LAB MANUAL

Course: CSC211-Data Structures and Algorithms

Department of Computer Science

Java Learning Procedure

1) Stage J (Journey inside-out the concept)
2) Stage a_1 (Apply the learned)
3) Stage V (Verify the accuracy)
4) Stage a_2 (Assess your work)

COMSATS University Islamabad

Table of Contents

Lab #	Topics Covered	Page #
Lab # 01	C++ Revision (Ms. Memoona)	3
Lab # 02	Singly Linked List (Dr. yasir)	10
Lab # 03	Circular Linked list (Dr. yasir)	19
Lab # 04	Doubly Linked List (Dr. yasir)	28
Lab # 05	Stacks (Dr. yasir)	36
Lab # 06	**Lab Sessional 1**	
Lab # 07	Queues (Dr. yasir)	39
Lab # 08	Recursion (Dr. Inayat)	41
Lab # 09	Sorting (Mr. Zulfiqar)	49
Lab # 10	Sorting (Mr. Zulfiqar)	53
Lab # 11	Hashing (Ms. Memoona)	58
Lab # 12	**Lab Sessional 2**	
Lab # 13	Binary Trees (Dr. Inayat)	71
Lab # 14	AVL Trees (Ms. Memoona)	83
Lab # 15	Heaps (Dr. Inayat)	98
Lab # 16	Graphs (Ms. Memoona)	106
	Terminal Examination	

LAB # 01

Statement Purpose:

This lab will give you an overview of Java language.

Activity Outcomes:

This lab teaches you the following topics:

- Basic syntax of Java
- Data types and operators in Java
- Control flow statements in Java
- Arrays and Functions

Instructor Note:

Introduction to Programming in C++

- https://ece.uwaterloo.ca/~dwharder/aads/Tutorial/
- http://www.cplusplus.com

1) Stage J (Journey)

Introduction

C++, as we all know is an extension to C language and was developed by **Bjarne stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
3. Exception Handling is there in C++.
4. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
5. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

2) Stage a1 (apply)

Lab Activities:

Activity 1: **Write a Hello World program in Java.**

Solution:

```java
package lab1;
public class HelloWorld {
	public static void main(String args[])
	{
		System.out.println("HelloWorld ";
		System.out.println("My first program in DS lab");
	}
}
```

Activity 2: **Write a program to use different data types in Java**

Solution:

```
package lab1;
public class DataTypes {
     public static void main(String args[]){
            double a=5.5;           // initial value: 5
            int b=3;               // initial value: 3
            int c=2;               // initial value: 2
            double result;          // initial value undetermined
            a = a + b;
            result = a-c;
            System.out.println(result);
          }
     }
```

Activity 3: Write a program to use string data type in Java

Solution:

```
package lab1;
public class StringType {
     public static void main(String args[])
     {
          String mystring;
          mystring = "This is the initial string content";
          System.out.println(mystring);
          mystring ="This is a different string content";
          System.out.println(mystring);
     }
 }
```

Activity 4: Write a program to use arithmetic operators in Java

Solution:

```
package lab1;
public class Operators {
     public static void main(String args[])
     {
          int a, b=3;
          a = b;
          a+=2;              // equivalent to a=a+2
          System.out.println(a);
     }
}
```

Activity 5: Write a program to use relational operators in Java

Solution:

```
package lab1;
public class RelationalOperators {
     public static void main(String args[])
     {
     int a,b,c;
     a=2;
     b=7;
     c = (a>b) ? a : b;
     System.out.println(c);
     }
 }
```

Activity 6: Write a program to use if-else statement in Java

Solution:

```
package lab1;
import java.util.Scanner;
public class IfElse {
	public static void main(String args[])
	{
		int x;
		Scanner sc=new Scanner(System.in);
		System.out.println("Please Enter a Value");
		x=sc.nextInt();
		if (x > 0)
		System.out.println( "x is positive");
		else if (x < 0)
		System.out.println( "x is negative");
		else
		System.out.println( "x is 0");

	}
}
```

Activity 7: Write a program to use while loop

Solution:

```
package lab1;
public class WhileLoop {
	public static void main(String args[])
	{
		int n = 10;
		while (n>0) {
		System.out.println(n);
		--n;
		}
		System.out.println("liftoff");
		}
	}
```

Activity 8: Write a program to use do-while loop in Java

Solution:

```
package lab1;
import java.util.Scanner;
public class DoWhileLoop {
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String str;
		do {
		System.out.println("Enter text: ");
		str=sc.nextLine();
		System.out.println("You entered: "+ str);
		} while (!(str.equals("bye")));
	}
}
```

Activity 9: Write a program that will add two numbers in a function and call that function in main.

Solution:

```
package lab1;
public class Functions {
	public static void main(String args[])
	{
		int z;
		z = addition(5,3);
		System.out.println("The result is "+ z);

	}
	static int  addition (int a, int b)
	{
	int r; r=a+b; return r;
	}

}
```

Activity 10: Write a program to use the concept of Arrays in Java

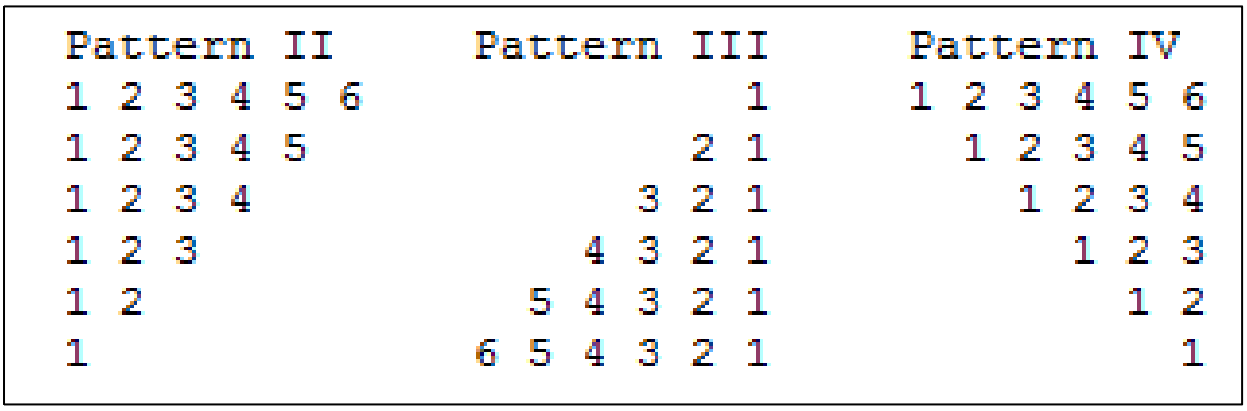
Solution:

```
package lab1;
public class Arrays {
	public static void main(String args[])
	{
		int foo []= {16, 2, 77, 40, 50};
		int n, result=0;
		for (n=0 ; n<5 ; ++n )
		{
		result += foo[n];
		}
		System.out.println(result);
	}
}
```

3) Stage V (verify)

Home Activities:

1. Use nested loops that print the following patterns in three separate programs.



```
Pattern II          Pattern III           Pattern IV
1 2 3 4 5 6                   1           1 2 3 4 5 6
1 2 3 4 5                   2 1             1 2 3 4 5
1 2 3 4                   3 2 1               1 2 3 4
1 2 3                   4 3 2 1                 1 2 3
1 2                   5 4 3 2 1                   1 2
1                   6 5 4 3 2 1                     1
```

2. (Printing numbers in a pyramid pattern) Write a nested for loop that prints the following output:

```
                1
             1  2   1
          1  2  4   2 1
       1  2  4  8   4  2  1
    1  2  4  8  16  8  4  2 1
 1 2  4  8 16  32 16  8  4 2 1
1 2 4  8 16 32  64 32 16  8 4 2 11
2 4 8 16 32 64 128 64 32 16 8 4 2 1
```

4) Stage a2 (assess)

Lab Assignment and Viva voce

1. Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays **the student with the highest score**.
2. Write a program that displays all the numbers from 100 to 1000, **ten per line, that are divisible by 5 and 6.**
3. Write a program to generate Fibonacci Series like given below using loop up to given number by user. If user enter number = 20, your output must be

   Fibonacci Series up to 20

   0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

[Hint: add the last two number in the series to get new number of the series**]**

LAB # 02

Statement Purpose:

The purpose of this lab session is to acquire skills in working with singly linked lists.

Activity Outcomes:

This lab teaches you the following topics:

- This lab covers the flowing topics
- Basic syntax of Arrays in Java
- Operations on Arrays
- Traversing
- Insertion
- Searching
- Deleting
- Searching
- ArrayList

Instructor Note:

NULL

1) Stage J (Journey)

<u>Introduction</u>

2) Stage a1 (apply) Lab Activities:

Activity 1:

- **Write a program to copy content of one array to another array**

Solution:

```
package lab2;
public class ArrayCopying {
    public static void main(String args[])
    {
        double[] myList = {1.9, 2.9, 3.4, 3.5};
        double[] myList2 = {1.1, 2.5, 3.7, 3.10};
        myList2=myList;
        for (int i=0;i<myList2.length;i++)
        System.out.println(myList2[i]);
    }
}
```

Activity 2:

- **Write a program to pass array to methods**

Solution:

```
package lab2;
public class ArraystoMethods {
    public static void main(String args[])
    {
        //int[] list = {3, 1, 2, 6, 4, 2};
        //printArray(list);
        printArray(new int[]{3, 1, 2, 6, 4, 2});
    }
    public static void printArray(int[] array) {
          for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
          }
        }
}
```

Activity 3:

- **Write a program to pass array to methods by value**

Solution:

```
package lab2;
public class ArrayPassbyValues {
    public static void main(String[] args) {
        int x = 1;
        int[] y = new int[10];
        y[0]=232;
        m(x, y);
        System.out.println("x is " + x);
        System.out.println("y[0] is " + y[0]);
      }
      public static void m(int number, int[] numbers) {
        number = 1001; // Assign a new value to number
        numbers[0] = 5555; // Assign a new value to
numbers[0]
      }
 }
```

Activity 4:

- **Write a program to traverse elements of an array**

Solution:

```
package lab2;
public class ArrayTraversal {
    public static void main(String args[])
    {
        double[] myList = {1.9, 2.9, 3.4, 3.5};
        for (int i=0;i<myList.length;i++)
        System.out.println(myList[i]);
    }

 }
```

Activity 5:

- **Write a program to insert an element of an array at specified position.**

Solution:

```
package lab2;
public class ArrayInsertion {
    public static void main(String args[])
    {
        double[] myList=new double[8];
        myList[0] =1.9;
        myList[1] =1.2;
        myList[2] =1.6;
        myList[3] =1.1;
        int k=0;
        double value=1.12;
        if(k>=myList.length){
            System.out.println("The postion should be
less than the size");
            return;
        }
        int j=myList.length-2;
        while(j>=k){
            myList[j+1]=myList[j];
            j--;
        }
        myList[k]=value;
        for(int i=0;i<myList.length;i++)
        System.out.println(myList[i]);
    }
 }
```

Activity 6:

- **Write a program to delete an elements of an array from specified position.**

Solution:

```
package lab2;
public class ArrayDeletion {
    public static void main(String args[])
    {
        double[] myList=new double[8];
        myList[0] =1.9;
        myList[1] =1.2;
        myList[2] =1.6;
```

```
          myList[3] =1.1;
          int k=3;
          if(k>=myList.length){
               System.out.println("The postion should be
less than the size");
               return;
          }
          int j=k;
          while(j<myList.length-1){
               myList[j]=myList[j+1];
               j++;
          }
          for(int i=0;i<myList.length;i++)
          System.out.println(myList[i]);
     }
 }
```

Activity 7:

- **Write a program to search an element in an array.**

Solution:

```
package lab2;
public class ArraySearching {
     public static void main(String args[])
     {
          double[] myList = {1.9, 2.9, 3.4, 3.5};
          double val=3.5;
          int pos=0;
          boolean found=false;
          for (int i=0;i<myList.length;i++)
               if(myList[i]==val){
               found=true;
               pos=i;
               break;
               }
               if(found)
               System.out.println(val +" found at postion
"+pos);
               else
               System.out.println(val+" not found ");
     }
}
```

Activity 8:

- **Write a program to elaborate ArrayList**

Solution:

```
package lab2;
import java.util.ArrayList;
public class ArrayListExample {
public static void main(String args[]){
    ArrayList<String> alist = new ArrayList<String>();
    ArrayList list=new ArrayList();
    alist.add("Apple");
    alist.add("Banana");
    alist.add("Mango");
    alist.add("Grapes");
    System.out.println(alist);
    System.out.println(alist.get(1));

}
}
```

3) Stage V (verify)

Home Activities:

1. Write a program that prompts the user to enter ten integer values. The program stores the values in an array, finds and displays the smallest and largest values in the array.
2. Write a program that stores all prime numbers from 100 to 500 in array.
3. Write a program that asks the user to enter a number and generate Fibonacci Series like given below using loop up to given number and stores the results in an array. If user enter number = 20, your output must be **0   1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181.** Your program then displays the values stored in the array.
4. Write a program that uses at least five methods of the ArrayList.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

- Write a program that tells whether two consecutive elements of the array are same/duplicates. The program prints the positions and the value.
- Write a program that prints duplicates (if any) of each element of the array.
- Write a program that asks the user to enter ten values and stores the values in an array. The program then asks the user to enter a value and finds all occurrences of the value i.e., prints index numbers. Prints not found if the no occurrence is found.

LAB # 03

Statement Purpose:

The purpose of this lab session is to acquire skills in working with singly linked lists.

Activity Outcomes:

This lab teaches you the following topics:

Creation of singly linked list
Insertion in singly linked list
Deletion from singly linked list
Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A list is a finite ordered set of elements of a certain type.
The elements of the list are called cells or nodes.
A list can be represented statically, using arrays or, more often, dynamically, by allocating and releasing memory as needed. In the case of static lists, the ordering is given implicitly by the one-dimension array. In the case of dynamic lists, the order of nodes is set by pointers. In this case, the cells are allocated dynamically in the heap of the program. Dynamic lists are typically called linked lists, and they can be singly- or doubly-linked.

The structure of a node may be:

```
class Node {
        int value;
        Node next;
}
```

Activity 1:

Creation of linked list and insertion to singly linked list

Solution:

Consider the following steps which apply to *dynamic* lists:

1. Initially, the list is empty. First node is created, and the next field is set to null. This node is the head and tail node.

```
Node node = new Node(nodeValue)
node.next = null;
head = node;
tail = node;
```

2. For creating more nodes and insertion at different positions, some conditions are checked. If the node is to be inserted at the start following steps are performed.

```
node.next = head;
head = node;
```

3. If the node is to be inserted at the last position, the following steps are carried out.

```
tail.next = node;
tail = node;
```

4. For any other position, following steps are performed for insertion

```
Node tempNode = head;
int indexCounter = 0;
while (indexCounter < nodeLocation - 1) {
      tempNode = tempNode.next;
      indexCounter++;
      }
     Node nextNode = tempNode.next;
     tempNode.next = node;
     node.next = nextNode;
```

Activity 2:

Accessing nodes of linked list

Solution:

This can be accomplished with starting from head node till null is found in the next field.

```
if (head == null) {
    System.out.println("The list is empty!");
} else {
```

```
                Node tempNode = head;
                while (tempNode.next != null) {
                    System.out.print(tempNode.value + " ");
                    tempNode = tempNode.next;
                }
                System.out.print(tempNode.value + " ");
            }
}
```

Activity 3:

Searching singly linked list at a specific node/index

Solution:

To do this we need to loop over the list till the desired index. We also need checks on index i.e., it should be in the range.

```
Node tempNode = head;
    if(index<0 || index > size)
    {
        System.out.println("Please a valid index");
        return;
    }
        int counter = 0;
        while (counter < index) {
            counter++;
            tempNode = tempNode.next;
        }
        System.out.println("The value at index "+index
+" is "+tempNode.value);

    }
```

Activity 4:

Searching a specific value in a singly linked list.

Solution:

For this purpose, we need to scan the list till the value is found. If we reach at the end of the list and the value is not found then the not found will be printed. Note that this code will print only the first occurrence.

Activity 5:

Deleting a node from single linked list

Solution:

When we are to remove a node from a list, there are some aspects to take into account: (i) list may be empty; (ii) list may contain a single node; (iii) list has more than one node. And, also, deletion of the first, the last or a node given by its key may be required. Thus we have the following cases:

1. Check if list is empty

```
if (head == null)
        System.out.println("The list is empty!");
        return;
```

2. Delete head node

```
 if (index <= 0)
        head = head.next;
```

3. Removing the last node of a list

```
Node tempNode = head;
while (tempNode.next != tail) {
          tempNode = tempNode.next;
         }
       tempNode.next = null;
       tail = tempNode;
```

4. Removing a node given by a *key*

```
Node tempNode = head;
int counter = 0;
while (counter < index - 1) {
        tempNode = tempNode.next;
        counter++;
        }
     Node nextNode = tempNode.next.next;
     tempNode.next = nextNode;
```

Activity 6:

Complete deletion of single linked list

Solution:

For a complete deletion of a list, we have to set head equal to null and both head and tail are at same place.

```
head = tail = null;
```

3) Stage V (verify)

Home Activities:

1. Write a function that prints all nodes of a linked list in the reverse order.
2. Write a function which reverses the order of the linked list.
3. Write a function which rearranges the linked list by group nodes having even numbered and odd numbered value in their data part.

4. Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

4) Stage a$_2$ (assess)

Lab Assignment and Viva voce

LAB # 04

Statement Purpose:

This lab session is intended to help you develop the operations on doubly-linked lists.
In this lab session we will also enhance singly linked lists with another feature: we'll make them circular. And, as was the case in the previous lab session, we will show how to implement creation, insertion, and removal of nodes.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of doubly linked list
- Insertion in doubly linked list
- Deletion from doubly linked list
- Traversal of all nodes
- Creation of circular linked list
- Insertion in circular linked list
- Deletion from circular linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A doubly-linked list is a (dynamically allocated) list where the nodes feature two relationships: successor and predecessor.

Activity 1:

Creation of doubly linked list and insertion into doubly linked list

Solution:

Consider the following steps which apply to doubly linked list.

1. Initially, the list is empty. First node is created, and the next and previous fields are set to null. This node is the head and tail node.

```
Node node = new Node(nodeValue)
node.previous = null;
node.next = null;
head = node;
tail = node;
```

2. For creating more nodes and insertion at different positions, some conditions are checked. If the node is to be inserted at the head side following steps are performed.

```
node.next = head;
head.previous = node;
head = node;
```

3. If the node is to be inserted at the tail side, the following steps are carried out.

```
node.previous = tail;
tail.next = node;
tail = node;
```

For any other position, following steps are performed for insertion.

```
      Node tempNode = head;
            int counter = 0;
            while (counter < nodeLocation - 1) {
                tempNode = tempNode.next;
                counter++;
            }
            Node nextNode = tempNode.next;
            tempNode.next = node;
            node.previous = tempNode;
            node.next = nextNode;
            nextNode.previous = node;
```

Activity 2:

Accessing nodes of doubly linked list

Solution:

The nodes of a doubly linked list can be accessed either from head side in forward direction or from tail side in backward direction. For forward direction we need to loop over the list until we reach null in the next field. Only last node can have null in the next field.

```
Node tempNode = head;
        if (tempNode == null) {
            System.out.println("The list is empty");
            return;
        }
        while (tempNode.next != null) {
            System.out.print(tempNode.value + " ");
            tempNode = tempNode.next;
        }
        System.out.print(tempNode.value + "\n");
```

To access elements in backward direction, we need to loop over the list starting from tail until we reach null in the previous field. Only head node has null in the previous field.

```
Node tempNode = tail;
        if (tempNode == null) {
            System.out.println("The list is empty");
            return;
        }
        while (tempNode.previous != null) {
            System.out.print(tempNode.value + " ");
            tempNode = tempNode.previous;
        }
        System.out.print(tempNode.value + "\n");
```

Activity 3:

Deletion from doubly linked list

Solution:

Deleting a node from doubly linked list may have the following cases.

Delete the hade node.

In this case we need to move head node to the next of the head and set the previous field of the new head to null.

```
     head = head.next;
     head.previous= null;
```

Delete tail node

In this case we need to move the tail to previous of the tail and set the next field of the new tail to null.

```
        tail = tail.previous;
        tail.next = null;
```

Delete intermediate node

In this case we need to search for the desired location and set the previous and next fields of the nodes on the right and left sides of the deleted node.

```
            int counter = 0;
            Node tempNode = head;
            while (counter < nodeLocation - 1) {
                tempNode = tempNode.next;
                counter++;
            }
            Node nextNode = tempNode.next.next;
            tempNode.next = nextNode;
            nextNode.previous = tempNode;
```

Activity 4:

Deleting complete doubly linked list

Solution:

To do this we need to set head equal to tail equal to null.

```
                head = tail = null;
```

Activity 5:

Searching a node in doubly linked list.

Solution:

Two cases are considered here. In the first case, we can get the value of a specified node. For this purpose, we need to loop over the list to the desired index.

```
        public void get(int index) {
        int counter = 0;
        Node tempNode = head;
        if(index<0 || index > size)
    {
         System.out.println("Please enter a valid
index");
         return;
    }
        while (counter < index) {
            tempNode = tempNode.next;
            counter++;
        }
        System.out.println(tempNode.value);
}
```

In the second case, we can search a specific value in the list. For this purpose, loop will be executed till the desired value is found or the last node is reached.

```
public void search(int value) {
        int counter = 0;
        boolean isFound=false;
        Node tempNode = head;
                while (counter<=size) {
                 if(tempNode.value==value){
                     isFound=true;
                     break;
                 }
                    tempNode = tempNode.next;
                    counter++;
                }
                if(isFound)
        System.out.println("The value " +  value + " is
found in Node " + counter);
                else
        System.out.println("The value " +  value + "
not found ");
    }
```

Activity 6:

Operations on singly circular linked list.

Solution:

Singly circular list can be implemented in similar way as singly linked list. The difference is that the next of the singly circular list contains the address of the head node. This point should be considered when inserting elements or deleting elements at tail or head positions. Insertion in the middle of the list is similar to the simple list.
The insert can be written as follows.

```
public void insert(int nodeValue, int nodeLocation) {
     Node node = new Node(nodeValue);
     if (head == null) {
         node.next = node;
         head = node;
         tail = node;
            return;
     } else if (nodeLocation <= 0) {// insert from head
            node.next = head;
            head = node;
            tail.next = head;          // Important
     } else if (nodeLocation>= size){//insert from tail
            tail.next = node;
            tail = node;
            tail.next = head;
    } else {//insert somewhere in between head and tail
            Node tempNode = head;
            int indexCounter = 0;
            while (indexCounter < nodeLocation - 1) {
                tempNode = tempNode.next;
                indexCounter++;
            }
            Node nextNode = tempNode.next;
            tempNode.next = node;
            node.next = nextNode;
        }
        size++;
}
```

The delete function works as follows.

```
public void deleteNodeAt(int index) {
        if (head == null) {
            System.out.println("The list is empty!");
            return;
        } else if (index <= 0) {
            head = head.next;
            tail.next = head;
        } else if (index >= size) {
            Node tempNode = head;
            while (tempNode.next != tail) {
                tempNode = tempNode.next;
            }
            tempNode.next = head;
            tail = tempNode;
        } else {
            Node tempNode = head;
            int counter = 0;
            while (counter < index - 1) {
                tempNode = tempNode.next;
                counter++;
            }
            Node nextNode = tempNode.next.next;
            tempNode.next = nextNode;
        }
        size--;
    }
```

For searching in singly linked list, we used null to stop the loop either on tail side or head side for forward and backward directions respectively. In case of circular list, the loop is controlled using the size of the list. The function is defined as follows.

```
public void search(int value) {
        int counter = 0;
        boolean isFound = false;
        Node tempNode = head;
        while (counter <= size) {
            if (tempNode.value == value) {
                isFound = true;
                break;
            }
            tempNode = tempNode.next;
            counter++;
        }

        if (isFound) {
            System.out.println("The value " + 
tempNode.value + " is found in Node " + counter);
        } else {
            System.out.println("Value not found!");
        }

    }
```

Searching with a specific index is same as in the simple list.

```
public void get(int index) {
        Node tempNode = head;
        if(index<0 || index > size)
    {
         System.out.println("Please enter a valid
index");
         return;
    }
        int counter = 0;
        while (counter < index) {
            counter++;
            tempNode = tempNode.next;
        }
        System.out.println(tempNode.value);
    }
```

Activity 7:

Operations on doubly circular linked list.

Solution:

Doubly circular list can be implemented in similar way as doubly linked list except the head and tail nodes where the next and previous fields store addresses of the head and tail nodes for circular visits.
Different functions are shown as follows.

```java
public void insert(int nodeValue, int nodeLocation) {
        Node node = new Node(nodeValue);
        if (head == null) {
            node.previous = node;
            node.next = node;
            head = node;
            tail = node;
            return;
        } else if (nodeLocation == 0) {
            node.next = head;
            head.previous = node;
            node.previous = tail;
            tail.next = node;
            head = node;
        } else if (nodeLocation >= size) {
            node.previous = tail;
            tail.next = node;
            node.next = head;
            head.previous = node;
            tail = node;
        } else {
            Node tempNode = head;
            int counter = 0;
            while (counter < nodeLocation - 1) {
                tempNode = tempNode.next;
                counter++;
            }
            Node nextNode = tempNode.next;
            tempNode.next = node;
            node.previous = tempNode;
            node.next = nextNode;
            nextNode.previous = node;
        }
        size++;
}
```

```
public void printInReverse() {
        Node tempNode = tail;
        int counter = 0;
        while (counter <= size) {
            System.out.print(tempNode.value + " ");
            tempNode = tempNode.previous;
            counter++;
        }
    }
```

```
    // Get Value of specified indexs' node
    public void get(int index) {
        int counter = 0;
        Node tempNode = head;
        if(index<0 || index > size)
    {
         System.out.println("Please enter a valid
index");
         return;
    }
        while (counter < index) {
            tempNode = tempNode.next;
            counter++;
        }
        System.out.println(tempNode.value);
    }
```

```
// Search for node by giving specified value
    public void search(int value) {
        int counter = 0;
        boolean isFound = false;
        Node tempNode = head;
        while (counter <= size) {
            if(tempNode.value == value){
                isFound = true;
                break;
            }
            tempNode = tempNode.next;
            counter++;
        }
        if(isFound){
            System.out.println("The check value " +
tempNode.value + " is found in Node " + counter);
        } else {
            System.out.println("The search value is not
found!");
        }
    }
```

```
 // Remove node on specific location
    public void deleteNodeAt(int nodeLocation) {
        if (head == null) {
            System.out.println("The head is NULL, so
nothing to delete!");
        } else if (nodeLocation <= 0) {
            head = head.next;
            head.previous = tail;
            tail.next = head;
        } else if (nodeLocation >= size) {
            tail = tail.previous;
            tail.next = head;
            head.previous = tail;
        } else {
            int counter = 0;
            Node tempNode = head;
            while (counter < nodeLocation - 1) {
                tempNode = tempNode.next;
                counter++;
            }
            Node nextNode = tempNode.next.next;
            tempNode.next = nextNode;
            nextNode.previous = tempNode;
        }
        size--;
    }
```

LAB # 05

Statement Purpose:

This lab will introduce you the concept of Stack data structure.

Activity Outcomes:

This lab teaches you the following topics:

- Implementing stack using array
- Implementing stack using linked list
- How to push data onto the stack
- How to pop data from the stack
- How to access the top of the stack
- Conversion of infix to postfix using stack

Instructor Note:

`1) Stage J (Journey)

Activity 1:

Implementing stack using array

Solution:

Stack can be implemented using array or linked list. Array implementation needs predefined capacity whereas linked list implementation is dynamic.

Stack implementation using Array

To achieve this task, we need an array and a variable to point to the point of the array. Different functions will be used for stack operations.

To create array of the specified capacity, constructor is used. Initially the top is set to -1 which means that the stack is empty.

```
public class Stack_Array {
    int top = -1;
    int[] stack;
    public Stack_Array(int size) {
        stack = new int[size];
}
```

The push function works as follows. First, we need to check if the stack is full. If not full the value is placed on top, and top is incremented by 1.

```
public void push(int stackValue) {
        if (top == stack.length - 1) {
            System.out.println("Stack is full");
        } else {
            stack[++top] = stackValue;
        }
    }
```

The pop function first checks if the stack is empty. If not empty the top is decremented by 1.

```
public void pop() {
        if (top == -1) {
            System.out.println("Stack is empty");
        } else {
            top--;
        }
    }
```

The other functions related to stack are as follows.

```
public boolean isEmpty() {
        return top == -1;
    }
    public boolean isFull() {
        return top == stack.length - 1;
    }

    public int peek() {
        return stack[top];
    }
    public void print() {
        for (int i = 0; i <= top; i++) {
            System.out.println(stack[i] + " ");
        }
    }
```

Activity 2:

Implementing stack using linked list

Solution:

The same functions can be implemented using linked list. The push function can be implemented without the restrictions of checking the top of the stack.

```java
public void push(int nodeValue) {
        Node node = new Node();
        node.value = nodeValue;
        if (head == null) {
            node.next = null;
            head = node;
            tail = node;
            return;
        }
        node.next = head;
        head = node;
    }
```

The pop function still needs to check underflow.

```java
public void pop() {
        if (isEmpty()) {
            System.out.println("The Stack is 
empty!!!");
            return;
        }
        head = head.next;
    }
```

The other functions works in similar way as in previous activity.

```
public boolean isEmpty() {
        return head == null;
    }
    public int peek() {
        if (isEmpty()) {
            System.out.println("The Stack is empty!!!
(Will return -1)");
            return -1;
        }
        return head.value;
    }

    public void print() {
        if (isEmpty()) {
            System.out.println("The Stack is
empty!!!");
            return;
        }
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.value + " ");
            temp = temp.next;
        }
        System.out.println();
    }
```

Activity 3:

Conversion of infix to postfix notations using stack

Solution:
Either linked list or array implantation can be used for this purpose. The program follows the following rules.

The infix expression is scanned from left to right until end of the expression. The operands are passed directly to the output. Whereas, the operators are fist passed to the stacks.

2. Whenever an operand is encountered, it is added to the output, i.e. to the postfix expression.
3. Each time an operator is read, the stack is repeatedly popped and operands are passed to the output, until an operator is reached that has a lower precedence than the most recently read operator. The most recently read operator is then pushed onto the stack.
4. When end of the infix expression is reached, all operators-remaining in the stack are popped and passed to the output in the same sequence.
5. Parentheses can be used in the infix expression, but these are not used in the postfix expression. During conversion process, parentheses are treated as operators that have higher precedence than any other operator. The left parenthesis is pushed into the stack when encountered. The right parenthesis is never pushed to the stack. The left parenthesis is popped only when right parenthesis is encountered. The parentheses are not passed to the output postfix 'expressions; they are discarded.
7. When end of expression is reached, then all operators from stack arc popped and added to the output.

Following function has the desired steps discussed above.

```
private static String toPostfix(String infix)
  {
InfixToPostfix operators=new InfixToPostfix();
  char symbol;
  String postfix = "";
  for(int i=0;i<infix.length();++i)
  //while there is input to be read
  {
  symbol = infix.charAt(i);
  //if it's an operand, add it to the string
  if (Character.isLetter(symbol))
  postfix = postfix + symbol;
  else if (symbol=='(')
  //push
  {
  operators.push(symbol);
  }
  else if (symbol==')')
  //push everything back to (
  {
  while (operators.peek() != '(')
  {
  postfix = postfix + operators.pop();
  }
  operators.pop();         //remove '('
  }
  else
  //print operators occurring before it that have
greater precedence
  {
  while (!operators.isEmpty() &&
!(operators.peek()=='(') && prec(symbol) <= prec((char)
operators.peek()))
  postfix = postfix + operators.pop();
  operators.push(symbol);
  }
  }
  while (!operators.isEmpty())
  postfix = postfix + operators.pop();
  return postfix;
  }
```

3) Stage V (verify)

Home Activities:

Write a function that converts a evaluate postfix expression using stack. Use linked list for stack implementation.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 06

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

- Implementation of queue using array
- Implementation of queue using linked list
- Operations on queue
- Implementation of circular queue using array

Instructor Note:Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue.*

Enqueue – place an element at the tail of the queue.

Dequeue – take out an element form the front of the queue.

Delete – delete the whole queue.

1) Stage a1 (apply)

Lab Activities:

Activity 1:

Implementation of queue using array

Solution:

Array implantation of queue is defined with a specific size of the array. Both ends are controlled with front and rear variables. We also need to check full or empty status. Different functions are explained here.

The constructor is used to create queue of a specified size. Initially both ends are set to -1 that indicates that the queue is empty.

```
public LinearQueue_Array(int size) {
        queue = new int[size];
        lastIndex = -1;
        firstIndex = -1;
    }
```

The enqueue and dequeue functions are implemented as follows.

```
public void enQueue(int value) {
        if (lastIndex < queue.length - 1) {
            queue[++lastIndex] = value;
            firstIndex = (firstIndex == -1) ? 0 :
firstIndex;
            return;
        }
        System.out.println("Queue is Full or it may
have filled once but not emptied. To use again, empty
first!");

    }
```

The dequeue and dequeue functions are implemented as follows.

```
public void deQueue() {
        if (firstIndex <= lastIndex) {
            firstIndex++;
        }
        if (firstIndex > lastIndex) {
            firstIndex = lastIndex = -1;
        }
```

Elements of queue can be accessed with the following code

```
public void printQueue() {
        try {
     for (int i = firstIndex; i <= lastIndex; i++) {
          System.out.print(queue[i] + " ");
            }
        } catch (Exception ex) {
  System.out.println("Cannot read deleted Queue!!!");
        }
        System.out.println();
}
```

Other associated functions are as follows.

```
public int peekFirst() {
        if (isEmpty()) {
            return -1;
        }
        return queue[firstIndex];
    }

    public int peekLast() {
        if (isEmpty()) {
            return -1;
        }
        return queue[lastIndex];
    }

    public boolean isEmpty() {
        return firstIndex == -1 || queue == null;
    }

    public boolean isFull() {
        return lastIndex == queue.length - 1 || queue
== null;
    }
    public void deleteQueue() {
        queue = null;

    }
```

Activity 2:

Implementation of queue using linked list

To achieve this goal, we need linked list with the desired variables as discussed in previous labs. The push or enqueue function checks whether the queue is empty initially or contains elements. Here we don't need to check the overflow as the linked list can be extended dynamically. The code works as follows.

```
public void push(int nodeValue) {
        Node node = new Node();
        node.value = nodeValue;
        if (isEmpty()) {
            node.next = null;
            head = tail = node;
            size++;
            return;
        }
        tail.next = node;
        node.next = null;
        tail = node;
        size++;
}
```

The pop/dequeue operation needs to check whether the queue is empty or has elements do dequeue.

```
public void pop() {
        if (isEmpty()) {
         System.out.println("Queue is empty");
         return;
        }
            head = head.next;
            size--;
}
```

The elements of queue can be accessed using the following code.

```
public void printQueue() {
        Node tempNode = head;
        int counter=0;
        while (counter<getCurrentSize()) {
            System.out.print(tempNode.value + " ");
            counter++;
            tempNode = tempNode.next;
        }
}
```

Activity 3:

Implementation of circular queue using array

Some additional checks are used when implementing circular queue using array. Following are supporting functions that are used during enqueue and dequeue operations.

```
public CircularQueue_Array(int size) {
        this.size = size;
        queue = new int[size];
        front = rear = -1;
        System.out.println("The Circular queue is 
created Successfully!");
    }

    // is Empty?
    // The queue will be empty if rear = -1
    public boolean isEmpty(){
        return rear == -1;
    }

    // is Full?
    // Will be full if front = 0 and rear = last 
possible index, i.e; rear = size - 1
    // Also, will be full if rear + 1 = front (if 
circulated)
```

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

The enqueue function works as follows.

```
public void enQueue(int value){
        if(isFull()){
            System.out.println("The queue is Full!");
        } else if(isEmpty()){ // If enqueing for the
first time - we need to increment both front and rear
            front++;
            rear++;
            queue[rear] = value;
        } else {
            if(rear + 1 == size){ // if true, then we
have free location at left of queue, queue is full from
right
                rear = 0;
            } else { // else we are simply moving
towards the right
                rear++;
            }
            queue[rear] = value;
        }


    }
```

The dequeue function works as shown below.

```
public int deQueue(){
        if(isEmpty()){
            System.out.println("The queue is empty at
the moment! will return -1");
            return -1;
        } else {
            int result = queue[front];
            queue[front] = -1; // every -1 in the queue
means that is a free space
            // Below code decides where the front
variable points to
```

```
            if(front == rear){ // It means we are
removing the only element from the queue
                front = rear = -1;
            } else if(front + 1 == size){ // if the
queue has to circualte
                front = 0;
            } else {
                front++;
            }
            return result;
        }
    }
```

2) **Stage V (verify)**

Home Activities:

- Write a program that implements circular queue using linked list
- Write a program that implements double ended queue using linked list

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 07

Statement Purpose:

This lab will introduce you the concept of Binary Trees and Binary Search Trees

Activity Outcomes:

This lab teaches you the following topics:

How to inset elements to binary tree
How to search in binary trees.
How to traverse binary trees.
How to insert and delete data from binary search trees
How to traverse and search the binary search trees
How to delete from binary trees

Instructor Note:

Stage J (Journey)

Introduction

A tree is a Non-Linear Data Structure which consists of set of nodes called vertices and set of edges which links vertices

A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

Recursive definition:

A binary tree is a finite set of nodes that is either empty or consists of a root and maximum of two disjoint binary trees called the left subtree and the right subtree. The binary tree may also have only right or left subtree.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minimum effort

Terminology:

Root Node: The starting node of a tree is called Root node of that tree

Terminal Nodes: The node which has no children is said to be terminal node or leaf.

Non-Terminal Node: The nodes which have children is said to be Non-Terminal

Nodes

Degree: The degree of a node is number of sub trees of that node

Depth: The length of largest path from root to terminals is

said to be depth or height of the tree

Siblings: The children of same parent are said to be siblings.
 Ancestors: The ancestors of a node are all the nodes along the path from the root to the node

Binary Search Trees

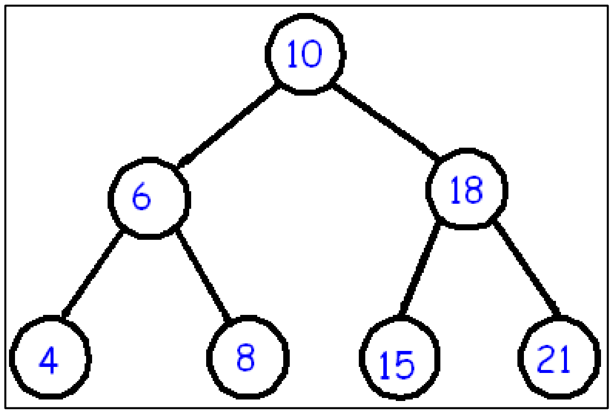
We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.

A BST is a binary tree where nodes are ordered in the following way:

Each node contains three parts (left subtree, right subtree and data part

The data/keys in the left subtree are less then the data/key in its parent node, in short L < P;
The keys in the right subtree are greater the key in its parent node, in short P < R;

In the left tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees are again BST; the above definition is recursively applied to all internal nodes:



An example of binary search tree.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a complete program to insert data into binary tree.

Solution:

To insert data to a binary tree, the recursive procedure is used. Following code demonstrates the insertion process.

```
void insertNode(int value) {
         BinaryNode node = new BinaryNode(value);
         if (root == null) {
              root = node;
              return;
         }
         Queue<BinaryNode> queue = new LinkedList<>();
         queue.add(root);
         while (!(queue.isEmpty())) {
             BinaryNode currentNode = queue.remove();
             if (currentNode.left != null) {
                 queue.add(currentNode.left);
             } else {
                 currentNode.left = node;
                 return;
             }
             if (currentNode.right != null) {
                 queue.add(currentNode.right);
             } else {
                 currentNode.right = node;
                 return;
             }
         }
        }
```

Activity 2:

Write a program to search data in a binary tree.

Solution:

Searching is also carried out using the recursive procedure. For values smaller than root the left side is scanned and for greater values right side is scanned. The following code demonstrates the procedure.

```
void searchNode(int searchValue) {
        Queue<BinaryNode> queue = new LinkedList<>();
        queue.add(root);
        while (!(queue.isEmpty())) {
            BinaryNode currentNode = queue.remove();
            if (currentNode.value == searchValue) {
                System.out.println("The value is found in
Tree!");
                return;
            }
            if (currentNode.left != null) {
                queue.add(currentNode.left);
            }
            if (currentNode.right != null) {
                queue.add(currentNode.right);
            }
        }
```

```
            System.out.println("The value is NOT there in tree!");
        }
```

Activity 3:

Write a program to delete a node from binary tree.

Solution:

Different cases are check when node from a binary tree is deleted. Following code demonstrates the procedure.

```
 void delete(int deleteValue) {
        Queue<BinaryNode> queue = new LinkedList<>();
        queue.add(root);
        BinaryNode nodeToDelete;
        while (!(queue.isEmpty())) {
            BinaryNode currentNode = queue.remove();
            if (currentNode.value == deleteValue) {
                nodeToDelete = currentNode;
                BinaryNode depestNode = getDepestNode();
                nodeToDelete.value = depestNode.value;
                deleteDepestNode(depestNode);
                break;
            }
            if (currentNode.left != null) {
                queue.add(currentNode.left);
            }
            if (currentNode.right != null) {
                queue.add(currentNode.right);
            }
        }
```

Different supporting functions used in the above program are as follows.

```
private BinaryNode getDepestNode() {
        Queue<BinaryNode> queue = new LinkedList<>();
        queue.add(root);
        BinaryNode depestNode = null;
        while (!(queue.isEmpty())) {
            BinaryNode currentNode = queue.remove();
            if (currentNode.left != null) {
                queue.add(currentNode.left);
            }
            if (currentNode.right != null) {
                queue.add(currentNode.right);
            }
            depestNode = currentNode;
        }
        return depestNode;
    }
```

Activity 4:

Write a program for in-order, pre-order and post-order and level order traversal of binary search trees

Solution:

Recursive procedures are used to traverse BST in different orders.

Pre-order

```
private void preOrder(Node node) {
        if (node == null) {
            return;
        }
        System.out.print(node.value + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
```

Post-order

```
private void postOrder(Node node) {
        if (node == null) {
            return;
        }
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.value + " ");
    }
```

In-order

```
private void inOrder(Node node) {
        if (node == null) {
            return;
        }
        inOrder(node.left);
        System.out.print(node.value + " ");
        inOrder(node.right);
        }
```

Level order

```
void printLevelOrder() {
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        while (!(queue.isEmpty())) {
            Node currentNode = queue.remove();
            System.out.print(currentNode.value + " ");
            if (currentNode.left != null) {
                queue.add(currentNode.left);
            }
            if (currentNode.right != null) {
                queue.add(currentNode.right);
            }
        }
        }
```

Activity 5:

Write a complete program to insert data into binary search tree.

Solution:

To insert data to a binary search tree, the recursive procedure is used. Following code demonstrates the insertion process. First root is checked and then other nodes are inserted.

```
void insert(int insertionValue) {
        if (root == null) {
            root = new Node(insertionValue);
            return;
        }
        insert(root, insertionValue);
    }
```

```
private Node insert(Node node, int nodeValue) {
        if (node == null) {
            Node newNode = new Node(nodeValue);
            return newNode;
        } else if (nodeValue > node.value) {
            node.right = insert(node.right, nodeValue);
        } else {
            node.left = insert(node.left, nodeValue);
        }
        return node;
}
```

Activity 6:

Write a program to search data in a binary search tree.

Solution:

Searching is also carried out using the recursive procedure. For values smaller than root the left side is scanned and for greater values right side is scanned. The following code demonstrates the procedure.

```
void search(int value) {
        search(value, root);
    }

    private void search(int value, Node node) {
        if (node == null) {
System.out.println("The searched value was not found in tree!");
            return;
        }
        if (value > node.value) {
            search(value, node.right);
        } else if (value < node.value) {
            search(value, node.left);
        } else {
System.out.println("The search value " + node.value + " is
present in the Tree!");
        }
```

Activity 7:

Write a program to delete a node from binary search tree.

Solution:

Different cases are check when node from a binary tree is deleted. Following code demonstrates the procedure.

```
 void delete(int value) {
        DeleteNode(root, value);
    }

    private Node DeleteNode(Node node, int value) {
        if (node == null) {
            System.out.println("The node to delete was not 
found!");
            return null;
        }
        if (value < node.value) {
            node.left = DeleteNode(node.left, value);
        } else if (value > node.value) {
            node.right = DeleteNode(node.right, value);
        } else {
//          <------- case 1 ------>
            if (node.left != null && node.right != null) {
                Node temp = node;
                Node minNodeFromRight = 
getMinimumNode(temp.right);
                temp.value = minNodeFromRight.value;
                node.right = DeleteNode(node.right, 
minNodeFromRight.value);
            } //          <----- case 2 ----->
            else if (node.left != null) {
                node = node.left;
            } else if (node.right != null) {
                node = node.right;
//          <----- case 3 ----->
            } else {
                node = null;
            }
        }
        return node;
}
```

Different supporting functions used in the above program are as follows.

```
private Node getMinimumNode(Node node) {
        if (node.left == null) {
            return node;
        }
        return getMinimumNode(node.left);
}
```

Activity 8:

Write a program for in-order, pre-order and post-order and level order traversal of binary search trees

Solution:

Recursive procedures are used to traverse BST in different orders.

Pre-order

```
private void preOrder(Node node) {
        if (node == null) {
            return;
        }
        System.out.print(node.value + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
```

Post-order

```
private void postOrder(Node node) {
        if (node == null) {
            return;
        }
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.value + " ");
    }
```

In-order

```
private void inOrder(Node node) {
        if (node == null) {
            return;
        }
        inOrder(node.left);
        System.out.print(node.value + " ");
        inOrder(node.right);
        }
```

Level order

```
void printLevelOrder() {
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        while (!(queue.isEmpty())) {
            Node currentNode = queue.remove();
            System.out.print(currentNode.value + " ");
            if (currentNode.left != null) {
                queue.add(currentNode.left);
            }
            if (currentNode.right != null) {
                queue.add(currentNode.right);
            }
        }
        }
```

3) Stage V (verify)

Home Activities:

- Write a program that implements circular queue using linked list
- Write a program that implements double ended queue using linked list

4) Stage a2 (assess)

Lab Assignment and Viva voce

LAB # 08

Statement Purpose:

This lab will introduce you the concept of AVL trees

Activity Outcomes:

This lab teaches you the following topics:

- How to insert and delete data from AVL trees
- How to search and traverse AVL trees

Instructor Note:

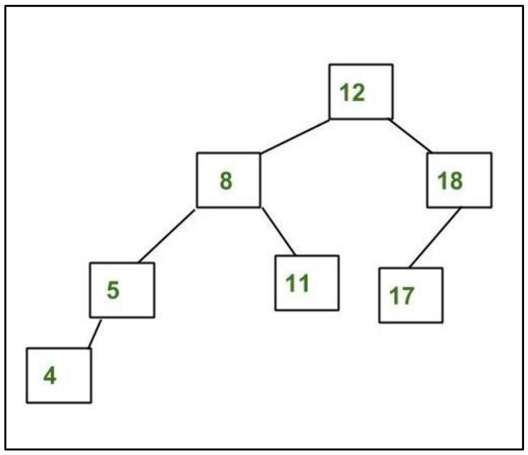
Stage J (Journey)

Introduction

AVL Tree:

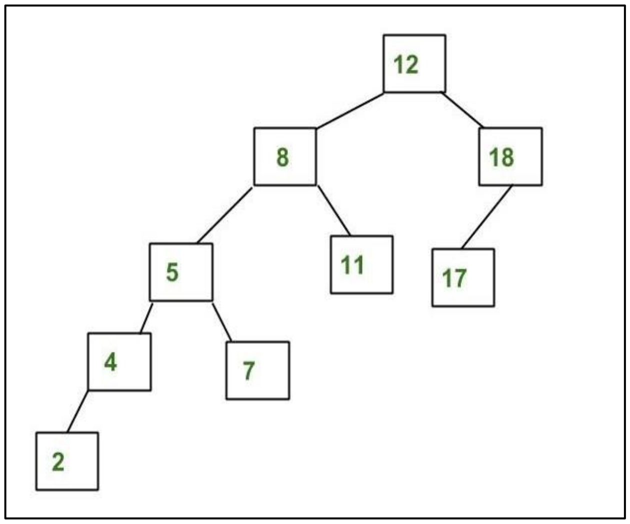
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program to insert a node in AVL tree while balancing.

Solution:

To insert data to AVL tree, the recursive procedure is used like BST. After inserting an element tree balancing is ensured. Following code demonstrates the insertion process. The structure of the program consists of two classes. One is used for tree construction and the other is used for maintaining functions and main.

```
class Node {
int item, height;
Node left, right;
Node(int d) {
 item = d;
 height = 1;
}
}
```

```
public class AVLTree {
Node root;
int height(Node N) {
 if (N == null)
   return 0;
 return N.height;
}
int max(int a, int b) {
 return (a > b) ? a : b;}
```

Following function is used to insert an element to AVL tree.

```
Node insertNode(Node node, int item) {
 if (node == null)
   return (new Node(item));
 if (item < node.item)
   node.left = insertNode(node.left, item);
 else if (item > node.item)
   node.right = insertNode(node.right, item);
 else
   return node;
 node.height = 1 + max(height(node.left), height(node.right));
 int balanceFactor = getBalanceFactor(node);
 if (balanceFactor > 1) {
   if (item < node.left.item) {
     return rightRotate(node);
   } else if (item > node.left.item) {
     node.left = leftRotate(node.left);
     return rightRotate(node);
   }
 }
 if (balanceFactor < -1) {
   if (item > node.right.item) {
     return leftRotate(node);
   } else if (item < node.right.item) {
     node.right = rightRotate(node.right);
     return leftRotate(node);
   }
 }
 return node;
}
```

Following rotations are used for balancing the tree.

```
Node rightRotate(Node y) {
 Node x = y.left;
 Node T2 = x.right;
 x.right = y;
 y.left = T2;
 y.height = max(height(y.left), height(y.right)) + 1;
 x.height = max(height(x.left), height(x.right)) + 1;
 return x;
}
Node leftRotate(Node x) {
 Node y = x.right;
 Node T2 = y.left;
 y.left = x;
 x.right = T2;
 x.height = max(height(x.left), height(x.right)) + 1;
 y.height = max(height(y.left), height(y.right)) + 1;
 return y;
}
```

Following function is used for calculating balancing factor.

```
int getBalanceFactor(Node N) {
 if (N == null)
   return 0;
 return height(N.left) - height(N.right);
}
```

Following function is used to print the tree.

```
private void printTree(Node currPtr, String indent, boolean
last) {
 if (currPtr != null) {
   System.out.print(indent);
   if (last) {
     System.out.print("R----");
     indent += "   ";
   } else {
     System.out.print("L----");
     indent += "|  ";
   }
   System.out.println(currPtr.item);
   printTree(currPtr.left, indent, false);
   printTree(currPtr.right, indent, true);
 }
}
```

Following is the main function to execute the program.

```
public static void main(String[] args) {

 AVLTree tree = new AVLTree();
 tree.root = tree.insertNode(tree.root, 5);
 tree.root = tree.insertNode(tree.root, 7);
 tree.root = tree.insertNode(tree.root, 6);
 tree.root = tree.insertNode(tree.root, 9);
 tree.root = tree.insertNode(tree.root, 21);
 tree.root = tree.insertNode(tree.root, 61);
 tree.root = tree.insertNode(tree.root, 8);
 tree.root = tree.insertNode(tree.root, 11);
 tree.root = tree.insertNode(tree.root, 82);
 tree.printTree(tree.root, "  ", true);
```

Activity 2:

Write a program to delete a node from AVL tree while balancing.

Solution:

Following function is used to delete an element from AVL tree. After deleting the node balancing is ensured. This function is the part of the program discussed in activity 1. The rotation functions are used here as well.

```
Node deleteNode(Node root, int item) {
 if (root == null)
   return root;
 if (item < root.item)
   root.left = deleteNode(root.left, item);
 else if (item > root.item)
   root.right = deleteNode(root.right, item);
 else {
   if ((root.left == null) || (root.right == null)) {
     Node temp = null;
     if (temp == root.left)
       temp = root.right;
     else
       temp = root.left;
     if (temp == null) {
       temp = root;
       root = null;
     } else
       root = temp;
   } else {
     Node temp = nodeWithMimumValue(root.right);
```

```
      root.item = temp.item;
      root.right = deleteNode(root.right, temp.item);
    }
  }
  if (root == null)
    return root;
  root.height = max(height(root.left), height(root.right)) + 1;
  int balanceFactor = getBalanceFactor(root);
  if (balanceFactor > 1) {
    if (getBalanceFactor(root.left) >= 0) {
      return rightRotate(root);
    } else {
      root.left = leftRotate(root.left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root.right) <= 0) {
      return leftRotate(root);
    } else {
      root.right = rightRotate(root.right);
      return leftRotate(root);
    }
  }
  return root; }
```

Following supported function is also used.

```
Node nodeWithMimumValue(Node node) {
  Node current = node;
  while (current.left != null)
    current = current.left;
  return current; }
```

4) Stage V (verify)

Home Activities:

- Search the minimum number in AVL tree
- Search the maximum number in the AVL tree
- Search the minimum subtree in the AVL tree

5) Stage a2 (assess)

6) Lab Assignment and Viva voce

LAB # 09

Statement Purpose:

This lab will introduce you the concept of Heap data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to implement heap
- How to insert and delete data from heap
- How to find minimum value

Instructor Note:

Stage J (Journey)

Introduction

Why Heaps?

Queues are a standard mechanism for ordering tasks on a first-come, first-served basis. In many situations, simple queues are inadequate, since first in/first out scheduling has to be overruled using some priority criteria like:

- In a sequence of processes, process P_2 may need to be executed before process P_1 for the proper functioning of a system, even though P_1 was put on the queue of waiting processes before P_2.
- Banks managing customer information often will remove or get the information about the customer with the minimum bank account balance.
- In shared printer queue list of print jobs, must get next job with highest *priority*, and higher-priority jobs always print before lower-priority jobs.

In situations like these, a modified queue or *priority queue*, is needed in which elements are dequeued according to their priority and their current queue positions. The problem with a priority queue is in finding an efficient implementation which allows relatively fast enqueuing and dequeuing. Since elements may arrive randomly to the queue, there is no guarantee that the front elements will be the most likely to be dequeued and that the elements put at the end will be the last candidates for dequeuing. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases.

Consider an example of a printer; if the three jobs have been submitted to print, the jobs have sizes 100, 10, 1 page. The average waiting time for FIFO service, (100+110+111)/3 = 107 time units and average waiting time for shortest job first service, (1+11+112)/3 = 41 time units.

So a heap is an excellent way to implement a priority queue as **binary heap** is a complete or almost complete **binary tree** which satisfies the **heap** ordering property. The ordering can be one of two types: the min-**heap** property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root and max-heap each node is smaller than its parent node.

Heaps

Heap has the following properties:

- The value of each node is not less than the values stored in each of its children
- The tree is an almost complete binary tree.

These two properties define a ***max heap*** and if ―less‖ in the first property is replaced with ―greater‖; then the definition specifies a ***min heap.***
Heap is generally preferred for priority queue implementation by array list because it provide better performance as a method **getHighestPriority()** to access the element with highest priority can be implemented in **O(1)** time, method to **insert()** new element can be implemented in **O(Logn)** time and **deleteHighestPriority()** can also be implemented in **O(Logn)** time.

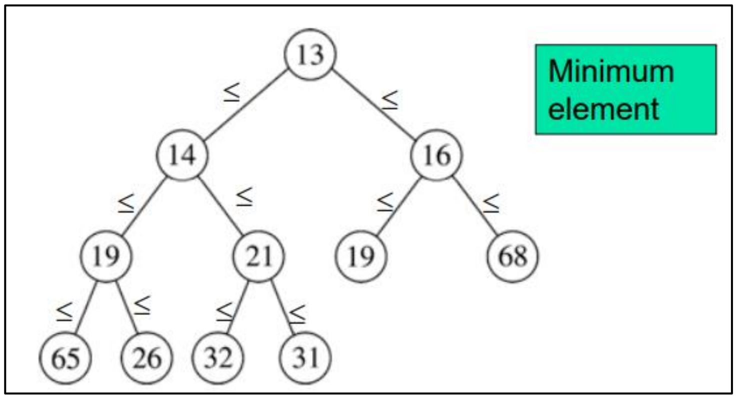
Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra’s shortest path algorithm, Prim’s Minimum Spanning Tree, etc
3) All queue applications where priority is involved

A Binary Heap is a Binary Tree with following properties:
1) It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

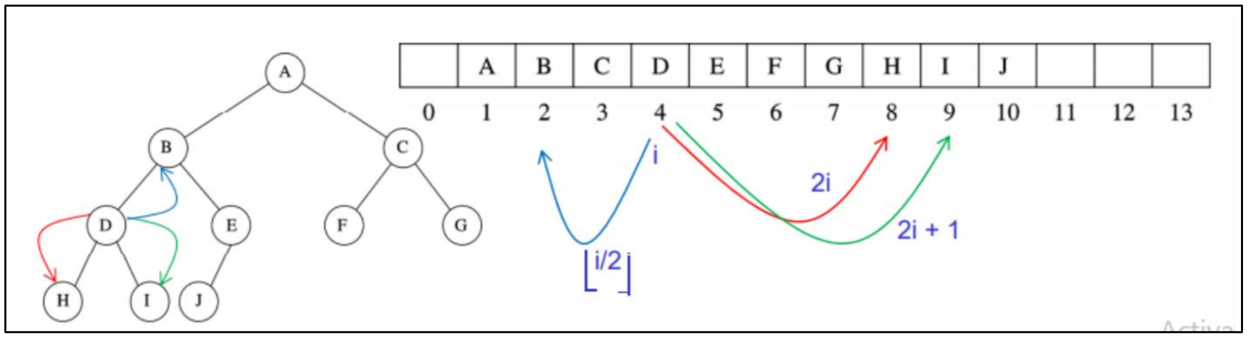
Examples of Min-Heap



How is Binary Heap represented?

Given element at position i in the array:

- Left child (i) = at position 2i
- Right child(i) = at position 2i + 1
- Parent(i) = at position ⌊ i / 2 ⌋



2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program to insert and delete the data from a heap while maintaining its properties.

Solution:

The structure of the class consists of different members used.

```
class Heap {
    // declare array and variables
private int[] heapData;
private int sizeOfHeap;
private int heapMaxSize;
private static final int FRONT = 1;
```

Use constructor to initialize heapData array

```
public Heap(int heapMaxSize)  {
this.heapMaxSize = heapMaxSize;
this.sizeOfHeap = 0;
heapData = new int[this.heapMaxSize + 1];
heapData[0] = Integer.MIN_VALUE;
}
```

Create getParentPos() method that returns parent position for the node

```
private int getParentPosition(int position)  {
return position / 2;
 }
```

Create getLeftChildPosition() method that returns the position of left child

```
private int getLeftChildPosition(int position)  {
return (2 * position);
}
```

Create getRightChildPosition() method that returns the position of right child

```
private int getRightChildPosition(int position)  {
return (2 * position) + 1;
}
```

Check whether the given node is leaf or not

```
private boolean checkLeaf(int position)  {
if (position >= (sizeOfHeap / 2) && position <= sizeOfHeap) {
return true;
        }
return false;
 }
```

Create swapNodes() method that perform swapping of the given nodes of the heap

```
private void swap(int firstNode, int secondNode)   {
int temp;
temp = heapData[firstNode];
heapData[firstNode] = heapData[secondNode];
heapData[secondNode] = temp;
}
```

Create minHeapify() method to heapify the node for maintaining the heap property

```
private void minHeapify(int position)   {

        //check whether the given node is non-leaf and greater
than its right and left child
if (!checkLeaf(position)) {
if (heapData[position] >heapData[getLeftChildPosition(position)]
|| heapData[position]
>heapData[getRightChildPosition(position)]) {

                // swap with left child and then heapify the
left child
if (heapData[getLeftChildPosition(position)]
<heapData[getRightChildPosition(position)]) {
swap(position, getLeftChildPosition(position));
minHeapify(getLeftChildPosition(position));
                }

                // Swap with the right child and heapify the
right child
else {
swap(position, getRightChildPosition(position));
minHeapify(getRightChildPosition(position));
                }
            }
}
}
```

Following function demonstrates the insertion into a heap.

```
public void insertNode(int data)   {
if (sizeOfHeap>= heapMaxSize) {
return;
}
heapData[++sizeOfHeap] = data;
int current = sizeOfHeap;

while (heapData[current] <heapData[getParentPosition(current)])
{
swap(current, getParentPosition(current));
current = getParentPosition(current);
 }
}
```

Create displayHeap() method to print the data of the heap

```
public void displayHeap()  {
System.out.println("PARENT NODE" + "\t" + "LEFT CHILD NODE" +
"\t" + "RIGHT CHILD NODE");
for (int k = 1; k <= sizeOfHeap / 2; k++) {
System.out.print(" " + heapData[k] + "\t\t" + heapData[2 * k] +
"\t\t" + heapData[2 * k + 1]);
System.out.println();
        }
 }
```

Create designMinHeap() method to construct min heap

```
public void designMinHeap()  {
for (int position = (sizeOfHeap / 2); position >= 1; position--)
{
minHeapify(position);
        }
 }
```

Create removeRoot() method for removing minimum element from the heap

```
public int removeRoot()  {
int popElement = heapData[FRONT];
heapData[FRONT] = heapData[sizeOfHeap--];
minHeapify(FRONT);
return popElement;
    }
}
```

Create MinHeapJavaImplementation class to create heap

```
public class MinHeap{
public static void main(String[] arg)  {
    int heapSize;
    Scanner sc = new Scanner(System.in);

    System.out.println("Enter the size of Min Heap");
    heapSize = sc.nextInt();
    Heap heapObj = new Heap(heapSize);
    for(int i = 1; i<= heapSize; i++) {
        System.out.print("Enter "+i+" element: ");
        int data = sc.nextInt();
        heapObj.insertNode(data);
    }
sc.close();
heapObj.designMinHeap();
System.out.println("The Min Heap is ");
heapObj.displayHeap();
System.out.println("After removing the minimum element(Root
Node) "+heapObj.removeRoot()+", Min heap is:");
heapObj.displayHeap();

    }
 }
```

1) **Stage V (verify)**

Home Activities:

Revise the heap definition of above class to implement a max-heap. The member function **removemin** should be replaced by a new function called **removemax**.

2) **Stage a2 (assess)**

3) **Lab Assignment and Viva voce**

LAB # 10

Statement Purpose:

This lab will introduce you the concept of graphs

Activity Outcomes:

This lab teaches you the following topics:
Implementation of graphs using adjacency matrix
Implementation of graphs using adjacency list
Traversing graphs using Breadth First and Depth First Search

Instructor Note:

Stage J (Journey)

Introduction

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Implementation of graphs using adjacency matrix with Breadth First Search and Depth First Search.

Solution:

This implementation will use two-dimensional array to create adjacency matrix. Following code demonstrates the class and other related functions involved.

```
public class GraphAdjacencyMatrix{
static class Graph
{
     int v;
```

```
        int e;
        int[][] adj;
        Graph(int v, int e)
        {
                this.v = v;
                this.e = e;
                adj = new int[v][v];
                for(int row = 0; row < v; row++)
                        Arrays.fill(adj[row], 0);
        }
```

```
void addEdge(int start, int e)
        {
                adj[start][e] = 1;
                adj[e][start] = 1;
        }
```

```
        void BFS(int start)
        {
                boolean[] visited = new boolean[v];
                Arrays.fill(visited, false);
                List<Integer> q = new ArrayList<>();
                q.add(start);
                visited[start] = true;
                int vis;
                while (!q.isEmpty())
                {
                        vis = q.get(0);
                        System.out.print(vis + " ");
                        q.remove(q.get(0));
                        for(int i = 0; i < v; i++)
                        {
                                if (adj[vis][i] == 1 && (!visited[i]))
                                {
                                        q.add(i);
                                        visited[i] = true;
                                }
                        }
                }
        }
```

```
void DFS(int start, boolean[] visited)
        {
            System.out.print(start + " ");
            visited[start] = true;
            for (int i = 0; i < adj[start].length; i++) {
                if (adj[start][i] == 1 && (!visited[i])) {
                    DFS(i, visited);
                }
            }
        }
}
```

```
public static void main(String[] args)
{
      int v = 5, e = 4;
      Graph G = new Graph(v, e);
      G.addEdge(0, 1);
      G.addEdge(0, 2);
      G.addEdge(1, 3);
      G.BFS(0);
      boolean[] visited = new boolean[v];
      System.out.println();
      G.DFS(0, visited);
}
}
```

Activity 2:

Implementation of graphs using adjacency list with Breadth First and Depth First Search

Solution:

Following procedures are involved in the implementation.

```
public class Graph {
      private int V; // No. of vertices
      private LinkedList<Integer> adj[];
      Graph(int v)
      {
            V = v;
            adj = new LinkedList[v];
            for (int i = 0; i < v; ++i)
                  adj[i] = new LinkedList();
      }
```

```
void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }
```

```
void DFSUtil(int v, boolean visited[])
    {
        visited[v] = true;
        System.out.print(v + " ");
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    void DFS(int v)
    {
        boolean visited[] = new boolean[V];
        DFSUtil(v, visited);
    }
```

```
void BFS(int s)
    {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<Integer>();
        visited[s]=true;
        queue.add(s);
        while (queue.size() != 0)
        {
            s = queue.poll();
            System.out.print(s+" ");
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
    public static void main(String args[])
    {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        System.out.println("Following is DFS");
        g.DFS(2);
        System.out.println("Following is BFS");
        g.BFS(2);
    }
}
```

1) **Stage V (verify)**

Home Activities:

2) **Stage a_2 (assess)**
3) **Lab Assignment and Viva voce**

LAB # 11

Statement Purpose:

This lab will introduce you with the algorithms for Minimum Spanning Tree and Single Source Shortest path.

Activity Outcomes:

This lab teaches you the following topics:

Kruskal Algorithm
Prims Algorithm
Dijkstra’s Algorithm

Instructor Note:

Stage J (Journey)

Introduction

A tree is a Non-Linear Data Structure which consists of set of nodes called vertices and set of edges which links vertices

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a complete program to implement Kruskal Algorithm.

Solution:

Following code demonstrates the Kruskal Algorithm.

```
public class Kruscal {
     class Edge implements Comparable<Edge>
     {
          int src, dest, weight;
          public int compareTo(Edge compareEdge)
          {
               return this.weight - compareEdge.weight;
          }
     };
```

```
class subset
     {
          int parent, rank;
     };
```

```
     int V, E; // V-> no. of vertices & E->no.of edges
     Edge edge[]; // collection of all edges
     Kruscal(int v, int e)
     {
          V = v;
          E = e;
          edge = new Edge[E];
          for (int i = 0; i < e; ++i)
               edge[i] = new Edge();
     }
```

```
int find(subset subsets[], int i)
     {
          if (subsets[i].parent != i)
               subsets[i].parent
                    = find(subsets, subsets[i].parent);

          return subsets[i].parent;
     }
```

```
void Union(subset subsets[], int x, int y)
     {
          int xroot = find(subsets, x);
          int yroot = find(subsets, y);
          if (subsets[xroot].rank
               < subsets[yroot].rank)
               subsets[xroot].parent = yroot;
          else if (subsets[xroot].rank
                    > subsets[yroot].rank)
               subsets[yroot].parent = xroot;
          else {
               subsets[yroot].parent = xroot;
               subsets[xroot].rank++;
          }
     }
```

```
void KruskalMST()
     {
          Edge result[] = new Edge[V];
          int e = 0;
          int i = 0;
          for (i = 0; i < V; ++i)
               result[i] = new Edge();
          Arrays.sort(edge);
          subset subsets[] = new subset[V];
          for (i = 0; i < V; ++i)
               subsets[i] = new subset();
          for (int v = 0; v < V; ++v)
          {
               subsets[v].parent = v;
               subsets[v].rank = 0;
          }

          i = 0; // Index used to pick next edge

          // Number of edges to be taken is equal to V-1
          while (e < V - 1)
          {
               Edge next_edge = edge[i++];
               int x = find(subsets, next_edge.src);
               int y = find(subsets, next_edge.dest);
               if (x != y) {
                    result[e++] = next_edge;
                    Union(subsets, x, y);
               }
          }

          // print the contents of result[] to display
          // the built MST
          System.out.println("Following are the edges in "
                             + "the constructed MST");
          int minimumCost = 0;
          for (i = 0; i < e; ++i)
          {
               System.out.println(result[i].src + " -- "
                                  + result[i].dest
                                  + " == " + result[i].weight);
               minimumCost += result[i].weight;
          }
          System.out.println("Minimum Cost Spanning Tree "
                             + minimumCost);
     }
```

```
public static void main(String[] args)
     {
          int V = 4; // Number of vertices in graph
          int E = 5; // Number of edges in graph
          Kruscal graph = new Kruscal(V, E);
          graph.edge[0].src = 0;
          graph.edge[0].dest = 1;
          graph.edge[0].weight = 10;
          graph.edge[1].src = 0;
          graph.edge[1].dest = 2;
          graph.edge[1].weight = 6;
          graph.edge[2].src = 0;
          graph.edge[2].dest = 3;
          graph.edge[2].weight = 5;
          graph.edge[3].src = 1;
          graph.edge[3].dest = 3;
          graph.edge[3].weight = 15;
          graph.edge[4].src = 2;
          graph.edge[4].dest = 3;
          graph.edge[4].weight = 4;
          graph.KruskalMST();
     }
}
```

Activity 2:

Write a complete program to implement Prim's Algorithm.

Solution:

Following code demonstrates the Prims Algorithm.

```
public class MSTPrims {
     private static final int V = 5;
     int minKey(int key[], Boolean mstSet[])
     {
          int min = Integer.MAX_VALUE, min_index = -1;
          for (int v = 0; v < V; v++)
               if (mstSet[v] == false && key[v] < min) {
                    min = key[v];
                    min_index = v;
               }
          return min_index;
     }
     void printMST(int parent[], int graph[][])
     {
          System.out.println("Edge \tWeight");
          for (int i = 1; i < V; i++)
               System.out.println(parent[i] + " - " + i + "\t" +
graph[i][parent[i]]);
     }
```

```
void primMST(int graph[][])
     {
          int parent[] = new int[V];
          int key[] = new int[V];
          Boolean mstSet[] = new Boolean[V];
          for (int i = 0; i < V; i++) {
               key[i] = Integer.MAX_VALUE;
               mstSet[i] = false;
          }
          parent[0] = -1;
          for (int count = 0; count < V - 1; count++) {
               int u = minKey(key, mstSet);
               mstSet[u] = true;
               for (int v = 0; v < V; v++)
                    if (graph[u][v] != 0 && mstSet[v] == false
&& graph[u][v] < key[v]) {
                         parent[v] = u;
                         key[v] = graph[u][v];
                    }
          }
          printMST(parent, graph);
     }
```

```
public static void main(String[] args)
     {
          MSTPrims t = new MSTPrims();
          int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
                                        { 2, 0, 3, 8, 5 },
                                        { 0, 3, 0, 0, 7 },
                                        { 6, 8, 0, 0, 9 },
                                        { 0, 5, 7, 9, 0 }
};

          t.primMST(graph);
     }
}
```

Activity 3:

Write a complete program to implement Dijkstra's Algorithm.

Solution:

Following code demonstrates the Dijkstra's Algorithm..

```
public class Dijkstra {
     private int dist[];
     private Set<Integer> settled;
     private PriorityQueue<Node> pq;
     private int V;
     List<List<Node> > adj;
     public Dijkstra(int V)
     {
          this.V = V;
          dist = new int[V];
          settled = new HashSet<Integer>();
          pq = new PriorityQueue<Node>(V, new Node());
     }
```

```
public void dijkstra(List<List<Node> > adj, int src)
     {
          this.adj = adj;
          for (int i = 0; i < V; i++)
          dist[i] = Integer.MAX_VALUE;
          pq.add(new Node(src, 0));
          dist[src] = 0;
          while (settled.size() != V) {
               if (pq.isEmpty())
                    return;
               int u = pq.remove().node;
               if (settled.contains(u))
                    continue;
               settled.add(u);
               e_Neighbours(u);
          }
     }
```

```
private void e_Neighbours(int u)
     {
          int edgeDistance = -1;
          int newDistance = -1;
          for (int i = 0; i < adj.get(u).size(); i++) {
               Node v = adj.get(u).get(i);
               if (!settled.contains(v.node)) {
                    edgeDistance = v.cost;
                    newDistance = dist[u] + edgeDistance;
                    if (newDistance < dist[v.node])
                         dist[v.node] = newDistance;
                    pq.add(new Node(v.node, dist[v.node]));
               }
          }
     }
```

```
public static void main(String arg[])
     {

          int V = 5;
          int source = 0;
          List<List<Node> > adj = new ArrayList<List<Node> >();
          for (int i = 0; i < V; i++) {
               List<Node> item = new ArrayList<Node>();
               adj.add(item);
          }
          adj.get(0).add(new Node(1, 9));
          adj.get(0).add(new Node(2, 6));
          adj.get(0).add(new Node(3, 5));
          adj.get(0).add(new Node(4, 3));
          adj.get(2).add(new Node(1, 2));
          adj.get(2).add(new Node(3, 4));
          Dijkstra dpq = new Dijkstra(V);
          dpq.dijkstra(adj, source);
          System.out.println("The shorted path from node :");

          for (int i = 0; i < dpq.dist.length; i++)
               System.out.println(source + " to " + i + " is "
                                   + dpq.dist[i]);
     }
}
```

```
class Node implements Comparator<Node> {
     public int node;
     public int cost;
     public Node() {}
     public Node(int node, int cost)
     {
          this.node = node;
          this.cost = cost;
     }

     @Override public int compare(Node node1, Node node2)
```

```
        {

            if (node1.cost < node2.cost)
                return -1;

            if (node1.cost > node2.cost)
                return 1;

            return 0;
        }
}
```

1) **Stage V (verify)**

Home Activities:

2) **Stage a_2 (assess)**
3) **Lab Assignment and Viva voce**

LAB # 12

Statement Purpose:

This lab will introduce you the concept of Hashing

Activity Outcomes:

This lab teaches you the following topics:

- How to implement hash function
- How to create hash tables
- How to insert, search and retrieve values from hash tables

Instructor Note:

Stage J (Journey)

Introduction

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given key to a smaller number and uses the small number as index in a table called hash table.

Example:

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

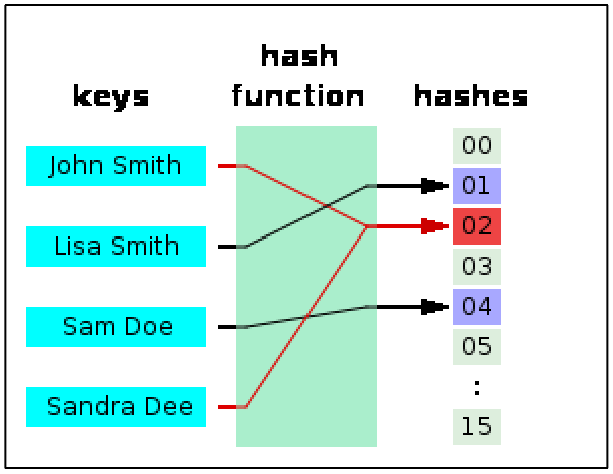
With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table. This solution has many practical limitations. First problem with this solution is extra space
required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hash Function:

A function that converts a given big number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table. For example, a person's name, having a variable length, could be hashed to a single integer. The values returned by a hash function are called hash values, hash codes, hash sums, checksums or simply hashes as shown in figure below:



A good hash function should have following properties

1) Efficiently computable

2) Should uniformly distribute the keys (Each table position equally likely for each key)

<u>Hash Table</u>:

An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table. ▪ **Open Addressing:** In open addressing, all elements are stored in the hash table itself.

  Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a function to compute a simple hash function.

Solution:

To insert data to a binary tree, the recursive procedure is used. Following code demonstrates the insertion process.

Activity 2:

Write a function to insert an item in a hash table.

Solution:

To insert data to a binary tree, the recursive procedure is used. Following code demonstrates the insertion process.

Activity 3:

Write a function to search an item from hash table.

Solution:

To insert data to a binary tree, the recursive procedure is used. Following code demonstrates the insertion process.

Activity 4:

Write a complete program to create a hash table using simple division hash function. Also perform searching and deletion in hash table.

Solution:

To insert data to a binary tree, the recursive procedure is used. Following code demonstrates the insertion process.

1) Stage V (verify)

Home Activities:

Consider –key mod 7‖ as a hash function and a sequence of keys as 50, 700, 76, 85, 92, 73, 101. Create hash table for to store these keys using linear probing.

2) Use mid square hash function to store the data of employees of a company.

Employee ids are:
4176, 5678, 5469, 1245, 8907

3) Apply folding method to store the above data.

2) Stage a_2 (assess)

3) Lab Assignment and Viva voce

Consider the data with keys: 24, 42, 34, 62, 73. Store this data into a hash table of size 10 using quadratic probing to avoid collision.

2) Store the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in an empty hash table of size **13 using double hashing.** Use h(key) = key % 13 as a first hash function while the second hash function is: h_p(key) = 1 + key % 12

LAB # 13

Statement Purpose:

This lab will introduce you the concept of Sorting Algorithms

Activity Outcomes:

This lab teaches you the following topics:

Bubble Sort
Insertion Sort.
Selection Sort
Merge Sort
How to traverse and search the binary search trees
How to delete from binary trees

Instructor Note:

Stage J (Journey)

Introduction

Bubble Sort

Bubble sort is a simple and well-known sorting algorithm. It is used in practice once in a blue moon and its main application is to make an introduction to the sorting algorithms. Bubble sort belongs to O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is stable and adaptive.

Algorithm

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

Insertion Sort

Insertion sort belongs to the O(n2) sorting algorithms. Unlike many sorting algorithms with quadratic complexity, it is actually applied in practice for sorting small arrays of data. For instance, it is used to improve quicksort routine. Some sources notice, that people use same algorithm ordering items, for example, hand of cards.

Algorithm

Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains **first element** of the array and unsorted one contains the rest. At every step, algorithm takes **first element** in the unsorted part and **inserts** it to the right place of the sorted one. When unsorted part becomes **empty**, algorithm *stops*.

Selection Sort

Selection sort is one of the O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes. Selection sort is notable for its programming simplicity and it can over perform other sorts in certain situations (see complexity analysis for more details).

Algorithm

The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is **empty**, while unsorted one contains **whole array**. *At every step*, algorithm finds **minimal element** in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes **empty**, algorithm *stops*.

When algorithm sorts an array, it swaps first element of unsorted part with minimal element and then it is included to the sorted part. This implementation of selection sort in **not stable**In case of linked list is sorted, and, instead of swaps, minimal element is linked to the unsorted part, selection sort is **stable**.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a complete program to bubble sort.

Solution:

The following code demonstrates bubble sort.

```
public static int[] bubbleSort(int[] list) {
        for (int i = 0; i < list.length; i++) {
            for (int j = 0; j < list.length - 1; j++) {
                if (list[j] > list[j + 1]) {
                    int temp = list[j];
                    list[j] = list[j + 1];
                    list[j + 1] = temp;
                }
            }
        }
        return list;
}
```

```
private static int partition(int[] list, int start, int end) {
        int pivot = end;
        int i = start - 1;
        for (int j = start; j <= end; j++) {
            if (list[j] <= list[pivot]) {
                i++;
                int temp = list[i];
                list[i] = list[j];
                list[j] = temp;
            }
        }
        return i;
}
```

Activity 2:

Write a complete program to insertion sort.

Solution:

The following code demonstrates insertion sort.

```
public static int[] insertionSort(int[] list) {
        for (int i = 0; i < list.length; i++) {
            int temp = list[i];
            int j = i - 1;
            while (j >= 0 && list[j] > temp) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = temp;
        }
        return list;
}
```

Activity 3:

Write a complete program to selection sort.

Solution:

The following code demonstrates selection sort.

```
public static int[] slectionSort(int[] list) {
        for (int i = 0; i < list.length; i++) {
            int SNI = i; // Supposition : SNI => smallest number
index
            for (int j = i + 1; j < list.length; j++) {
                if (list[j] < list[SNI]) {
                    SNI = j;
                }
            }
            int temp = list[SNI];
            list[SNI] = list[i];
            list[i] = temp;
        }
        return list;
}
```

Activity 4:

Write a complete program to merge sort.

Solution:

The following code demonstrates merge sort.

```
public static void mergeSort(int[] list, int left, int right) {
        if (right > left) {
            int mid = (left + right) / 2;
            mergeSort(list, left, mid);
            mergeSort(list, mid + 1, right);
            merge(list, left, mid, right);
        }
    }
```

```
 private static void merge(int[] list, int left, int mid, int
right) {
        int[] leftTempArr = new int[mid - left + 2];
        int[] rightTempArr = new int[right - mid + 1];
        for (int i = 0; i <= mid - left; i++) {
            leftTempArr[i] = list[left + i];
        }
        for (int i = 0; i < right - mid; i++) {
            rightTempArr[i] = list[(mid + 1) + i];
        }

        leftTempArr[mid - left + 1] = Integer.MAX_VALUE;
        rightTempArr[right - mid] = Integer.MAX_VALUE;
        int i = 0, j = 0;
        for (int k = left; k <= right; k++) {
            if (leftTempArr[i] < rightTempArr[j]) {
                list[k] = leftTempArr[i];
                i++;
            } else {
                list[k] = rightTempArr[j];
                j++;
            }
        }
    }
```

1) **Stage V (verify)**

Home Activities:

2) **Stage a_2 (assess)**
3) **Lab Assignment and Viva voce**

LAB # 14

Statement Purpose:

This lab will introduce you the concept of searching algorithms

Activity Outcomes:

This lab teaches you the following topics:

Linear Search
Binary Search

Instructor Note:

Stage J (Journey)

Introduction

A tree is a Non-Linear Data Structure which consists of set of nodes called vertices and set of edges which links vertices

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a complete program to implement linear search algorithm.

Solution:

Following code demonstrates linear search algorithm.

```
public static void linearSearch(int[] list, int searchValue) {
        for (int i = 0; i < list.length; i++) {
            if (list[i] == searchValue) {
                System.out.println("The number is found on index
: " + i);
                return;
            }
        }
        System.out.println("The number is not found in the 
list!");
}
```

Activity 2:

Write a complete program to implement binary search algorithm.

Solution:

Following code demonstrates binary search algorithm.

```
public static void binarySearch(int[] list, int searchValue) {
        SortAlgorithm.slectionSort(list);
        int start = 0, end = list.length - 1;
        int mid = end / 2;
        while (start != end) {
            if (list[mid] == searchValue) {
                System.out.println("The number is found on index 
: " + mid);
                return;
            } else {
                if (searchValue > list[mid]) {
                    start = mid;
                } else {
                    end = mid;
                }
                mid = (end + start) / 2;
            }
        }
}
```

1) **Stage V (verify)**

Home Activities:

2) **Stage a2 (assess)**
3) **Lab Assignment and Viva voce**