Explanation

Trees

Non - linear data structure
Hierarchical nature
Binary and n-ary tree
Bst
Different types - full, perfect, complete
Traversals - in, pre, post, level dfs, bfs
A tree is a graph
Balanced binary tree

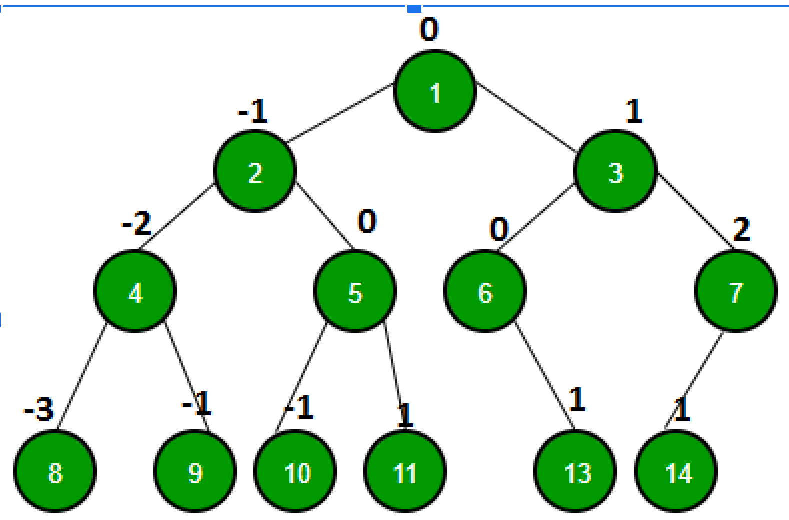
N - nodes n-1 edges => tree (directed)
N - nodes x edges => graph

```
Class node{
Public:
	Int val;
	node* left;
	node* right;
	node(int d)
	{
		Val = d;
		Left  right = NULL;
	}
};
Class tree{
public:
	node* root;
	// FUNCTIONS
};
```

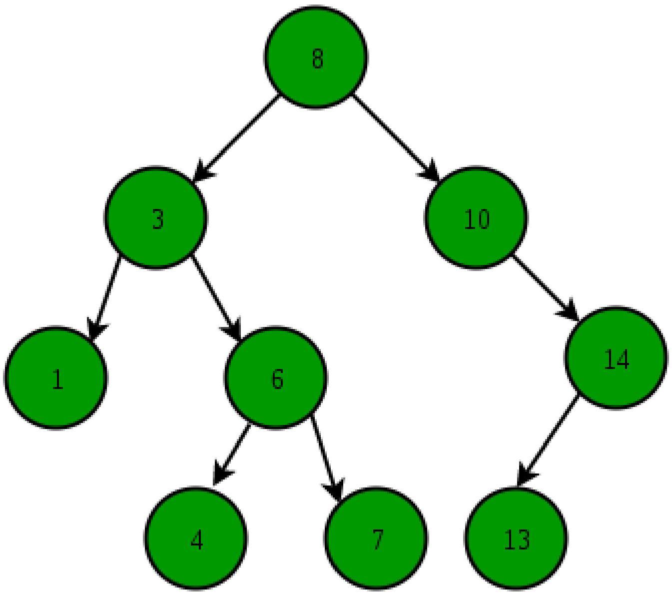
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

```
          1
     2         3
4         5
```

Que. level order and pre order are given, can you construct a unique binary tree



Left view: 1, 2, 4, 8
Right view: 1, 3, 7, 14
Top view: 8, 4, 2, 1, 3, 7
Bottom view: 8, 4, 10, 6, 14, 7



Left: 8, 3, 1, 4
Right: 8, 10, 14, 13
Top: 1, 3, 8, 10, 14
Bottom: 1, 4, 6, 13, 14

Level - 8, 3, 10, 1, 6, 14, 4, 7, 13
Level spiral - 8, 10, 3, 1, 6, 14, 13, 7, 4

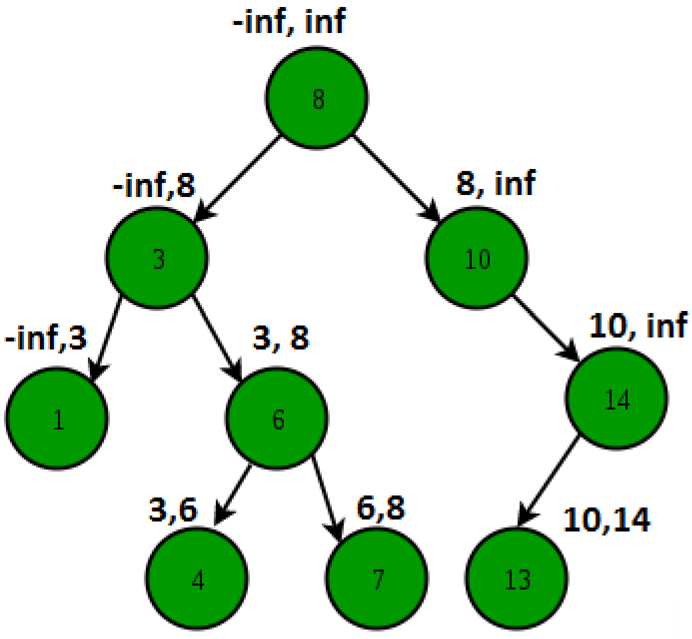
Lca - lowest common ancestor
3 cases -

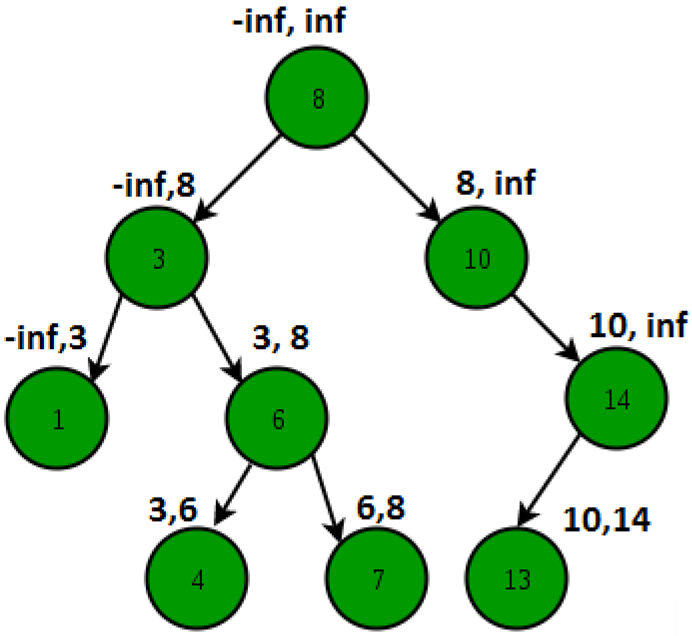
Diameter of tree - 3 cases
Identical or mirror trees

Construct tree from inorder and preorder -
Find root from preorder and then check in inorder left and right subtree and do same for them also

BST - binary search tree
Search
Insert
Delete
Inorder successor
Inorder predecessor

Check if binary tree is bst or not
By using preorder



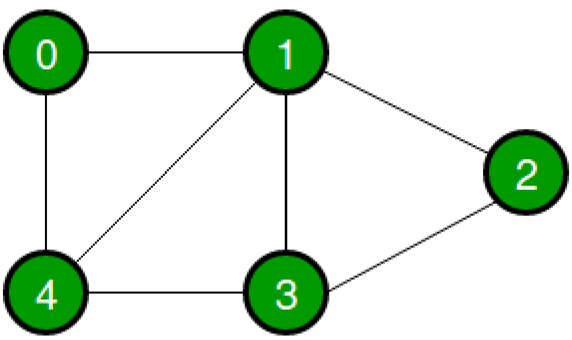


Tree construction
Level order
Pre order 8 3 1 N N 6 4 N N 7 N N 10 N 14 13 N N N

node* root
root->left root->right root->val

N nodes/vertices
Tree - n-1
Graph - (n-1) - (n(n-1)/2)

Matrix = O(v*v)
List = O(e)



Bfs - 0, 4, 1, 3, 2
Dfs - 0, 4, 1, 3, 2

Adj list
0 -> 1,4
1 -> 0,2,3,4
2 -> 1,3
3 -> 1,2,4
4 -> 0,1,3

N - vertices
M - edges

```
vector<vector<int>> v(n+1)
for(int i=0;i<m;i++)
{
	Int x,y;
	cin>>x>>y;
	v[x].push_back(y);
	v[y].push_back(x);
}
```

GRAPHS

Fig. 7.7 A directed graph

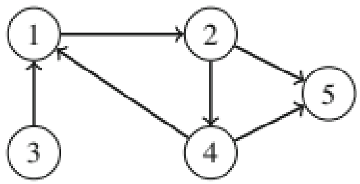


Fig. 7.8 A weighted graph

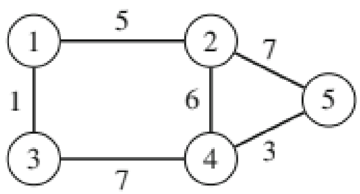


Fig. 7.9 Degrees of nodes

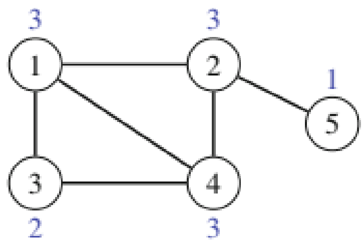


Fig. 7.10 Indegrees and outdegrees

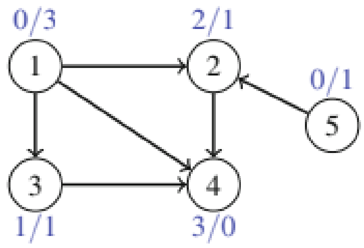
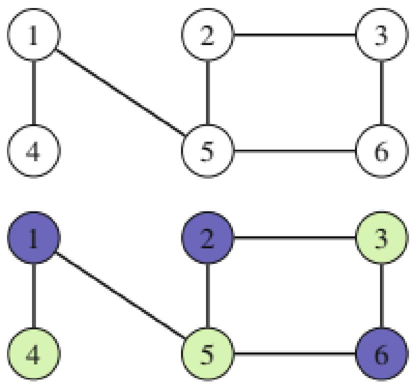


Fig. 7.11 A bipartite graph and its coloring



Connectivity Check A graph is connected if there is a path between any two nodes of the graph. Thus, we can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.

Cycle Detection A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited.

Bipartiteness Check The idea is to pick two colors X and Y , color the starting node X, all its neighbors Y , all their neighbors X, and so on. If at some point of the search we notice that
two adjacent nodes have the same color, this means that the graph is not bipartite.
Otherwise the graph is bipartite and one coloring has been found.

Fig. 7.3 A cycle of three nodes

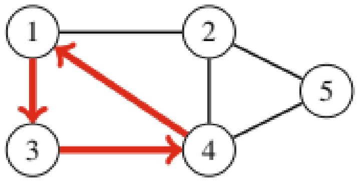


Fig. 7.4 The left graph is connected, the right graph is not

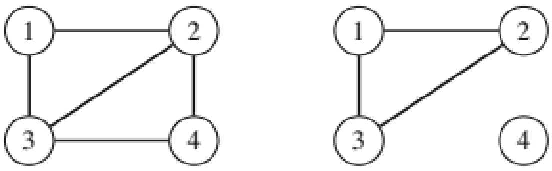


Fig. 7.5 A graph with three components

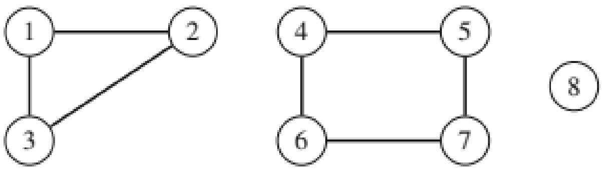
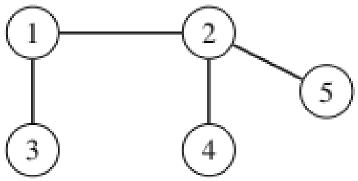


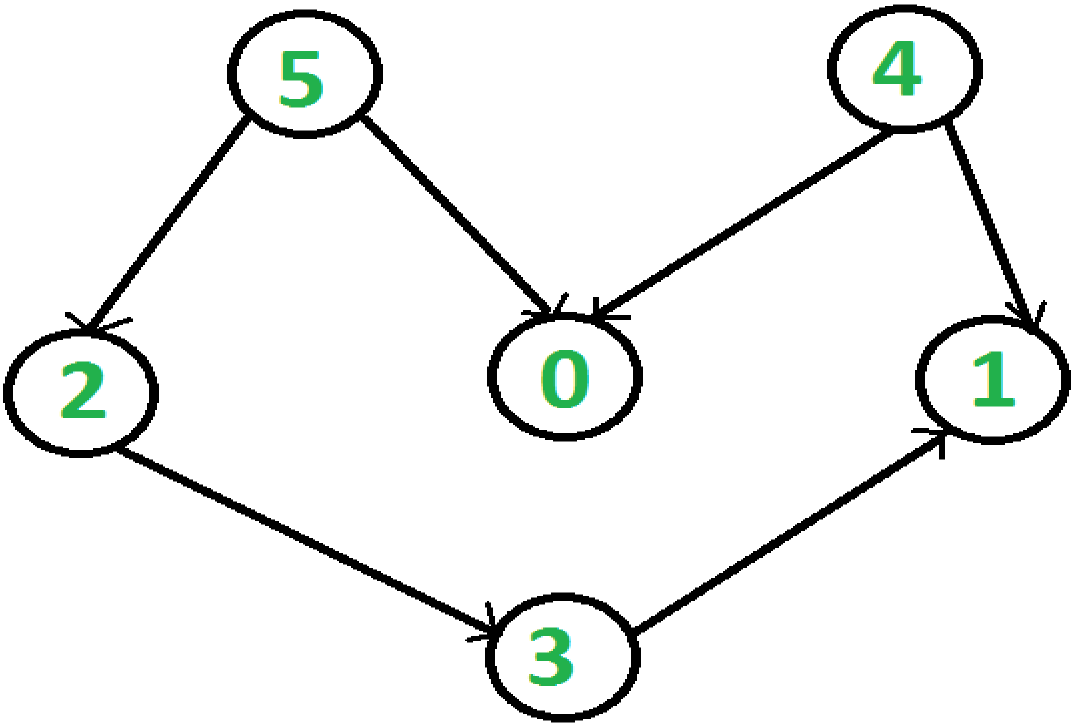
Fig. 7.6 A tree



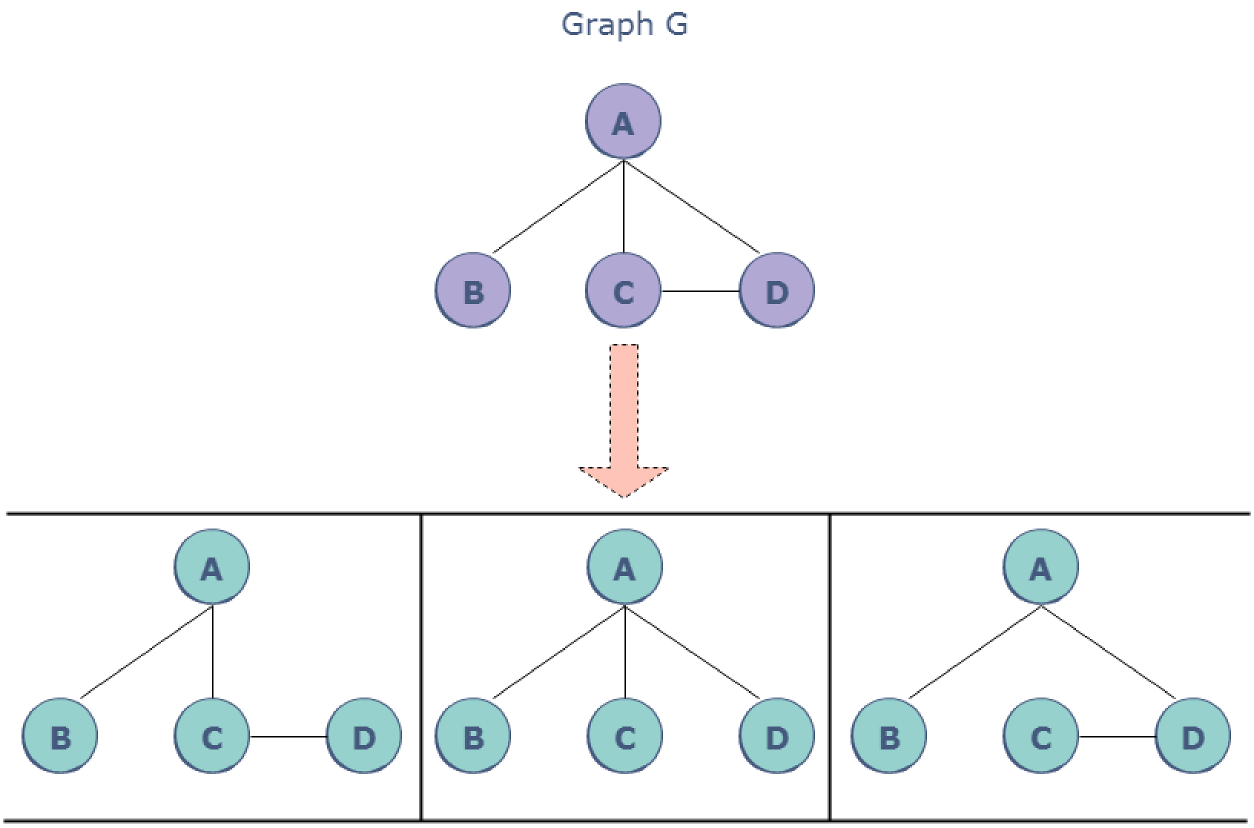
Complete graph - edges = n*(n-1)/2
Indegree, outdegree

https://practice.geeksforgeeks.org/problems/steps-by-knight5927/1
https://practice.geeksforgeeks.org/problems/rat-in-a-maze-problem/1

Topological sort - https://www.geeksforgeeks.org/topological-sorting/



5 4 2 0 3 1
4 5 0 2 3 1
4 5 2 0 3 1



Kruskal - https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

Prim - https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/