmpty())
        return -1;
        int t=arr[0];
        for(int i=0;i<size-1;i++)
            arr[i] = arr[i+1];
        size--;
        return t;
    }
    void printqueue(){
        if(isEmpty())
        return;
        for(int i=0;i<size;i++)
            System.out.print(arr[i]+" ");
        System.out.println();
    }
    int peek(){
        if(isEmpty())
        return -1;
        return arr[size];
    }

}
```

Queues using Circular Arrays

Circular Arrays

An array is called circular if we consider the first element as next of the last element. Circular arrays are used to implement queue

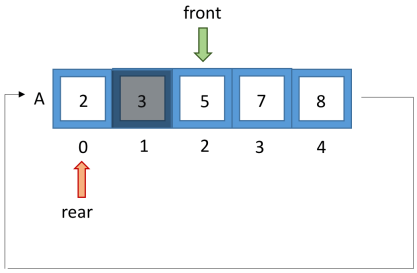
An efficient solution is to deal with circular arrays using the same array. If a careful observation is run through the array, then after n-th index, the next index always starts from 0 so using mod operator, we can easily access the elements of the circular list, if we use (i)%n and run the loop from i-th index to n+i-th index. and apply mod we can do the traversal in a circular array within the given array without using any extra space.



Circular representation of an array

Explanation:

In `enqueue()` it will insert the element at `rear` index and in `dequeue()` it increments `front` index(element is ignored while printing which is same as deleting).
With the use of `% operator` the the circulation of rear & front is achieved.
We can simply check if next index of rear is front or not, to check if queue is full.




```
class q2{
    int cap,front,rear;
    int arr[];
    q2(int c){
        cap = c;
        arr = new int[cap];
        front = rear = -1;

    }
    boolean isEmpty(){
        return (front == -1 && rear == -1);
    }
    boolean isFull(){
        return ((rear+1)%cap == front);
    }
    void enqueue(int x){
        if(isFull()){System.out.println("Full");return;}
        if(front==-1)
            front++;
        rear = (rear+1)%cap;
        arr[rear] = x;
    }
    void dequeue(){
        if(isEmpty())return;
        if(front==rear)
            front = rear = -1;
        else
            front = (front+1)%cap;
    }
    int getfront(){
        if(isEmpty())
        return -1;
        return arr[front];
    }
    int getrear(){
        if(isEmpty())
        return -1;
        else
            return arr[rear];
    }
    void printqueue(){
        if(isEmpty()) return;
        if(front<=rear)
            for(int i=front;i<=rear;i++)
                System.out.print(arr[i]+" ");
        else{
            for(int i = front;i<cap;i++)
                System.out.print(arr[i]+" ");
            for(int i=0;i<=rear;i++)
                System.out.print(arr[i]+" ");
        }
        System.out.println();
    }
}
```

Queues using LinkedList

Front and rear nodes act as pointer and work in the same way as in circular arrays

```
class QNode {
    int key;
    QNode next;
    public QNode(int key)
    {
        this.key = key;
        this.next = null;
    }
}

class Queue {
    QNode front, rear;
    int size;
    public Queue()
    {
        size = 0;
        this.front = this.rear = null;
    }

    void enqueue(int key)
    {
        size++;
        QNode temp = new QNode(key);
        if (this.rear == null) {
            this.front = this.rear = temp;
            return;
        }

        this.rear.next = temp;
        this.rear = temp;
    }

    void dequeue()
    {

        if (this.front == null)
            return;
        size--;
        QNode temp = this.front;
        this.front = this.front.next;
        if (this.front == null)
            this.rear = null;
    }
    int front(){
        return front.key;
    }
    int size(){
        return size;
    }
    void printqueue(){
        QNode i = front;
        do{
            System.out.print(i.key+" ");
            i = i.next;
        }while(i!=null);
        System.out.println();
    }
}
```

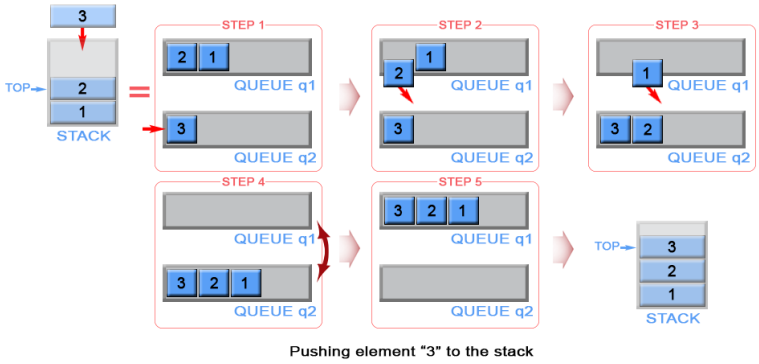
Stacks using Queues

Method

Stack can be made by using two queues. Let it be `q1` & `q2`

In `push(x)` operation, x is enqueued into q2 and the elements in q1 are enqueued after x.

Then q2 is swapped with q1, Now q2 is empty and q1 is having the current set



Pushing element "3" to the stack

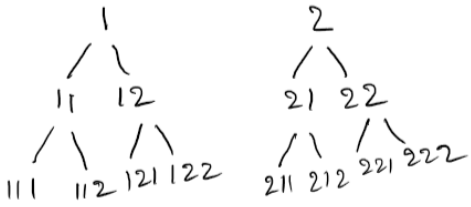
```
import java.util.*;
import java.util.Queue;
class p3 {

  static class Stack {

    static Queue<Integer> q1 = new LinkedList<Integer>();
    static Queue<Integer> q2 = new LinkedList<Integer>();


    static int curr_size;

    Stack() //const
    {
      curr_size = 0;
    }

    static void push(int x)
    {
      curr_size++;


      q2.add(x); //10


      while (!q1.isEmpty()) {
        q2.add(q1.peek());
        q1.remove();
      }


      Queue<Integer> q = q1;
      q1 = q2;
      q2 = q;
    }

    static void pop()
    {


      if (q1.isEmpty())
        return;
      q1.remove();
```

```
      curr_size--;
    }

    static int top()
    {
      if (q1.isEmpty())
        return -1;
      return q1.peek();
    }

    static int size() //O(1)
    {
      return curr_size;
    }
  }
```

Permutation & Combinations of 2 strings using queues

Method

Let s1 & s2 be the two strings, both s1 & s2 are added to queue then an element is polled using `q.poll()` and stored in a temp.
Now temp is printed, then two elements (temp + s1 & temp + s2) concatenated strings added to again into the queue.




```
static void printFirstN(int n, String s1, String s2)
  {
      Queue<String> q = new LinkedList<>();

      q.add(s1);
      q.add(s2);

      for(int i = 0; i < n; i++)
      {
          String curr = q.poll();

          System.out.print(curr+ " ");

          q.add(curr + s1);
          q.add(curr + s2);
      }

  }
```

Reversing a Queue (Recursion)

```
static Queue<Integer> reverseQueue(Queue<Integer> q)
{
    // Base case
    if (q.isEmpty())
        return q;
    // Dequeue current item (from front)
    int data = q.peek();
    q.remove();
    // Reverse remaining queue
    q = reverseQueue(q);
    // Enqueue current item (to rear)
```

```
    q.add(data);
    return q;
}
```

Deques

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

Implementation

Using Array

```
// Deque implementation in Java

class Deque {
  static final int MAX = 100;
  int arr[];
  int front;
  int rear;
  int size;

  public Deque(int size) {
    arr = new int[MAX];
    front = -1;
    rear = 0;
    this.size = size;
  }


  boolean isFull() {
    return ((front == 0 && rear == size - 1) || front == rear + 1);
  }

  boolean isEmpty() {
    return (front == -1);
  }


  void insertfront(int key) {
    if (isFull()) {
      System.out.println("Overflow");
      return;
    }

    if (front == -1) {
      front = 0;
      rear = 0;
    }

    else if (front == 0)
      front = size - 1;

    else
      front = front - 1;

    arr[front] = key;
  }
```

```
  void insertrear(int key) {
    if (isFull()) {
      System.out.println(" Overflow ");
      return;
    }

    if (front == -1) {
      front = 0;
      rear = 0;
    }

    else if (rear == size - 1)
      rear = 0;

    else
      rear = rear + 1;

    arr[rear] = key;
  }

  void deletefront() {
    if (isEmpty()) {
      System.out.println("Queue Underflow\n");
      return;
    }

    // Deque has only one element
    if (front == rear) {
      front = -1;
      rear = -1;
    } else if (front == size - 1)
      front = 0;

    else
      front = front + 1;
  }

  void deleterear() {
    if (isEmpty()) {
      System.out.println(" Underflow");
      return;
    }

    if (front == rear) {
      front = -1;
      rear = -1;
    } else if (rear == 0)
      rear = size - 1;
    else
      rear = rear - 1;
  }

  int getFront() {
    if (isEmpty()) {
      System.out.println(" Underflow");
      return -1;
    }
    return arr[front];
  }

  int getRear() {
    if (isEmpty() || rear < 0) {
      System.out.println(" Underflow\n");
      return -1;
    }
    return arr[rear];
  }
}
```

Using Circular Arrays

```
int getFront()
{
    if(isEmpty()) return -1;
    else return front;
}
int getRear()
{
    if(isEmpty()) return;
    else return (front+size-1)%cap;
}
void insertFront(int x)
{
    if(isFull()) return;
    front = (fornt + cap-1)%cap;
    arr[front] = x;
    size++;
}
void insertRear()
{
    if(isFull()) return;
    int t_rear = (front + size)%cap;
    arr[new_rear] = x;
    size++;
}
void deleteFront()
{
    if(isEmpty()) return;
    front = (front + 1)%cap;
    size--;
}
void deleteRear()
{
    if(isEmpt()) return;
    size--;
}
```

ArrayDeque

The ArrayDeque in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as Array Double Ended Queue or Array Deck. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.

```
ArrayDeque<Integer> name = new ArrayDeque<Integer>();
```

Basic Operations

Adding Elements:

- **add()**
- **addFirst()**
- **addLast()**
- **offer()**
- **offerFirst()**
- **offerLast()**

Peeking Elements:

- **getFirst()**
- **getLast()**
- **peek()**
- **peekFirst()**
- **peekLast()**

Removing Elements:

- **remove()**
- **removeFirst()**
- **removeLast()**
- **poll()**
- **pollFirst()**
- **pollLast()**
- **pop()**

> Online References
> Queues using array
> Queues using circular array
> Queues using linkedlist
> Stacks using queues
> Reversing a queue
> Deques
> Deque using circular
> ArrayDeque
> Practice Problems Queues
> Quiz on Queues