ASSIGNMENT II

COUSE CODE : CSE214

COURSE TITLE :

ALGORITHM

SUBMITTED TO :

SUBROTO NAG PINKU

DEPARTMENT OF CSE,

DAFFODIL INTERNATIONAL UNIVERSITY .

SUBMITTED BY :

SHAZID NAWAS SHOVON

ID : **191-15-12929**

SECTION : O -14

DEPARTMENT OF CSE ,

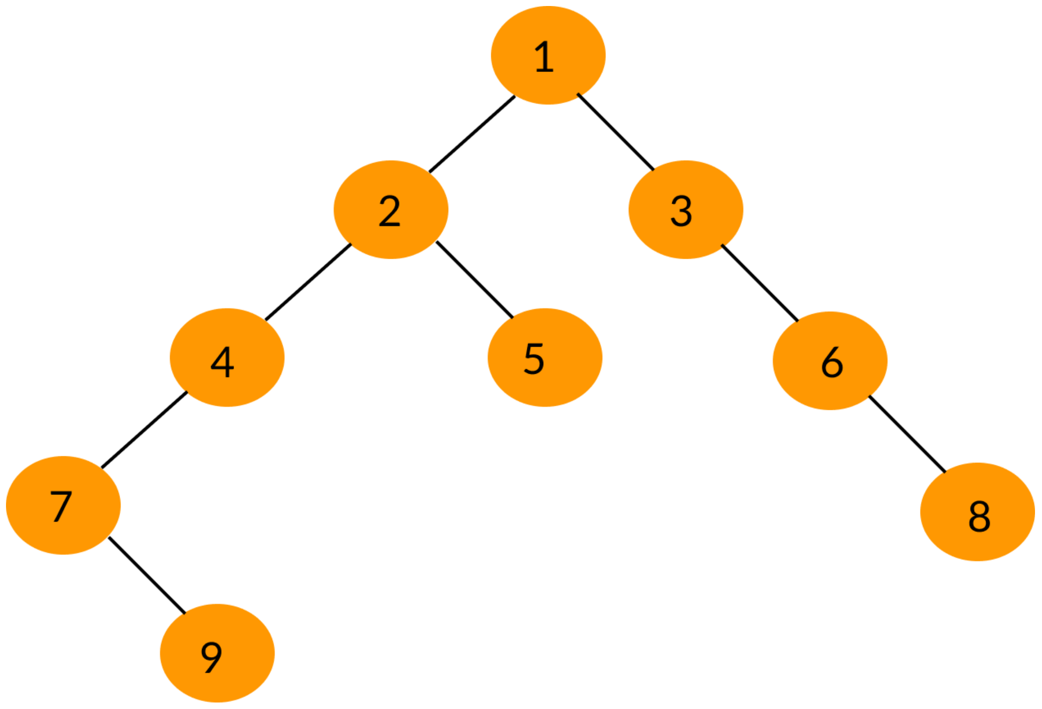
DAFFODIL INTERNATIONAL UNIVERSITY .

Full Tree Traversal

Binary tree traversal can be done in the following ways.

- Inorder traversal
- Preorder traversal
- Postorder traversal

Consider the given binary tree,



Inorder Traversal: 7 9 4 2 5 1 3 6 8

Preorder Traversal: 1 2 4 7 9 5 3 6 8

Postorder Traversal: 9 7 4 5 2 8 6 3 1

Inorder Traversal: For binary search trees (BST), Inorder Traversal specifies the nodes in non-descending order. In order to obtain nodes from BST in non-increasing order, a variation of inorder traversal may be used where inorder traversal is reversed.

Preorder Traversal: Preorder traversal will create a copy of the tree. Preorder Traversal is also used to get the prefix expression of an expression.

Postorder Traversal: Postorder traversal is used to get the postfix expression of an expression given

Algorithm for binary tree traversal

Inorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Visit and print the root node.
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Preorder(root)

- Visit and print the root node.
- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Postorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).
- Visit and print the root node.

Program for binary tree traversals in inorder, preorder, and postorder traversals is given below.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

/* Structure for a node */
struct node
{
int data;
struct node *left;
struct node *right;
};

/* Function to create a new node */
struct node *newNode(int data)
{
struct node *temp = (struct node *) malloc(sizeof(struct node));
temp -> data = data;
temp -> left = NULL;
temp -> right = NULL;
return temp;
};
```

```
/* Function to insert a node in the tree */
void insert_node(struct node *root, int n1, int n2, char lr)
{
if(root == NULL)
return;
if(root -> data == n1)
{
switch(lr)
{
case 'l' :root -> left = newNode(n2);
break;
case 'r' : root -> right = newNode(n2);
break;
}
}
else
{
insert_node(root -> left, n1, n2, lr);
insert_node(root -> right, n1, n2, lr);
}
}

/* Function to print the inorder traversal of the tree */
void inorder(struct node *root)
{
if(root == NULL)
```

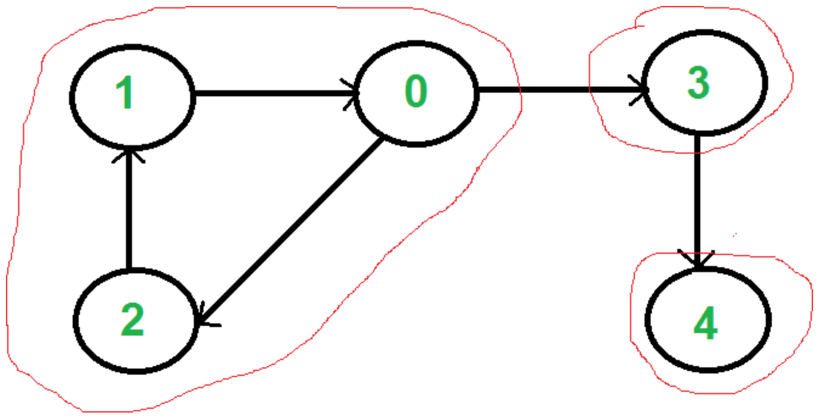
```
return;
inorder(root -> left);
cout << root -> data << " ";
inorder(root -> right); }
/* Function to print the preorder traversal of the tree */
void preorder(struct node *root)
{
if(root == NULL)
return;
cout << root -> data << " ";
preorder(root -> left);
preorder(root -> right);
}

/* Function to print the postorder traversal of the tree */
void postorder(struct node *root)
{
if(root == NULL)
return;
postorder(root -> left);
postorder(root -> right);
cout << root -> data << " ";
}
/* Main Function */
int main()
{
struct node *root = NULL;
```

```
int n;
cout <<"\nEnter the number of edges : ";
cin >> n;
cout << "\nInput the nodes of the binary tree in order \n\nparent-child-
left(or)right-\n\n";
while(n--)
{
char lr;
int n1,n2;
cin >> n1 >> n2;
cin >>lr;
if(root == NULL)
{
root = newNode(n1);
switch(lr)
{
case 'l' :root -> left = newNode(n2);
break;
case 'r' : root -> right = newNode(n2);
break;
}
}
else
{
insert_node(root,n1,n2,lr);
}
}
cout <<"\nInorder Traversal : ";
```

```
inorder(root);
cout << endl;
cout <<"\nPreorder Traversal : ";
preorder(root);
cout << endl;
cout <<"\nPostorder Traversal : ";
postorder(root);
cout << endl;
return 0;
}
```

Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



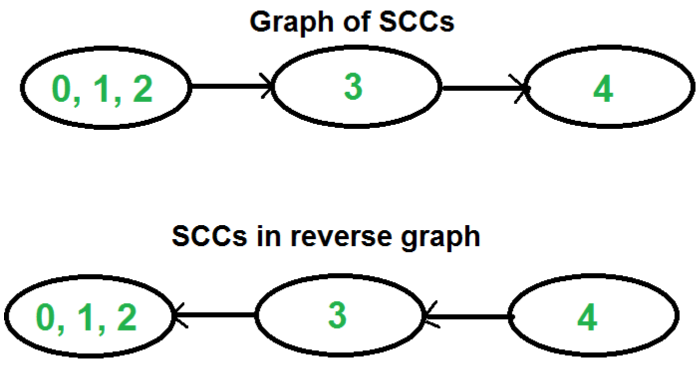
We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.
1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we

get vertices in stack as 1, 2, 4, 3, 0.
2) Reverse directions of all arcs to obtain the transpose graph.
3) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC (See this for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.
In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to

achieve and that is all needed to print SCCs one by one.



Articulation points or cut vertices in a graph

A vertex in an undirected connected graph is an articulation point or cut vertex if and only if removing it, and the edges connected to it, splits the graph into multiple components.

Hint: Apply **Depth First Search** on a graph.

- Construct the DFS tree.
- A node which is visited earlier is a "parent" of those nodes which are reached by it and visited later.
- If any child of a node does not have a path to any of the ancestors of its parent, it means that removing this node would make this child disjoint from the graph. This means that this node is an articulation point.
- There is an exception: the root of the tree. If it has more than one child, then it is an articulation point, otherwise not.

Now for a child, this path to the ancestors of the node would be through a back-edge from it or from any of its children.

Complexity

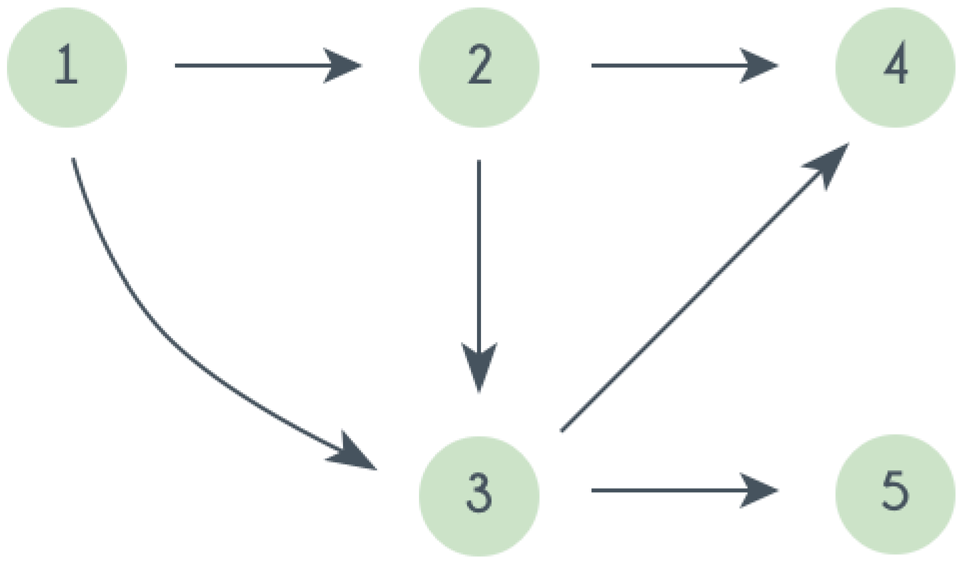
- Worst case time complexity: **Θ(V+E)**
- Average case time complexity: **Θ(V+E)**
- Best case time complexity: **Θ(V+E)**
- Space complexity: **Θ(V)**

Topological sorting

Topological sorting of vertices of a **Directed Acyclic Graph** is an ordering of the vertices v1,v2,...vn

in such a way, that if there is an edge directed towards vertex vj from vertex vi, then vi comes before vj.

For example consider the graph given below:



A topological sorting of this graph is: 1 2 3 4 5
There are multiple topological sorting possible for a graph. For the graph given above one another topological sorting is: 1 2 3 5 4
In order to have a topological sorting the graph must not contain any cycles. In

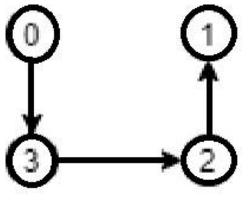
order to prove it, let's assume there is a cycle made of the vertices v1,v2,v3...vn. That means there is a directed edge between vi and vi+1 (1≤i<n) and between vn and v1. So now, if we do topological sorting then vn must come before v1 because of the directed edge from vn to v1. Clearly, vi+1 will come after vi, because of the directed from vi to vi+1, that means v1 must come before vn. Well, clearly we've reached a contradiction, here. So topological sorting can be achieved for only directed and acyclic graphs.

Le'ts see how we can find a topological sorting in a graph. So basically we want to find a permutation of the vertices in which for every vertex vi, all the vertices vj having edges coming out and directed towards vi comes before vi. We'll maintain an array T that will denote our topological sorting. So, let's say for a graph having N vertices, we have an array in_degree[] of size N whose ith element tells the number of vertices which are not already inserted in T and there is an edge from them incident on vertex numbered i. We'll append vertices vi to the array T, and when we do that we'll decrease the value of in_degree[vj] by 1 for every edge from vi to vj. Doing this will mean that we have inserted one vertex having edge directed towards vj. So at any point we can insert only those vertices for which the value of in_degree[] is 0.

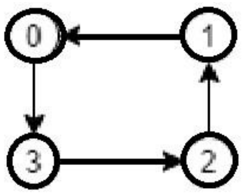
Cycle Finding

A graph cycle is when there is a "loop" or circular reference. This can be a series of edges that connect back to an origin vertex. If a graph has a cycle it is a cyclic graph.

To determine if a graph has a cycle, we can traverse the graph and look for a back edge. A back edge is one that connects a vertex to an already visited ancestor.



Acyclic graph (no back edeges)



Cyclic graph (introducing back edeges)

To detect a cycle in a directed graph (i.e to find a back edge), we can use depth-first search (with some introduction of local state to tell we if a back edge occurs):

[We will maintain 3 buckets of vertices: white, grey, & black buckets. (We can also colour vertices instead)]

• The white bucket will contain all of the unvisited vertices. At the start of our traversal, this means every vertex is in the white bucket.

• Before visiting a vertex, we will move it from the white bucket into the grey bucket.

• After fully visiting a vertex, it will get moved from the grey bucket into the black bucket.

• We can skip over vertices already in the black bucket, if we happen to try and visit them again.

• When visiting the children/descendants of a vertex, if we come to a descendant vertex that is already in the grey bucket - that means we have found a back edge/cycle.

• This means the current vertex has a back edge to it's ancestor - as we only arrived at the current vertex via it's ancestor. So we have just determined that there is more than one path between the two (a cycle). To detect a cycle in an undirected graph, it is very similar to the approach for a directed graph. However, there are some key differences:

• We no longer colour vertices/maintain buckets.

• We have to make sure to account for the fact that edges are bidirectional - so an edge to an ancestor is allowed, if that ancestor is the parent vertex.

• We only keep track of visited vertices (similar to the grey bucket).

• When exploring/visiting all descendants of a vertex, if we come across a vertex that has already been visited then we have detected a cycle. Time complexity is O(v + e) for an adjacency list. Space complexity is O(v). For an adjacency matrix, the time & space complexity would be O(v^2).

Component Finding

Component finding can be processed by using both DFS and BFS. When all nodes are connected, we can go from one node to another node. We will run a loop outside BFS and DFS and will see for every node it is visited or not, otherwise will do BFS or DFS on that node. The result will be how many times we do BFS or DFS.

Here is an example of a tree given below. If we start BFS or DFS from 1 then 1-6 all will be visited, so we don’t need to do BFS or DFS for 2,3,…6. But 7is not connected with any node and is not visited yet. So we will do BFS or DFS for 7. We have to visit all the nodes like this. After 7, all of our nodes will be visited. For 2 components, we do BFS and DFS process for 2 times. This is how the whole process works for tree in finding components .