DATA STRUCTURES

WENYE LI

CUHK-SZ

OUTLINE

- Stack and Implementation
- Queue and Implementation
- Examples

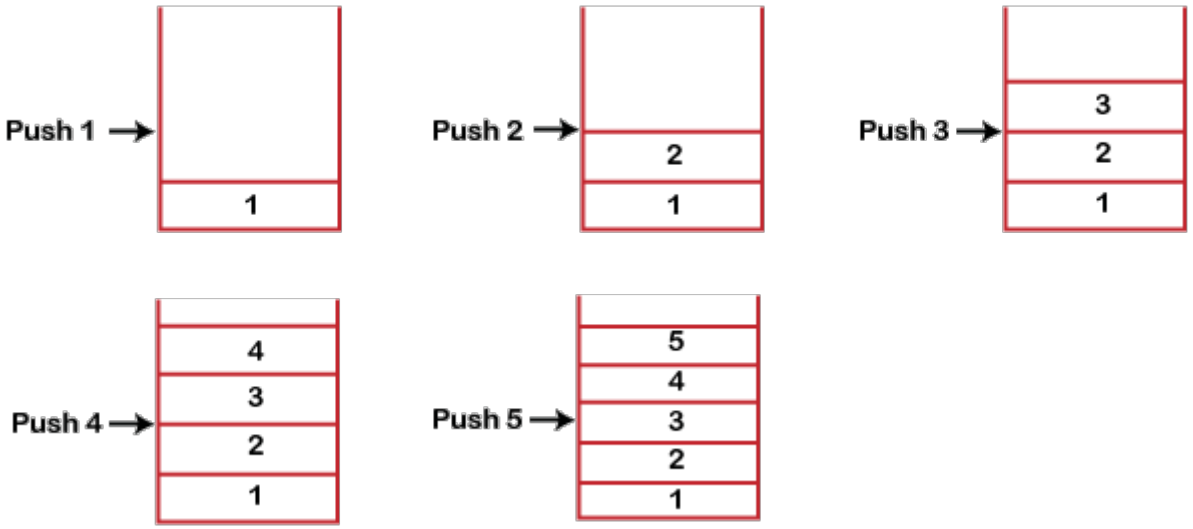
STACK

- Stack as a data structure is not related to the stack memory area !
  - They are completely different things.
  - Refer to stack as a data structure by Stack, and the stack memory area by Stack memory.

- Key points:
  - It is called as stack because it behaves like a real-world stack, piles of books, etc.
  - A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
  - It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO (Last In, First Out) or FILO (First In, Last Out).

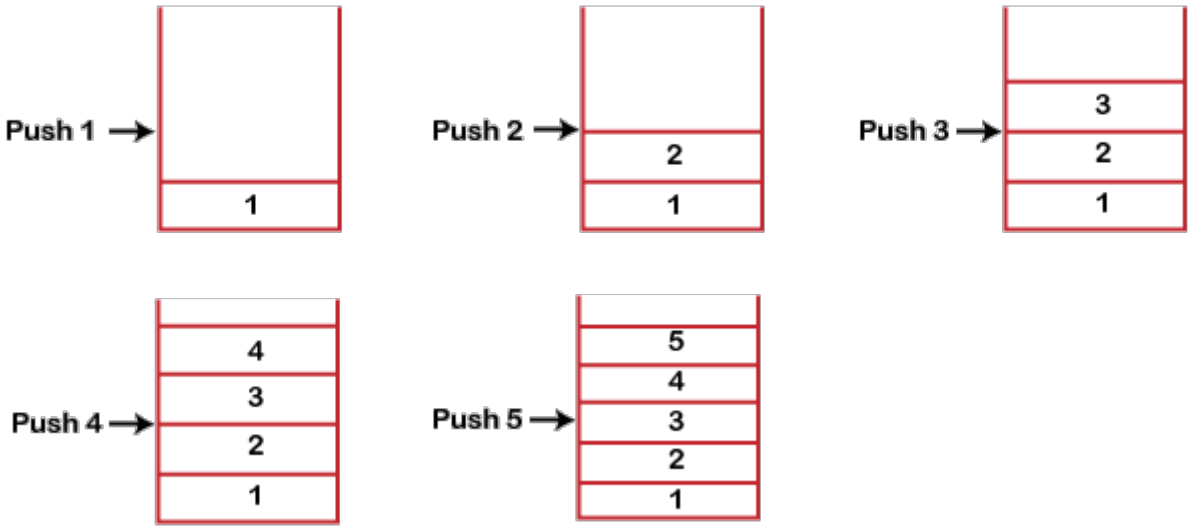
STACK

- Stack works on the LIFO pattern.
  - In the below figure there are five memory blocks in the stack; the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty. We are pushing the elements one by one until the stack becomes full.



STACK

- It goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.
- For the delete operation, it follows the LIFO pattern. The value 1 is entered first, so it will be removed only after the deletion of all the other elements.



STACK

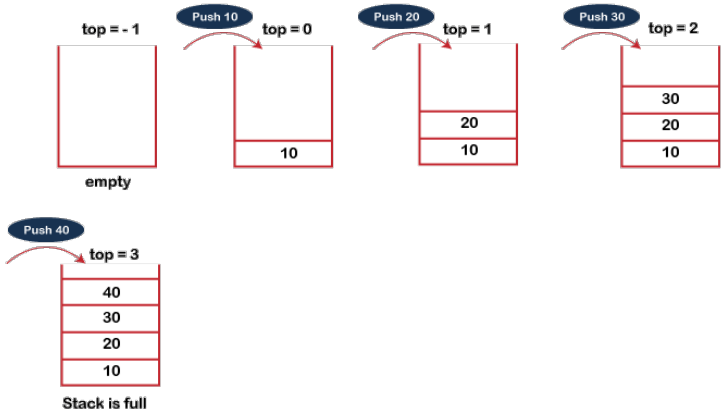
- A Stack is a linear data structure that follows LIFO principle.
- Stack has one end. It contains only one pointer top pointer pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.

- Therefore, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

OPERATIONS ON STACK

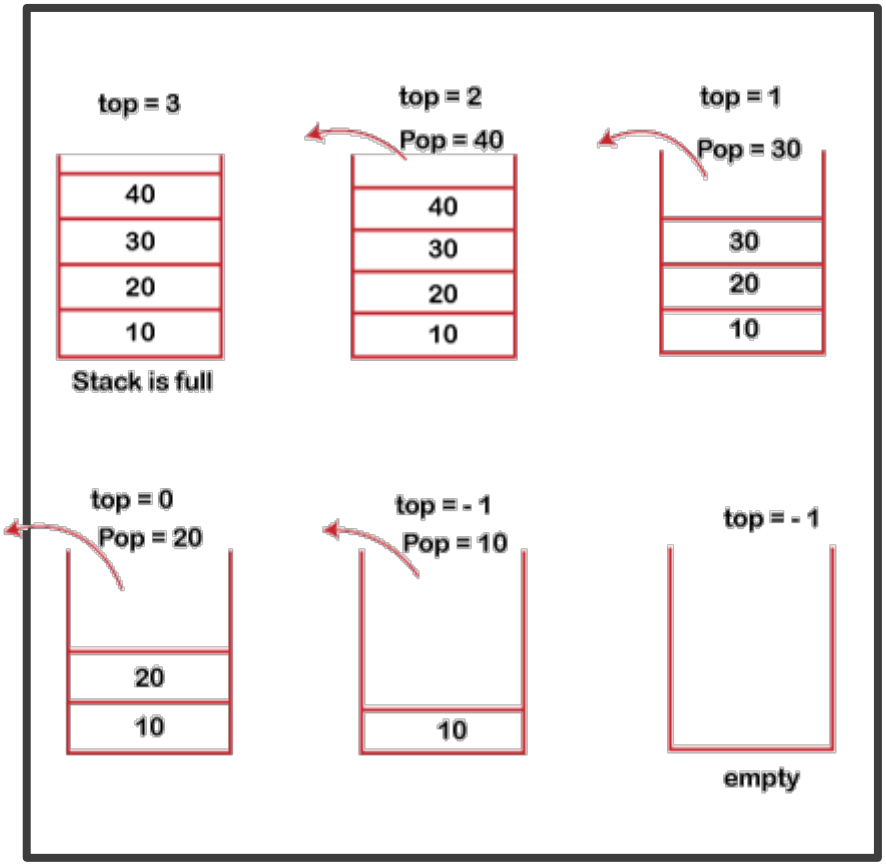
- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.'
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack.

PUSH OPERATION



- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the max size of the stack.

POP OPERATION



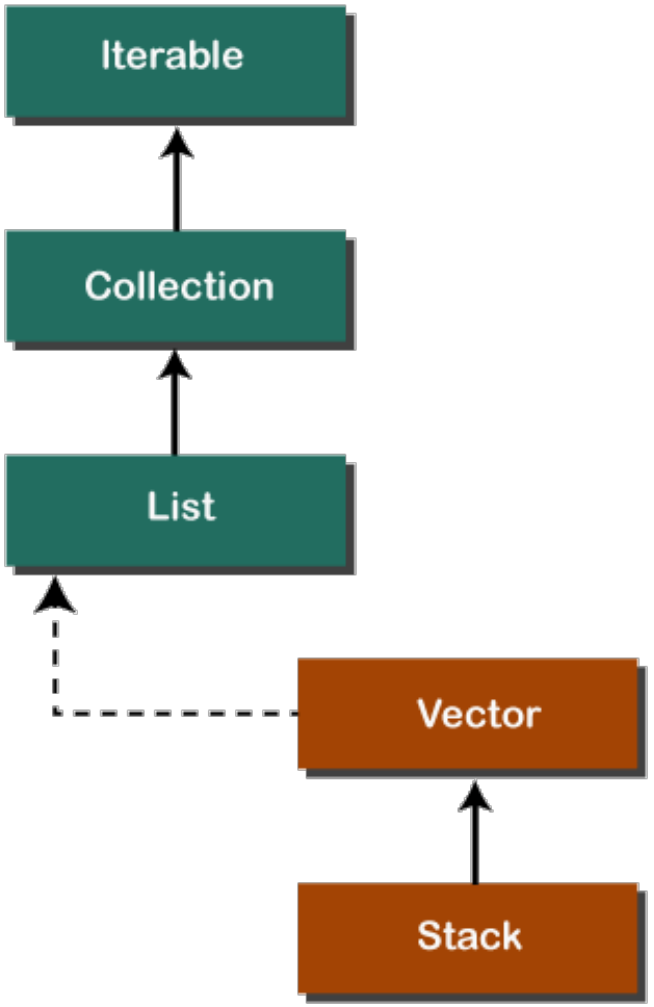
- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top.
- Once pop operation is performed, the top is decremented by 1, i.e., top=top-1.

OPERATIONS ON STACK

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

STACK IN JAVA

- In Java, Stack is a class that falls under the Collection framework that extends the Vector class.
- It also implements interfaces List, Collection, Iterable, Cloneable, Serializable.
- It represents the LIFO stack of objects.
- Before using the Stack class, we must import the java.util package.



```java
import java.util.Stack;

public class StackEmptyMethodExample
{
    public static void main(String[] args)
    {
        //creating an instance of Stack class
        Stack<Integer> stk = new Stack<>();
        // checking stack is empty or not
        boolean result = stk.empty();
        System.out.println("Is the stack empty? " + result);
        // pushing elements into stack
        stk.push(78);
        stk.push(113);
        stk.push(90);
        stk.push(120);
        //prints elements of the stack
        System.out.println("Elements in Stack: " + stk);
        result = stk.empty();
        System.out.println("Is the stack empty? " + result);
    }
}
```

Output:

```
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false
```

```java
import java.util.*;
public class StackPushPopExample {
    public static void main(String args[]) {
        //creating an object of Stack class
        Stack <Integer> stk = new Stack<>();
        System.out.println("stack: " + stk);
        //pushing elements into the stack
        pushelmnt(stk, 20);
        pushelmnt(stk, 13);
        pushelmnt(stk, 89);
        pushelmnt(stk, 90);
        pushelmnt(stk, 11);
        pushelmnt(stk, 45);
        pushelmnt(stk, 18);
        //popping elements from the stack
        popelmnt(stk);
        popelmnt(stk);
        //throws exception if the stack is empty
        try {
            popelmnt(stk);
        } catch (EmptyStackException e) {
            System.out.println("empty stack");
        }
    }
    //performing push operation
    static void pushelmnt(Stack stk, int x) {
        //invoking push() method
        stk.push(new Integer(x));
        System.out.println("push -> " + x);
        //prints modified stack
        System.out.println("stack: " + stk);
    }
    //performing pop operation
    static void popelmnt(Stack stk) {
        System.out.print("pop -> ");
        //invoking pop() method
        Integer x = (Integer) stk.pop();
        System.out.println(x);
        //prints modified stack
        System.out.println("stack: " + stk);
    }
}
```

Output:

```
stack: []
push -> 20
stack: [20]
push -> 13
stack: [20, 13]
push -> 89
stack: [20, 13, 89]
push -> 90
stack: [20, 13, 89, 90]
push -> 11
stack: [20, 13, 89, 90, 11]
push -> 45
stack: [20, 13, 89, 90, 11, 45]
push -> 18
stack: [20, 13, 89, 90, 11, 45, 18]
pop -> 18
stack: [20, 13, 89, 90, 11, 45]
pop -> 45
stack: [20, 13, 89, 90, 11]
pop -> 11
stack: [20, 13, 89, 90]
```

```java
import java.util.Stack;

public class StackPeekMethodExample
{
    public static void main(String[] args)
    {
        Stack<String> stk = new Stack<>();
        // pushing elements into Stack
        stk.push("Apple");
        stk.push("Grapes");
        stk.push("Mango");
        stk.push("Orange");
        System.out.println("Stack: " + stk);
        // Access element from the top of the stack
        String fruits = stk.peek();
        //prints stack
        System.out.println("Element at top: " + fruits);
    }
}
```

Output:

```
Stack: [Apple, Grapes, Mango, Orange]
Element at the top of the stack: Orange
```

```java
import java.util.Stack;
public class StackSearchMethodExample
{
    public static void main(String[] args)
    {
        Stack<String> stk = new Stack<>();
        //pushing elements into Stack
        stk.push("Mac Book");
        stk.push("HP");
        stk.push("DELL");
        stk.push("Asus");
        System.out.println("Stack: " + stk);
        // Search an element
        int location = stk.search("HP");
        System.out.println("Location of Dell: " + location);
    }
}
```

```java
import java.util.Stack;

public class StackSizeExample
{
    public static void main (String[] args)
    {
        Stack stk = new Stack();
        stk.push(22);
        stk.push(33);
        stk.push(44);
        stk.push(55);
        stk.push(66);
        // Checks the Stack is empty or not
        boolean rslt = stk.empty();
        System.out.println("Is the stack empty or not? " + rslt);
        // Find the size of the Stack
        int x = stk.size();
        System.out.println("The stack size is: " + x);
    }
}
```

Output:

```
Is the stack empty or not? false
The stack size is: 5
```

```java
import java.util.Iterator;
import java.util.Stack;
public class StackIterationExample1
{
    public static void main (String[] args)
    {
        //creating an object of Stack class
        Stack stk = new Stack();
        //pushing elements into stack
        stk.push("BMW");
        stk.push("Audi");
        stk.push("Ferrari");
        stk.push("Bugatti");
        stk.push("Jaguar");
        //iteration over the stack
        Iterator iterator = stk.iterator();
        while(iterator.hasNext())
        {
            Object values = iterator.next();
            System.out.println(values);
        }
    }
}
```

Output:

```
BMW
Audi
Ferrari
Bugatti
Jaguar
```

```java
import java.util.*;
public class StackIterationExample2
{
    public static void main (String[] args)
    {
        //creating an instance of Stack class
        Stack <Integer> stk = new Stack<>();
        //pushing elements into stack
        stk.push(119);
        stk.push(203);
        stk.push(988);
        System.out.println("Iteration over the stack using forEach() Method:");
        //invoking forEach() method for iteration over the stack
        stk.forEach(n ->
        {
            System.out.println(n);
        });
    }
}
```

Output:

```
Iteration over the stack using forEach() Method:
119
203
988
```

```java
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Stack;

public class StackIterationExample3
{
    public static void main (String[] args)
    {
        Stack <Integer> stk = new Stack<>();
        stk.push(119);
        stk.push(203);
        stk.push(988);
        ListIterator<Integer> ListIterator = stk.listIterator(stk.size());
        System.out.println("Iteration over the Stack from top to bottom:");
        while (ListIterator.hasPrevious())
        {
            Integer avg = ListIterator.previous();
            System.out.println(avg);
        }
    }
}
```

Output:

```
Iteration over the Stack from top to bottom:
988
203
119
```

IMPLEMENTATION

- We make a static array with size 100, which is the maximum size the Stack can have. Initially, our Stack is empty.
- The pseudo-code is presented in C-style.

```
struct CharStack1000
{
    char buffer[ 1000 ];
    int top = -1;
    // Default value of top is -1 when declaring the stack.
    // -1 means our stack is empty
};

void push( CharStack1000 &stack , char newElement )
{
    ++stack.top;
    stack.buffer[ stack.top ] = newElement;
}

char front( CharStack1000 &stack )
{
    return stack.buffer[ stack.top ];
}

void pop( CharStack1000 &stack )
{
    --stack.top;
}

int size( CharStack1000 &stack )
{
    return ( stack.top + 1 ); // simple
}

bool isEmptyStack( CharStack1000 &stack )
{
    return ( stack.top == -1 );
}
```

```java
import static java.lang.System.exit;
// Create Stack Using Linked list
class StackUsingLinkedlist {
    // A linked list node
    private class Node {
        int data; // integer data
        Node link; // reference variable Node type
    }
    // create global top reference variable global
    Node top;
    // Constructor
    StackUsingLinkedlist() {
        this.top = null;
    }
    // Utility function to add an element x in the stack
    public void push(int x) { // insert at the beginning
        // create new node temp and allocate memory
        Node temp = new Node();
        // check if stack (heap) is full. Then inserting an
        // element would lead to stack overflow
        if (temp == null) {
            System.out.print("\nHeap Overflow");
            return;
        }
        // initialize data into temp data field
        temp.data = x;
        // put top reference into temp link
        temp.link = top;
        // update top reference
        top = temp;
    }
    // Utility function to check if the stack is empty or not
    public boolean isEmpty() {
        return top == null;
    }
    // Utility function to return top element in a stack
    public int peek() {
        // check for empty stack
        if (!isEmpty()) {
            return top.data;
        } else {
            System.out.println("Stack is empty");
            return -1;
        }
    }
    // Utility function to pop top element from the stack
    public void pop() { // remove at the beginning
        // check for stack underflow
        if (top == null) {
            System.out.print("\nStack Underflow");
            return;
        }
        // update the top pointer to point to the next node
        top = (top).link;
    }
    public void display() {
        // check for stack underflow
        if (top == null) {
            System.out.printf("\nStack Underflow");
            exit(1);
        } else {
            Node temp = top;
            while (temp != null) {
                // print node data
                System.out.printf("%d->", temp.data);
                // assign temp link to temp
                temp = temp.link;
            }
        }
    }
}
public class GFG {
    public static void main(String[] args) {
        StackUsingLinkedlist obj = new StackUsingLinkedlist();
        obj.push(11);
        obj.push(22);
        obj.push(33);
        obj.push(44);
        // print Stack elements
        obj.display();
        // print Top element of Stack
        System.out.printf("\nTop element is %d\n", obj.peek());
        // Delete top element of Stack
        obj.pop();
        obj.pop();
        // print Stack elements
        obj.display();
        // print Top element of Stack
        System.out.printf("\nTop element is %d\n", obj.peek());
    }
}
```

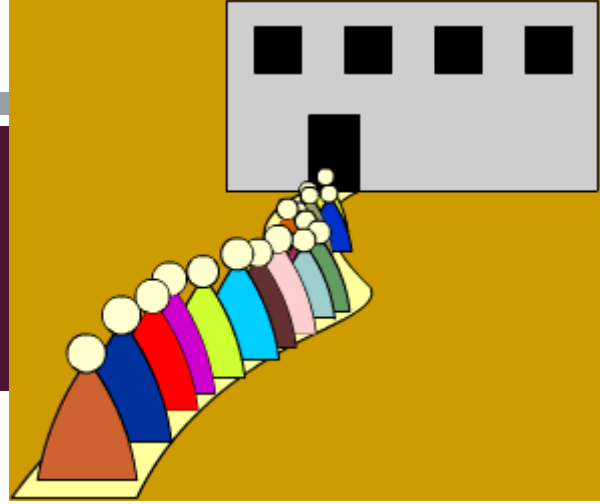
Output:

```
44->33->22->11->
Top element is 44
22->11->
Top element is 22
```

OUTLINE

- Stack and Implementation
- Queue and Implementation
- Examples

LINEAR QUEUE



- Queue: an ordered list which enables
  - insert operations to be performed at one end called REAR
  - delete operations to be performed at another end called FRONT

- Queue is referred to be as First In First Out list (FIFO).
  - For example, people waiting in line for a rail ticket form a queue.

QUEUE APPLICATIONS

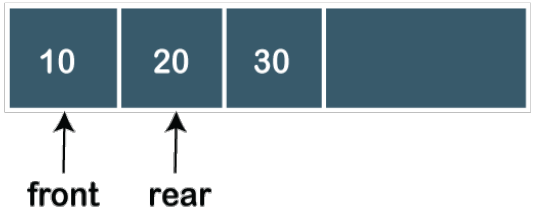
- Widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- Used as buffers in most of the applications like MP3 media player, CD player, etc.
- Used to maintain the play list in media players in order to add and remove the songs from the play-list.
- Used in operating systems for handling interrupts.

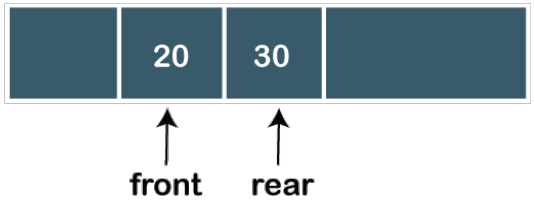
QUEUE APPLICATIONS

- Suppose we have a printer shared between various machines in a network. The printer can serve a single request at a time.
  - When any print request comes from the network, and if the printer is busy, the printer's program will put the print request in a queue.
  - If the requests are available in the queue, the printer takes a request from the front of the queue, and serves it.

- The processor in a computer is used as a shared resource. There are multiple requests that the processor must execute, but the processor can execute a single process at a time.
  - The processes are kept in a queue for execution.

LINEAR QUEUE





- An insertion takes place from the rear while the deletion occurs from the front.
  - The elements are inserted from the rear. To insert more elements in a queue, the rear value gets incremented on every insertion.
  - The front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.
- Drawback: insertion is done only from the rear end.
  - If the first three elements are deleted from the queue, we cannot insert more elements even though the space is available in a linear queue. The linear queue shows the overflow condition as the rear is pointing to the last element of the queue.

QUEUE OPERATIONS

- Enqueue: Insert the element at the rear end of the queue. It returns void.
- Dequeue: Delete from the front-end of the queue. It returns the element which has been removed from the front-end.
- Peek: Return the element, which is pointed by the front pointer in the queue but does not delete it.
- Queue overflow (isfull): When the queue is completely full, then it shows the overflow condition.
- Queue underflow (isempty): When the queue is empty, it throws the underflow condition.

ALGORITHM TO INSERT AN ELEMENT

- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

ALGORITHM TO DELETE AN ELEMENT

- If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

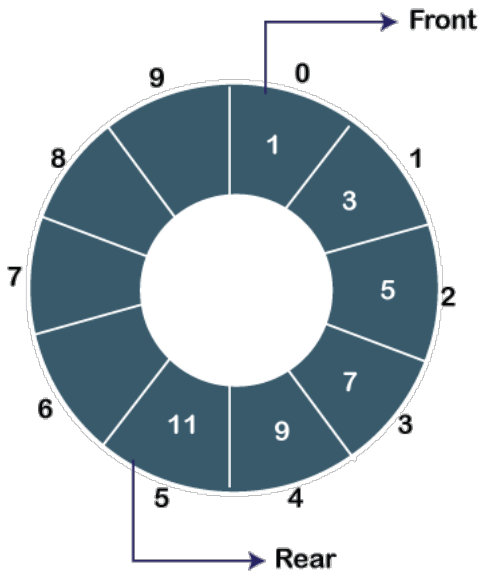
IMPLEMENTATION

- Sequential allocation: The sequential allocation in a queue can be implemented using an array.

- Linked list allocation: The linked list allocation in a queue can be implemented using a linked list.

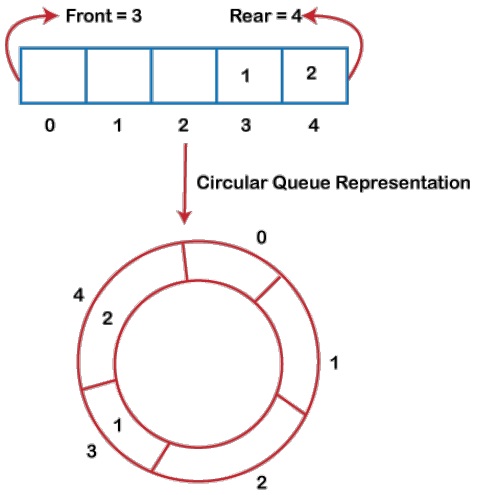
```java
// A class to represent a queue
class Queue {
    private int[] arr;      // array to store queue elements
    private int front;      // front points to the front element in the queue
    private int rear;       // rear points to the last element in the queue
    private int capacity;   // maximum capacity of the queue
    private int count;      // current size of the queue

    // Constructor to initialize a queue
    Queue(int size) {
        arr = new int[size];
        capacity = size;
        front = 0;
        rear = -1;
        count = 0;
    }

    // Utility function to dequeue the front element
    public void dequeue() {
        // check for queue underflow
        if (isEmpty())
        {
            System.out.println("Underflow\nProgram Terminated");
            System.exit(1);
        }

        System.out.println("Removing " + arr[front]);

        front = (front + 1) % capacity;
        count--;
    }

    // Utility function to add an item to the queue
    public void enqueue(int item) {
        // check for queue overflow
        if (isFull()) {
            System.out.println("Overflow\nProgram Terminated");
            System.exit(1);
        }

        System.out.println("Inserting " + item);

        rear = (rear + 1) % capacity;
        arr[rear] = item;
        count++;
    }

    // Utility function to return the front element of the queue
    public int peek() {
        if (isEmpty()) {
            System.out.println("Underflow\nProgram Terminated");
            System.exit(1);
        }
        return arr[front];
    }

    // Utility function to return the size of the queue
    public int size() {
        return count;
    }

    // Utility function to check if the queue is empty or not
    public Boolean isEmpty() {
        return (size() == 0);
    }

    // Utility function to check if the queue is full or not
    public Boolean isFull() {
        return (size() == capacity);
    }
}

class Main {
    public static void main (String[] args) {
        // create a queue of capacity 5
        Queue q = new Queue(5);
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);
        System.out.println("The front element is " + q.peek());
        q.dequeue();
        System.out.println("The front element is " + q.peek());
        System.out.println("The queue size is " + q.size());

        q.dequeue();
        q.dequeue();
        if (q.isEmpty()) {
            System.out.println("The queue is empty");
        } else {
            System.out.println("The queue is not empty");
        }
    }
}
```

CIRCULAR QUEUE



- All the nodes are represented as circular.
  - Similar to the linear queue except that the last element of the queue is connected to the first element.
  - Also known as Ring Buffer as all the ends are connected to another end.
- Drawback that occurs in a linear queue is overcome.
  - If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

CIRCULAR QUEUE



- In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the queue. If we try to insert the element then it will show that there are no empty spaces in the queue.

- Solution: The rear is at the last position of the queue and front is pointing somewhere rather than the 0^{th} position.

APPLICATIONS OF CIRCULAR QUEUE

- **Memory management:** The circular queue managed memory more efficiently (than linear queue) by placing the elements in a location which is unused.

- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.

- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

ENQUEUE ALGORITHM

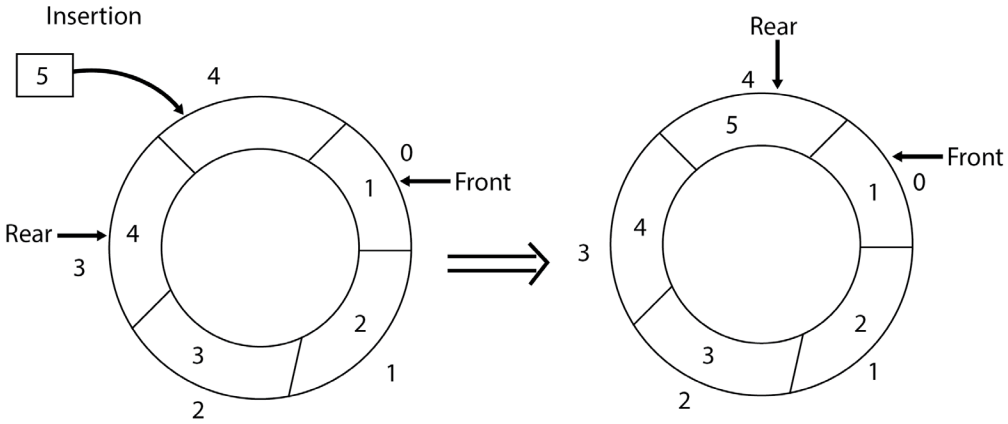
Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

- First, check whether the queue is full or not.
- Initially set both front and rear to -1. To insert the first element, both front and rear are set to 0.
- To insert a new element, the rear gets incremented.



DEQUEUE ALGORITHM

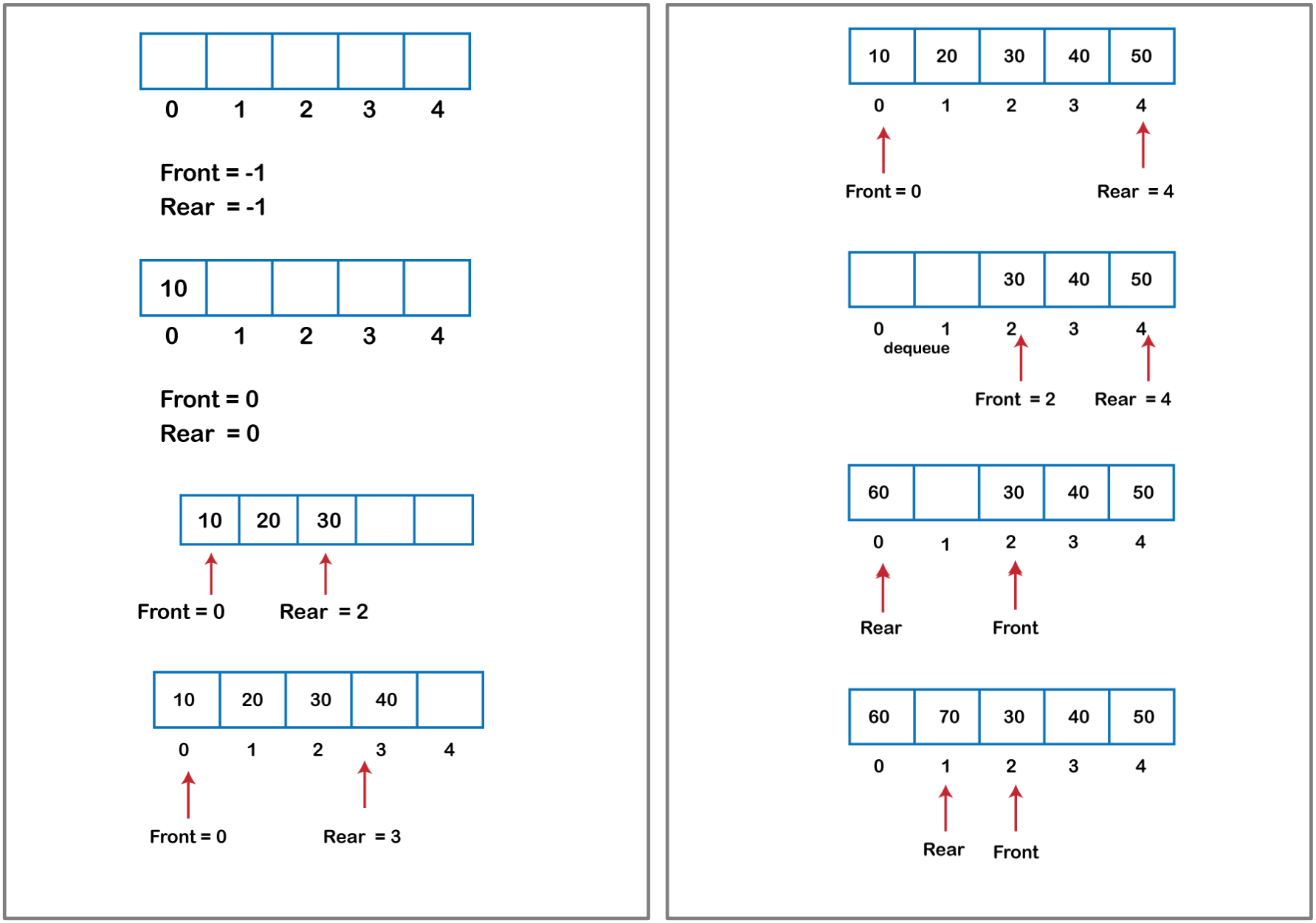
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT

- First, check whether the queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.



Implementation of Circular Queue using an array

```java
import java.io.*;
class CircularQ {
    int Q[] = new int[100];
    int n, front, rear;
    static BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
    public CircularQ(int nn) {
        n=nn;
        front = rear = 0;
    }
    public void enqueue(int v) {
        if((rear+1) % n != front) {
            rear = (rear+1)%n;
            Q[rear] = v;
        }
        else
            System.out.println("Queue is full !");
    }
    public int dequeue() {
        int v;
        if(front!=rear) {
            front = (front+1)%n;
            v = Q[front];
            return v;
        }
        else
            return -9999;
    }
    public void disp() {
        int i;
        if(front != rear) {
            i = (front +1) %n;
            while(i!=rear) {
                System.out.println(Q[i]);
                i = (i+1) % n;
            }
        }
        else
            System.out.println("Queue is empty !");
    }
    public static void main(String args[]) throws IOException {
        System.out.print("Enter the size of the queue : ");
        int size = Integer.parseInt(br.readLine());
        CircularQ call = new CircularQ(size);
        int choice;
        boolean exit = false;
        while(!exit) {
            System.out.print("\n1 : Add\n2 : Delete\n" +
                "3 : Display\n4 : Exit\n\nYour Choice : ");
            choice = Integer.parseInt(br.readLine());
            switch(choice) {
                case 1 :
                    System.out.print("\nEnter number to be added :");
                    int num = Integer.parseInt(br.readLine());
                    call.enqueue(num);
                    break;
                case 2 :
                    int popped = call.dequeue();
                    if(popped != -9999)
                        System.out.println("\nDeleted : " +popped);
                    else
                        System.out.println("\nQueue is empty !");
                    break;
                case 3 :
                    call.disp();
                    break;
                case 4 :
                    exit = true;
                    break;
                default :
                    System.out.println("\nWrong Choice !");
                    break;
            }
        }
    }
}
```

```
Enter the size of the queue : 5

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 31

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 28

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 9

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 56

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 3
31
28
9

1 : Add
2 : Delete
3 : Display
4 : Exit
```

PRIORITY QUEUE

- Behaves similarly to normal queue except that each element has some priority
  - The element with the highest priority would come first in a priority queue.
  - The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

- Only supports comparable elements
  - The elements are either arranged in an ascending or descending order.

- For example, suppose we have values 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, 1 has the highest priority while 22 has the lowest priority.

CHARACTERISTICS OF PRIORITY QUEUE

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

CHARACTERISTICS OF PRIORITY QUEUE

- Consider a priority queue with the following values: **1, 3, 4, 8, 14, 22**
- All values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:
  - **poll():** remove the highest priority element from the priority queue. The '1' element has the highest priority, so it will be removed from the priority queue.
  - **add(2):** Insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
  - **poll():** Remove '2' element from the priority queue as it has the highest priority queue.
  - **add(5):** Insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

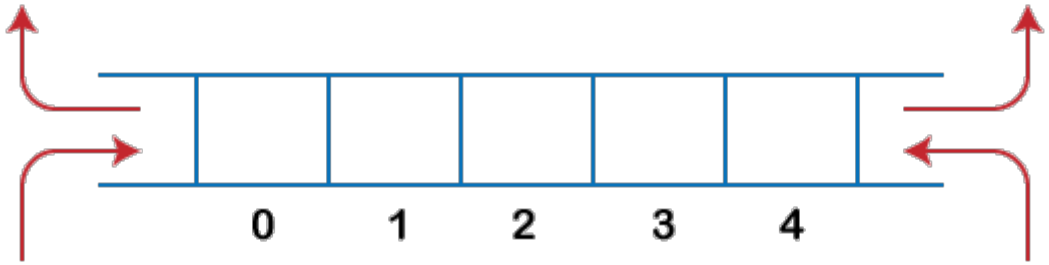
IMPLEMENTATION

- The priority queue can be implemented in four ways:
  - arrays
  - linked list
  - heap data structure
  - binary search tree

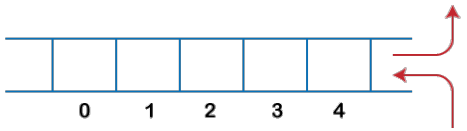
- The heap data structure is the most efficient way (will come back later).

DEQUE

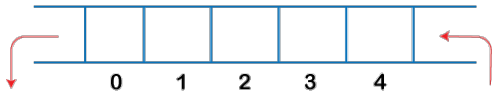
- **Double Ended Queue** is a linear data structure, a generalized queue.
  - Does not follow the FIFO principle.
  - Insertion and deletion can occur from both ends.



DEQUE EXAMPLES

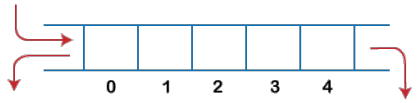


- Can be used both as stack and queue.
  - The insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end.

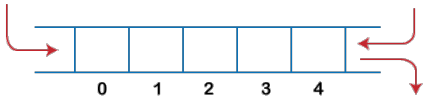


  - The insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end.

DEQUE EXAMPLES



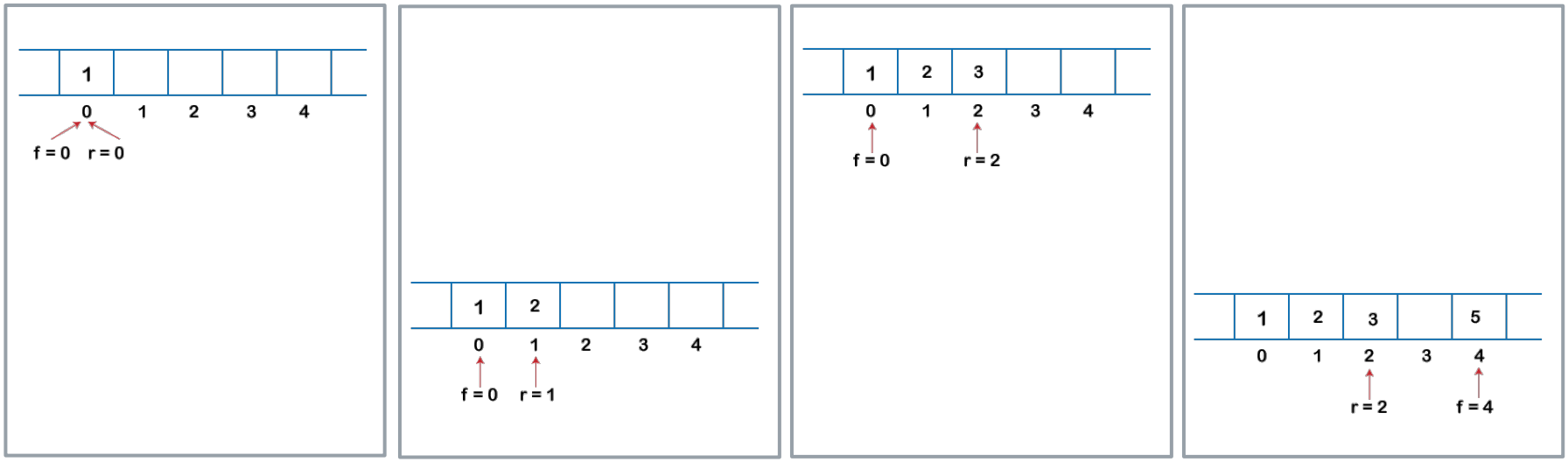
- **Input-restricted queue:** some restrictions are applied to the insertion. The insertion is applied to one end while the deletion is applied from both the ends.



- **Output-restricted queue:** some restrictions are applied to the deletion operation. The deletion can be applied only from one end, whereas the insertion is possible from both ends.

DEQUE OPERATIONS

- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.



ENQUEUE

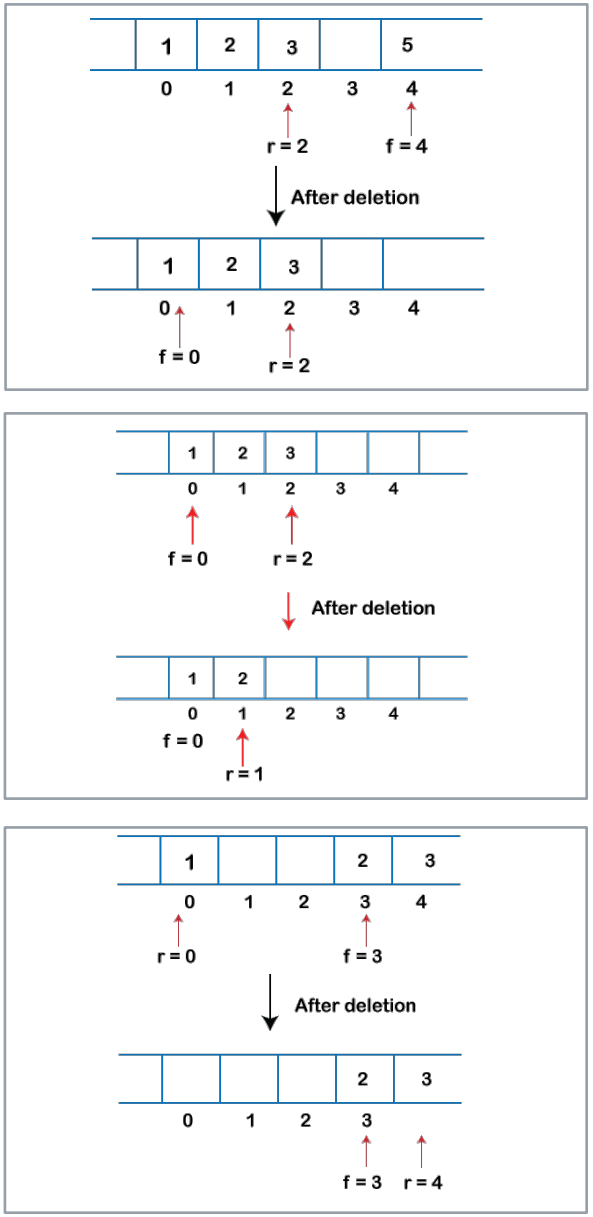
- Initially, the deque is empty. Both front and rear are -1, i.e., *f = -1* and *r = -1*.
- As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**
- To insert the element from the rear end, increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.
- To again insert the element from the rear, first increment the rear, and now rear points to the third element.
- To insert the element from the front end, and insert an element from the front, decrement the value of front by 1. Then the front points to -1 location, which is not any valid location in an array. So, set the front as **(n-1),** which is equal to 4 as n is 5.

DEQUEUE

- Suppose the front is pointing to the last element. To delete an element from the front, set **front=front+1**. Currently, the front is 4, and it becomes 5 which is not valid. Therefore, if front points to the last element, then front is set to 0 for delete operation.

- To delete the element from rear end, then decrement the rear value by 1, i.e., **rear=rear-1**.

- Suppose the rear is pointing to the first element. To delete the element from the rear end, set **rear=n-1** where **n** is the size of the array.



OUTLINE

- Stack and Implementation
- Queue and Implementation
- Examples

```java
// A Linked List Node
class Node {
    int data;       // integer data
    Node next;      // pointer to the next node
    public Node(int data) {
        // set data in the allocated node and return it
        this.data = data;
        this.next = null;
    }
}
class Queue {
    private static Node rear = null, front = null;
    // Utility function to dequeue the front element
    public static int dequeue() {    // delete at the beginning
        if (front == null) {
            System.out.print("\nQueue Underflow");
            System.exit(1);
        }
        Node temp = front;
        System.out.printf("Removing %d\n", temp.data);
        // advance front to the next node
        front = front.next;
        // if the list becomes empty
        if (front == null) {
            rear = null;
        }
        // deallocate the memory of the removed node and
        // optionally return the removed item
        int item = temp.data;
        return item;
    }
    // Utility function to add an item to the queue
    public static void enqueue(int item) {    // insertion at the end
        // allocate a new node in a heap
        Node node = new Node(item);
        System.out.printf("Inserting %d\n", item);
        // special case: queue was empty
        if (front == null) {
            // initialize both front and rear
            front = node;
            rear = node;
        } else {
            // update rear
            rear.next = node;
            rear = node;
        }
    }
    // Utility function to return the top element in a queue
    public static int peek() {
        // check for an empty queue
        if (front != null) {
            return front.data;
        } else {
            System.exit(1);
        }

        return -1;
    }
    // Utility function to check if the queue is empty or not
    public static boolean isEmpty() {
        return rear == null && front == null;
    }
}

class Main {
    public static void main(String[] args) {
        Queue q = new Queue();
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);
        q.enqueue(4);
        System.out.printf("The front element is %d\n", q.peek());
        q.dequeue();
        q.dequeue();
        q.dequeue();
        q.dequeue();
        if (q.isEmpty()) {
            System.out.print("The queue is empty");
        } else {
            System.out.print("The queue is not empty");
        }
    }
}
```

Queue Implementation using a Linked List

```java
import java.util.ArrayDeque;
import java.util.Queue;
// Implement stack using two queues
class Stack<T> {
    Queue<T> q1, q2;
    // Constructor
    public Stack() {
        q1 = new ArrayDeque<>();
        q2 = new ArrayDeque<>();
    }
    // Insert an item into the stack
    void add(T data) {
        // Move all elements from the first queue to the second queue
        while (!q1.isEmpty()) {
            q2.add(q1.peek());
            q1.poll();
        }
        // Push the given item into the first queue
        q1.add(data);
        // Move all elements back to the first queue from the second queue
        while (!q2.isEmpty()) {
            q1.add(q2.peek());
            q2.poll();
        }
    }
    // Remove the top item from the stack
    public T poll() {
        // if the first queue is empty
        if (q1.isEmpty()) {
            System.out.println("Underflow!!");
            System.exit(0);
        }
        // return the front item from the first queue
        T front = q1.peek();
        q1.poll();
        return front;
    }
}
class Main {
    public static void main(String[] args) {
        int[] keys = { 1, 2, 3, 4, 5 };
        // insert the above keys into the stack
        Stack<Integer> s = new Stack<Integer>();
        for (int key: keys) {
            s.add(key);
        }
        for (int i = 0; i <= keys.length; i++) {
            System.out.println(s.poll());
        }
    }
}
```

IMPLEMENTING STACK USING A QUEUE

```java
import java.util.Stack;
// Implement a queue using two stacks
class Queue<T> {
    private Stack<T> s1, s2;
    // Constructor
    Queue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }
    // Add an item to the queue
    public void enqueue(T data) {
        // Move all elements from the first stack to the second stack
        while (!s1.isEmpty()) {
            s2.push(s1.pop());
        }

        // push item into the first stack
        s1.push(data);

        // Move all elements back to the first stack from the second stack
        while (!s2.isEmpty()) {
            s1.push(s2.pop());
        }
    }
    // Remove an item from the queue
    public T dequeue() {
        // if the first stack is empty
        if (s1.isEmpty())
        {
            System.out.println("Underflow!!");
            System.exit(0);
        }

        // return the top item from the first stack
        return s1.pop();
    }
}
class Main {
    public static void main(String[] args) {
        int[] keys = { 1, 2, 3, 4, 5 };
        Queue<Integer> q = new Queue<Integer>();
         // insert above keys
        for (int key: keys) {
            q.enqueue(key);
        }
        System.out.println(q.dequeue());    // print 1
        System.out.println(q.dequeue());    // print 2
    }
}
```

IMPLEMENTING A QUEUE USING TWO STACKS

EXAMPLE

- **The Josephus Problem**

- There are n people standing in a circle waiting to be executed. After the first man is executed, k - 1 people are skipped and the k-th man is executed. Then again, k-1 people are skipped and the k-th man is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last man remains, who is given freedom.
- The task is to choose the place in the initial circle so that you survive, given n and k.

THANKS