```
class Node

{

    int data;

    Node next;

    Node(int key)

    {

        this.data = key;

        next = null;

    }

}

class MergeSort

{

    // Function to merge sort

    static Node mergeSort(Node head)

    {

        if (head.next == null)

            return head;



        Node mid = findMid(head);

        Node head2 = mid.next;

        mid.next = null;

        Node newHead1 = mergeSort(head);

        Node newHead2 = mergeSort(head2);
        Node finalHead = merge(newHead1, newHead2);


        return finalHead;

    }

    // Function to merge two linked lists

    static Node merge(Node head1, Node head2)

    {

        Node merged = new Node(-1);

        Node temp = merged;


        // While head1 is not null and head2 is not null

        while (head1 != null && head2 != null) {

            if (head1.data < head2.data) {

                temp.next = head1;

                head1 = head1.next;

            }

            else {

                temp.next = head2;

                head2 = head2.next;

            }

            temp = temp.next;

        }

        // While head1 is not null
        while (head1 != null) {
            temp.next = head1;
            head1 = head1.next;
            temp = temp.next;
        }
        // While head2 is not null
        while (head2 != null) {
            temp.next = head2;
            head2 = head2.next;
            temp = temp.next;
        }
        return merged.next;
    }
    // Find mid using The Tortoise and The Hare approach
    static Node findMid(Node head)
    {
        Node slow = head, fast = head.next;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
```

```
static void printList(Node head)

{

    while (head != null) {

        System.out.print(head.data + " ");

        head = head.next;

    }

}



// Driver Code

public static void main(String[] args)

{

    Node head = new Node(7);

    Node temp = head;

    temp.next = new Node(10);

    temp = temp.next;

    temp.next = new Node(5);

    temp = temp.next;

    temp.next = new Node(20);

    temp = temp.next;

    temp.next = new Node(3);

    temp = temp.next;

    temp.next = new Node(2);

    temp = temp.next;
```

```

        // Apply merge Sort

        head = mergeSort(head);

        System.out.print("\nSorted Linked List is: \n");

        printList(head);

    }

}
```

curr2 → data <= curr1 → data

3 <= 5

ei → next = curr2

ei = curr2

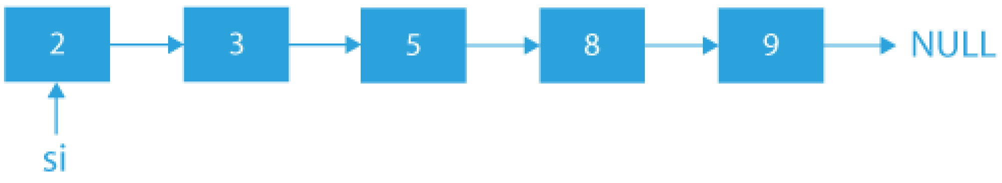
Move, curr2 forward

curr2 = curr2 → next

Now curr2, becomes NULL
Join ei with curr1

ei → next = curr1

After joining our linked list looks like



Return Si,

Our linked list is sorted now

```
class PrepBytes{



static class DQueNode

{

    int value;

    DQueNode next;

    DQueNode prev;
```

```
}



static class deque

{



    private DQueNode head;

    private DQueNode tail;



    public deque()

    {

        head = tail = null;

    }



    boolean isEmpty()

    {

        if (head == null)

            return true;



        return false;

    }



    int size()

    {
```

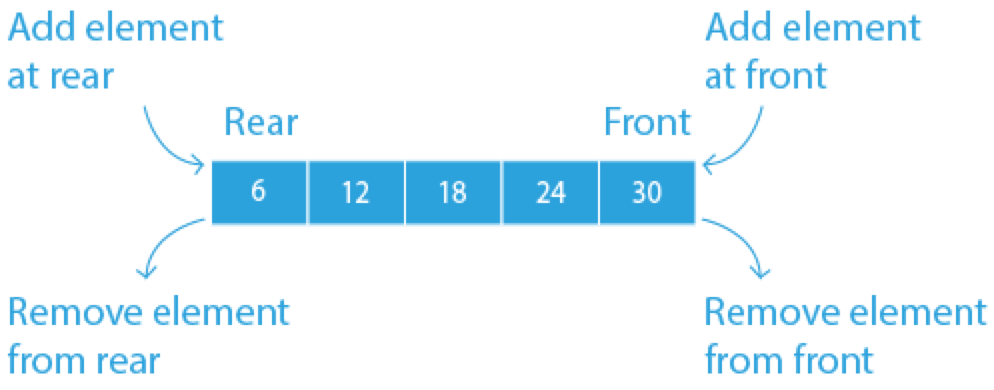
```
    if (!isEmpty())

    {

        DQueNode temp = head;

        int len = 0;



        while (temp != null)

        {

            len++;

            temp = temp.next;

        }

        return len;

    }

    return 0;

}



void insert_first(int element)

{



    DQueNode temp = new DQueNode();

    temp.value = element;



    if (head == null)
    {

        head = tail = temp;

        temp.next = temp.prev = null;

    }

    else

    {

        head.prev = temp;

        temp.next = head;

        temp.prev = null;

        head = temp;

    }

}



void insert_last(int element)

{



    DQueNode temp = new DQueNode();

    temp.value = element;



    if (head == null)

    {

        head = tail = temp;

        temp.next = temp.prev = null;
    }

    else

    {

        tail.next = temp;

        temp.next = null;

        temp.prev = tail;

        tail = temp;

    }

}



void remove_first()

{



    if (!isEmpty())

    {

        DQueNode temp = head;

        head = head.next;

        head.prev = null;



        return;

    }

    System.out.print("List is Empty");

}
```

```
void remove_last()

{



    if (!isEmpty())

    {

        DQueNode temp = tail;

        tail = tail.prev;

        tail.next = null;



        return;

    }

    System.out.print("List is Empty");

}



void display()

{



    if (!isEmpty())

    {

        DQueNode temp = head;



        while (temp != null)
```

```
            {

                System.out.print(temp.value + " ");

                temp = temp.next;

            }



            return;

        }

        System.out.print("List is Empty");

    }

}



static class Stack

{

    deque d = new deque();



    public void push(int element)

    {

        d.insert_last(element);

    }



    public int size()

    {

        return d.size();
    }


    public void pop()

    {

        d.remove_last();

    }


    public void display()

    {

        d.display();

    }

}


static class Queue

{

    deque d = new deque();


    public void enqueue(int element)

    {

        d.insert_last(element);

    }


    public void dequeue()
    {
        d.remove_first();
    }

    public void display()
    {
        d.display();
    }

    public int size()
    {
        return d.size();
    }
}

public static void main(String[] args)
{

    Stack stk = new Stack();

    stk.push(7);
    stk.push(14);
    System.out.print("Stack: ");
    stk.display();



    System.out.println();



    stk.pop();

    System.out.print("Stack: ");

    stk.display();



    System.out.println();



    Queue que = new Queue();



    que.enqueue(12);

    que.enqueue(24);

    System.out.print("Queue: ");

    que.display();



    System.out.println();



    que.dequeue();

    System.out.print("Queue: ");

    que.display();
    System.out.println();

    System.out.println("Size of stack is " +

                    stk.size());

    System.out.println("Size of queue is " +

                    que.size());

}

}
```

What is Deque?

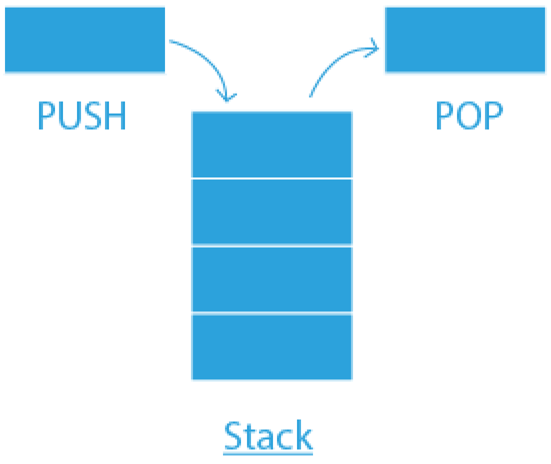
Deque is a double ended queue, i.e. a special kind of queue in which insertion and deletion can be done at the both rear as well as front end of the queue.You can implement both stack and queue by using deque.



Dequeue

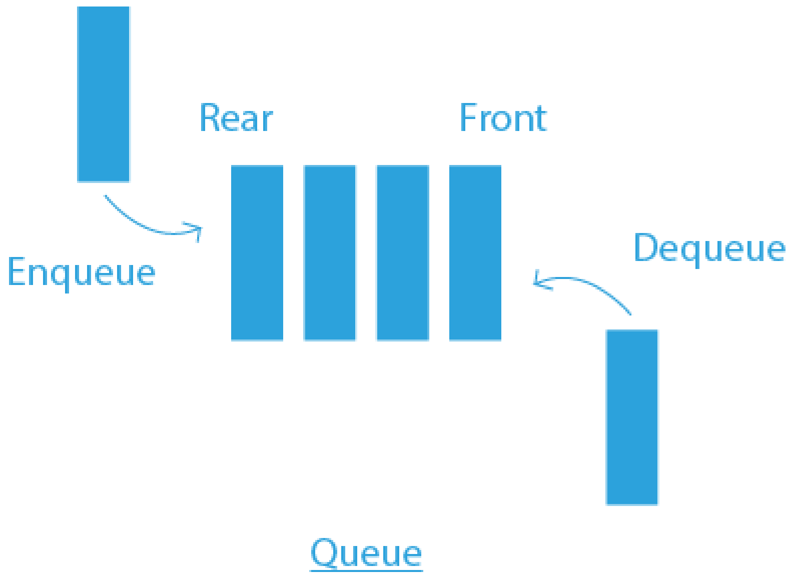
What is stack?

Stack follows the principle of LIFO (Last in First out) i.e. element which is inserted at last will be removed first. The operation for insertion of elements in stack is known as Push operation and the operation for deletion of element in stack is known as Pop operation.



What is Queue?

Queue follows the principle of FIFO (First in First out) i.e. element which is inserted first will be removed first. The operation for insertion of elements is known as enqueue operation and the operation for deletion of elements is known as dequeue operation.



Functions of deque that works same as the functions of stack and queue:

Deque	**Stack**	**Queue**
size()	size()	size()

Deque	**Stack**	**Queue**
isEmpty()	isEmpty()	isEmpty()
Insert_First()	–	–
Insert_Last()	Push()	Enqueue()
Remove_First()	–	Dequeue()
Remove_Last()	Pop()	–

Operations of Deque

- **size():** This function returns the size of the deque.
- **isEmpty():** This function returns true if the deque is empty else false.
- **Insert_First(element):** This function will insert an element in the deque at the front end.
- **Insert_Last(element):** This function will insert an element in the deque at the rear end.
- **Remove_First():** This function will remove the element from the deque which is present at the front end.
- **Remove_Last():** This function will remove the element from the deque which is present at the rear end.