**Narayan Asati**

**(MIT2021072).**

**Question1:-**Represent the following sets by using BST: A={7,2,4,9,15,6,11},

B={12,3,16,1,5,8,10}. Merge both the sets into new set C (test if merge operation is possible). Write the procedure to display C.

**We divide the problem into steps.**

**Step-1:-**First we implement Set A, Set B, and Set C by using BST.

* In java, we can implement a set using BST which is called TreeSet.

TreeSet:-TreeSet is one of the most notable Java implementations of the SortedSet interface that stores data in a Tree. Whether or not an explicit comparator is provided, a set maintains the order of the components by using their natural ordering. If the Set interface is to be correctly implemented, this must be compatible with equals.

//Set<Integer> setA=new TreeSet<>();

//Set<Integer> setB=new TreeSet<>();

//Set<Integer> setC=new TreeSet<>();

**Step-2:-**In this we are adding an element in TreeSet-A and in TreeSet-B respectively with the help of the set.add(element) function in java.

* add(element):-This function will add the supplied element to the TreeSet in the same sorting order as when it was created. Entries that are duplicated will not be added.

//setA.add(element);

//setB.add(element);

**Step-3:-** Merging both set-A and set-B in set-C with the help of function addAll(object).

* addAll(Collection c):-To append all of the elements from the stated collection to the existing set, use the java.util.Set.addAll(Collection C) method. The elements are inserted at random and in no particular order.

//setc.addAll(setA); merging setA in setC(union of setC and SetA)

//stec.addAll(setB);merging setB in setC(union of setC and SetA)

* At the time of merging, duplicate elements will be removed from the set. that is the set property.

**Step-4:-**Display set-C

* It will Display in order traversal of BST or in a sorted order that is a specialty of TreeSet.

**Code:-**

**import java.util.\*;**

**public class TreeSetEx{**

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

**Set<Integer> setA=new TreeSet<>();// implement setA**

**Set<Integer> setB=new TreeSet<>();// implement setB**

**Set<Integer> setC=new TreeSet<>();// implement setC**

**setA.add(7);//adding element to setA**

**setA.add(2);**

**setA.add(4);**

**setA.add(9);**

**setA.add(15);**

**setA.add(6);**

**setA.add(11);**

**setB.add(12);//adding element to setB**

**setB.add(3);**

**setB.add(16);**

**setB.add(1);**

**setB.add(5);**

**setB.add(8);**

**setB.add(10);**

**System.out.println("SetA Representaion Using BST");**

**System.out.println(setA);//printing a element in sorted order**

**System.out.println("SetB Representaion Using BST");**

**System.out.println(setB);**

**setC.addAll(setA);**

**setC.addAll(setB);**

**System.out.println("Merging Of setA and setB in setC");**

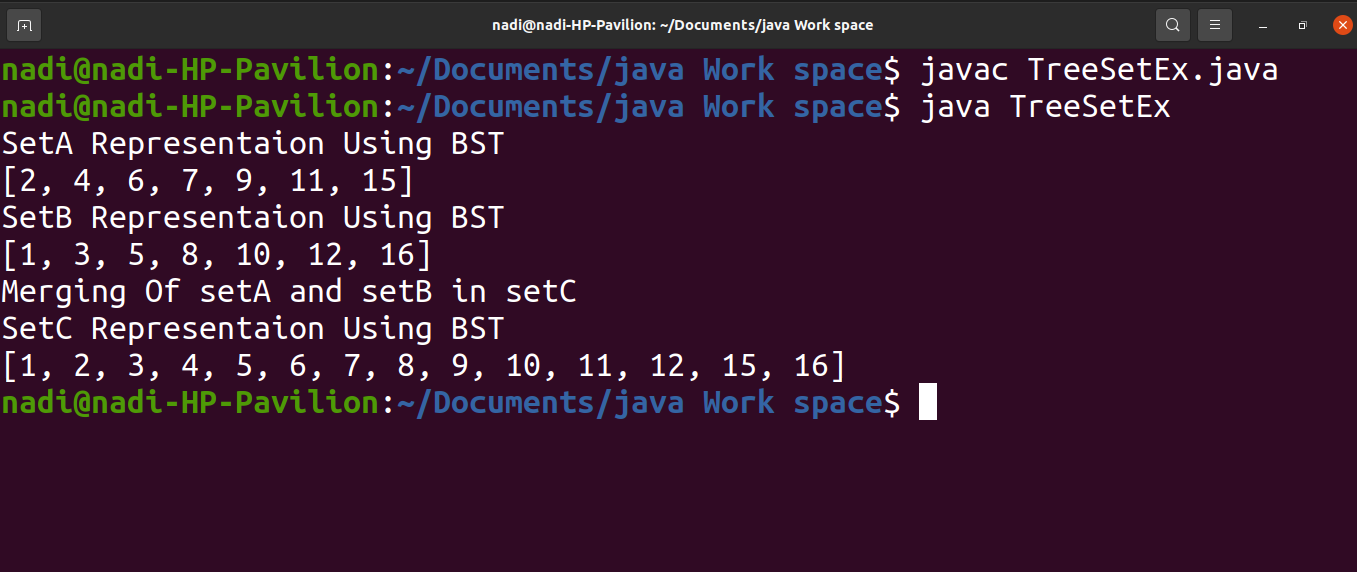
**System.out.println("SetC Representaion Using BST");**

**System.out.println(setC);**

**}**

**}**

**Output:-**

****

**Q 6 :-Write the algorithm to find the longest path in a Graph and Estimate the complexity of the algorithm.**

The longest path problem is the challenge of finding a simple path in a graph with the maximum length; in other words, the problem is to discover the longest simple path among all feasible simple paths in the graph. For an unweighted graph, finding the longest path in terms of the number of edges suffices; for a weighted graph, edge weights must be used instead. To illustrate how we came up with this technique, we'll start by creating an algorithm for computing the single-source longest path in an unweighted directed acyclic graph (DAG), then generalize it to compute the longest path in any DAG, unweighted or weighted.

I would do a Dynamic Programming algorithm. Denote *L*(*u*) to be the longest valid path starting at node *u*. Your base case is *L*(*n*-1) = [*n*-1] (i.e., the path containing only node *n*-1). Then, for all nodes *s* from *n*-2 to 0, perform a BFS starting at *s* in which you only allow traversing edges (*u*,*v*) such that *v* > *u*. Once you hit a node for which you've already started at (i.e., a node *u* such that you've already computed *L*(*u*)),

*L*(*s*) = longest path from *s* to *u* + *L*(*u*) out of all possible *u* > *s*.

The answer to your problem is the node *u* that has the maximum value of *L*(*u*), and this algorithm is O(*E*), where *E* is the number of edges in your graph. I don't think you can do faster than this asymptotically

Actually, the "BFS" isn't even a BFS: it's simply traversing the edges (*s*,*v*) such that *v* > *s* (because you have already visited all nodes *v* > *s*, so there's no traversal: you'll immediately hit a node you've already started at.

**algorithm:-**

**longest\_path\_increasing\_nodes():**

L = Hash Map whose keys are nodes and values are paths (list of nodes)

L[n-1] = [n-1] # base case

longest\_path = L[n-1]

for s from n-2 to 0: # recursive case

L[s] = [s]

for each edge (s,v):

if v > s and length([s] + L[v]) > length(L[s]):

L[s] = [s] + L[v]

if L[s] > longest\_path:

longest\_path = L[s]

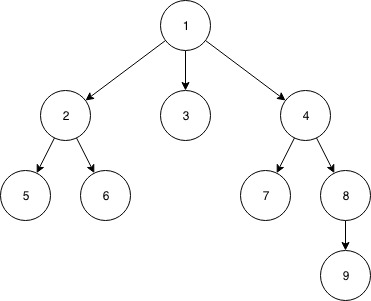
return longest\_path

Q8. Traverse a general tree (rooted) in all siblings together (bfs) manner and write the procedure for it. Discuss the space complexity of tree representation first.

In a general rooted tree, each node may have any number of childs connected to it. So we need to do bfs on general rooted tree.

To represent the general rooted tree, we need to store it in the same way as we store graphs in the adjacency list. So it will take space of O(n+2m) since for each node we need to store their adj nodes. Along with the adjacency list, we also need to be visited the array to keep track of visited nodes while doing bfs traversal. This will take O(n) space.

Example-1:-



Input:-

No. of vertices=9

No. of edges=8

Edges between (u,v) vertices.

1--->3

1--->2

1 --->4

2--->5

2--->6

4--->7

4--->8

8--->9

output:-[1,4,3,2,5,6,7,8,9]

Algorithm:-

BFT(V){

visited(V)=1;

add(V,Q);

while(Q is not empty){

x=delete(Q);

print(x);

For all w adjacent to x

{

if(w is not visited)

{ visited(w)=1;

add(w,Q);}

}

}

Pseudo code:-

public static void BFS(int x,ArrayList<ArrayList<Integer>> a,int[] visited,ArrayList<Integer> bfstraverse){

Queue<Integer> q1 = new LinkedList<Integer>();

visited[x]=1;

q1.add(x);

while(!q1.isEmpty())

{int p=q1.poll();

bfstraverse.add(p);

for(int i=0;i<a.get(p).size();i++){

if(visited[a.get(p).get(i)]==0){

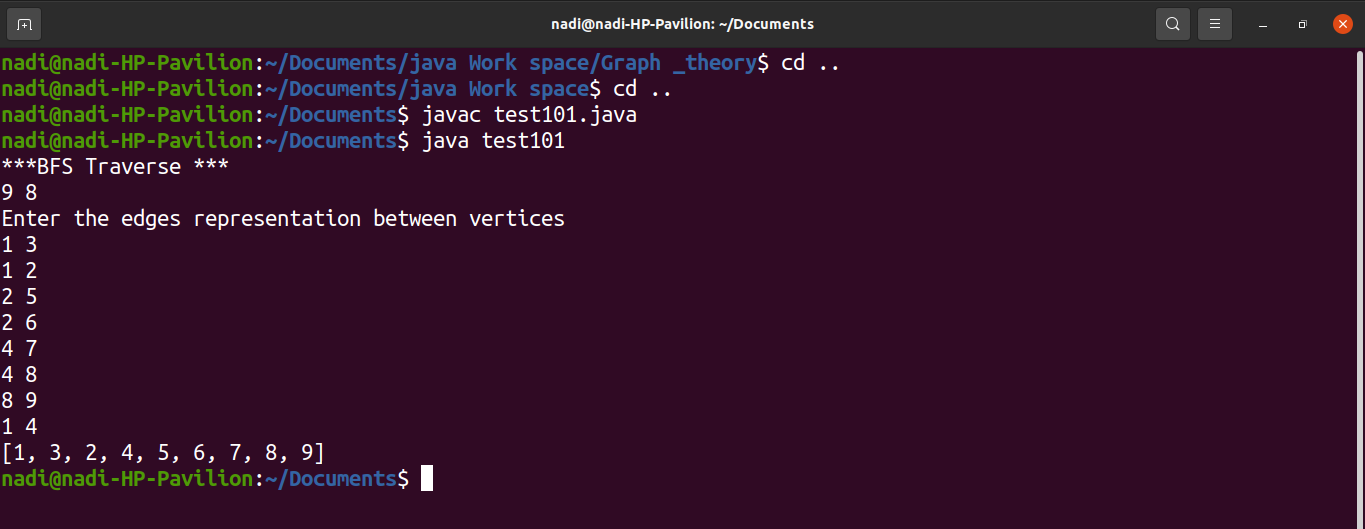
visited[a.get(p).get(i)]=1;

q1.add(a.get(p).get(i));}

}

}

}



Time complexity:-O(v+e)

Space Complexity:-

* To represent the general rooted tree using adjacency list will be O(v + 2e).
* Using an adjacency matrix it will take O(n2) space.