CS 601 - Advanced Algorithms
Programming Assignment 2

Question 1:

Code Report :

To execute program please execute below command

bash SearchingAlgortihms.bash

Input is accessed from input.txt file. From main function we call *readTextFile(), constructTree(),insertIntoTree,BFS() and DFS()*

readTextFile()

Using scanner objects we retrieve each line as a string and store it in an arraylist of string(listOfInput). If the file is not accessible we throw an error.

constructTree()

This function is for constructing binary trees. It iterates over the listOfInput string array and forms the binary tree. Each input till it reach “*” is split and converted to two integers and sent to the insertIntoTree function to build the tree accordingly. We used the BFS algorithm to find the element where the successor needs to be added. Once we found a parent we added the child as left if left is null,if left is not null we added it as right child. Element after * is stored as goalElement, which is a search element in the tree.

insertIntoTree()

This function is for building trees. It takes predecessorValue and successorValue as input and uses bfs to search predecessor and add successor value to predecessor as child.

BFS()

Breadth first search algorithm is implemented using a queue.

Algorithm used for this is as follows :

```
while (queue is not empty)

{

remove a node from the queue;
 if (node is a goal node) return success;
put all children of node onto the queue;

}

return failure;
```

We use a flag element to check failure cases. Initially we add root to the queue and check whether the queue is empty. If it is not empty we poll the queue and check with goalElement, if we find a goal element we break and if not we push all the children of the polled element to the queue. Finally If the goal element is not found we return a false message.

DFS()

Depth First Algorithm uses stack:

Algorithm used for implementation is :

```
while (stack is not empty)

 {

 remove (pop) the top node from the stack;
if (node is a goal node) return success;
Push all children of node onto the stack;

}

return failure;
```

We use a flag element to check failure cases. Initially we add the root to the stack and check whether the stack is empty. If it is not empty we pop stack and check with goalElement, if we find a goal element we break and if not we push all the children of popped element to stack. Finally If the goal element is not found we return a false message.

Time Complexity

Time complexity of each function is defined below

Main Function

Operations	**Costs**	**Time**
readTextFile();	c1	1
constructTree();	c2	1
BFS()	c3	1
DFS()	c4	1

readTextFile Function

File fileObject = **new** File(System.*getProperty*("user.dir") + "/src/com/advancealgo/assignmenttwo/input.txt");	c5	1

Scanner scanObject = **new** Scanner(fileObject);	c6	1
while (scanObject.hasNextLine())	c7	n+3(*,#,search element)
String characterElement = scanObject.nextLine();	c8	n+3
listOfInput.add(characterElement);	c9	n+3
scanObject.close();	c10	1

catch (FileNotFoundException e) {
 	
 System.***out***.println("Error occurred while accessing file");
 	
 e.printStackTrace();
 }
	c11	1

constructTree Function

for (**int** i = 0(c12); i < *listOfInput*.size()(c13); i++c(14))	c12	1
	c13	n+4
	c14	n+3
String characterElement = *listOfInput*.get(i);	c15	n+3
if (!characterElement.equals("*"))	c16	n+1
String[] arrayOfcharacterElement = characterElement.split(",", 2);	c17	n

int predecessorValue = Integer.*parseInt*(arrayOfcharacterElement[0]);	c18	n
int successorValue = Integer.parseInt(arrayOfcharacterElement[1]);	c19	n
if (root == null)	c20	n
root = new TreeNode(predecessorValue);	c21	1
root.left = new TreeNode(successorValue);	c22	1
else	c23	n
insertIntoTree(predecessorValue,successorValue);	c24	n
else	c24	1
goalElement = Integer.parseInt(listOfInput.get(i + 1));	c25	1
break;	c26	1

insertIntoTree Function

Queue<TreeNode> queue = new LinkedList<TreeNode>();	c27	n
queue.add(root);	c28	n
while (!queue.isEmpty())	c29	n
TreeNode tempNode = queue.poll();	c30	n^2
if (tempNode.data == predecessorValue)	c31	n^2

if (tempNode.left == null)	c32	n^2
tempNode.left = new TreeNode(successorValue);	c33	n^2
else	c34	n^2
tempNode.right = new TreeNode(successorValue);	c35	n^2
if (tempNode.left != null)	c36	n^2
queue.add(tempNode.left);	c37	n^2
if (tempNode.right != null)	c38	n^2
queue.add(tempNode.right);	c39	n^2

BFS Function

boolean flag = false;	c40	1
Queue<TreeNode> queue = new LinkedList<TreeNode>();	c41	1
while (!queue.isEmpty())	c42	*n*
TreeNode tempNode = queue.poll();	c43	*n*
if (tempNode.data == goalElement)	c44	*n*
System.out.println("BFS: Success");	c45	*1*
flag = true;	c46	*1*

break;	c47	*1*

if (tempNode.left != null) {
 	
queue.add(tempNode.left);
}
	c48	*n*

if (tempNode.right != null) {
 	
queue.add(tempNode.right
);
}
	c49	*n*

if (!flag)
 	
System.out.println("BFS:
Failure");
	c50	*1*

DFS Function

boolean flag = false;
Stack<TreeNode> stack =
new Stack<TreeNode>();
stack.push(root)
	c51	1

while (!stack.isEmpty())	c52	n
TreeNode tempNode = *stack.pop();* *if* *(tempNode.data ==* *goalElement)*	c53	n
	c56	1

System.out.println("DFS: Success");
 	
flag = true;
 	
break;		

if (tempNode.right != null) {
 	
stack.push(tempNode.right);
}
if (tempNode.left != null) {
 	
stack.push(tempNode.left);
}
	c57	n

if (!flag)
 	
System.out.println("DFS: Failure");
	c56	1

f(n) can be obtained by adding all costs * time

As program is very lengthy we are considering operation

which $\overset{has}{\wedge}$ higher magnitude and all i's are replaced with c

$$f(n) = c7(n+3) + c8(n+3) + c9(n+3) + c13(n+4) + c14(n+3) + c15(n+3)$$

$$c16(n+1) + c17(n) + c18(n) + c19(n) + c20(n) + c23(n) + c24(n)$$

$$c27(n) + c28(n) + c29(n) + c30(n^2) + c31(n^2) + c32(n^2) +$$

$$c33(n^2) + c34(n^2) + c35(n^2) + c36(n^2) + c37(n^2) + 38(n^2) +$$

$$c39(n^2) + c42(n) + c43(n) + c44(n) + c48(n) + c49(n) +$$

$$+ c52(n) + c53(n) + c57(n) + c$$

we make all n^2, n, and c together

$$= n^2(c30 + c31 + c32 + c33 + c34 + c35 + c36 + c37 + c38 + c3$$

$$+ n(c7 + c8 + c9 + c13 + c14 + c15 + c16 + c17 + c18 + c19 + c2$$

$$c23 + c24 + c27 + c28 + c29 + c42 + c43 + c44 + c48 + c49$$

$$c52 + c53 + c57) + c$$

$$= n^2 c'' + n c' + c$$

consider c', c'', & c are constants ~~only higher as~~ .

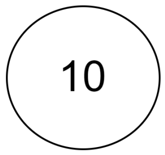
as $\overset{n}{\wedge}$ increases the whole runtime increase, for n^2

so consider only higher magnitudes

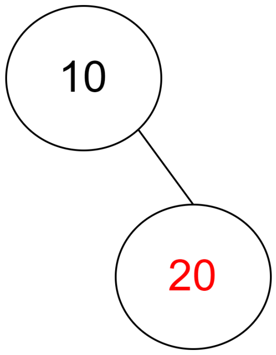
$$\Rightarrow \boxed{f(n) = O(n^2)}$$

Question 2:
Create a Red-Black tree by inserting the following numbers into the tree. Show all your work step by step and for EACH number that you insert, explain how you treated that, and which rules you applied. USE DIFFERENT COLORS (BLACK AND RED) and color nodes once you draw each resulting tree. 10, 20, 8, 17, 19, 25, 60, 100.

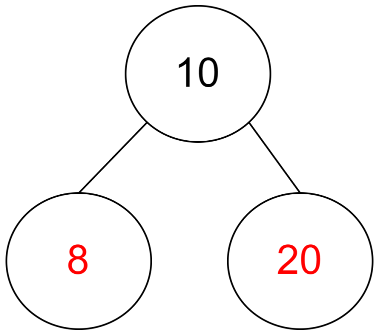
⇒ Inserting first node 10.
⇒ Rule: If the tree is empty , create a new node as root node and color it as black.



⇒ Inserting next node 20.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is black , then exit.

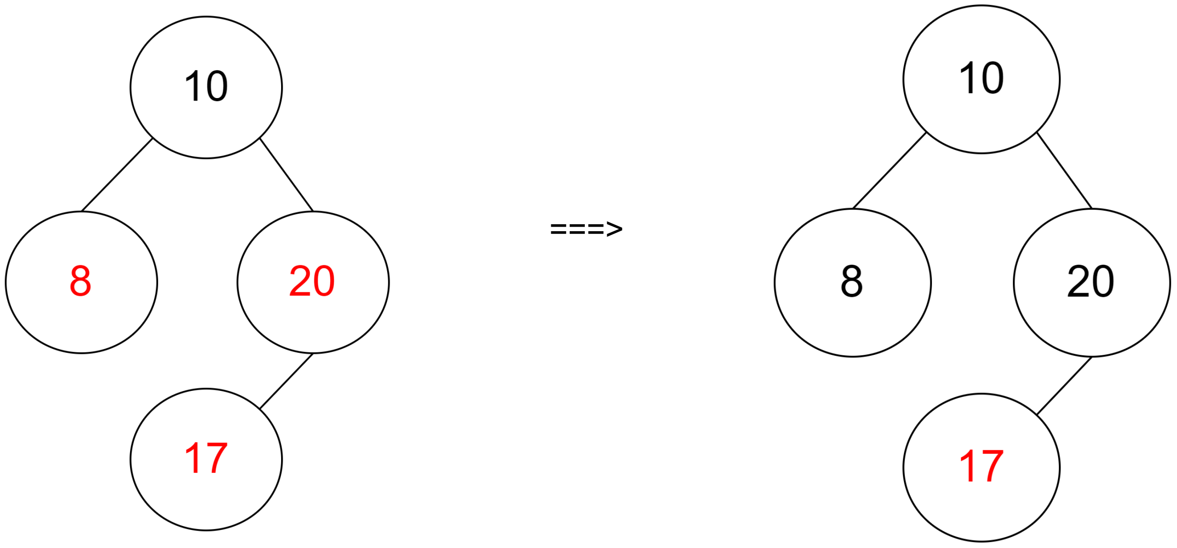


⇒ Inserting next node 8.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is black , then exit.



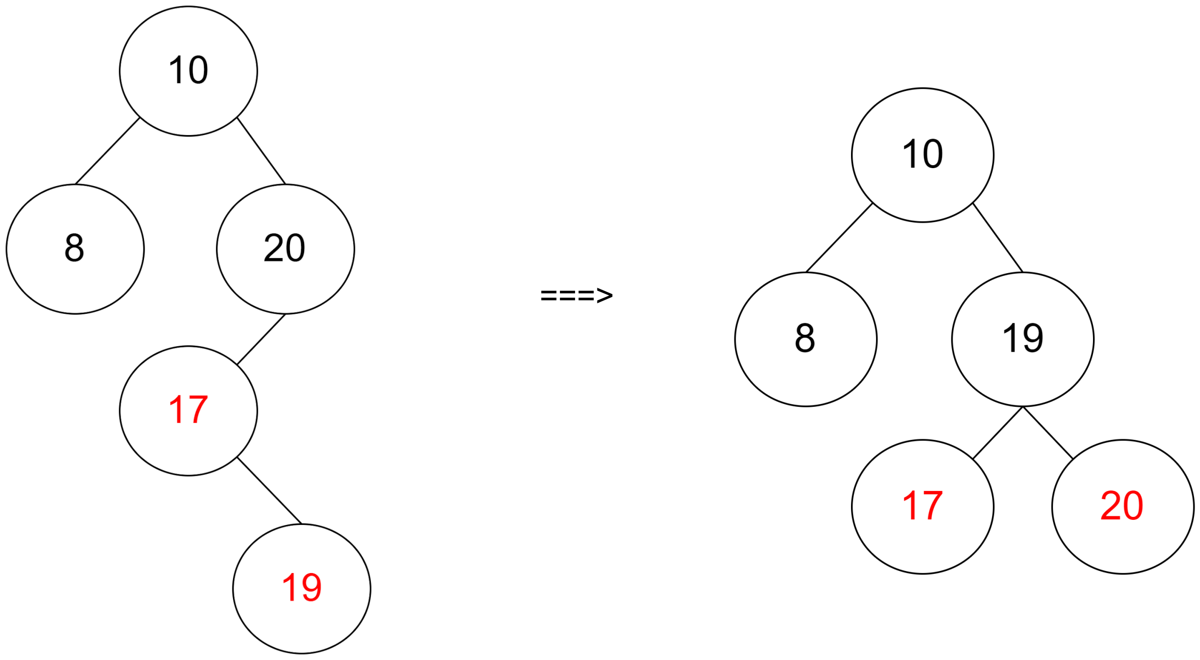
⇒ Inserting next node 17.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is red, there comes the red-red conflict.

⇒ Rule: If there is a red-red conflict check the uncle node (sibling of parent node). If the uncle node is red then recolor and check , if the grandparent node (parent of parent node) of the new node is not root node then recolor it and check.



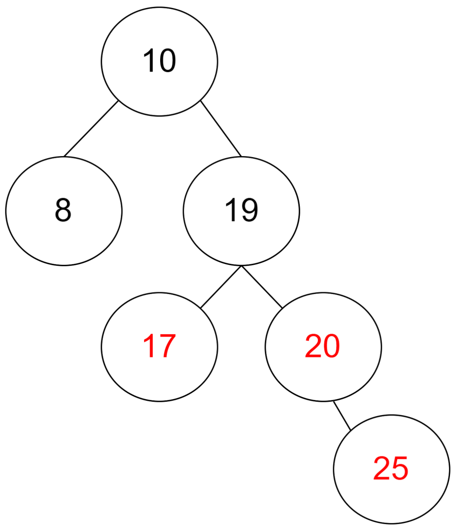
⇒ Inserting next node 19.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is red, there comes the red-red conflict.

⇒ Rule: If there is a red-red conflict check the uncle node (sibling of parent node). If the uncle node is black or null then do suitable rotation and recolor. (LR rotation)

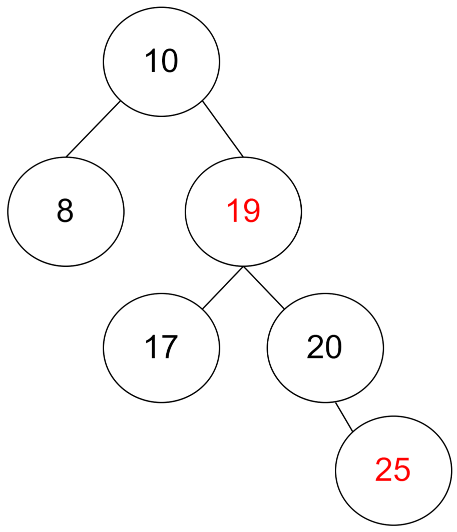


⇒ Inserting next node 25.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is red, there comes the red-red conflict.

⇒ Rule: If there is a red-red conflict check the uncle node (sibling of parent node). If the uncle node is red then recolor and check , if the grandparent node (parent of parent node) of the new node is not root node then recolor it and check.

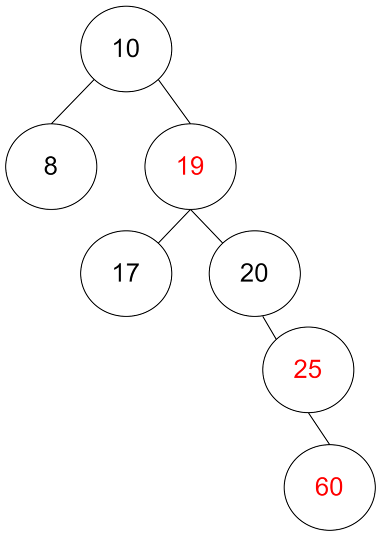


===>

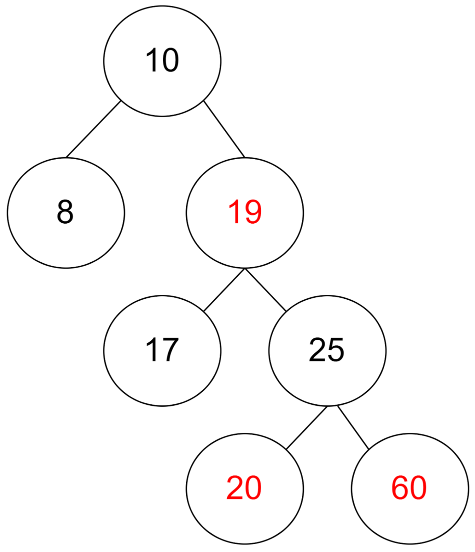


⇒ Inserting next node 60.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is red, there comes the red-red conflict.

⇒ Rule:If there is a red-red conflict check the uncle node (sibling of parent node). If the uncle node is black or null then do suitable rotation and recolor. (RR Rotation)

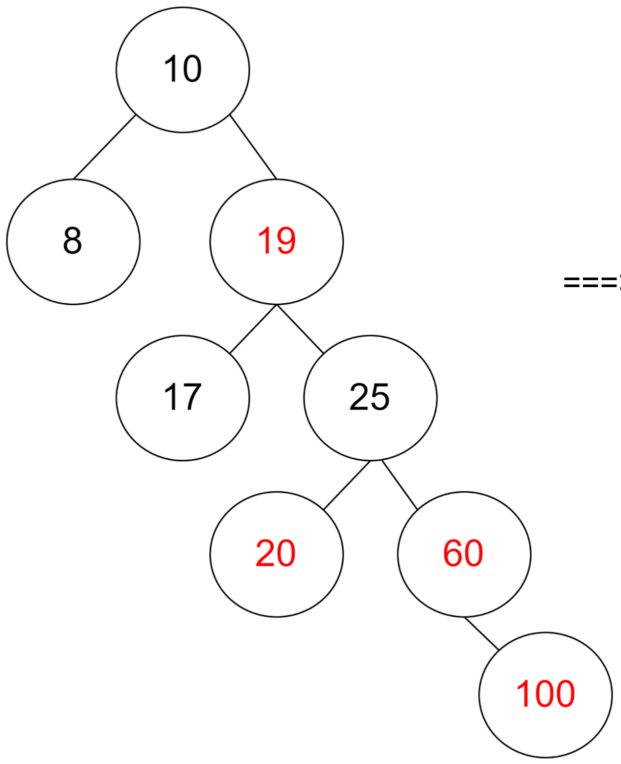


===>

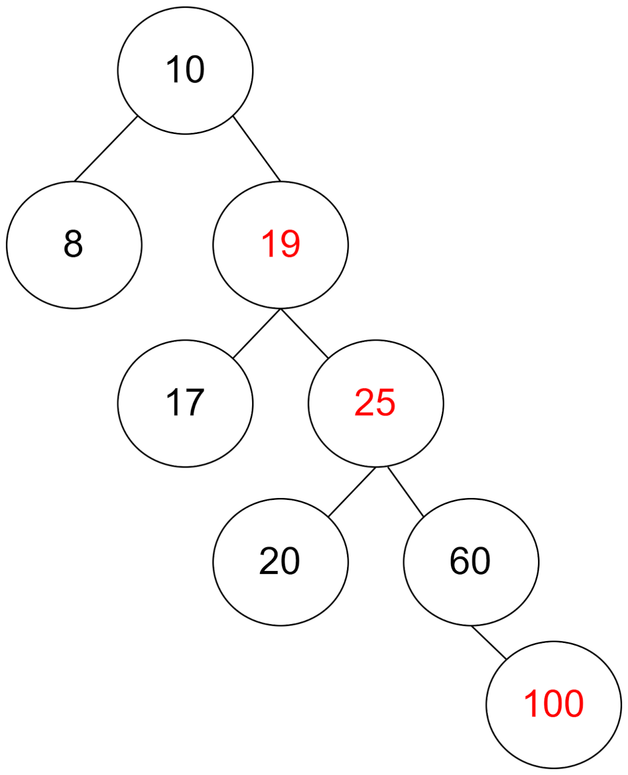


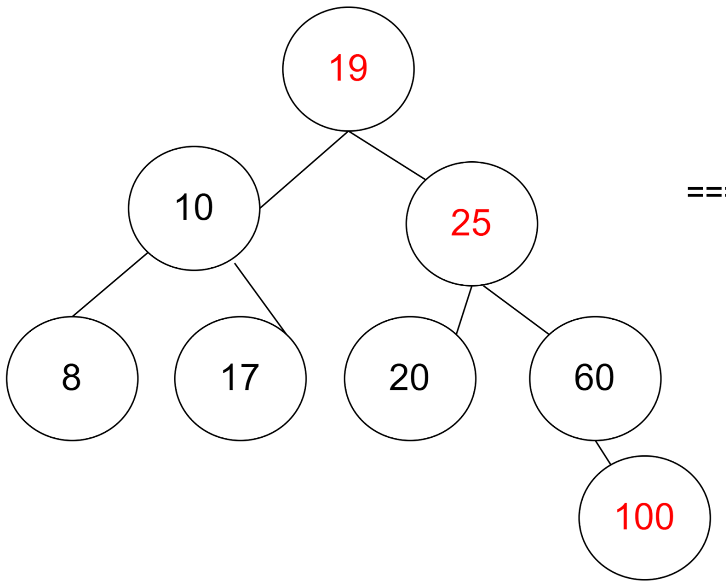
⇒ Inserting next node 100.
⇒ Rule: If the tree is not empty , create a new leaf node and color it as red. If the parent of the new node is red, there comes the red-red conflict.

⇒ Rule:If there is a red-red conflict check the uncle node (sibling of parent node). If the uncle node is black or null then do suitable rotation. (RR Rotation)

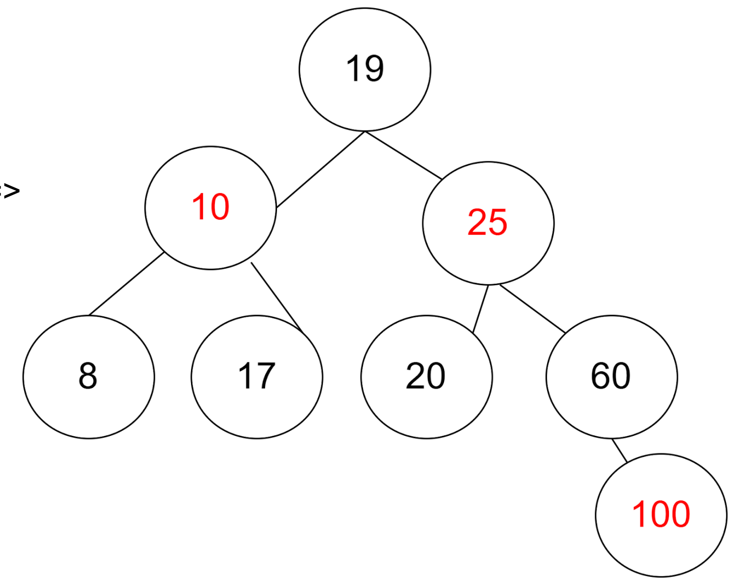


===>





===>

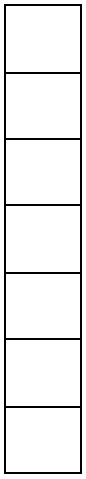


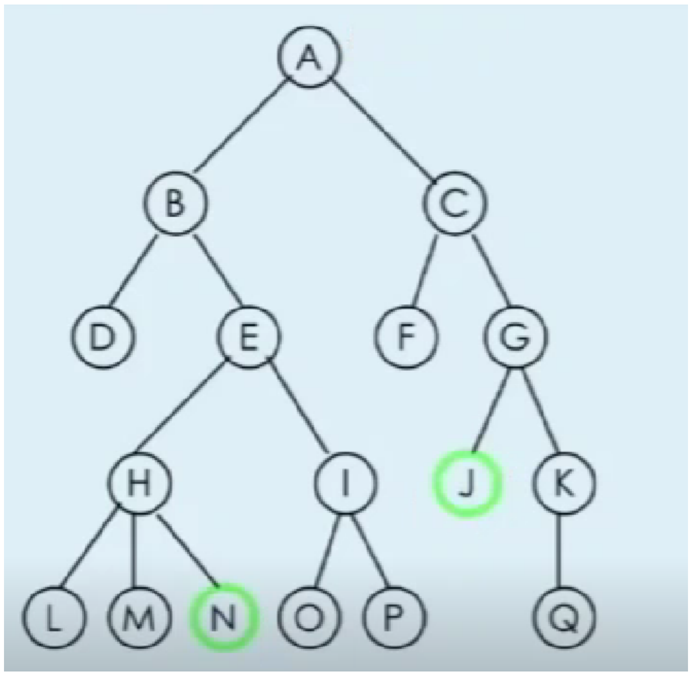
Practice Question (DFS) :

Given a Tree we need to search for the goal node (N or J).

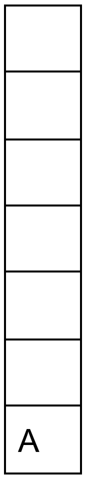
Step by Step Procedure :

1)Initially the stack is empty.



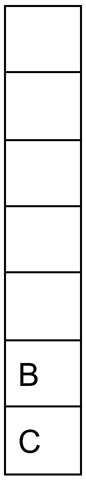


2) Push the root node on to stack, i.e, A

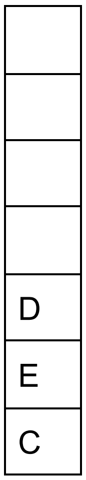


3)As stack is not empty we need to remove top node from stack, i.e, A

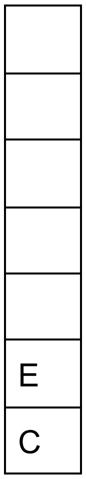
A is not equal to our goal node which is N, so we push children of A to stack. Children of A is B and C (first push right and than left)



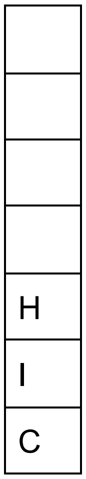
4) Stack is not empty so B is removed from stack and compared with goal node
B is not equal to goal node so children of B is pushed into stack



5) Stack is not empty so D is removed from stack and compared with goal node D is not equal to the goal node so children of D are pushed into the stack. D has no children so nothing is pushed to stack.



6) Stack is not empty so E is removed from stack and compared with goal node.E is not equal to the goal node so children of E are pushed into the stack.



7) Stack is not empty so H is removed from stack and compared with goal node.H is not equal to the goal node so children of H are pushed into the stack.

L
M
N
I
C

8) Stack is not empty so L is removed from stack and compared with goal node.L is not equal to the goal node so children of L are pushed into the stack. L is not having any children .

M
N
I
C

9) Stack is not empty so M is removed from stack and compared with goal node.M is not equal to the goal node so children of M are pushed into the stack. M is not having any children .

N
I
C

10) Stack is not empty so N is removed from stack and compared with goal node.N is equal to the goal node so we return success. I and C remained in the stack.

I
C

Practice Question (BFS)

1) Initially Queue is empty.

2) Putting the root node A on the queue.

A						

3) Removing A From queue and check whether it is equal to goal node(i.e, J)
   A is not equal to J so we push children of A into the queue.

B	C					

4) Removing B From queue and check whether it is equal to goal node(i.e, J)
   B is not equal to J so we push children of B into the queue.

C	D	E				

5) Removing C From queue and check whether it is equal to goal node(i.e, J)
   C is not equal to J so we push children of C into the queue.

D	E	F	G			

6) Removing D From queue and check whether it is equal to goal node(i.e, J)
   D is not equal to J so we push children of D into the queue. D has no children.

E	F	G				

7) Removing E From queue and check whether it is equal to goal node(i.e, J)
   E is not equal to J so we push children of E into the queue.

F	G	H	I			

8) Removing F From queue and check whether it is equal to goal node(i.e, J)
   F is not equal to J so we push children of F into the queue. F has no children.

G	H	I				

9) Removing G From queue and check whether it is equal to goal node(i.e, J)
G is not equal to J so we push children of G into the queue.

H	I	J	K			

10) Removing H From queue and check whether it is equal to goal node(i.e, J)
H is not equal to J so we push children of H into the queue.

I	J	K	L	M	N	

11) Removing I From queue and check whether it is equal to goal node(i.e, J)
I is not equal to J so we push children of I into the queue.

J	K	L	M	N	O	P

12) Removing J From queue and check whether it is equal to goal node(i.e, J)
J is equal to J, So we return a success message.

And queue will have below elements

K	L	M	N	O	P	