

Data Structure and Algorithms

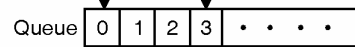
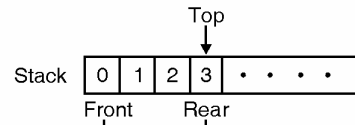
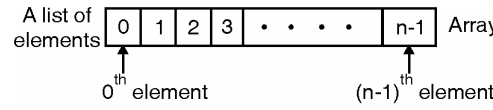
Q .1 a. *What are linear and non linear data structure?*

(20)

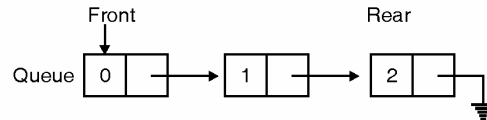
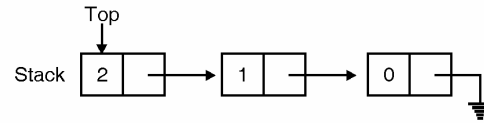
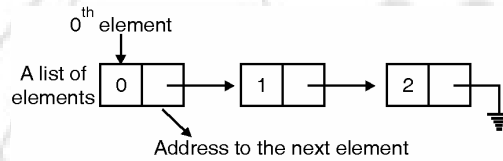
Ans: Linear and Non-Linear Data Structure :

Linear :

Elements are arranged in a linear fashion (one dimension). All one-one relation can be Handled through linear data structures. Lists, stacks and queues are examples of linear data structure. Fig.shows the representation of linear data structure.



Representation of Linear data structures in an Array

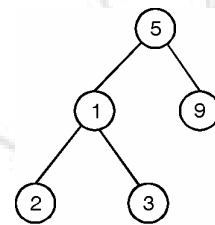


(b) Representation of Linear data structures through Linked structure

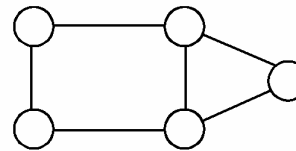
Fig. : Representation of Linear data structures

Non-Linear :

All one-many, many-many or many-many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and table are examples of non-linear data structures (Refer Fig.). Representation of binary tree in linked and array structure is given in Fig.



Tree (a)



Graph (b)

5	9	1
6	4	13
2	5	0
9	6	11

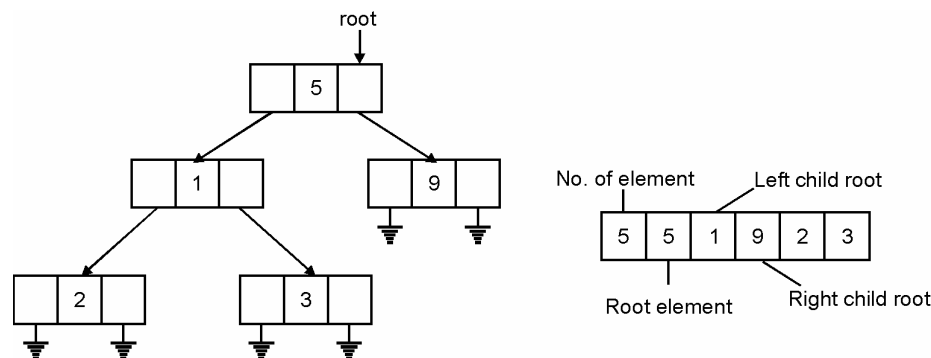
Table (c)

Fig. : Non-Linear data structure

(a) Representation of the binary tree through linked structure

(b) Representation of binary tree through an array

Data Structure and Algorithms



Q.1 b. **Why is it necessary to analyze an algorithm?**

Ans: when programmer builds an algorithm during design phase of software development life cycle, he/she might not be able to implement it immediately. This is because programming comes in later part of software development life cycle. There is a need to analyze the algorithm at the stage. This will help in forecasting time of execution and amount of primary memory algorithm might occupy when it is implemented.

Analysis of algorithm means developing a formula or prediction of how fast the algorithm works based on problem size.

Problem size could be :

- i) The numbers of inputs /outputs in an algorithm.
- ii) The numbers of operations involved in the algorithm.

Q.1. c. **What are Asymptotic notation?**

Ans:

There are three types of asymptotic notations:

O notation: explanation with graph

θ notation: explanation with graph

Ω notation: explanation with graph

Q.1 d. **What is an Abstract data type?**

Ans:

Definition: A set of data values and associated operations that are precisely specified independent of any particular implementation.

One of the simplest abstract data types is the stack. The operations $new()$, $push(v, S)$, $top(S)$, and $popOff(S)$ may be defined with axiomatic semantics as following.

1. $new()$ returns a stack
2. $popOff(push(v, S)) = S$
 $top(push(v, S)) = v$

QUEUE:

Definition: A collection of items in which only the earliest added item may be accessed. Basic operations are add (to the tail) or enqueue and delete (from the head) or dequeue. Delete returns the item removed. Also known as "first-in, first-out" or FIFO.

Formal Definition: It is convenient to define delete or dequeue in terms of remove

Data Structure and Algorithms

and a new operation, front. The operations new(), add(v, Q), front(Q), and remove(Q) may be defined with *axiomatic semantics* as follows.

1. new() returns a queue
 2. front(add(v, new())) = v
 3. remove(add(v, new())) = new()
 4. front(add(v, add(w, Q))) = front(add(w, Q))
 5. remove(add(v, add(w, Q))) = add(v, remove(add(w, Q)))
- where Q is a queue and v and w are values.

Also known as FIFO.

Q.2.a. Explain how a stack can be used to execute computation of mathematical expressions.

For example consider the expression $(a+b)*(b-c)$.

(10)

Ans: Algorithm for “Infix(with parenthesis) to Postfix Conversion” :

- Step 1 :** opstk = the empty stack;
- Step 2 :** While(not end of input) repeat following steps
- Step 2.1 :** symb = next input character
- Step 2.2 :** if(symb is an operand) then
Add symb to postfix string
Else do following steps
- Step 2.2.1 :** while(!empty(opstk) && prcd(stacktop(opstk),symb))
Repeat following instructions
Topsymb = pop(opstk)
Add topsymb to the postfix string
- Step 2.2.2 :** if(empty(opstk) | |symb !=')')
push (opstk, symb);
else
topsymb = pop(opstk);
- Step 3 :** while(!empty(opstk)) repeat following instructions
- Step 3.1 :** Topsymb = pop(opstk);
- Step 3.2 :** Add topsymb to the postfix string
- Step 4 :** Stop

Q .2.b. Write the program to create single link list and display the list.

(10)

Ans: /*Program for static implementation of linked list*/

```
// class Nodetype for each node of
// the linked list
class Nodetype
{
int info,next;
Nodetype(int i,int n)
{
info=i;
next=n;
}
}
//-----
-----
```

Data Structure and Algorithms

```

class Operations
{
    int maxnodes=10;
    Nodetype node[]=new Nodetype[maxnodes];
    int avail;
    int list=-1;
    void createlist()
    {
        /*link all available nodes together*/

        int i;
        avail=0;
        for(i=0;i<maxnodes-1;i++)
            node[i]=new Nodetype(0,i+1);
        node[maxnodes-1]=new Nodetype(0,-1);
    } /* end createlist */
    int getnode()
    {
        /*obtain a node from available list and return its index */
        int p;
        if(avail== -1)
        {
            System.out.println("\nEmpty Linked List");
        }
        p=avail;
        avail=node[avail].next;
        return(p);
    } /*end getnode*/
}
//-----
void freenode(int p)
{
    /*accept index of a node and return that node to the available list*/
    node[p].next=avail;
    node[p].info=0;
    avail=p;
} /*end freenode*/
//-----
void display()
{
    /*display all elements of linked list*/
    int i;
    int temp;
    if(list== -1)
        System.out.println("\nEmpty linked list");
    else
    {
        temp=list;
        //System.out.println("\n"+temp);
    }
}

```

Data Structure and Algorithms

```

        System.out.println();
        while(temp!=-1)
        {
            System.out.print("-->" + node[temp].info + " | " + node[temp].next + " | ");
            temp = node[temp].next;
        }
    }
}
/*end display*/
//-----
void insertbeg(int x)
{
    /*insert new node at the beginning of linked list*/
    int q;
    q=getnode();
    node[q].info=x;
    node[q].next=list;
    list=q;
    display();
}
}

```

Q.3 a. *Explain Map Abstract data type.*

(10)

Ans: Map is an object that stores key/value pairs. Given a key, you can find its value. Keys must be unique, but values may be duplicated. The HashMap class provides the primary implementation of the map interface. The HashMap class uses a hash table to implement the map interface. This allows the execution time of basic operations, such as get() and put() to be constant. This code shows the use of HashMap. In this program HashMap maps the name to account balances.

```

import java.util.*;

public class HashMapDemo {

    public static void main(String[] args){

        HashMap hm = new HashMap();
        hm.put("Rohit", new Double(3434.34));
        hm.put("Mohit", new Double(123.22));
        hm.put("Ashish", new Double(1200.34));
        hm.put("Khariwal", new Double(99.34));
        hm.put("Pankaj", new Double(-19.34));
        Set set = hm.entrySet();

        Iterator I = set.iterator();
    }
}

```

Data Structure and Algorithms

```

while(i.hasNext()){
Map.Entry me = (Map.Entry)i.next();
System.out.println(me.getKey() + " : " + me.getValue() );
}

//deposit into Rohit's Account
double balance = ((Double)hm.get("Rohit")).doubleValue();
hm.put("Rohit", new Double(balance + 1000));

System.out.println("Rohit new balance : " + hm.get("Rohit"));

}
}

```

Output Screen:

Rohit : 3434.34

Ashish : 1200.34

Pankaj : -19.34

Mohit : 123.22

Khariwal : 99.34

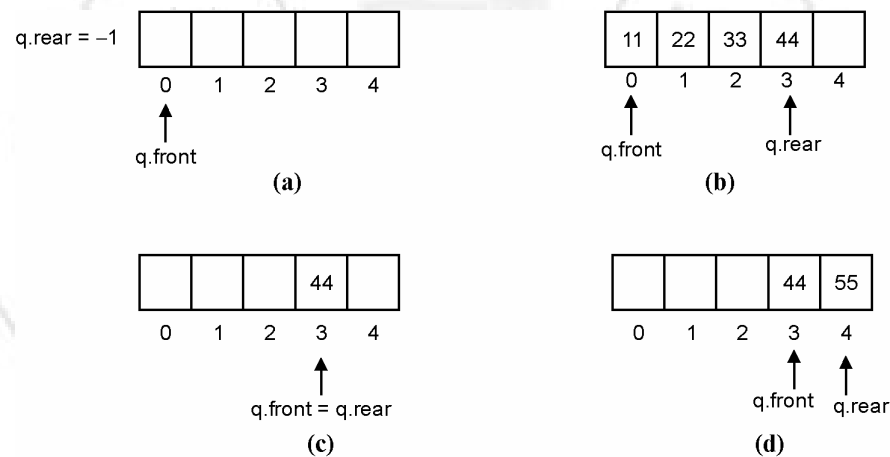
Rohit new balance : 4434.34

Q.3 b. **What are circular and priority queues.**

(10)

Ans: Circular Queue :

Consider following queue.



Let us first see what has happened with queue shown above. Fig. shows an array of 5 elements used to represent a queue. Initially (Fig. (a)), the queue is empty. In Fig. (b) items 11,22,33,44 have been inserted. In Fig. (c) three items have been deleted, and in Fig. (d) one new item '55' has been inserted. The value of $q.front$ is 3 and value of $q.rear$ is 4, so that there are only $4-3+1=2$ elements in the queue. Since the size of the array is 5 there should be room for the queue to expand without the worry of overflow.

Data Structure and Algorithms

However, to insert 66 into the queue, $q.rear$ must be increased by 1 to 5 and $q.items[5]$ must be set to the value 66. But $q.items$ is an array of only five elements, so that the insertion cannot be made. Clearly, the array representation discussed is not acceptable.

One solution is to modify the *remove* operation so that when an item is deleted, the entire queue is shifted to the beginning of the array. This method however is too inefficient.

Each deletion involves moving every remaining element of the queue. If a queue contains 500 or 1000 elements, this is clearly too high a price to pay.

Another solution is to view the array that holds the queue as a circle rather than as a straight line. That is we imagine the first element of the array as immediately following the last element of the array. This implies that even if the last element is occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty.

Let us look at previous queue (Fig (d)), where we could not insert 66 into the queue even though the queue was not full. Now assume that a circular queue contains two elements in positions 3 and 4 of a five element array. The situation of Fig. (d) is reproduced as Fig.

(a). Although the array is not full, its last element is occupied. If item 66 is now inserted into the queue, it can be placed in position 0 of the array, as shown in Fig.(b). The first item of queue is in $q.items[3]$, which is followed in the queue by $q.items[4]$ and $q.items[0]$.

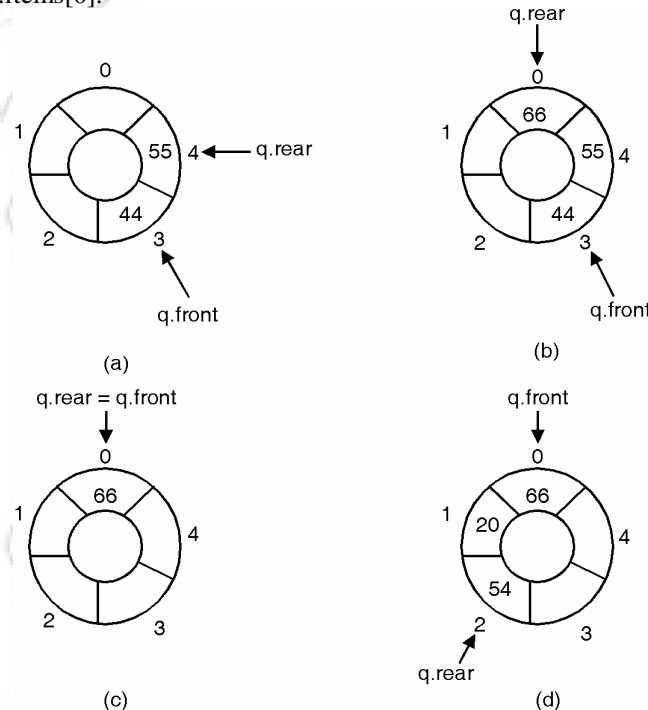
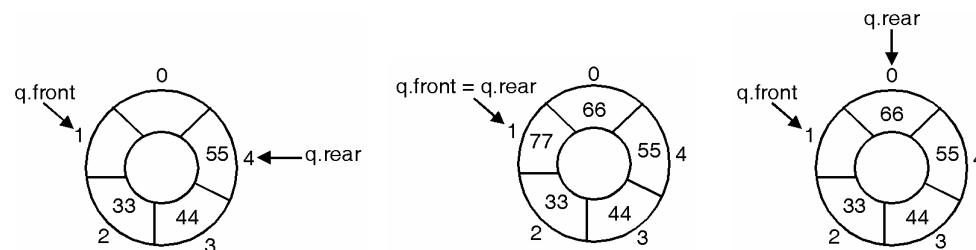


Fig. (c) and (d) show the status of the queue as first two items 44 and 55 are deleted, then 20 and 54 are inserted.

Unfortunately, it is difficult under this representation to determine when the queue is empty. The condition $q.rear < q.front$ is no longer valid as a test for the empty queue. One way of solving this problem is to establish the convention that the value of $q.front$ is the array index immediately preceding the first element of the queue rather than the index of the first element itself. Thus since $q.rear$ is the index of the last element of the queue, the condition $q.front == q.rear$ implies that the queue is empty.

The insert operation involves testing for overflow, which occurs when the entire array is occupied by items of the queue and an attempt is made to insert yet another item into the queue.

Data Structure and Algorithms



For example, consider the queue of Fig. (a). There are 3 elements into the queue: 33, 44, 55 in $q.items[2]$, $q.items[3]$, $q.items[4]$, respectively. Since the last element of the array occupies $q.items[4]$, $q.rear$ equals 4. Since the first element of the queue is in $q.items[2]$, $q.front$ equals 1. In Fig. (b) and (c) items 66 and 77 are inserted into the queue. At that point, the array is full and an attempt to perform any more insertions causes an overflow. But this is indicated by the fact that $q.front$ equals $q.rear$, which is precisely the condition for underflow. There is no way to distinguish between the empty queue and the full queue under this implementation. Such a situation is clearly unsatisfactory.

One solution is to sacrifice one element of the array and to allow a queue to grow only as large as one less than the size of the array. Thus, if an array of 100 elements is declared as a queue, the queue may contain upto 99 elements. An attempt to insert a hundredth element into a queue causes an overflow.

Priority Queue Operating principle: we insert element in random order but removed element which has highest priority.

Two types of priority queues:

- i Ascending priority queue: smallest element in queue removed first.
- ii Descending priority queue: largest element in queue removed first.

Q .4 a. *What is a binary search tree. suppose the following list of letters is inserted in order into an empty BST J R D G T E M h P A F Q Find final tree..* (10)

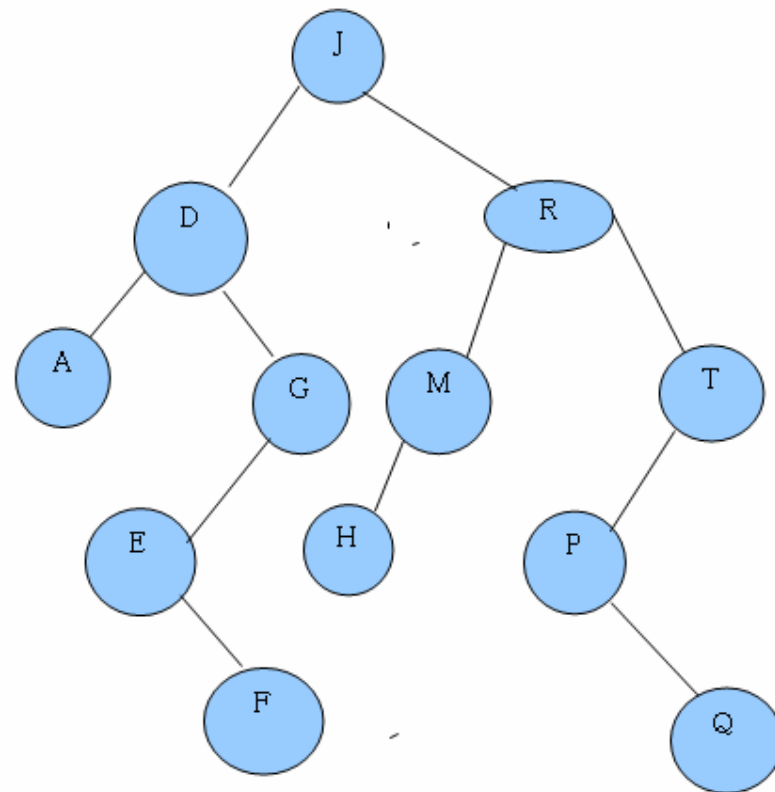
Ans: Binary Search tree

Searching a binary tree is almost like binary search! The difference is that instead of searching an array and defining the middle element ourselves, we just follow the appropriate pointer!

Dictionary search operations are easy in binary trees. The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees - recursive structure, recursive algorithm.

Data Structure and Algorithms

PROBLEM:



Q .4.b. Write the program to create a binary tree and preorder, inorder and postorder travelsal. (10)

Ans:

```

class node
{
public int data;
public node left;
public node right;
}
Class SimpleBinaryTree
{
node root,parent;
public SimpleBinaryTree()
{
root=null;
parent=null;
}
public void insert(node root,node New)
{
if (New.data<root.data)
{
if(root.left==null)
root.left=New;
else

```

Data Structure and Algorithms

```

        insert(root.left,New);
    }
    if(New.data>root.data)
    {
        if(root.right==null)
            root.right=New;
        else
            insert(root.right,New);
    }
}
public void search(node temp,int key)
{
    int found=0;
    if(temp==null)
        System.out.println("No BST Found");
    if(temp.data==key)
        found=1;
    else if(temp.data>key)
    {
        while(temp.left!=null&&found==0)
        {
            if(temp.data!=key)
                temp=temp.left;
            else
            {
                found=1;
                break;
            }
        }
    }
    else
    {
        while(temp.right!=null&&found==0)
        {
            if(temp.data!=key)
                temp=temp.right;
            else
            {
                found=1;
                break;
            }
        }
    }
    if(found==1)
    {
        System.out.println("Key Found");
    }
    else
    {
        System.out.println("Key absent");
    }
}
public void inorder(node temp)
{
    if(temp!=null)

```

Data Structure and Algorithms

```

        {
            inorder(temp.left);
            System.out.print("\t"+temp.data);
            inorder(temp.right);
        }
    }
    public void preorder(node temp)
    {
        if(temp!=null)
        {
            System.out.print("\t"+temp.data);
            inorder(temp.left);
            inorder(temp.right);
        }
    }
    public void postorder(node temp)
    {
        if(temp!=null)
        {
            inorder(temp.left);
            inorder(temp.right);
            System.out.print("\t"+temp.data);
        }
    }
}

```

Q.5 a. Explain selection sort and write a program to implement selection sort.

(10)

Ans:

```

static void selection(int n)
{
    int i,j,indx,large;
    for(i=n-1;i>0;i--)
    {
        large=a[0];
        indx=0;
        for(j=1;j<=i;j++)
            if(a[j]>large)
            {
                large=a[j];
                indx=j;
            }
        a[indx]=a[i];
        a[i]=large;
        System.out.println("Elements in Array after pass "+(n-i));
        for(int l=0;l<n;l++)
            System.out.print("\t"+a[l]);
    }
}

```

Data Structure and Algorithms

Q.5.b. Write an algorithm and explain with an example radix sort method.

(10)

Ans:

```

import java.lang.*;
import java.io.*;

public class RadixSort{

public static void radixSort(int[] arr){
    if(arr.length == 0)
        return;
    int[][] np = new int[arr.length][2];
    int[] q = new int[0x100];
    int i,j,k,l,f = 0;
    for(k=0;k<4;k++){
        for(i=0;i<(np.length-1);i++){
            np[i][1] = i+1;
            np[i][1] = -1;
            for(i=0;i<q.length;i++){
                q[i] = -1;
            }
            for(f=i=0;i<arr.length;i++){
                j = ((0xFF<<(k<<3))&arr[i])>>(k<<3);
                if(q[j] == -1)
                    l = q[j] = f;
                else{
                    l = q[j];
                    while(np[l][1] != -1)
                        l = np[l][1];
                    np[l][1] = f;
                    l = np[l][1];
                }
                f = np[f][1];
                np[l][0] = arr[i];
                np[l][1] = -1;
            }
            for(l=q[i=j=0];i<0x100;i++){
                for(l=q[i];l!=-1;l=np[l][1])
                    arr[j++] = np[l][0];
            }
        }
    }
}

public static void main(String[] args){
    int i;
    int[] arr = new int[15];
    System.out.print("original: ");
    for(i=0;i<arr.length;i++){
        arr[i] = (int)(Math.random() * 1024);
        System.out.print(arr[i] + " ");
    }
    radixSort(arr);
    System.out.print("\nsorted: ");
}

```

Data Structure and Algorithms

```

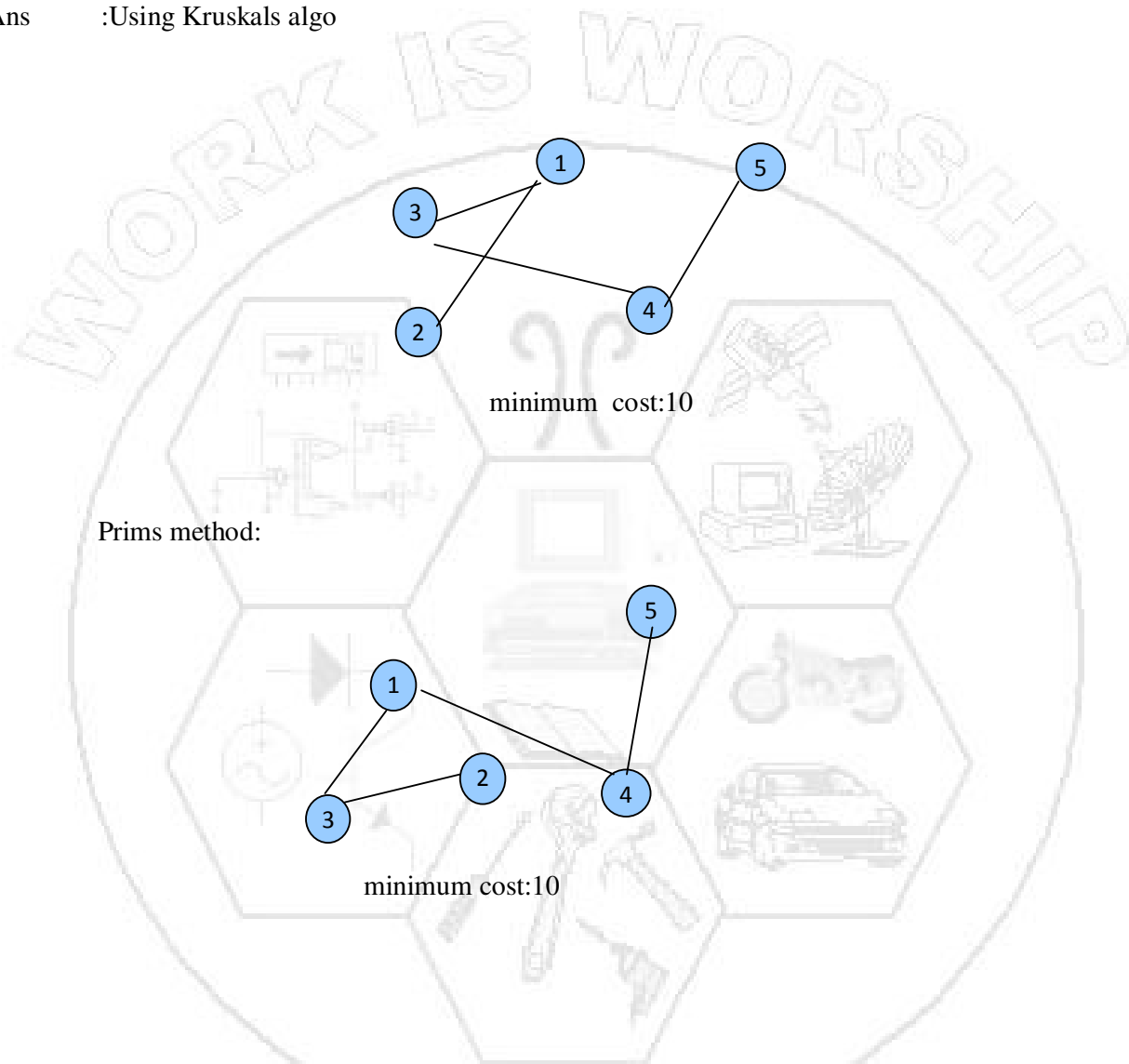
for(i=0;i<arr.length;i++)
    System.out.print(arr[i] + " ");
System.out.println("\nDone ;-");
}
}

```

Q.6 a. Find out the minimum cost spanning tree for below graph using kruskal and prisms method.

(10)

Ans :Using Kruskals algo



Q.6 .b . Write any pattern matching algorithm and explain it with suitable example.

(10)

Ans:

Code:

```

class Knuth_Morris_Pratt extends PatternMatchingAlgorithm {
    public int next[];
    public void preprocess(char p[], int m) {
        int j, t;
        next = new int[m+1];
    }
}

```

Data Structure and Algorithms

```

j = 0; t = -1; next[0] = -1;
while (j < m) {
    while (t >= 0 && p[j] != p[t]) t = next[t];
    t++; j++;
    if (j < m && p[j] == p[t]) next[j] = next[t];
    else
        next[j] = t;
}
//for (j=0; j <= m; j++)System.out.println("next[" + j + "] = " + next[j]);
}
void find(char t[], int n, char p[], int m) throws Exception {
    boolean trivial[] = new boolean[m];
    int i, j;
    int cnt = 0;

    preprocess(p,m);
    clear_table(trivial, m);
    pause(0, trivial, -1, false, false, cnt);
    j = 0;
    for (i = 0; i < n; i++) {
        while (j == m || (j >= 0 && p[j] != t[i])) {
            if (j < m) cnt++;
            if (stopRequested) return;
            if (j < m) pause(i-j, trivial, j, true, false, cnt);
            j = next[j];
            for (int k = 0; k < m; k++) trivial[k] = (k < j);
            pause(i-j, trivial, -1, false, false, cnt);

            if (j >= 0) {
                cnt++;
                pause(i-j, trivial, j, false, (j+1==m), cnt);
            }
        }
        ++j;
    }
}
}
}
}

```

Q.7. Write short notes on

(20)

a. Red-Black trees.

Ans:

Red-Black Tree Definition

Red-black trees have the following properties:

1. Every node is colored either red or black.
2. Every leaf (NIL pointer) is black.
3. If a node is red then both its children are black.
4. Every single path from a node to a descendant leaf contains the same number of black nodes.

Rotations

The basic restructuring step for binary search trees are left and right rotation:

1. Rotation is a local operation changing $O(1)$ pointers.
2. An in-order search tree before a rotation *stays* an in-order search tree.

Data Structure and Algorithms

3. In a rotation, one subtree gets one level closer to the root and one subtree one level further from the root.

b. *Text compression*

Ans:

Huffman coding :

Given the frequencies of each character in a file, the Huffman tree can be constructed as follows:

For each symbol to be encoded, make the binary tree consisting of a single root. Each of these root nodes should hold the frequency (or relative frequency) of one symbol. (Ultimately, these nodes will be the leaves of our Huffman tree)

2. Choose the two trees with the smallest root values and merge these into a new tree, as in the following diagram. Here the trees labeled A and H are chosen. The root of the new tree holds the value 3, obtained as the sum of the values stored in the roots of its left and right subtrees.

3. Continue the process, always forming a new tree from the two trees with the lowest root values. Here we merge the two trees each with 3 stored in the root are Four trees remain with root values 6,4,4,and 6.

4. Continue this process, only a single tree remains. This final tree is a representation of the Huffman code. Remember always merge the two subtrees which have the smallest root values

Now merge the two trees with root values 6. Two trees remain

Here is the final tree, with each branch labeled either 0 or 1.

The following is a general algorithm:

For each character c to be encoded
 create a binary tree, consisting of a single root node r such that $\text{content}(r) = f(c)$,
 the frequency of c
 for $i = 1$ to $n-1$
 {
 From the set of binary trees S , remove the two trees, x and y ,
 with minimal values, f_1 and f_2 in the root
 create a tree with root z such that
 x is the left child of z
 y is the right child of z
 $\text{content}(z) = f_1 + f_2$
 add this new tree with root z to S .
 }

Using the Huffman tree to determine the codewords.

Once the Huffman tree is created, it is easy to determine the codeword for each character.

Topdown Method

Data Structure and Algorithms

An ordinary, recursive tree traversal with an additional parameter can also be used to determine the codewords:

```
void traverse(string s, tree root)
```

Each time that the function is called recursively, pass it string *s* concatenated with either '0' or '1.' When visiting a leaf, simply output string *s* – *s* is the codeword for character associated with the leaf.

Encoding the file:

Simply translate each character using a table of codewords:

Example:

Character	Codeword
A	0000
H	0001
-	001
E	10
L	11
S	01

Translate: "SHE-SELLS....."

01000110001011011101

S H E - S E L L S

c. **Graph Traversal**

Ans:

i Breadth first search

ii Depth first search

i. Breadth First Search

BFS is graph traversal method in this method we can start searching from any vertex. Vertex v_1 in the graph will be visited first then all the vertices adjacent to v_1 will be travelled suppose adjacent to v_1 are so $v_2, v_3 \dots v_n$ will be printed first then again from v_2 the adjacent vertices will be printed this processes will be continue for all the vertices to get encountered to keep track of all the vertices and their adjacent vertices.

Algorithm-

Step 1: Start with any node and mark it as visited.

Step 2: find the adjacent nodes to the node mark in step1 and them in Queue

Step 3: Visit the node which is at the front of Queue delete this node from Queue.

Step 4: Repeat step3 still the Queue is not empty.

Step 5: Stop.

Example:

ii. Depth first search

DFS, the next vertex to be visited will be the adjacent vertex (to the starting point), which has the highest depth value and then the next adjacent vertex with the next higher depth value and so on till all the adjacent nodes for that vertex are not visited. We repeat this same procedure for every visited vertex of the graph.

Algorithm:-

Step-1: Set the starting point of traversal and push it inside the stack

Data Structure and Algorithms

Step-2: Pop the stack and add the popped vertex to the list of visited vertices

Step-3: Find the adjacent vertices for the most recently visited vertex(from the Adjacency Matrix)

Step-4: Push these adjacent vertices inside the stack (in the increasing order of their depth) if they are not visited and not there in the stack already

Step-5: Go to step-2 if the stack is not empty

d. **Recursion**

Ans: Definition :

Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attributes that allows a method to call itself. A method that calls itself is called to be recursion.

Example :

e. **Comparison of sorting algorithms.**

Ans:

Insertion sort

Insertion sort is good only for sorting small arrays (usually less than 100 items). In fact, smaller the array, the faster insertion sort is compared to any other sorting algorithm.

However, being an $O(n^2)$ algorithm, it becomes very slow very quick when the size of the array increases. It was used in the tests with arrays of size 100.

Shell sort

Shell sort is a rather curious algorithm, quite different from other fast sorting algorithms. It's actually so different that it even isn't an $O(n \log n)$ algorithm like the others, but instead it's something between $O(n \log^2 n)$ and $O(n^{1.5})$ depending on implementation details.

Given that it's an *in-place* non-recursive algorithm and it compares very well to the other algorithms, shell sort is a very good alternative to consider.

Heap sort

Heap sort is the other (and by far the most popular) *in-place* non-recursive sorting algorithm used in this test. Heap sort is not the fastest possible in all (nor in most) cases, but it's the *defacto* sorting algorithm when one wants to make *sure* that the sorting will not take longer than $O(n \log n)$. Heap sort is generally appreciated because it is trustworthy: There aren't any "pathological" cases which would cause it to be unacceptably slow. Besides that, sorting *in-place* and in a non-recursive way also makes sure that it will not take extra memory, which is often a nice feature.

One interesting thing I wanted to test in this project was whether heap sort was considerably faster or slower than shell sort when sorting arrays.

Merge sort

The virtue of merge sort is that it's a truly $O(n \log n)$ algorithm (like heap sort) and that it's stable (iow. it doesn't change the order of equal items like eg. heap sort often does). Its main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the memory requirements.

In this test I helped merge sort a bit by giving it the second array as parameter so that it wouldn't have to allocate and deallocate it each time it was called (which would have probably slowed it down somewhat, especially with arrays of the bigger items). Also,

Data Structure and Algorithms

instead of doing the basic "merge to the second array, copy the second array to the main array" procedure like the basic algorithm description suggests, I simply merged from one array to the other alternatively (and in the end copied the final result to the main array only if it was necessary).

Quicksort

Quicksort is the most popular sorting algorithm. Its virtue is that it sorts *in-place* (even though it's a recursive algorithm) and that it usually is very fast. One reason for its speed is that its inner loop is very short and can be optimized very well.

The main problem with quicksort is that it's not trustworthy: Its worse-case scenario is

$O(n^2)$ (in the worst case it's as slow, if not even a bit slower than insertion sort)

and the pathological cases tend to appear too unexpectedly and without warning, even if an optimized version of quicksort is used (as I noticed myself in this project).

I used two versions of quicksort in this project: A plain vanilla quicksort, implemented as the most basic algorithm descriptions tell, and an optimized version (called MedianHybridQuickSort in the source code below). The optimized version chooses the median of the first, last and middle items of the partition as its pivot, and it stops partitioning when the partition size is less than 16. In the end it runs insertion sort to finish the job.

The optimized version of quicksort is called "Quick2" in the bar charts.

The fourth number distribution (array is sorted, except for the last 256 items which are random) is very expectedly a pathological case for the vanilla quicksort and thus was

skipped with the larger arrays. Very unexpectedly this distribution was

pathological for the optimized quicksort implementation too, with larger arrays, and thus I also skipped it in the

worst cases (because else it would have affected negatively the scale of the bar charts). I don't have any explanation of why this happened.

f. Searching algorithm

Ans:

Linear Search:

In computer science, linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search.

Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) may be much more efficient.

Binary search:

a binary search or half-interval search algorithm locates the position of an item in a sorted array. Binary search works by comparing an input value to the middle element of the array. The comparison determines whether the element equals the input, less than the input or greater. When the element being compared to equals the input the search stops and typically returns the position of the element. If the element is not equal to the input then a comparison is made to determine whether the input is less than or greater than the element. Depending on which it is the algorithm then starts over but only searching the top or bottom subset of the array's elements. If the input is not located within the array the algorithm will usually output a unique value indicating this.

Binary search algorithms typically halve the number of items to check with each successive

Data Structure and Algorithms

iteration, thus locating the given item (or determining its absence) in logarithmic time. A binary search is a dichotomic divide and conquer search algorithm. It is useful to find where an item is in a sorted array. For example, if you had an array for contact information, with people's names, addresses, and telephone numbers sorted by name, you could use binary search to find out a few useful facts: whether the person's information is in the array, what the person's address is, and what the person's telephone number is.

Binary search will take far fewer comparisons than a linear search, but there are some downsides. Binary search can be slower than using a hash table. If items are changed, the array will have to be re-sorted so that binary search will work properly, which can take so much time that the savings from using binary search aren't worth it. If you can tell ahead of time that a few items are disproportionately likely to be sought, putting those items first and using a linear search could be much faster.

