Introduction

Introduction to Data structures

Data Structure is an arrangement of data in computer's memory (or sometimes on a disk) so that we can retrieve it & manipulate it correctly and efficiently.

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a particular data model depends on the two factors -

- Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
- Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Introduction to Data structures

Data Structure is an arrangement of data in computer's memory (or sometimes on a disk) so that we can retrieve it & manipulate it correctly and efficiently.

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a particular data model depends on the two factors -

- Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
- Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Categories of Data Structure:

The data structure can be sub divided into major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure:

A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory. First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as
arrays.

The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

The common examples of linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non linear Data Structure:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non Linear Data Structures are listed below:

- Graphs
- family of trees and
- table of contents

Data structures are essential in almost every aspect where data is involved. In general, algorithms that involve efficient data structure is applied in the following areas:

Numerical analysis, Operating system, Statistical analysis, Compiler Design, Operating System, Database Management System, Statistical analysis package, Numerical Analysis, Graphics, Artificial Intelligence, Simulation

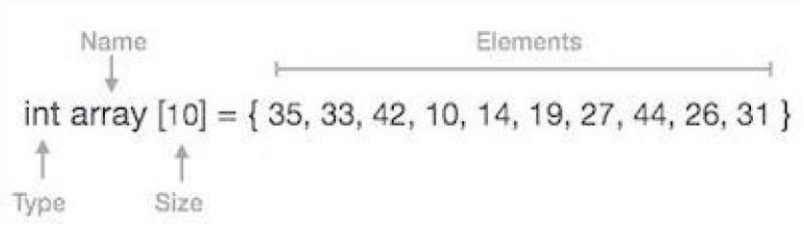
Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

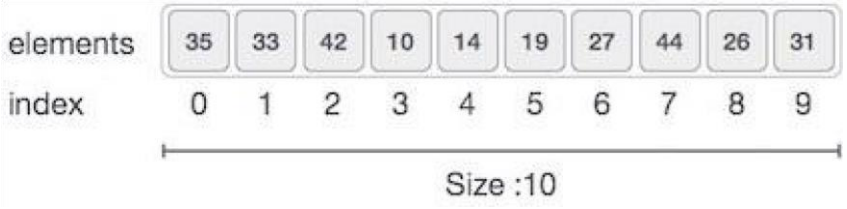
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation:(Storage structure)

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Singly Linked List

Limitations of array over linked list:

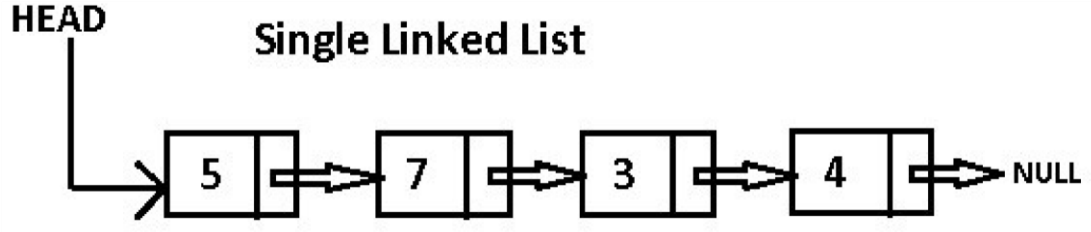
- Array does not grow dynamically (i.e length has to be known) Inefficient memory management.
- In ordered array insertion is slow.
- In both ordered and unordered array deletion is slow.
- Large number of data movements for insertion & deletion
- Which is very expensive in case of array with large number of elements.

Linked List:

We can overcome the drawbacks of the sequential storage.

If the items are explicitly ordered, that is each item contained within itself the address of the next Item.

In Linked Lists elements are logically adjacent, need not be physically adjacent. Node is divide into two part.



Operation can be performed on linked List:

1. insert()
2. insertAtFirst()

3. insertAtLast()
4. insertByPos()
5. deleteAtFirst()
6. deleteAtLast()
7. deleteByPos()
8. deleteByVal()
9. deleteAllNodes()
10. reverse()
11. findMidElement()
12. remodeDuplicateNodes() etc…..

```
//------------------------------Node Class----------------//
public class Node {
   private int data;
   private Node next;

   public Node(int data) {
      this.data = data;
      next = null;
   }

   public int getData() {
      return data;
   }

   public void setData(int data) {
      this.data = data;
   }

   public Node getNext() {
```

```
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    @Override
    public String toString() {
        return data + "";
    }
}

//-------------------------Linked List  Class---------------------------//
public class LinkedList {
    private Node head;

    public LinkedList() {
        head = null;
    }

    public boolean insert(int data) {
        Node newNode = new Node(data);
        if( newNode == null) {
            return false;
        }

        //check if the list is empty
        if(head == null) {
```

```
            head = newNode;
            return true;
        }

        //insert node at the end of the list
        Node last = head;
        while( last.getNext() != null ) {
            last = last.getNext();
        }

        //last is referring to last node in the list
        last.setNext(newNode);

        return true;
    }

    public void display() {
        Node temp = head;
        while(temp != null) {
            System.out.print( temp.getData()  + " ");
            temp = temp.getNext();
        }
        System.out.println();

    }

    public boolean insert(int data, int position) {

        if(position <= 0) {
```

```
    return false;
}

Node newNode = new Node(data);
if(newNode == null) {
    return false;
}

if(position == 1) {
    newNode.setNext(head);
    head = newNode;
    return true;
}

//position is other than 1

//1. Locate node at pos - 1 i.e. prev
Node prev = head;
for(int i = 1; i < position - 1; i++) {
    prev = prev.getNext();
    if(prev == null) {
        return false;
    }
}

//2
newNode.setNext(prev.getNext());
//3
prev.setNext(newNode);
```

```
    return true;
}

boolean deleteByPosition(int position) {

    if(head == null || position <= 0) {
        return false;
    }

    if(position == 1) {
        head = head.getNext();
        return true;
    }

    //if pos > 1
    Node prev = head;
    for(int i = 1; i < position - 1; i++) {
        prev = prev.getNext();
        if(prev == null) {
            return false;
        }
    }
    Node del = prev.getNext();
    if(del == null) {
        return false;
    }

    prev.setNext(del.getNext());
```

```
    return true;
  }

  boolean deleteByVal(int data) {

    if(head == null) {
      return false;
    }

    if(head.getData() == data) {
      head = head.getNext();
      return true;
    }

    Node prev = head, del = head;
    while(del.getData() != data) {
      prev = del;
      del = del.getNext();
      if(del == null) {
        return false;
      }
    }

    prev.setNext(del.getNext());
    return true;
  }

  void displayRev() {
```

```
    Node temp = head;
    Node [] stack = new Node[100];
    int top = -1;

    while(temp != null) {
       stack[++top] = temp;
       temp = temp.getNext();
    }

    while(top != -1) {
       System.out.print(stack[top--] + " ");
    }
    System.out.println();
 }

 void displayRev(Node node) {
    if(node == null) {
       System.out.println();
       return;
    }

    displayRev( node.getNext() );
    System.out.print(node.getData() + " ");
 }

 void reverse() {

    if(head == null || head.getNext() == null) {
       return;
```

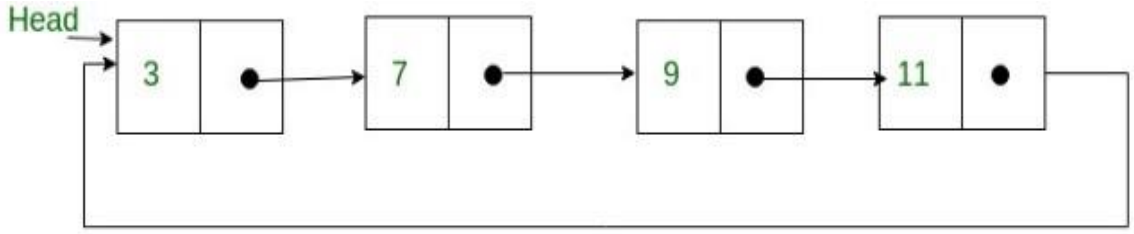
```
      }

      Node n1 = head, n2 = head.getNext();
      Node n3;

      while(n2 != null) {
         n3 = n2.getNext();
         n2.setNext(n1);
         n1 = n2;
         n2 = n3;
      }

      head.setNext(null);
      head = n1;
   }

   Node getHead() {
      return head;
   }
}

//--------------------------------TesterSLL Class-------------------------------/
public class Main {
   public static void main(String [] args) {

      LinkedList l1 = new LinkedList();

      l1.reverse();
```

```
l1.display();
l1.insert(10);
l1.insert(20);
l1.insert(30);
l1.insert(40);
l1.display();
l1.insert(50, 1);
l1.display();
l1.insert(60, 4);
l1.display();
l1.insert(70, 7);
l1.display();
l1.insert(80, -1);
l1.display();
l1.insert(90, 9);
l1.display();
l1.insert(20);
l1.insert(20, 5);
l1.display();
l1.displayRev();
l1.displayRev( l1.getHead() );
//l1.reverse();
//l1.display();
//l1.displayRev();
/*
while(l1.deleteByVal(20))
   ;
l1.display();
*/
```

```
        /*l1.deleteByVal(50);
        l1.display();
        l1.deleteByVal(60);
        l1.display();
        l1.deleteByVal(70);
        l1.display();
        l1.deleteByVal(70);
        l1.display();*/

        /*
        l1.deleteByPosition(1);
        l1.display();
        l1.deleteByPosition(3);
        l1.display();
        l1.deleteByPosition(5);
        l1.display();
        l1.deleteByPosition(5);
        l1.display();
        l1.deleteByPosition(10);
        l1.display();*/

    }
}
```

Singly Circular Linked List

A circular linked list is a linked list in which the last node points to the head or front node making the data structure to look like a circle. A circularly linked list node can be implemented using singly linked.




```
//---------------------------------Node Class----------------------//
public class Node {
   private int data;
   private Node next;

   public Node(int data) {
      this.data = data;
      next = null;
   }
   public int getData() {
      return data;
   }
   public void setData(int data) {
      this.data = data;
   }
   public Node getNext() {
      return next;
   }
   public void setNext(Node next) {
      this.next = next;
   }
```

```
}
//--------------------------CircularLinkedList Class-----------------//


public class SCLL {
	private Node head; //ref to first node of list

	public SCLL()
	{
		this.head=null;//empty list creation
	}
	public Node getHead()
	{
		return this.head;
	}
	public void setHead(Node head)
	{
		this.head=head;
	}

	boolean insertByVal(int data){

	Node newNode=new Node(data);

	if(newNode==null)//mem alloc fails...issue in node creatioin
	{
		//System.out.println("error in  node creation");
		return false;
	}
```

```
//chk if list is empty
if(this.head==null)//list is empty ...so new node becomes head
{
        this.head=newNode;
        newNode.setNext(head);
        return true;//exit
}

//traverse list till last node
Node temp=head;
while(temp.getNext()!=head)
{
        temp=temp.getNext();
}
//connect last node with new  node n newnode with head
temp.setNext(newNode);
newNode.setNext(head);
return true;

}

 void display() {
 Node temp=head;
 System.out.println();
do
 {
          System.out.print(temp.getData()+ "   ");
          temp=temp.getNext();
```

```
} while(temp!=head);

}



boolean insertByPos(int data,int pos){
        Node newNode=new Node(data);
        if(newNode==null)// issue in node creation
                {
                //System.out.println("error in  node creation");
                return false;
                }

        if(pos==1) {
        //chk if list is empty
        if(this.head==null)//list is empty ...so new node becomes head
                {
                        head=newNode;
                        head.setNext(head);
                        return true;
                }
                else //non empty list
                {
                        newNode.setNext(head);
                        Node temp=head;
                        while(temp.getNext()!=head)
                        {
                                temp=temp.getNext();
```

```
                }
                temp.setNext(newNode);
                head=newNode;
                return true;
            }
            }
            else    //pos other than first
            {
                Node prev=head;
                for(int i=1;i<pos-1;i++)
                {
                if(prev.getNext()==head)//out of bound pos
                        return false;
                    prev=prev.getNext();
                }
                newNode.setNext(prev.getNext());
                prev.setNext(newNode);
                return true;
            }
    }

    boolean  deleteByVal(int data)
    {
        Node del,prev,temp;
        del=prev=temp=head;
        if(del.getData()==data) //chk match with head or first
        {
          //traverse list till last node
           while(temp.getNext()!=head)
```

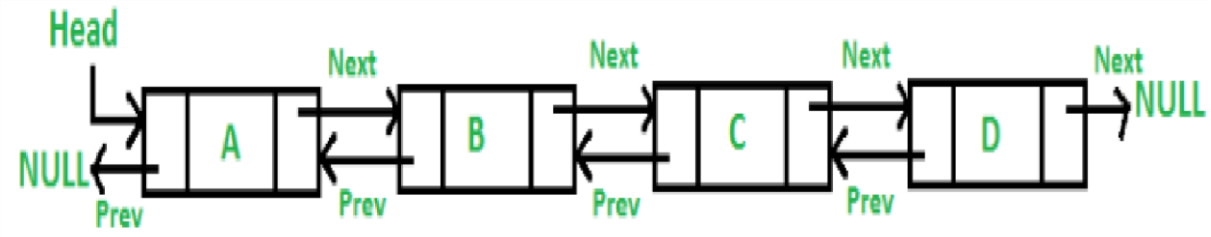
```
                {
                        temp=temp.getNext();//forward by one node
                }
               //connect last node with second
               temp.setNext(head.getNext());
                head=head.getNext();
                return true;
            }
        else
        {
        //locate deletable node by matching data
             while(del.getData()!=data)
                {
                        if(del.getNext()==head) //unmatched not found
                        {
                                return false;
                        }
                        prev=del;
                        del=del.getNext();
                }
                prev.setNext(del.getNext());
                return true;
        }
    }
    boolean deleteByPos(int pos)
    {
        Node del,prev,temp;
        del=temp=prev=head;
        if(head==null)//empty list
```

```
            return false;

        if(pos==1)
        {
            if(head.getNext()==head)//only one node in list
            {
                head=null;
                return true;
            }
            //non empty list
            //locate last node
            while(temp.getNext()!=head)
            {
                temp=temp.getNext();
            }
              // connect last n second node
            temp.setNext(head.getNext());
               head=head.getNext();//update head
            return true;
        }
        else //other than first pos
        {
             //set prev to pos-1
            for(int i=1;i<pos-1;i++)
            {
                if(prev.getNext()==head)//out of bound
                {
                    return false;
                }
```

```
                    prev=prev.getNext();
                }
                del=prev.getNext();
                prev.setNext(del.getNext());
                return true;
            }
            }
        }
}
```

Doubly Linked List

A doubly linked list is a linked list data structure that includes a link back to the previous node in each node in the structure. This is contrasted with a singly linked list where each node only has a link to the next node in the list. Doubly linked lists also include a field and a link to the next node in the list.




```
//--------------------Doubly Node Class-----------------------------//
public class Node {
   private int data;
   private Node prev, next;

   public Node(int data) {
      this.data = data;
```

```
        prev = next = null;
    }

    public int getData() {
        return data;
    }

    public void setData(int data) {
        this.data = data;
    }

    public Node getPrev() {
        return prev;
    }

    public void setPrev(Node prev) {
        this.prev = prev;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }
}
//---------------------------- DoublyLinkedList Class-----------------//
public class DoublyLinkedList {
```

```
private Node head;

public DoublyLinkedList() {
   head = null;
}

public boolean insert(int data) {

   Node newNode = new Node(data);
   if(newNode == null) {
      return false;
   }

   //list is empty
   if(head == null) {
      head = newNode;
      return true;
   }

   //list is not empty
   //locate last node
   Node last = head;
   while(last.getNext() != null) {
      last = last.getNext();
   }

   last.setNext(newNode);
   newNode.setPrev(last);
```

```
    return true;
  }
  public boolean insert(int data, int position) {
    if(position <= 0) {
      return false;
    }

    Node newNode = new Node(data);
    if(newNode == null) {
      return false;
    }

    if(position == 1) {
      if(head != null ) {
        newNode.setNext(head);
        head.setPrev(newNode);
      }
      head = newNode;
      return true;
    }

    //position is other than 1

    Node prevNode = head, nextNode = null;
    for(int i = 1; i < position - 1; i++) {
      prevNode = prevNode.getNext();
      if(prevNode == null) {
        return false;
      }
```

```
      }
      nextNode = prevNode.getNext();

      //link the newnode
      newNode.setPrev(prevNode);
      prevNode.setNext(newNode);
      if(nextNode != null) {
         newNode.setNext(nextNode);
         nextNode.setPrev(newNode);
      }

      return true;
   }

   public void display() {
      Node temp = head;

      while(temp != null) {
         System.out.print(temp.getData() + " ");
         temp = temp.getNext();
      }
      System.out.println();
   }

   public boolean deleteByPosition(int position) {

      if( (head == null || position <= 0 ) || ( head == null && position > 1))
{
         return false;
      }
```

```
    if(position == 1) {
       head = head.getNext();
       if(head != null) {
          head.setPrev(null);
       }
       return true;
    }

    Node del = head;
    for(int i = 1; i < position; i++) {
       del = del.getNext();
       if(del == null) {
          return false;
       }
    }

    del.getPrev().setNext(del.getNext());
    if(del.getNext() != null) {
       del.getNext().setPrev(del.getPrev());
    }

    return true;
 }

 public boolean deleteByVal(int data) {
    if(head == null) {
       return false;
    }
```

```
      if(head.getData() == data) {
         head = head.getNext();
         if(head != null) {
            head.setPrev(null);
         }
         return true;
      }

      //data is not present in first node
      //Locate the del node
      Node del = head;
      while(del.getData() != data) {
         del = del.getNext();
         if(del == null) {
            return false;
         }
      }

      //del is now referring to the node which is to be deleted
      del.getPrev().setNext(del.getNext());
      if(del.getNext() != null) {
         del.getNext().setPrev(del.getPrev());
      }

      return true;
   }
}
//------------------------Tester DLL Class------------------//
```

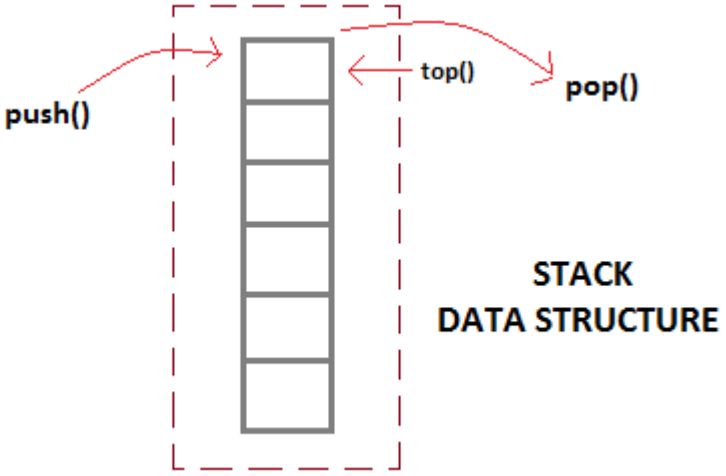
```
public class TesterDLL {

   public static void main(String [] args) {

      DoublyLinkedList dll = new DoublyLinkedList();

      dll.insert(10);
      dll.insert(20);
      dll.insert(30);
      dll.insert(40);
      dll.display();

      dll.insert(50, 1);
      dll.display();
      dll.insert(60, 4);
      dll.display();
      dll.insert(70, 7);
      dll.display();
      dll.insert(50, 9);
      dll.display();

      dll.deleteByPosition(1);
      dll.display();
      dll.deleteByPosition(3);
      dll.display();
      dll.deleteByPosition(5);
      dll.display();
      dll.deleteByPosition(5);
      dll.display();
```

```
    }
}
```

Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. It is an ordered list in which insertion and deletion are done at one end, called a top. The last element inserted is the first one to

be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.



Applications of Stack:

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML implemented

  Indirect applications
- Auxiliary data structure for other algorithms (Example: Tree traversal s)
- Component of other data structures

  (Example: Simulating queues, Queues)

Following are operations are performed with stack.

Stack Operations:

- When an element is inserted in a stack, the concept is called a push.
- When an element is removed from the stack, the concept is called pop.
- Trying to pop out an empty stack is called underflow (treat as Exception).
- Trying to push an element in a full stack is called overflow (treat as Exception).

Push Operation

The process of putting a new data element onto the stack is known as a Push Operation.

Push operation involves a series of steps −

Step 1 − Checks if the stack is full.

Step 2 − If the stack is full, produces an error and exit.

Step 3 − If the stack is not full, increments top to a point next empty space.

Step 4 − Adds data element to the stack location, where the top is pointing.

Step 5 − Returns success.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

Step 3 − If the stack is not empty, accesses the data element at which top is pointing.

Step 4 − Decreases the value of top by 1.

Step 5 − Returns success.

Stack Implementation using Array:

Push Operation

- In a push operation, we add an element into the top of the stack.
- Increment the variable Top so that it can now refer to the next memory location.
- Add an element at the position of the incremented top. This is referred to as adding a new element at the top of the stack.
- Throw an exception if Stack is full.

Pop Operation

- Remove the top element from the stack and decrease the size of a top by 1.
- Throw an exception if Stack is empty.

In the array, we add elements from left to right and use a variable to keep track of the index of the top element. The array storing the stack elements may become full. A push operation will then throw a full stack exception. Similarly, if we try deleting an element from an empty stack it will throw a stack empty exception.

```
//*----------Stack Class----------*/
public class Stack {
   private int [] arr;
   private int size;
   private int top;

   public Stack() {
      size = 5;
```

```
    arr = new int[5];
    top = -1;
}

public Stack(int size) {
    this.size = size;
    arr = new int[size];
    top = -1;
}

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

public boolean isFull() {
    return top == (size - 1);
}

public boolean push(int data) {
    if( isFull() ) {
        return false;
    }

    arr[++top] = data;
    return true;
}

public int pop() {
    if(isEmpty()) {
```

```
         return -999;
      }

      return arr[top--];
   }

   public int peek() {
      if(isEmpty()) {
         return -999;
      }

      /*int data = pop();
      push(data);
      return data;*/

      return arr[top];
   }

   /*public void display() {
      for(int i = 0; i <= top; i++ ) {
         System.out.print(arr[i] + " ");
      }
   }*/
}

//*----------------------TesterMain Class------------------*/
public class Main {
   public static void main(String [] args) {
```

```
        Stack s = new Stack(4);
        System.out.println("Pop: " + s.pop());
        System.out.println("Push: " + s.push(0) );
        System.out.println("Push: " + s.push(20) );
        System.out.println("Push: " + s.push(30) );
        System.out.println("Push: " + s.push(40) );
        System.out.println("Push: " + s.push(50) );

        System.out.println("peek: " + s.peek());
        System.out.println("Pop: " + s.pop());
        /*System.out.println("Pop: " + s.pop());
        System.out.println("Pop: " + s.pop());
        System.out.println("Pop: " + s.pop());
        System.out.println("Pop: " + s.pop());*/


    }
}
```

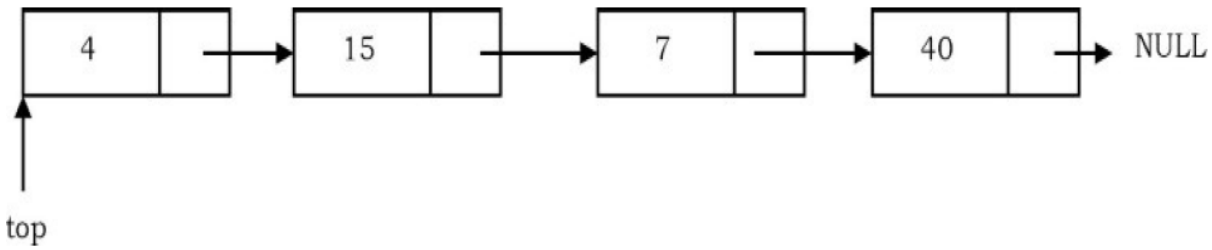
Stack Implementation using Linked List :

Stack implementation using Linked List in Java. Instead of using an array, we can also use a linked list to implement a Stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is the same for all the operations i.e. push, pop and peek.

In the linked list implementation of a Stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the Stack. A Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

- A push operation is implemented by inserting an element at the beginning of the list.

- A pop operation is implemented by deleting the node from the beginning (the header/top node).




```
//* ---------------------Stack Node Class-----------------*/
public class Node {
//data members
        private int data;
        private Node next;

        public Node()//def
        {
                this.data=0;
                this.next=null;
        }
        public Node(int data)//param.
        {
                this.data=data;
                this.next=null;
        }

        int getData()
        {
                return this.data;
        }

        Node getNext()
        {
```

```
		return this.next;
	}

	void setData(int data )//local var
	{
	  this.data=data;
	}

	void setNext(Node next) //localvar
	{
		this.next=next;
	}

}

//*---------------Stack Class with Operations--------------------*/

public class StackLiLi {
private Node top;

public StackLiLi()
{
	this.top=null;
}

boolean push(int data)
{
	Node newNode=new Node(data);
	if(newNode==null)
	{
		return false;
	}
	//chk if stk is empty
	if(top==null)
	{
		top=newNode;//set newnode as first top
```

```
                return true;
        }
        //if non empty stack then make newnode as top
        newNode.setNext(top);
        top=newNode;
        return true;
}

int pop()
{
        if(top==null) //empty stack
                return -999;
        //non empty stack
        int val=top.getData();
        top=top.getNext();
        return val;
}


int peek()
{
        if(top==null) //empty stack
                return -999;
        //non empty stack
        int val=top.getData();

        return val;
}

void display()
{
  Node temp=top;
  System.out.println();
  while(temp!=null)
  {
         System.out.print(temp.getData()+ "   " );
```

```
            temp=temp.getNext();
        }
}

}
//*------------------------Tester Main ----------------------*/

package com.stklili;

public class TesterStack {

        public static void main(String[] args) {
                StackLiLi s1=new StackLiLi();//default is 5 but we can pass size
also

                s1.push(10);
                s1.push(20);
                s1.push(30);
                System.out.println("peek : "+s1.peek());
                s1.push(40);
                s1.push(50);
                s1.display();
                s1.push(60);
                System.out.println("popped: "+s1.pop());
                System.out.println("popped: "+s1.pop());
                System.out.println("popped: "+s1.pop());
                System.out.println("popped: "+s1.pop());
                System.out.println("popped: "+s1.pop());
                System.out.println("popped: "+s1.pop());
                s1.display();
        }

}
```

Polish Notation:

These are notations to represent math equations.

- Infix Notation: A+B
- Prefix Notation: +AB
- Postfix Notation: AB+

To convert infix to prefix /postfix considers the priorities:

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Infix to postfix Evaluation rules:

1. Scan the infix exp. From left to right
2. If scan character is left parenthesis then push it into stack
3. If scan character is operand than that will display into postfix expression.
4. If scan character is operator than that will push into operator stack.

5. If scan character has higher precedence than stack operator then scanned operator will push into operator stack.
6. If scan character has less or equal precedence than stack operator then stack operator will pop out from stack to postfix expression and scanned operator will push into operator stack.
7. If scan character is right parenthesis then till left parenthesis of stack all operators will pop to postfix expression and left will remove from stack.
8. Repeat step 1 to step 6 until end of expression

Ex: A+ (B*C-(D/E^F)*G)*H

Check chart below as an example.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Infix to prefix Evaluation rules:

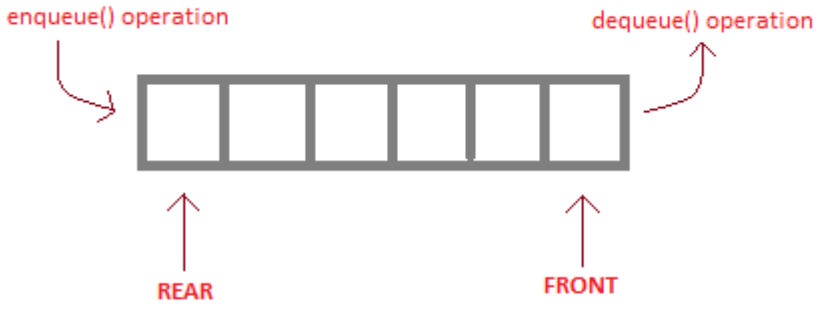
1. Reserve the input string or start from right of the infix expression
2. Examine the next element in the input.
3. If it is operand , add it to output string .
4. If it is closing parenthesis ,push it on to stack.
5. If it is an operator ,then
6. If stack is empty , push operator on stack.
7. If the top of stack is closing parenthesis , push operator on stack If it has same or higher priority then the top of stack push operator on stack
8. If it has lower priority the top, the pop stack and add it to post fix expression and push operator into stack
9. Else pop the operator from the stack and add it to output string If it is an opening parenthesis, pop operators from stack and ass them to output string until a closing parenthesis is encountered. pop and discard the closing parenthesis.
10. If there is more input go to step 2.
11. If there is no more input , unstack the remaining operators and them to output string.
12. Reserve the output string.

Queue

A Queue is an ordered collection of items into which new items may be inserted at rear end and items are deleted at one end called front.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Types of Queue:

1. Linear Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

Operations performed on Queue

ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the rear and add the element.

DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the front and increment the front.

Applications of Queue

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3.

Linear Queue implementation using Array:

```
//*---------------------------------Queue Class--------------------*/
public class Queue {
   private int [] arr;
   private int front, rear;
   private int size;

   public Queue(int size) {
      this.size = size;
```

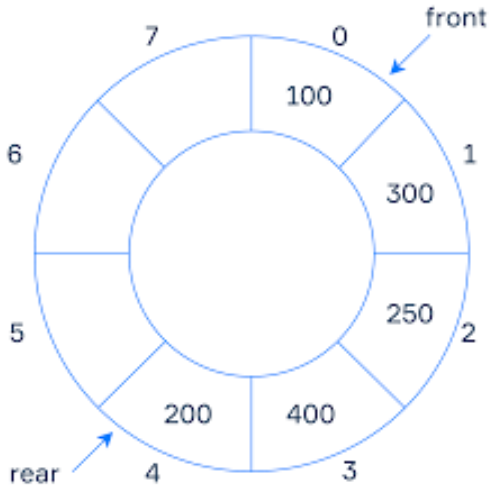
```
    this.arr = new int[size];
    front = rear = -1;
  }

  public boolean isEmpty() {
    return (front == -1 && rear == -1) || (front > rear);
  }

  public boolean isFull() {
    return rear == (size -1);
  }

  public boolean insert(int data) {
    if(isFull()) {
      return false;
    }

    arr[++rear] = data;
    if(front == -1) {
      front = 0;
    }

    //bulk move operation
    /*
    if(rear == size - 1 && front > 0 ) {
      //code to shift the data to the front of the queue
    }
     */
    return true;
  }
```

```
   public int delete() {
      if(isEmpty()) {
         return -999;
      }

      return arr[front++];
   }

   public void display() {
      for(int i = front; i <= rear; i++) {
         System.out.print( arr[i] + " ");
      }
   }
}
//*-----------------------Tester Main Class------------*/
public class Main {
   public static void main(String [] args) {

      CircularQueueNew cq = new CircularQueueNew(5);

      System.out.println("ins : " + cq.insert(10));
      System.out.println("ins : " + cq.insert(20));
      System.out.println("ins : " + cq.insert(30));
      System.out.println("ins : " + cq.insert(40));
      System.out.println("ins : " + cq.insert(50));
      System.out.println("ins : " + cq.insert(60));

      System.out.println("del : " + cq.delete());
      System.out.println("ins : " + cq.insert(60));
```

```
        System.out.println("del : " + cq.delete());
        System.out.println("ins : " + cq.insert(70));




        /*Queue q = new Queue(5);

        System.out.println("Ins: " + q.insert(10) );
        System.out.println("Ins: " + q.insert(20) );
        System.out.println("Ins: " + q.insert(30) );

        System.out.println("Del : " + q.delete());
        System.out.println("Del : " + q.delete());
        System.out.println("Del : " + q.delete());
        System.out.println("Del : " + q.delete());

        System.out.println("is Empty : " + q.isEmpty()  +" is Full : " + q.isFull());

        System.out.println("Ins: " + q.insert(40) );
        System.out.println("Ins: " + q.insert(50) );
        System.out.println("Ins: " + q.insert(60) );

        System.out.println("Del : " + q.delete());
        System.out.println("Del : " + q.delete());
        System.out.println("Del : " + q.delete());

        System.out.println("is Empty : " + q.isEmpty()  +" is Full : " + q.isFull());
*/      }
}
```

Circular Queue

The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer". One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.



Circular Queue implementation using Array:

```
//*-------------------------------- Circular Queue Class-----------------------*/

public class CQueue {
private int []arr;
private int front,rear;
private int size;
```

```
public CQueue()
{
        this.size=5;
        arr=new int [this.size];
        front=-1;
        rear=-1;
}
public CQueue(int size)
{
        this.size=size;
        arr=new int [this.size];
        front=-1;
        rear=-1;
}
public boolean isEmpty()
{
        return (front==rear);
}
public boolean isFull()
{
        return  ((front==-1&& rear==size-1)||((rear+1%size)==front));
}

boolean insert(int data)
{
        if(isFull())
          return false;
  else
  {       rear=(rear+1)%size;//cyclic increment
          arr[rear]=data;
```

```
            return true;
    }
}

public int delete()
{
        int val=-9999;
        if(isEmpty())
                return val;    //unpredicted data
        front=(front+1)%size;
        val = arr[front];
        return val;
}

public void display()
{
        int  i=front+1;
        while(i!=rear)
        {
                System.out.print(arr[i]+ "  ");
                i=(i+1)%size;
         }
        System.out.println(arr[i]);
}
public int peek()
{
        if(isEmpty())
                return -9999;         //unpredicted data
                /*
                 * int val=arr[front++]; return val;
```

```
                */
        return(arr[front]);
}
}
//*------------------------------ Tester Main Class-----------------------*/
public class TesterLinQArr {
        public static void main(String[] args) {
                CQueue   Q1=new CQueue(); //default size 5

                Q1.insert(10);
                Q1.insert(20);
                Q1.insert(30);
                //System.out.println(Q1.peek());
                Q1.insert(40);
                Q1.insert(50);
                Q1.display();
                System.out.println();
                System.out.println("deleted :"+Q1.delete());
                System.out.println("deleted :"+Q1.delete());

                Q1.insert(60);
                Q1.insert(70);//bubble prob
                Q1.display();
                System.out.println("deleted :"+Q1.delete());
                Q1.insert(70);
                Q1.display();
        }
}
```

Difference between Stack and Queue:

Sr.No	Stack	Queue
1	A Stack Data Structure works on Last In First Out (LIFO) principle.	A Queue Data Structure works on First In First Out (FIFO) principle.
2.	Push and pop operations are done from same end i.e "top"	Push and pop operations are done from same end i.e "rear" and "front" respectively
3.	You can implement multi-stack approach	Threre are four types of Queue i.e linear, circular, priority and dequeue.
4.	Application: • Used in infix to postfix conversion, • scheduling algorithms • depth first search and evaluation of an expression	Application: • Printer mainatains queue of documents to be printed • OS uses queues for many functionalities : Ready Queue, Waiting Queue, Message Queue • To implements algo like Breadth first search.

Trees

A tree consists of nodes connected by edges, which do not form cycle. For collection of nodes & edges to define as tree, there must be one & only one path from the root to any other node.

A tree is a connected graph of N vertices with N-1 Edges.

Tree Terminologies:

1. **Node:** A node stands for the item of information plus the branches of other items.

2. **Siblings:** Children of the same parent are siblings.

3. **Degree:** The number of sub trees of a node is called degree. The degree of a tree is the maximum degree of the nodes in the tree.

4. **Leaf Nodes:** Nodes that have the degree as zero are called leaf nodes or terminal nodes. Other nodes are called non terminal nodes.

5. **Ancestor:** The ancestor of a node are all the nodes along the path from the root to that node.

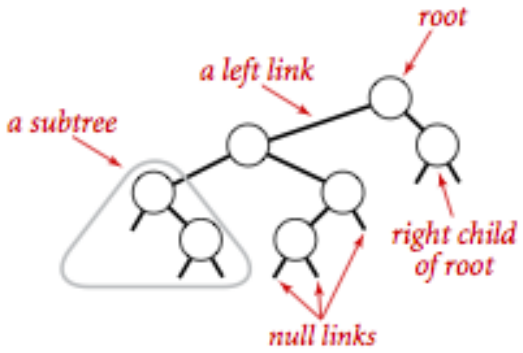
6. **Level:** The level of a node is defined by first letting the root of the tree to be level = 1or level=0.

7. **Height/Depth:** The height or depth of the tree is defined as the maximum level of any node in the tree.

Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.



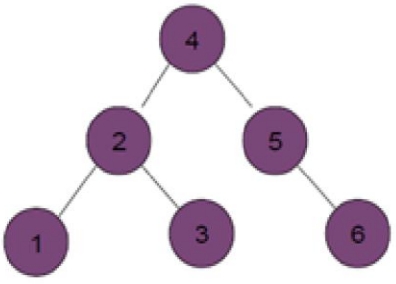
Anatomy of a binary tree

Tree traversals:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree ,

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

e.g.



- Preorder(Root-Left-Right)
  421356
- Inorder(Left-Root-Right)
  123456
- Postorder(Left-Right-Root)
  132654

Preorder:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Algorithm

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Inorder:

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

Algorithm

1. Traverse the left subtree, i.e., call Inorder (left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder (right-subtree)

Postorder:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Algorithm

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

```
//*---------------------------BST Node Class---------------------------*/

public class Node {
   private int data;
   private Node left, right;

   public Node(int data) {
      this.data = data;
      left = right = null;
   }

   public int getData() {
      return data;
   }

   public void setData(int data) {
      this.data = data;
   }

   public Node getLeft() {
      return left;
   }

   public void setLeft(Node left) {
      this.left = left;
   }

   public Node getRight() {
      return right;
   }
   public void setRight(Node right) {
      this.right = right;
   }
```

```
}//*--------------------------Tester main Class--------------------------*/

public class BinarySearchTree {
   private Node root;

   public Node getRoot() {
      return root;
   }

   public BinarySearchTree() {
      root = null;
   }

   public boolean insert(int data) {

      Node newNode = new Node(data);
      if (newNode == null) {
         return false;
      }

      if (root == null) {
         root = newNode;
         return true;
      }

      Node temp = root;

      while (true) {
         if (data == temp.getData()) {
            return false;
         }
```

```
        if (data < temp.getData()) {
```

```
            //insert to left
            if (temp.getLeft() == null) {
               //newnode will become left child of temp
               temp.setLeft(newNode);
               return true;
            }
            temp = temp.getLeft();
         } else {
            //insert to right
            if (temp.getRight() == null) {
               //newnode will become right child of temp
               temp.setRight(newNode);
               return true;
            }
            temp = temp.getRight();
         }
      }


   }

   public void preOrder() {
      Node temp = root;
      Node[] stack = new Node[100];
      int top = -1;

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

      while (temp != null || top != -1) {
         while(temp != null) {
            System.out.print(temp.getData() + " ");
```

```
            stack[++top] = temp;
            temp = temp.getLeft();
        }

        temp = stack[top--];
        temp = temp.getRight();
    }
    System.out.println();
}

public void inOrder() {
    Node temp = root;
    Node [] stack = new Node[100];
    int top = -1;

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

    while(temp != null || top != -1) {
        while(temp != null) {
            stack[++top] = temp;
            temp = temp.getLeft();
        }

        temp = stack[top--];
        System.out.print(temp.getData() + " ");
        temp = temp.getRight();
    }
    System.out.println();
}

public void postOrder() {

    System.out.print("PostOrder: ");
```

```
    class Pair {
       public Node node;
       public char flag;
    }

    Node temp = root;
    Pair [] stack = new Pair[100];
    int top = -1;

    while(temp != null || top != -1) {

       while(temp != null) {
          Pair pair = new Pair();
          pair.node = temp;
          pair.flag = 'L';
          stack[++top] = pair;
          temp = temp.getLeft();
       }

       Pair pair = stack[top--];
       if(pair.flag == 'L') {
          temp = pair.node.getRight();
          pair.flag = 'R';
          stack[++top] = pair;
       }
       else {
          System.out.print(pair.node.getData() + " ");
       }

    }
    System.out.println();
 }

 public void inOrder(Node root) {
```

```
    if(root == null) {
       return;
    }

    inOrder(root.getLeft());
    System.out.print(root.getData() + " ");
    inOrder(root.getRight());
  }

  public void preOrder(Node root) {
    if(root == null) {
       return;
    }

    System.out.print(root.getData() + " ");
    preOrder(root.getLeft());
    preOrder(root.getRight());
  }

  public void postOrder(Node root) {
    if(root == null) {
       return;
    }

    postOrder(root.getLeft());
    postOrder(root.getRight());
    System.out.print(root.getData() + " ");
  }


  public boolean delete(int data) {
    if (root == null) {
       return false;
    }
```

```
//1 Locate the del node along with parent node
Node parent = root, del = root;
while (data != del.getData()) {
   parent = del;
   if (data < del.getData()) {
      del = del.getLeft();
   } else {
      del = del.getRight();
   }
   if (del == null) {
      return false;
   }
}

while (true) {

   //2. check if the del node is terminal node, if it is unlink it from the parent
   if (del.getLeft() == null && del.getRight() == null) {
      if (del == root) {
         root = null;
         return true;
      }

      if (parent.getLeft() == del) {
         parent.setLeft(null);
      } else {
         parent.setRight(null);
      }

      return true;
   }

   //del is non terminal node
```

```
        //shift the node down the tree
        if (del.getLeft() != null) {
           //find max from left
           Node max = del.getLeft();
           parent = del;

           while (max.getRight() != null) {
              parent = max;
              max = max.getRight();
           }

           //swap del with max
           int tempData = del.getData();
           del.setData(max.getData());
           max.setData(tempData);

           del = max;

        } else {
           //find min from right
           Node min = del.getRight();
           parent = del;

           while(min.getLeft() != null) {
              parent = min;
              min = min.getLeft();
           }

           int tempData = del.getData();
           del.setData(min.getData());
           min.setData(tempData);

           del = min;
        }
```

```
    }
  }

  public int count(Node root) {
    if(root == null) {
      return 0;
    }

    return 1 + count(root.getLeft()) + count(root.getRight());
  }
}
//*---------------------------Tester main Class---------------------------*/
public class Main {
  public static void main(String [] args) {

    BinarySearchTree bst = new BinarySearchTree();

    bst.preOrder();
    bst.inOrder();
    bst.postOrder();

    System.out.println( bst.insert(50) );
    System.out.println( bst.insert(20) );
    System.out.println( bst.insert(28) );
    System.out.println( bst.insert(80) );
    System.out.println( bst.insert(10) );
    System.out.println( bst.insert(60) );
    System.out.println( bst.insert(100) );
    System.out.println( bst.insert(15) );
    System.out.println( bst.insert(55) );
    System.out.println( bst.insert(70) );
    System.out.println( bst.insert(15) );
    System.out.println( bst.insert(12) );
```

```
System.out.println( bst.insert(18) );
System.out.println("count = " + bst.count(bst.getRoot()));

bst.preOrder();
bst.preOrder(bst.getRoot());
System.out.println();

bst.inOrder();
bst.inOrder(bst.getRoot());
System.out.println();

bst.postOrder();
bst.postOrder(bst.getRoot());
System.out.println();

System.out.println();

System.out.println("del 50: " + bst.delete(50) );
bst.preOrder();
bst.inOrder();
bst.postOrder();
System.out.println();

System.out.println("del 28: " + bst.delete(28) );
bst.preOrder();
bst.inOrder();
bst.postOrder();
System.out.println();

System.out.println("del 20: " + bst.delete(20) );
bst.preOrder();
bst.inOrder();
bst.postOrder();
System.out.println();
```

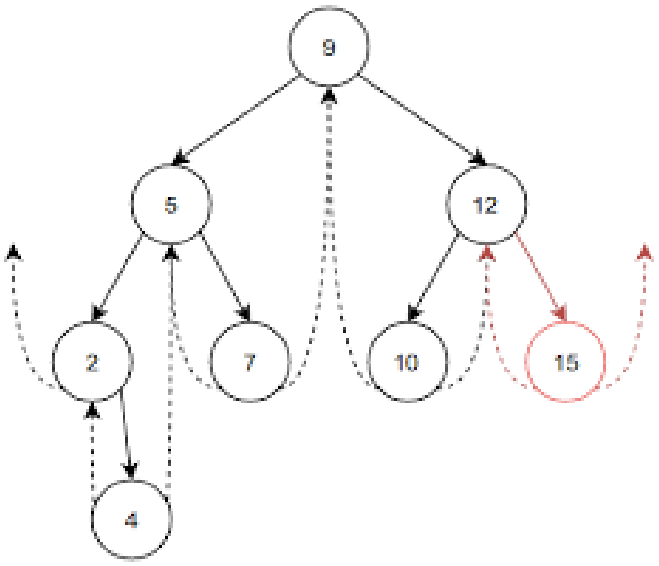
```
        System.out.println("del 55: " + bst.delete(55) );
        bst.preOrder();
        bst.inOrder();
        bst.postOrder();
        System.out.println();

        System.out.println("del 55: " + bst.delete(55) );
        bst.preOrder();
        bst.inOrder();
        bst.postOrder();
        System.out.println();
        System.out.println("count = " + bst.count(bst.getRoot()));
    }
}
```

Threaded Binary Search Tree

A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list representation, there is a number of NULL pointers than actual pointers. Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play any role except indicating that there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as threads.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.




```
//*------------------------ Threded BST Node Class---------------*/
```

```
public class Node {
	private int data;
	private Node left;
	private Node right;
	private char Lflag;
	private char Rflag;
	public Node(int data) {
		super();
		this.data = data;
		this.left=null;
		this.right=null;
		this.Lflag='T';
		this.Rflag='T';
	}
	public int getData() {
		return data;
	}
	public void setData(int data) {
		this.data = data;
	}
```

```
public Node getLeft() {
        return left;
}
public void setLeft(Node left) {
        this.left = left;
}
public Node getRight() {
        return right;
}
public void setRight(Node right) {
        this.right = right;
}
public char getLflag() {
        return Lflag;
}
public void setLflag(char lflag) {
        Lflag = lflag;
}
public char getRflag() {
        return Rflag;
}
```

```
	public void setRflag(char rflag) {
		Rflag = rflag;
	}
}
//*---------------------------- Threaded BST Class ----------------------------*/
public class TBST {
	private Node root;
	public TBST() {
		super();
		this.root=null;
	}
	public Node getRoot() {
		return root;
	}
	public void setRoot(Node root) {
		this.root = root;
	}
	public boolean insert(int data){
		Node newnode=new Node(data);
		if(root==null){
			root=newnode;
```

```
                return true;
        }
        Node temp=root;
        while(temp.getData()!=data){
                if(data<temp.getData()){
                        //insert left
                        if(temp.getLflag()=='T'){
        //parent will become inorder successor of newnode
                                newnode.setRight(temp);
//parent inorder predecessor will become inorder predecessor of newnode
                                newnode.setLeft(temp.getLeft());
                                //newnode will  become left child of parent
                                temp.setLeft(newnode);
                                temp.setLflag('L');
                                return true;
                                }
                        temp=temp.getLeft();
                }
                else{
                        //insert right
                        if(temp.getRflag()=='T'){
```

```
            //parent will become inorder predecessor of newnode
                        newnode.setLeft(temp);
    //parent inorder successor will become inorder successor of newnode
                        newnode.setRight(temp.getRight());
    //newnode will become right child of parent
                        temp.setRight(newnode);
                        temp.setRflag('L');
                        return true;

                    }
                    temp=temp.getRight();
                }
            }
            return false;
    }

    public void preOrder(){
            Node temp=root;
            char flag='L';
            while(temp!=null){
                while(temp.getLflag()=='L' && flag=='L'){
```

```
                System.out.print(" "+temp.getData());
                temp=temp.getLeft();
            }
            if(flag=='L'){
                System.out.print(" "+temp.getData());
            }
            flag=temp.getRflag();
            temp=temp.getRight();
        }
    }
    public void inOrder(){
        Node temp=root;
        char flag='L';
        while(temp!=null){

            while(temp.getLflag()=='L' && flag=='L'){
                temp=temp.getLeft();
            }
            System.out.print(" "+temp.getData());
            flag=temp.getRflag();
            temp=temp.getRight();
```

```
        }
    }
    public boolean isRight(Node node){
        if(node==root){
            return false;
        }
        Node temp=root;
        while(true){
            if(node.getData()<temp.getData()){
                temp=temp.getLeft();
                if(temp==node){
                    return false;
                }
            }
            else{
                temp=temp.getRight();
                if(temp==node){
                    return true;
                }
```

```
                }
            }
        }
        public void postOrder(){
            Node temp=root;
            char flag='L';
            while(temp!=null){
                while(temp.getLflag()=='L' && flag=='L'){
                    temp=temp.getLeft();
                }
                flag=temp.getRflag();
                if(flag=='L'){
                    temp=temp.getRight();
                }
                else{
                    while(true){
                        System.out.print(" "+temp.getData());
                        boolean isRight=isRight(temp);
                        if(isRight){
                            while(temp.getLflag()=='L'){
                            temp=temp.getLeft();
```

```
                                        }
                                temp=temp.getLeft();
                                }
                                else{
                                        while(temp.getRflag()=='L'){
                                        temp=temp.getRight();
                                }
                                temp=temp.getRight();
                                break;
                        }
                }
        }
}


        public boolean deleteData(int data){
                Node del=root,parent=root;
                if(root==null){
                        return false;
```

```
        }
        while(true){
            while(del.getData()!=data){
                if(data<del.getData()){
                    //left
                    if(del.getLflag()=='T'){
                    return false;
                    }
                    parent=del;
                    del=del.getLeft();
                }
            else{
                //right
                if(del.getRflag()=='T'){
                    return false;
                }
                parent=del;
                del=del.getRight();
            }
        }
```

```
//check for terminal node
if(del.getLflag()=='T' && del.getRflag()=='T'){


        //check for root node
        if(del==root){
                root=null;
                return true;
        }
        //if del is not root
        //check exist at left ot right
        if(parent.getLeft()==del){
                //del is left child
                parent.setLeft(del.getLeft());
                parent.setLflag('T');
        }
        else{
                //right child


                parent.setRight(del.getRight());
                parent.setRflag('T');
        }
```

```
            return true;
        }
        //if del is not terminal node
        if(del.getLflag()=='L'){
            //find max data from left subtree
            Node max=del.getLeft();
            parent=del;
            while(max.getRflag()=='L'){
                parent=max;
                max=max.getRight();
            }
            //swap
            int temp=max.getData();
            max.setData(del.getData());
            del.setData(temp);
            del=max;
        }
        else{
            //find min data from right subtree
            Node min=del.getRight();
            parent=del;
```

```
                while(min.getLflag()=='L'){
                        parent=min;
                        min=min.getLeft();

                }
                int temp=min.getData();
                min.setData(del.getData());
                del.setData(temp);
                del=min;
            }
        }
        }
}
//*-------------Tested Main Class----------------------*/
public class Test {
        public static void main(String[] args){
                TBST t=new TBST();
                t.insert(50);
                t.insert(30);
                t.insert(90);
                t.insert(40);
```

```
                t.insert(60);
                t.insert(20);
                t.insert(100);
                t.insert(25);
                /*t.inOrder();
                System.out.println("\n\n----------------\n\n");
                //t.preOrder();
                t.deleteData(50);
                t.inOrder();
                System.out.println("\n\n----------------\n\n");
                t.deleteData(30);*/
                t.postOrder();
                System.out.println("\n\n----------------\n\n");
        }
}
```

Graph

Graphs provide the ultimate in data structure flexibility. A graph consists of a set of nodes, and a set of edges where an edge connects two nodes. Trees and lists can be viewed as special cases of graphs.

Graphs are used to model both real-world systems and abstract problems, and are the data structure of choice in many applications.

Here is a small sampling of the types of problems that graphs are routinely used for.

- Modeling connectivity in computer and communications networks.
- Representing an abstract map as a set of locations with distances between locations. This can be used to compute shortest routes between locations such as in a GPS routefinder.
- Modeling flow capacities in transportation networks to find which links create the bottlenecks.
- Finding a path from a starting condition to a goal condition. This is a common way to model problems in artificial intelligence applications and computerized game players.
- Modeling computer algorithms, to show transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
- Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

Definition

A Graph is a collection of nodes, which are called vertices 'V', connected in pairs by line segments, called Edges E.

Sets of vertices are represented as V(G) and sets of edges are represented as E(G).

So a graph is represented as G = (V, E).

There are two types of Graphs

- Undirected Graph
- Directed Graph

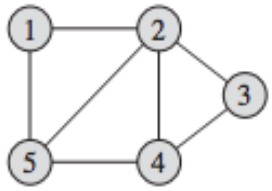
Directed Graph are usually referred as Digraph for Simplicity. A directed Graph is one, where each edge is represented by a specific direction or by a directed pair <v1, v2>.Hence <v1, v2> & <v2, v1> represents two different edges.

Terminology related to Graph:

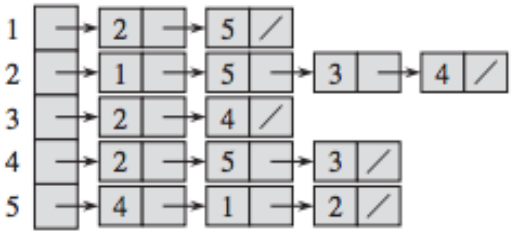
- Out – degree: The number of Arc exiting from the node is called the out degree of the node.
- In- Degree: The number of Arcs entering the node is the In degree of the node.
- Sink node: A node whose out-degree is zero is called Sink node.
- Path: path is sequence of edges directly or indirectly connected between two nodes.
- Cycle: A directed path of length at least L which originates and terminates at the same node in the graph is a cycle
-

The graphs can be implemented in two ways

1. Using Array
2. Using Linked List



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Graph traversal

Many graph applications need to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph traversal and is similar in

concept to a tree traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems.

1. **Depth first search**

   Our first method for organized graph traversal is called depth-first search (DFS). Whenever a vertex v is visited during the search, DFS will recursively visit all of v 's unvisited neighbors. Equivalently, DFS will add all edges leading out of v to a stack. The next vertex to be visited is determined by popping the stack and following that edge. The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a depth-first search tree. This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs.

2. **Breadth first search**

   Our second graph traversal algorithm is known as a breadth-first search (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

```
//*--------------------Graph class-------------*/
//Adjacency Matrix representation

public class Graph {

   private int noOfVertices;
   private char[] vertices;
   private byte[][] adjMat;

   public Graph(int noOfVertices) {
```

```
        this.noOfVertices = noOfVertices;
        vertices = new char[noOfVertices];
        adjMat = new byte[noOfVertices][noOfVertices];

        init();
    }

    private void init() {
        /*System.out.println("Enter values for vertices: ");
        Scanner scanner = new Scanner(System.in);*/

        for (int i = 0; i < noOfVertices; i++) {
            //vertices[i] = scanner.nextLine().charAt(0);
            vertices[i] = (char) (97 + i);
        }

        byte arr[] = {0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0,
0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0};
        int k= 0;

        for (int i = 0; i < noOfVertices; i++) {
            for (int j = 0; j < noOfVertices; j++) {
                adjMat[i][j] = arr[k++];
                /*
                System.out.println("Edge between " +  vertices[i] + " & " +
vertices[j] + ": ");
                adjMat[i][j] = scanner.nextByte();
                */
            }
```

```
    }


}

public void display() {
   System.out.print("  ");
   for(int i = 0; i < noOfVertices; i++) {
      System.out.print(vertices[i] + " ");
   }
   System.out.println();

   for(int i = 0; i < noOfVertices; i++) {
      System.out.print(vertices[i]  + " ");
      for(int j = 0; j < noOfVertices; j++) {
         System.out.print( adjMat[i][j] + " ");
      }
      System.out.println();
   }
}

public void bfs(int startVertex) {

   byte [] visited = new byte[noOfVertices];
   int [] queue = new int[100];
   int front = -1, rear = -1;
   queue[++rear] = startVertex;
   ++front;
```

```
    System.out.print("BFS: ");

    int curVertex;
    while( front <= rear) {

       curVertex = queue[front++];
       if(visited[curVertex] == 0) {
          System.out.print( vertices[curVertex]  + " ");
          visited[curVertex] = 1;

          for(int i = 0; i < noOfVertices; i++) {
             if(adjMat[curVertex][i] == 1 && visited[i] == 0) {
                queue[++rear] = i;
             }
          }
       }
    }
System.out.println();
 }

 public void dfs(int startVertex) {
    System.out.print("DFS: ");
    int [] visited = new int[noOfVertices];
    int [] stack = new int[100];
    int top = -1;

    int curVertex = startVertex;
    System.out.print( vertices[curVertex] + " ");
    visited[curVertex] = 1;
```

```
      stack[++top] = curVertex;

      while(top != -1) {
         curVertex = stack[top];
         for(int i = 0; i < noOfVertices; i++) {
            if(adjMat[curVertex][i] == 1 && visited[i] == 0) {
               System.out.print( vertices[i] + " ");
               visited[i] = 1;
               stack[++top] = i;
               curVertex = i;
               i = -1;
            }
         }
         top--;
      }
      System.out.println();
   }
}
//*-------------------------Tester Main ---------------------*/
public class Main {
   public static void main(String [] args) {
      Graph g = new Graph(9);
      g.display();
      g.bfs(0);
      g.bfs(5);
      g.dfs(0);
      g.dfs(3);
   }
}
```

Searching & Sorting Algorithms

Searching means finding out an element in array that meet some specified criteria. Sorting means rearranging all the items in array in increasing or decreasing order.

Sequential Search: This is simplest type of searching. The search starts at first record and move through each record until match is made or not.

Analysis of sequential search

The best case for sequential search is that it does one comparison, and matches X right away. In the worst case, sequential search does n comparisons, and either matches the last item in the list or doesn't match anything. The average case is harder to do.

The Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Analysis of Binary Search

In the base case, the algorithm will end up either finding the element or just failing and returning false. In both cases, the algorithm is going to take a constant time because only comparison and return statements are going to be executed.

```
import java.util.*;
public class BinarySearch {
public static void main(String[] args) {
   int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
   int item, location = -1;
   System.out.println("Enter the item which you want to search");
   Scanner sc = new Scanner(System.in);
   item = sc.nextInt();
   location = binarySearch(arr,0,9,item);
   if(location != -1)
   System.out.println("the location of the item is "+location);
   else
      System.out.println("Item not found");
   }
public static int binarySearch(int[] a, int beg, int end, int item)
{
   int mid;
   if(end >= beg)
   {
      mid = (beg + end)/2;
      if(a[mid] == item)
      {
         return mid+1;
      }
      else if(a[mid] < item)
      {
         return binarySearch(a,mid+1,end,item);
      }
      else
      {     return binarySearch(a,beg,mid-1,item);
      }
       }
   return -1;   } }
```

Introduction To Sorting

Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The

comparison operator is used to decide the new order of element in the respective data structure.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted.

- There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters.
- First parameter is the execution time of program, which means time taken for execution of program.
- Second is the space, which means space taken by the program.

Type of Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Shell Sort

Bubble Sort

- Bubble sorting is a simple sorting technique in which we arrange the elements of the list by forming pairs of adjacent elements
- Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements.

- Bubble Sort compares all the element one by one and sort them based on their values.
- It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
- Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

We can create a java program to sort array elements using bubble sort. Bubble sort algorithm is known as the simplest sorting algorithm.

In bubble sort algorithm, array is traversed from first element to last element. Here, current element is compared with the next element. If current element is greater than the next element, it is swapped.

```
void bubbleSort(int[] arr) {
    int n = arr.length;
     int temp = 0;
          for(int i=0; i < n; i++){
          for(int j=1; j < (n-i); j++){
                        if(arr[j-1] > arr[j]){
                      //swap elements
                          temp = arr[j-1];
                        arr[j-1] = arr[j];
                       arr[j] = temp;
               }
          }
          }
```

Selection Sort

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Selection Sort Applications

The selection sort is used when:

- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is O(n) as compared to O(n2) of bubble sort)

// Selection sort in Java

```
void selectionSort(int array[]) {
  int size = array.length;
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {
      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx]) {
        min_idx = i;
      }
    }

    // put min at the correct position
    int temp = array[step];
    array[step] = array[min_idx];
    array[min_idx] = temp;
  }
}
```

Insertion Sort

Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.

```
 void insertionSort(int array[]) {
  int size = array.length;

  for (int step = 1; step < size; step++) {
   int key = array[step];
   int j = step - 1;

   // Compare key with each element on the left of it until an element smaller than
   // it is found.
   // For descending order, change key<array[j] to key>array[j].
   while (j >= 0 && key < array[j]) {
    array[j + 1] = array[j];
    --j;
   }

   // Place key at after the element just smaller than it.
   array[j + 1] = key;
  }
 }
```

Insertion Sort Applications

The insertion sort is used when:

- the array is has a small number of elements
- there are only a few elements left to be sorted

Shell Sort

Shell sort is an algorithm that first sorts the elements far apart from each other and successively reduces the interval between the elements to be sorted. It is a generalized version of insertion sort.In shell sort, elements at a specific interval are sorted. The interval between the elements is gradually decreased based on the sequence used. The performance of the shell sort depends on the type of sequence used for a given input array.

// Shell sort in Java programming

```
 void shellSort(int array[], int n) {
 for (int interval = n / 2; interval > 0; interval /= 2) {
  for (int i = interval; i < n; i += 1) {
  int temp = array[i];
  int j;
  for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
   array[j] = array[j - interval];
  }
  array[j] = temp;
  }
 }
 }
```

HeapSort

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.

You must be wondering, how converting an array of numbers into a heap data structure will help in sorting the array.

Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list/array.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

// Java program for implementation of Heap Sort

```
public void sort(int arr[])
{
    int n = arr.length;

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i;  // Initialize largest as root
    int l = 2*i + 1;  // left = 2*i + 1
    int r = 2*i + 2;  // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;
```

```
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

Time Complexity:

Time Complexities of Searching & Sorting Algorithms:

	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$