3. GRAPHS AND TREES

Data Structures and Algorithms

19ECSC201

SOCSE, KLE TU, HUBBALLI-31
Prakash Hegade

"Grows from the root,
Leaf at the foot,
Data structures fame,
Tree is the name!"
- PH

Graphs

A graph G = (V, E) where V is the set of vertices and E is the set of edges.

A graph in which every edge is directed is called a directed graph or digraph. A graph in which every edge is undirected is called an undirected graph. A graph is said to be mixed graph if some of the edges are undirected and some of the edges are directed.

If there is a maximum of one edge between a pair of nodes in the graph, the graph is called simple graph. The total number of edges leaving a node is called outdegree of that node and the number of edges incident on a node is called indegree of that node. A simple digraph with no cycle is called acyclic graph.

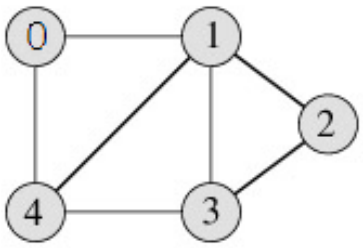
A directed tree is an acyclic graph, which has only one node with indegree 0 and all other nodes having indegree 1. Node with indegree 0 is called root node. All other nodes are reachable from root node. Reachable with only one edge are called children. A node with outdegree 0 is called a leaf node. Root and leaf nodes are called as external nodes and all other nodes are called internal nodes.

Computer Representation of Graphs

A graph can be represented as

- Adjacency Matrix
- Adjacency Linked List

A graph G(V, E)

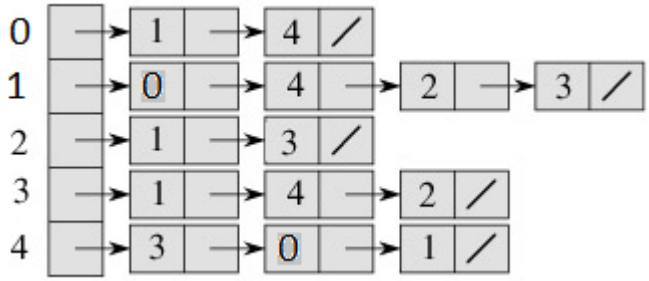


Adjacency Matrix Representation

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

The value is set to 1 if there is a path from one vertex to another, otherwise 0.

Adjacency List Representation



All the vertices attached are added as a node in the linked list. Example node number 0 is attached to 1 and 4.

An adjacency matrix uses O(n*n) memory. It has fast lookups to check for presence or absence of a specific edge, but slow to iterate over all edges.

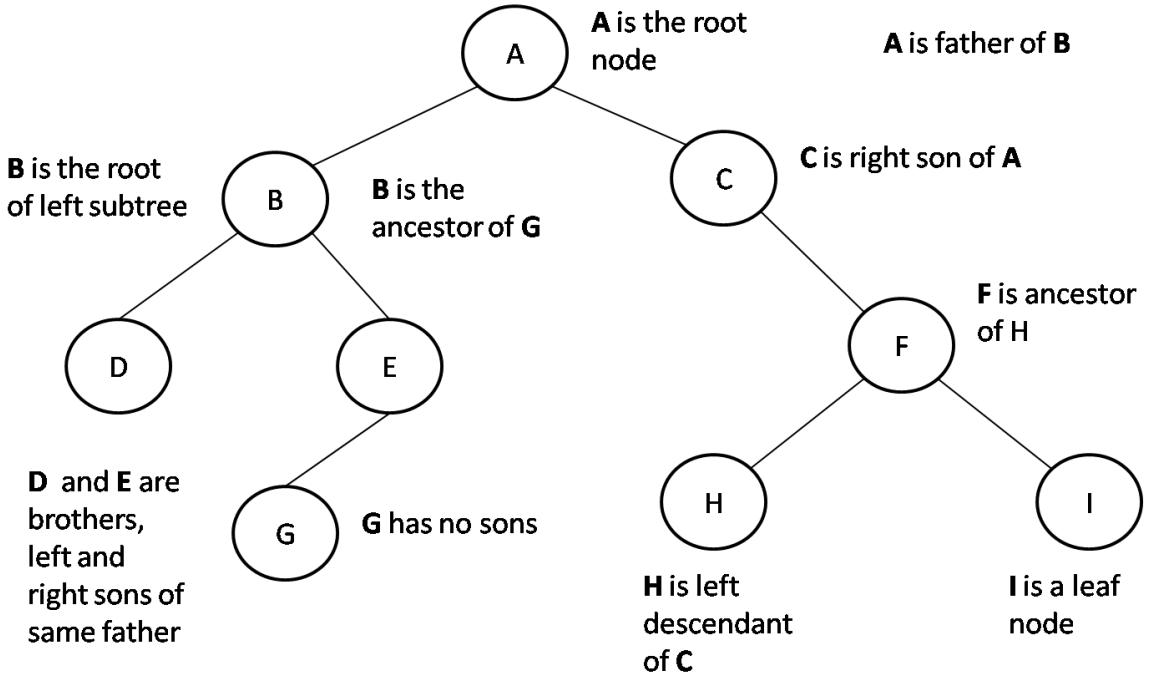
Adjacency lists use memory in proportion to the number edges, which might save a lot of memory if the adjacency matrix is sparse. It is fast to iterate over all edges, but finding the presence or absence specific edge is slightly slower than with the matrix.

Trees and Properties: Binary Tree

A Binary Tree:

- A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets
- The first subset contains a single element called root of the tree
- The other two subsets are themselves binary trees called left and right subtrees of the original tree
- Left or right subtree can be empty
- Each element of a binary tree is called a node of the tree

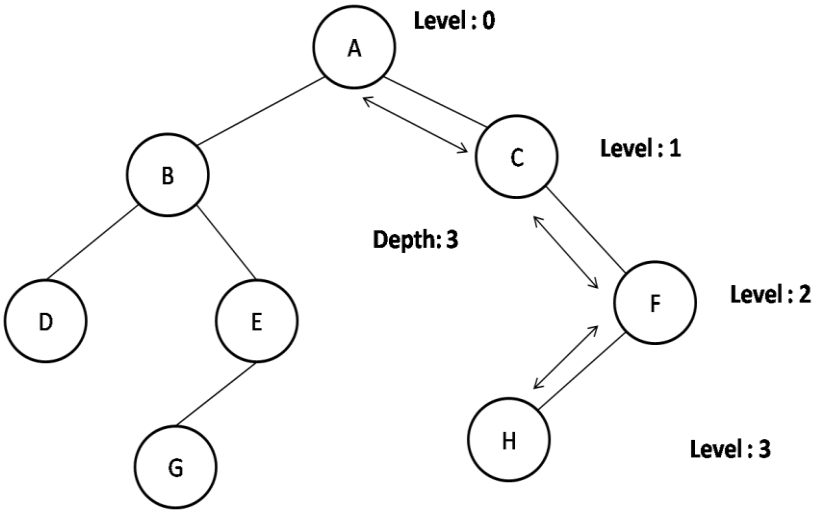
Tree Notions:



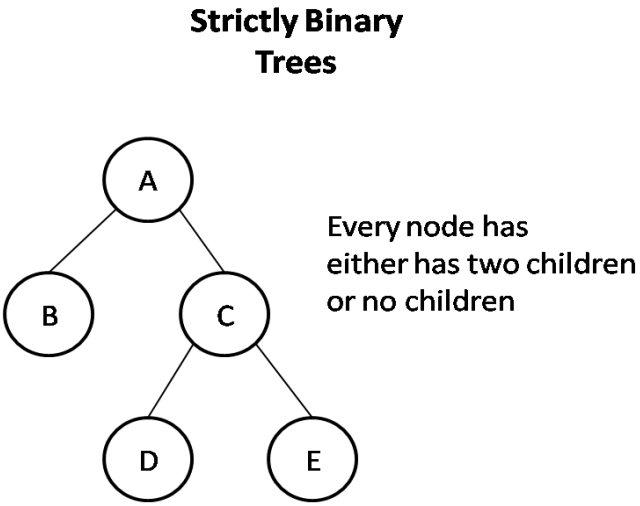
Traversing from leaf nodes to root node is called as climbing the tree and from root to leaf node is called descending the tree.

Levels in a tree: Root is at level 0. Level of any other node is one more than the level of its father.

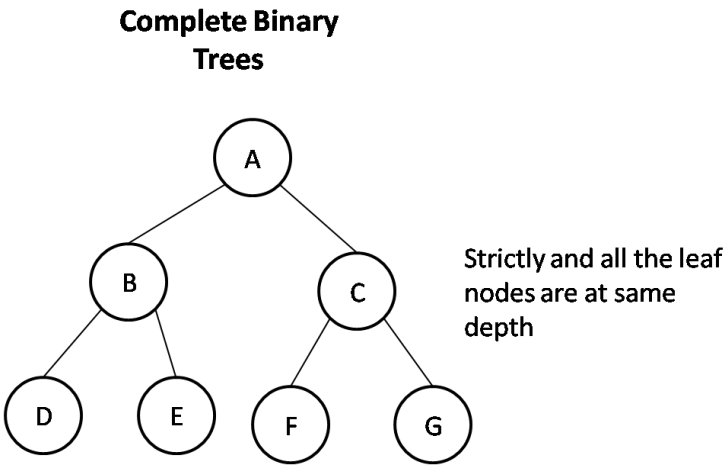
Depth: is maximum level of any leaf in the tree. It is the length of longest path from the root to any leaf.



Strictly Binary Tree: If every non-leaf node in a binary tree has a non-empty left and right subtree, the tree is termed as a strictly binary tree**.**



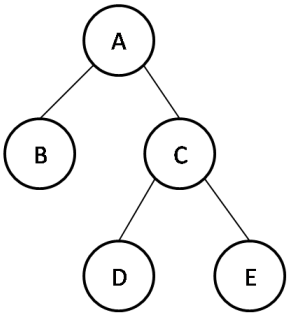
Complete Binary Tree: A complete binary tree of depth d, is the strictly binary tree all of whose leaves are at level d.



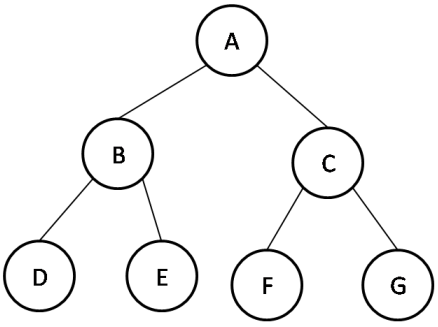
Almost Complete Binary Tree: A binary tree with depth d is an almost complete binary tree if,

- Any node **'n'** at level less than d-1 has 2 sons
- For any node **'n'** in the tree with a right descendant at level d, **'n'** must have a left son and every left descendant of **'n'** is either a leaf at level d or has two sons.

Almost Complete Binary Trees



Leaf nodes are at level d or d-1



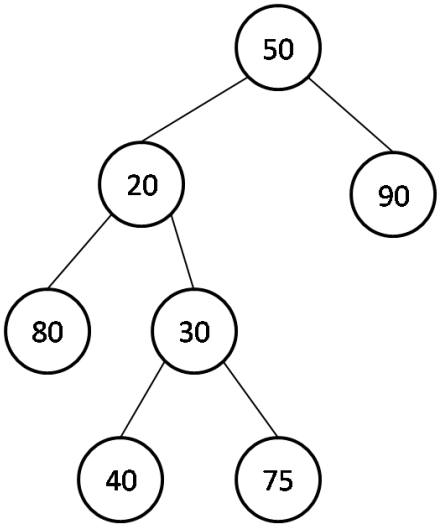
A right subtree/node can be present only if all its left subtrees/node is present

Tree Traversals

The binary tree can be traversed in following manners:

In-order: Traverse in Left, Root and Right
Pre-order: Traverse in Root, Left and Right
Post-order: Traverse in Left, Right and Root

Traverse the given tree in all the orders:

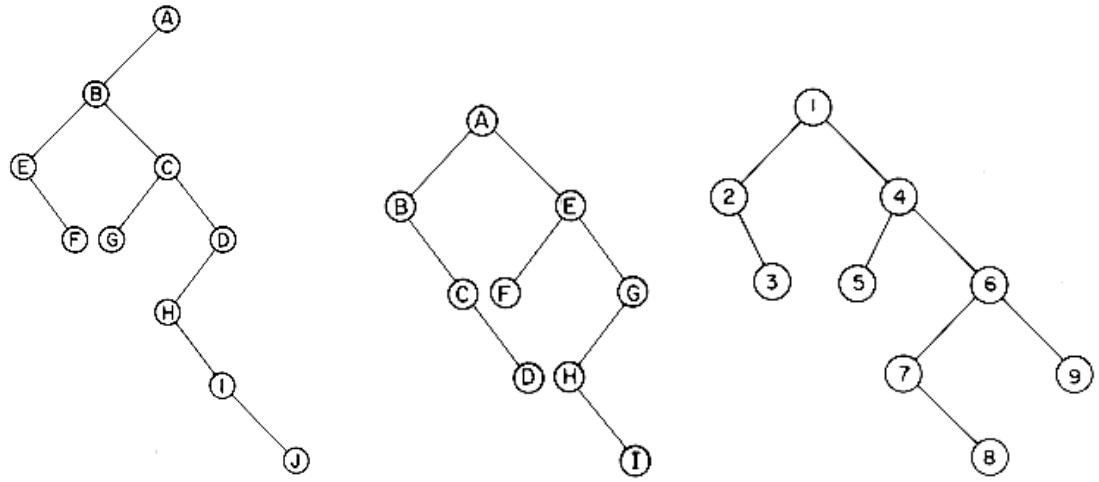


Inorder: 80 20 40 30 75 50 90

Preorder: 50 20 80 30 40 75 90

Postorder: 80 40 75 30 20 90 50

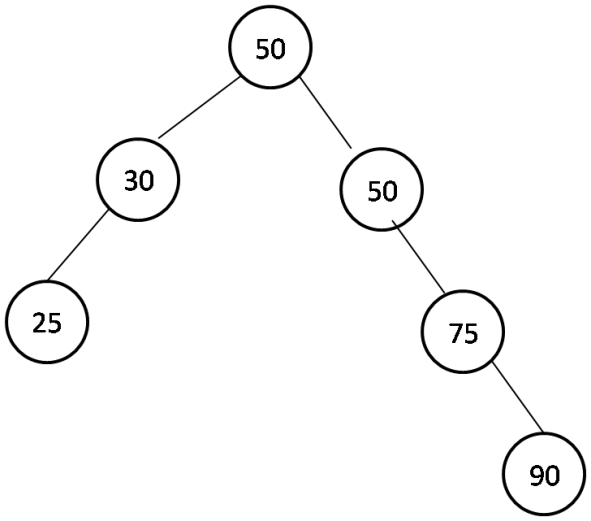
Write traversal for following trees:



Note: Refer class notes for more examples

Binary Search Tree (BST)

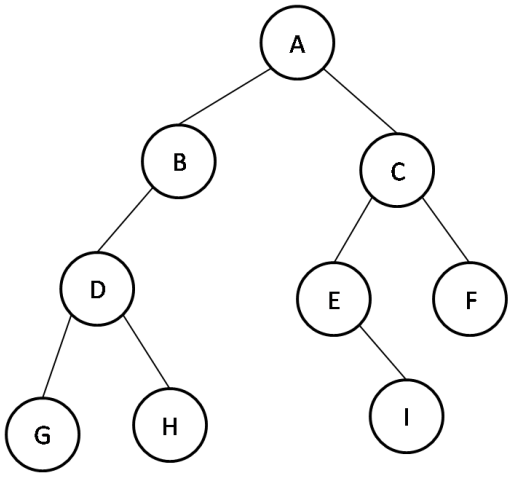
BST is created by taking first incoming element as root node and all the further items lesser than root are moved to left and greater than or equal to right side of the tree. Construct a binary search tree for: 50, 30, 50, 25, 75, 90



Exercise: Construct Binary search tree for following set of numbers:

- 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5
- 100, 350, 275, 275, 100, 117, 245, 356, 287, 100, 275
- B I N A R Y S E A R C H T R E E

Traverse the following tree in inorder, preorder and postorder:



Inorder: G D H B A E I C F

Preorder: A B D G H C E I F

Postorder: G H D B I E F C A

Note: refer class notes for more examples

Binary Search Tree Implementation

```
#include<stdio.h>
#include<stdlib.h>
struct tree
{
        int data;
        struct tree *left;
        struct tree *right;
};
typedef struct tree TREE;

TREE * insert_into_bst(TREE *, int);
void inorder(TREE *);
void preorder(TREE *);
void postorder(TREE *);
TREE * delete_from_bst(TREE *, int);

/*
Function Name: insert_into_bst
Input Params:  Root of the tree and data item to be inserted
Return Type:   Updated root of the tree
Description:   Inserts a node into a binary search tree at
               appropriate position
*/

TREE * insert_into_bst(TREE * root, int data)
{
        TREE *newnode,*currnode,*parent;
        // Dynamically allocate the memory using malloc
        newnode=(TREE*)malloc(sizeof(TREE));

        // Check if the memory allocation was successful
        if(newnode==NULL) {
```

```
		printf("Memory allocation failed\n");
		return NULL;
	}

	// Initialize the tree node elements
	newnode->data = data;
	newnode->left = NULL;
	newnode->right = NULL;

	// When the first insertion happens which is the root node
	if(root == NULL) {
		root = newnode;
		printf("Root node inserted into tree\n");
		return root;
	}

	// Traverse through the desired part of the tree using
	// currnode and parent pointers
	currnode = root;
	parent = NULL;
	while(currnode != NULL) {
		parent = currnode;
		if(newnode->data < currnode->data)
			currnode = currnode->left;
		else
			currnode = currnode->right;
	}

	// Attach the node at appropriate place using parent
	if(newnode->data < parent->data)
		parent->left = newnode;
	else
		parent->right = newnode;

	// print the successful insertion and return root
	printf("Node inserted successfully into the tree\n");
	return root;
}

/*
Function Name: inorder
Input Params:  Root of the tree
Return Type:   void
Description:   Recursively visits the tree in the order of
               Left, Root, Right
*/
void inorder(TREE *troot)
{
	if(troot != NULL)
	{
		inorder(troot->left);
		printf("%d\t",troot->data);
		inorder(troot->right);
	}
}
```

```
/*
Function Name: preorder
Input Params:  Root of the tree
Return Type:   void
Description:   Recursively visits the tree in the order of
               Root, Left, Right
*/
void preorder(TREE *troot)
{
        if(troot != NULL)
        {
                printf("%d\t",troot->data);
                preorder(troot->left);
                preorder(troot->right);
        }
}


/*
Function Name: postorder
Input Params:  Root of the tree
Return Type:   void
Description:   Recursively visits the tree in the order of
               Left, Right, Root
*/
void postorder(TREE *troot)
{
        if(troot != NULL)
        {
                postorder(troot->left);
                postorder(troot->right);
                printf("%d\t",troot->data);
        }
}



/*
Function Name: delete_from_bst
Input Params:  Root of the tree, item data to be deleted
Return Type:   Updated root of the tree
Description:   Deletes the specified data and re-adjusts the
               tree structure according to bst tree constraints
*/

TREE * delete_from_bst(TREE * root, int data)
{
    TREE * currnode, *parent, *successor, *p;
    // Check if the tree is empty
    if(root == NULL) {
        printf("Tree is empty\n");
        return root;
    }
```

```
    // Traverse and reach the appropriate part of the tree
    parent = NULL;
    currnode = root;
    while (currnode != NULL && data != currnode->data)
    {
        parent = currnode;
        if(data < currnode->data)
            currnode  = currnode->left;
        else
            currnode = currnode->right;
    }

    // If the data is not present in the tree
    if(currnode == NULL)  {
        printf("Item not found\n");
        return root;
    }

    // Check and manipulate if either left subtree is absent,
    // or right subtree is absent
    // or both are present
    if(currnode->left == NULL)
        p = currnode->right;
    else if (currnode->right == NULL)
        p = currnode->left;
    else
    {
         // Process of finding the inorder successor
        successor = currnode->right;
        while(successor->left != NULL)
            successor = successor->left;

        successor->left = currnode->left;
        p = currnode->right;
    }

    // The case of root deletion
    if (parent == NULL) {
        free(currnode);
        return p;
    }

    if(currnode == parent ->left)
        parent->left = p;
    else
        parent->right = p;

    free(currnode);
    return root;
}

int main() {
       TREE * root;
       root = NULL;
       int choice = 0, data = 0, count = 0;
```

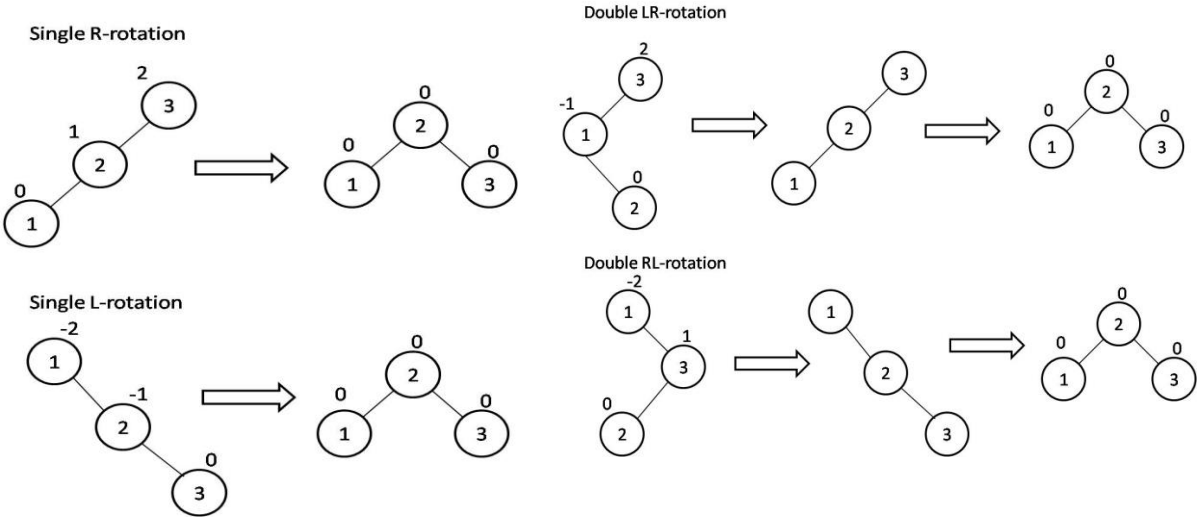
```
        while(1) {
         printf("\n******** Menu ************\n");
         printf("1-Insert into BST\n");
         printf("2-Inorder Traversal\n");
         printf("3-Preorder Traversal\n");
         printf("4-Postorder Traversal\n");
         printf("5-Delete from BST\n");
         printf("Any other option to exit\n");
         printf("*****************************\n");

         printf("Enter your choice\n");
         scanf("%d", &choice);

         switch(choice)
         {
             case 1: printf("Enter the item to insert\n");
                     scanf("%d", &data);
                     root = insert_into_bst(root, data);
                     break;

             case 2: if(root == NULL)
                         printf("Tree is empty\n");
                     else {
                         printf("Inorder Traversal is...\n");
                         inorder(root);
                     }
                     break;

             case 3: if(root == NULL)
                         printf("Tree is empty\n");
                     else {
                         printf("Preorder Traversal is...\n");
                         preorder(root);
                     }
                     break;

             case 4: if(root == NULL)
                         printf("Tree is empty\n");
                     else {
                         printf("Postorder Traversal is...\n");
                         postorder(root);
                     }
                     break;

             case 5: printf("Enter the item to be deleted\n");
                     scanf("%d", &data);
                     root = delete_from_bst(root, data);
                     break;
             default: printf("Exciting Code.\n");
                      exit(0);
          }
        }
        return 0;
}
```

AVL Trees

Properties:

- AVL tree is named after Adelson-Velskii and E. M. Landis
- A self-balancing binary search tree, and it was the first such data structure to be invented
- In an AVL tree, the heights of the two child sub-trees of any node differ by 0, 1 or -1 (called as balance factor)
- When balance factor is not 1, 0 or -1 rebalancing is done to restore this property.
- Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where *n* is the number of nodes in the tree prior to the operation.

Tree Rotations:

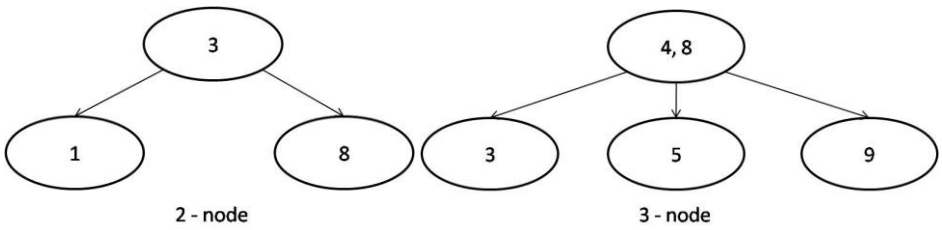


Refer class notes for more examples.

2-3 Trees

Properties

- A 2-3 Tree has either a
  - 2- node (2 children with 1 data key present) or it has a
  - 3-node (3 children with 2 data keys present)



- Tree is constructed bottom-up
- If the number of keys present in node equals 3, we promote middle key as parent and split the remaining to 2 children nodes

Note: Refer class notes for examples on AVL and 2-3 trees

Application of Trees

Trees in Future Semesters:

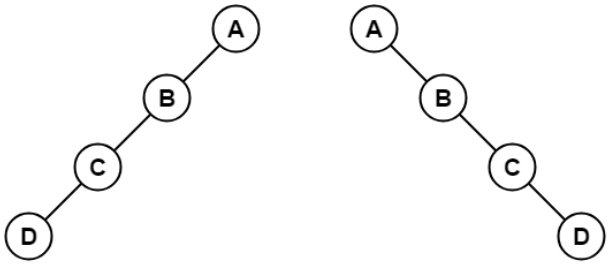
- Used in indexing databases – B trees (studied in Database Management System Course)
- Routing Algorithms (Studied in Networking Courses)
- File Systems (Studied in Operating Systems course)
- Compiler uses Abstract Syntax Tree and evaluation trees (Studied in Principles of Compiler Design course)

Tree Applications:

- As a data structure to search and sort the data
- Tree is the best data structure to store data which is **hierarchical** in nature
- Binary Search Tree - Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered
- Huffman Coding Tree - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats
- The organization of Morse code is a binary tree
- Look out! There is more and lot more!

Other Definitions

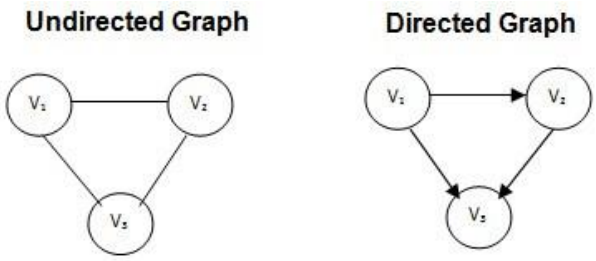
1. Skewed Binary Trees



Left Skewed Binary Tree Right Skewed Binary Tree

In a left skewed tree, most of the nodes have the left child without corresponding right child. In a right skewed tree, most of the nodes have the right child without corresponding left child.

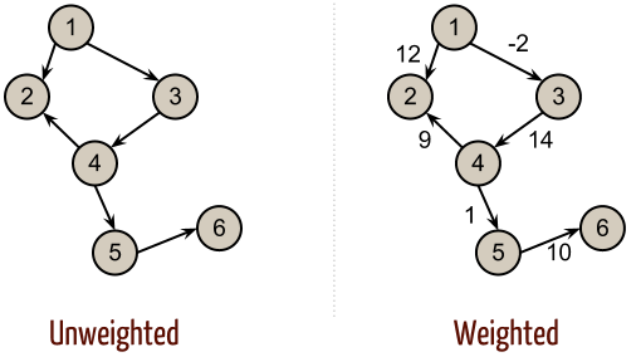
2. Directed and Undirected Graphs



An undirected graph is graph, i.e., a set of objects that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an undirected network. In contrast, a graph where the edges point in a direction is called a directed graph.

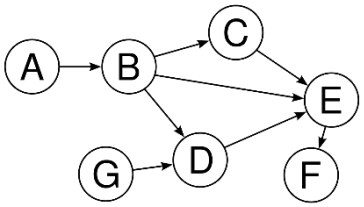
3. Weighted and Unweighted Graphs

A weighted graph refers to an edge-weighted graph that is a graph where edges have weights or values. Without the qualification of weighted, the graph is typically assumed to be unweighted.

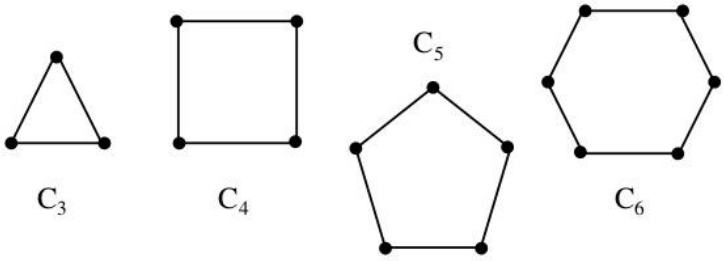


4. Cyclic and Acyclic Graphs

A directed acyclic graph (DAG) is a finite directed graph with no directed cycles. That is, it consists of finitely many vertices and edges, with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again.

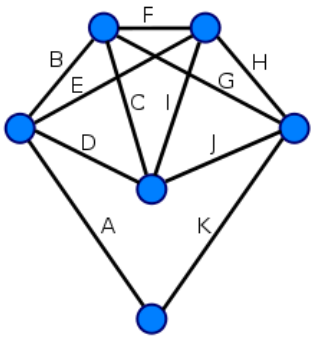


A cycle graph or circular graph is a graph that consists of a single cycle, or in other words, some number of vertices connected in a closed chain.



5. Euler Graph

An Eulerian cycle, Eulerian circuit or Euler tour in an undirected graph is a cycle that uses each edge exactly once.

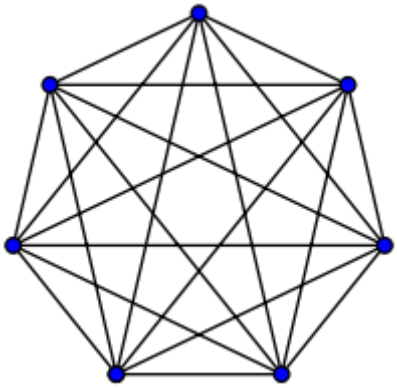


Every vertex of above graph has an even degree. Therefore, this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

The Königsberg Bridges is not Eulerian, therefore, a solution does not exist.

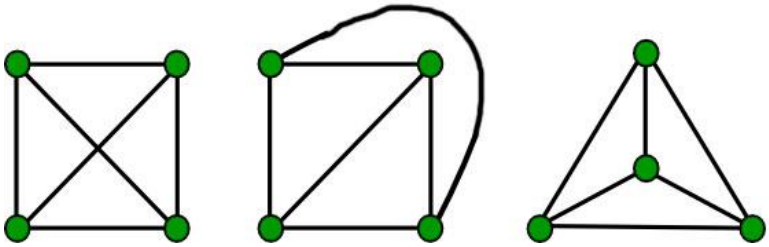
6. Complete Graph

A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.



7. Planar Graph

A planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.



~*~*~*~*~*~*~*~*~