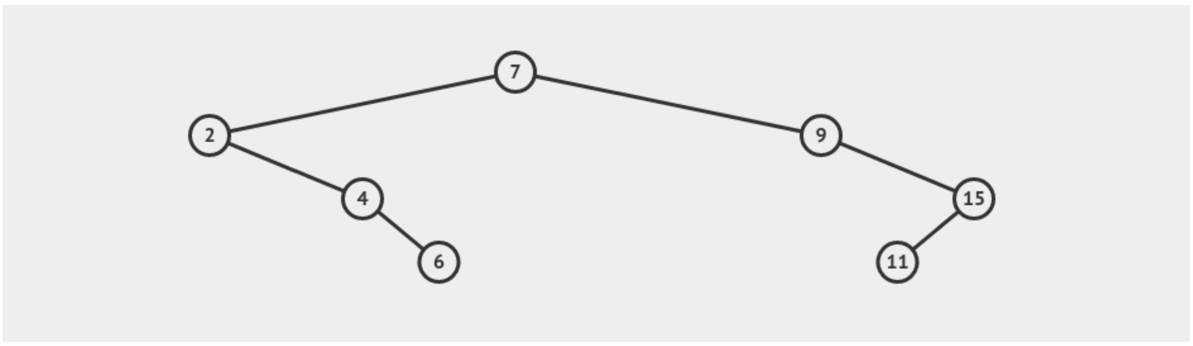
Members::

MIT2021072: Narayan Asati

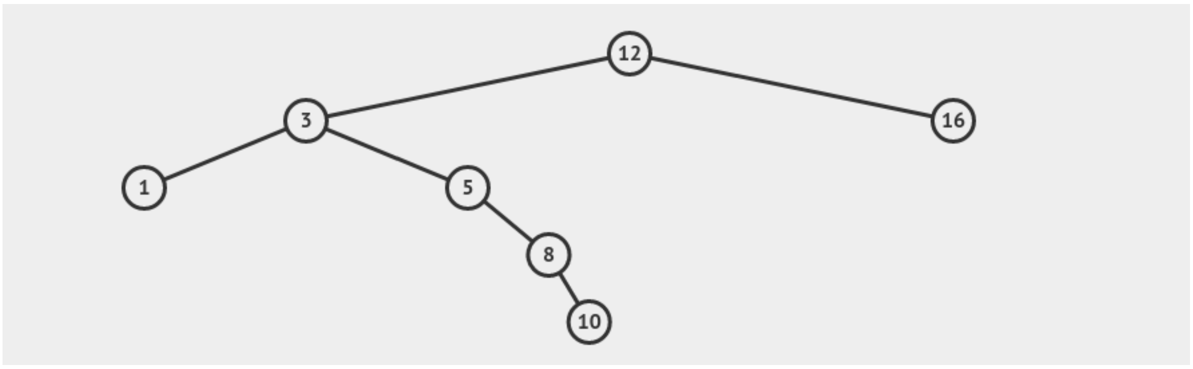
MIT2021117: Chandrakishor Singh

Q1. 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. Since, we are using BST for representing the set, the internal BST representation of the sets A and B will be as follows.

A = {7,2,4,9,15,6,11}



B = {12,3,16,1,5,8,10}



We will use the following node structure for representing the node.

```
Node:
    data: The integer value stored in the node
    left: The pointer to left subtree
    right: The pointer to right subtree
```

We will use following structure for representing the BST.

```
BST:
    head: A pointer to the head of the BST. It is a structure of Node type.
```

Below is the pseudo code for the insert operation for BST.

```
# root: head of BST
# data: data of the new node to be inserted
INSERT(root, data):
    IF root == NULL:
        RETURN Node(data)

    IF (root.data > data)
        root.right = INSERT(root.right, data)
    IF (root.data < data)
        root.left = INSERT(root.left, data)
```

Below is the pseudo code for the *is_member(x)* function for BST.

```
# root: head of BST
# data: data of the node to be searched
IS_MEMBER(root, data):
    IF root == NULL:
        RETURN False

    IF (root.data > data)
        RETURN IS_MEMBER(root.right, data)
    IF (root.data < data)
        RETURN IS_MEMBER(root.left, data)
    RETURN TRUE
```

To merge set A and B, we need to first ensure that A and B are disjoint. Only then merge operation will be possible. Hence, we will iterate over set A and will check whether the current element of set A is present in the set B or not(using *IS_MEMBER()* function). Then, we will iterate over both the sets A and B one-by-one and insert the current element into set C. Its pseudo code is as follows.

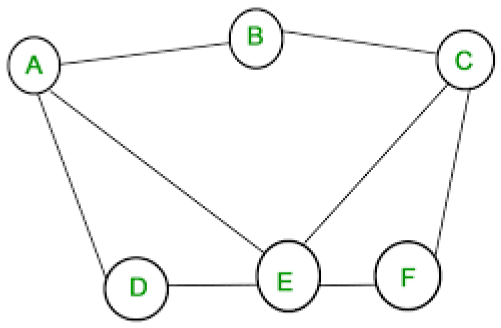
```
# A: Set A
# B: Set B
# rootA: Root of the BST of set A
# rootB: Root of the BST of set B

FOR node IN A:
    IF IS_MEMBER(rootB, node.data)
        PRINT "Sets are not disjoint. Not possible to merge them."
        EXIT

rootC = Create a new BST

FOR node IN A:
    INSERT(rootC, node.data)

FOR node IN B:
    INSERT(rootC, node.data)
```

To display the contents of set C, we will just iterate over C using following inorder traversal pseudo code.

```
# rootC: Root of the BST of set C

INORDER(rootC):
    IF rootC IS NULL:
        RETURN

    INORDER(rootC.left)
    PRINT rootC.data
    INORDER(rootC.right)
```

2. Represent a graph G as a combination of two sets V, E. If only these sets are used, how can DFS be implemented? Explain what additional information besides V and E may be required.

Let’s take an example of the following graph. It is an undirected graph and hence every edge in E will be a bidirectional edge. So, if (u, v) is in E then it means there is an edge from u to v and from v to u.



Here, V = { A, B, C, D, E, F }
E = { (A, B), (B, C), (C, F), (C, E), (E, F), (E, D), (E, A), (D, A) }

In order to implement DFS, there should be some way of knowing all the neighbours of a given node. If we are only given sets of vertices and edges then finding all nodes that are adjacent to a given node will require us to traverse all of the edges. Also an additional array called “visited” will be required to keep track of the vertices that have been visited during the DFS so far.

Hence, for finding all the adjacent nodes of a given node, the time complexity will be O(m), where m is the number of edges in the graph. This is not optimal as in DFS, we have to find the adjacent nodes for all the nodes hence the time complexity of such DFS will be O(mn), where n is the number of vertices.

Therefore, although it is possible to implement DFS using only the set V and E(and the extra “visited” array), it is not the optimal way to do it.

We can instead create an adjacency list which can efficiently give all the neighbours of a given node. Hence, we can first create an adjacency list by traversing all the edges once which will be of O(m) complexity. Then, we can efficiently do DFS which will be of O(m + n) time complexity.

If an undirected graph is given with n nodes, labeled from 1 to n, and m edges then the algorithm for creating an adjacency list is as follows.

```
1. Create a variable called “adjList” which is an array of dynamic arrays.
2. Initialize “adjList” to have n empty array.
3. For edge in E:
       // edge[0], edge[1] represents the end points of the edge
       a. adjList[edge[0]].append(edge[1])
       b. adjList[edge[1]].append(edge[0])
```

Then to do the DFS, we need one additional array called “visited” of size n which will keep track of the vertices that have been visited so far during the DFS traversal. Algorithm of DFS with starting node V is as follows.

DFS(V):

```
1. visited[V] = 1
2. PRINT(V)
3. FOR i in adjList[V]:
       a. IF visited[i] == 0:
              i.  DFS(i)
```

Q5. A transitive reduction of a directed graph G (V, E) is any graph G' with the same vertices but with minimum edges such that the transitive closure of G and G' are the same. How can G' be obtained,if G is given? Write the procedure.

We are assuming the given graph G(V, E) to be a directed acyclic graph and that the graph is represented using adjacency matrix. Here, n represents number of nodes in the graph and m represents the number of edges in the graph. Also, we are assuming that the nodes are labeled from 1 to n.

First we will find the "path matrix". This is a n x n matrix in which the entry at (i, j) will be 1 if and only if there is a path from node i to node j, otherwise it will be 0. It is easy to see that such a matrix will represent the transitive closure of the given graph.

Formally, a transitive closure of a graph G is another graph G1 such that there is an edge between node u, v if there is some path from u to v in graph G.

To find the path matrix, we will use warshall's algorithm. It takes the adjacency matrix as input and modifies it to give another matrix which represents the transitive closure of the graph. Its pseudo code is as follows.

```
# adjMat: adjacency matrix of the graph
# n: number of nodes in graph

TRANS_CLOSURE(adjMat, n):
FOR i = 1 to n:
    FOR j = 1 to n:
        IF i == j:
            SKIP
        IF adjMat[j][i] == 1:
            FOR k = 1 to n:
                IF adjMat[j][k] == 1:
                    adjMat[j][k] = adjMat[i][k]
```

The above algorithm modifies the adjacency matrix and now it represents the transitive closure of the given graph. Its time complexity is cubic.

Now, we need to find the transitive reduction of the given graph G.

Formally, the transitive reduction of the graph G is defined to be another graph, G1 such that the transitive closure of G and G1 are same and there is no other such graph with fewer arcs than G1 which satisfies this condition.

The main idea of finding the transitive reduction is to construct a graph which will have the same reachability as the original graph but has as few edges as possible. Note that if there is an edge from node i to node j and also an edge from node j to k, then we can remove the edge(if it exists) from node i to node k because, the two edges; i -> j and j -> k; are enough to form a path from node i to k. Hence the transitive closure of these two edges will be same as the transitive closure of edges i -> j, j-> k and i-> k.

Therefore, we will first create a graph which will have all the edges as contained in the transitive closure of graph G. Then we will remove the "redundant" edges as explained in the above paragraph to have as few edges as possible. The transitive closure can be obtained from warshall's algorithm as explained above. Then the following procedure will be used to remove the redundant edges in the same manner as explained in previous paragraph.

```
# pathMat: path matrix of the graph obtained via Warshall's algorithm
# n: number of nodes in graph

TRANS_REDUCE(pathMat, n):
    FOR j = 1 to n:
        FOR i = 1 to n:
            IF pathMat[i][j] == 1:
                FOR k = 1 to n:
                    IF pathMat[j][k] == 1:
                        m[i][k] = 0
```

The time complexity of this algorithm is also cubic.

Reference:

1. Harry Hsu. "An algorithm for finding a minimal equivalent graph of a digraph.", Journal of the ACM, 22(1):11-16, January 1975

Q6. Write the algorithm to find the longest path in a Graph and Estimate the complexity of the algorithm.

We assume that the given graph is a weighted directed acyclic graph with no multiple edges and self loops. Also, we are assuming that the weight of every edge of the graph is non-negative.

Since, a cyclic graph will have an "infinite longest path" as we can go through the cycle any number of times. Also, for the same reason, a graph with self loops will suffer from the same problem.

The main steps in this algorithm are as follows.

1. Find the topological order of the vertices.
2. Process the vertices in topological order and find the maximum cost to go to next level in topological order.
3. At the last level of topological ordering, the maximum cost will represent the maximum cost among any path.

A similar procedure can be used for unweighted directed graph. In such case, we can just assume that the cost of each edge is 1.

The detailed algorithm to find the longest path in a weighted directed acyclic graph is as follows. We are using adjacency list representation of the graph here.

```
# adjList: adjacency list of the given graph. each cell of the list has 3 values.
u, v and w which represents a edge from u to v with weight w

# n: number of nodes in the graph

# DFS function
FUNCTION DFS(node, visited, st):
    visited[node] = True
    FOR neighbour in adjList[node]:
        IF visited[neighbour] == False:
            DFS(neighbour, visited, st)
    st.push(node)

# Initialize
st = Create new Stack
visited = Create array of size n
FOR i = 1 to n:
    visited[i] = False

# Apply DFS
FOR i = 1 to n:
    IF visited[i] == False:
        DFS(i, visited, st)

# Store the topological order into an array
topOrder = Create array of size n
FOR i = 0 to n:
    topOrder[i] = st.pop()
```

The time complexity of topological sorting is O(V + E).

Now, after the topological sorting, the vertices are in required order. Now, we will process each node in this order and will find the max cost to reach to the next level of topological order from the set of nodes in the current level of topological order. We will maintain a "distance" array. Its ith cell will represent the max distance/cost to reach to the ith node of the graph from any of the nodes that come before ith node in topological order. Also another array called "parent" will be used. Its ith cell will represent the parent of the ith node. It will help in finding the actual path of the longest/maximum cost path.

Also, two extra variable will be maintained. One called "maxCost" to store the max cost to reach to any vertex and other called "maxCostNode" which will keep track of that maximum cost node.

Its pseudo code is as follows.

```
# adjList: adjacency list of the given graph. each cell of the list has 3 values.
u, v and w which represents a edge from u to v with weight w
# n: number of nodes in the graph
# distance: an array to store the distance to reach to all nodes
# parent: an array to store the parent nodes of all nodes
# maxCost: a variable to store the max cost/distance
# maxCostNode: a variable to store the node with max cost/distance

# initialize
FOR i = 1 to n:
    distance[i] = INF

distance[1] = 0
maxCost = 0
maxCostNode = -1
parent[1] = 1

# find the longest path
FOR i = 1 to n:
    FOR to, weight in adjList[i]:
        IF distance[to] < distance[i] + weight:
            distance[to] = distance[i] + weight
                parent[to] = i
                IF maxCost < distance[to]:
                    maxCost = max(maxCost, distance[to])
                    maxCostNode = to

# print the longest path in reverse order
node = maxCostNode
while parent[node] != node:
    PRINT node
    node = parent[node]
```

The above code prints the longest path in reverse order.

The time complexity of above procedure is O(V + E) as algorithm processes every node of the graph and then finds all of the adjacent nodes as well which is O(E) in worst case.

Hence, the overall time complexity is O(V + E).

Q7. Implement a trie to accept all three letter strings. What is the cost to add an element in this structure? Estimate it.

We are assuming here that the strings would be made up of only small english alphabets. The structure used for representing a node of the trie data structure is as follows.

```
Node:
    isWordEnd: A boolean variable
    arr: an array of Node type
```

The isWordEnd field will contain True if a word(the 3 letter string) ends at that node otherwise False. And the arr field will contain an array of size 26 such that each cell is also a Node of the trie structure.

The node initialization, insertion, search procedures are as follows.

Node initialization procedure:

```
CREATE_NODE():
    node = Create a new Node
    node.isWordEnd = False
    FOR i = 1 to 26:
        node.arr[i] = NULL

    RETURN node
```

Node insertion procedure:

```
INSERT(root, word):
    temp = root

    FOR i = 1 to word.size():
        index = word[i] - 'a'
        IF temp.arr[index] == NULL:
            temp.arr[index] = CREATE_NODE()

        temp = temp.arr[index]

    temp.isWordEnd = True
```

Node search procedure:

```
SEARCH(root, word):
    temp = root

    FOR i = 1 to word.size():
        index = word[i] - 'a'
        IF temp.arr[index] == NULL:
            RETURN False

        temp = temp.arr[index]

    RETURN temp.isWordEnd
```

The pseudo code to insert all three letter string is as follows.

```
root = CREATE_NODE()

FOR i = 'a' to 'z':
    FOR j = 'a' to 'z':
        FOR z = 'a' to 'z':
            insert(root, i + j + z)
```

Time Complexity:

Since, we are only inserting strings of length 3, the depth of trie will only be 3. Hence, at most we need to do 3 comparisons always. Therefore, the time complexity will be constant.

Its C++ implementation is as follows.

```
#include<bits/stdc++.h>
using namespace std;

const int SIZE = 26;

struct node{
    bool endOfWord;
    node* ar[SIZE];
};

node* getNode()
{
    node* n = new node;
    n->endOfWord = false;

    for(int i=0;i<SIZE;i++)
    n->ar[i] = NULL;

    return n;
}

void insert(node *root , string st)
```

```cpp
{
    node *tempRoot = root;

    for(int i=0;i<st.size();i++)
    {
        int index = st[i] - 'a';

        if(tempRoot->ar[index] == NULL)
        tempRoot->ar[index] = getNode();

        tempRoot = tempRoot->ar[index];
    }

    tempRoot->endOfWord = true;
}

bool search(node *root , string st)
{
    node *tempRoot = root;

    for(int i=0;i<st.size();i++)
    {
        int index = st[i] - 'a';

        if(tempRoot->ar[index] == NULL)
        return false;

        tempRoot = tempRoot->ar[index];
    }

    return tempRoot->endOfWord;
}

int main()
{
    node *root = getNode();

    for (int i = 0; i < 26; i++) {
        for (int j = 0; j < 26; j++) {
            for (int z = 0; z < 26; z++) {
                string s = string(1, char(i + 'a')) + string(1, char(j + 'a')) +
string(1, char(z + 'a'));
                insert(root, s);
            }
        }
    }
}
```

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.

We are assuming that the structure of the node of the tree is as follows.

```
Node:
    data: data present in the node
    children: a dynamic array which stores the links of child nodes of this node
```

Since, every node is storing only the data and the links to its children, the total space used for representing the tree will be n + e, where

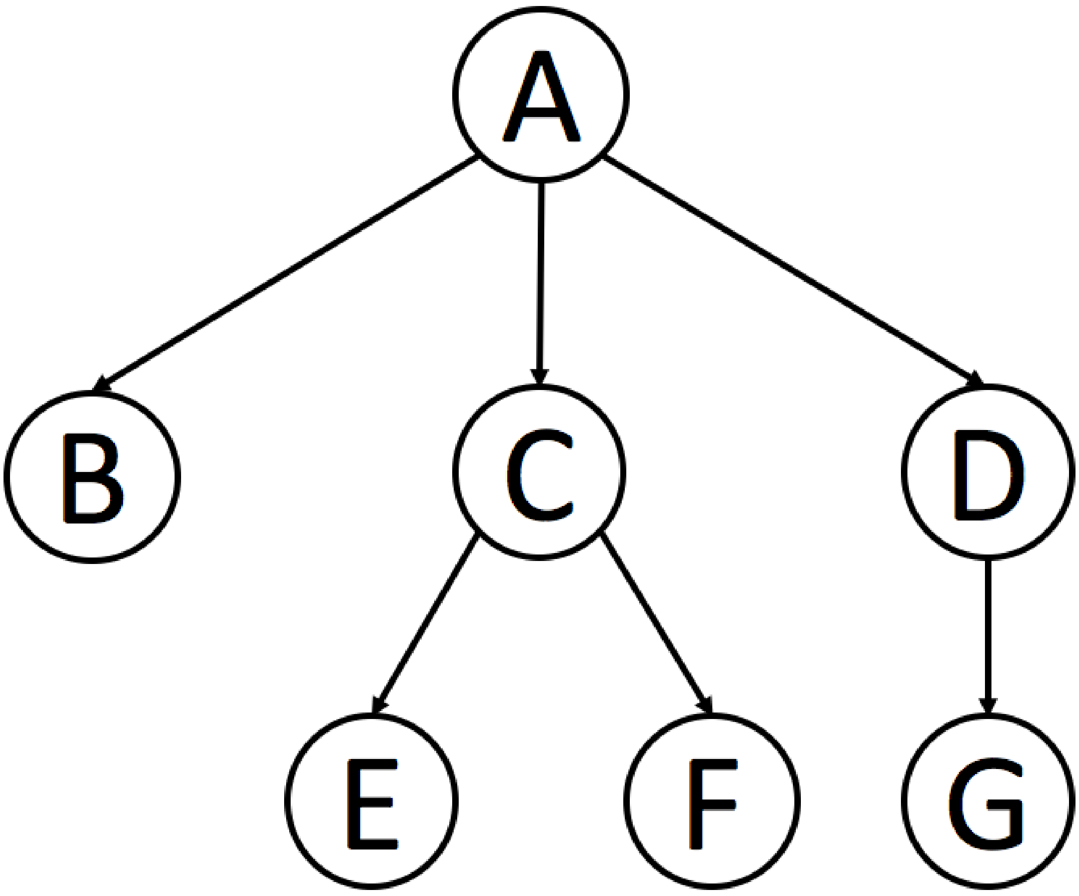
- n: number of nodes
- e: number of edges

Also, in a tree, e = n - 1. Hence the space complexity will be O(n) which is also equal to O(e).

The pseudo code for traversing the tree in BFS manner is as follows.

```
# root: Root of the tree

BFS(root):
    IF root == NULL:
        RETURN

    Q = Create new Queue
    Q.enqueue(root)

    WHILE Q IS NOT EMPTY:
        top = Q.dequeue()
        PRINT top.data

        IF top.children IS NOT EMPTY:
            FOR child in top.children:
                Q.enqueue(child)
```

Example: Let the given general tree be as follows.



The output of BFS traversal will be as follows.

```
A
B
C
D
E
F
G
```