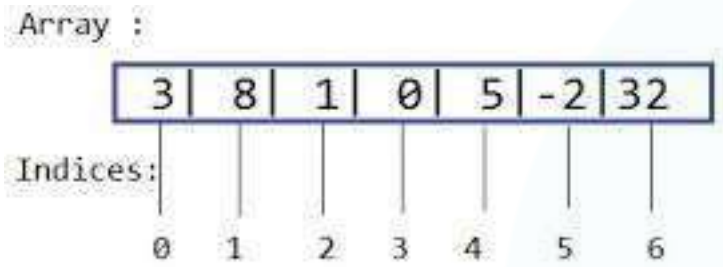
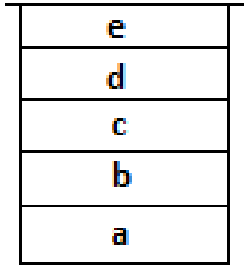
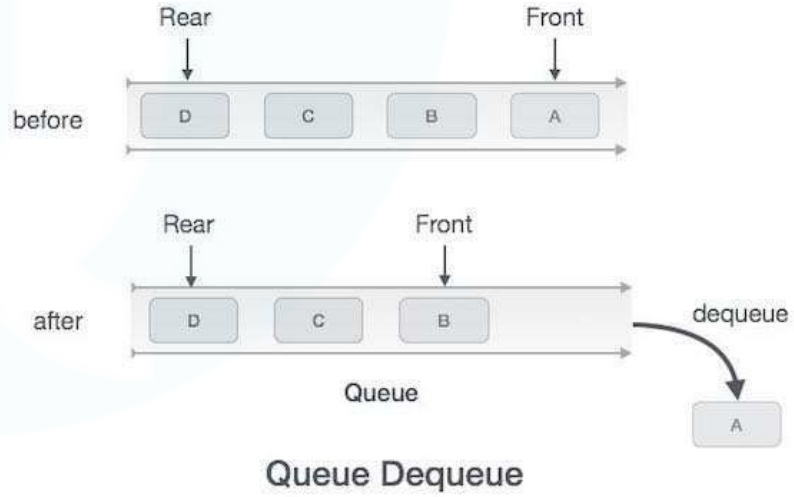
Tree

So far we discussed Linear data structures like





stac
k



Introduction to trees

- So far we have discussed mainly linear data structures – strings, arrays, lists, stacks and queues
- Now we will discuss a non-linear data structure called tree.
- Trees are mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents.
- Consider a parent-child relationship

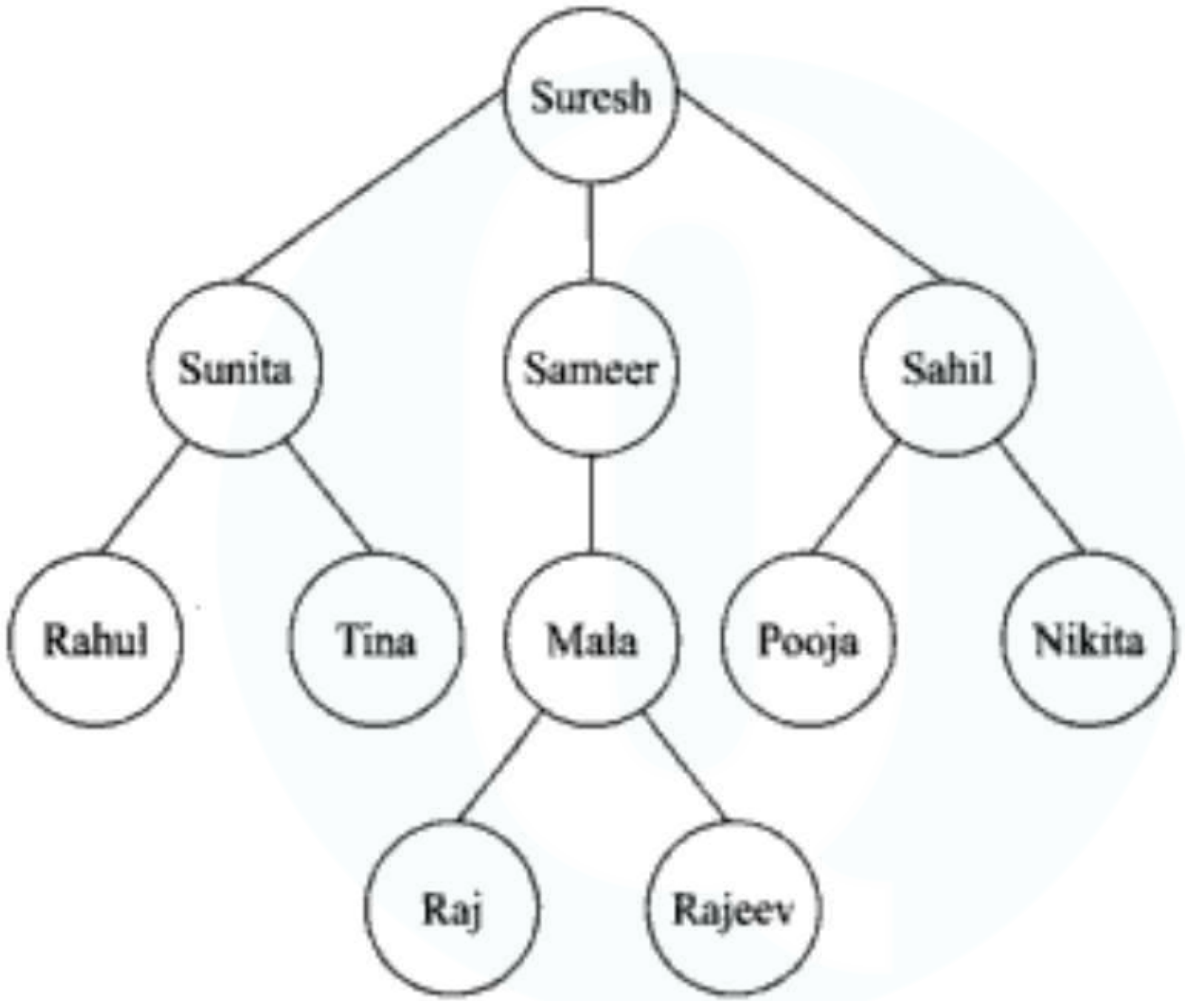
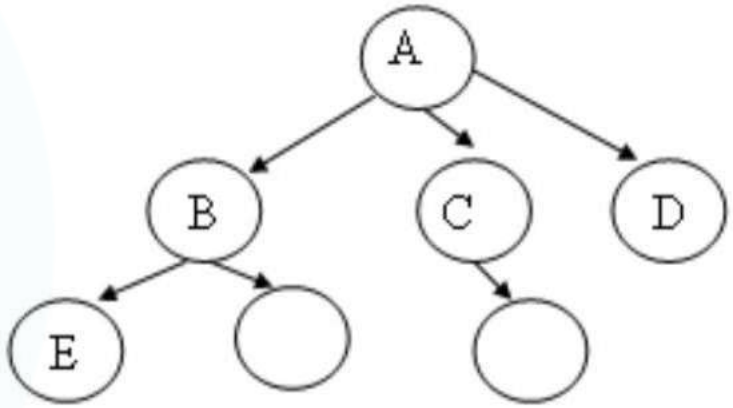


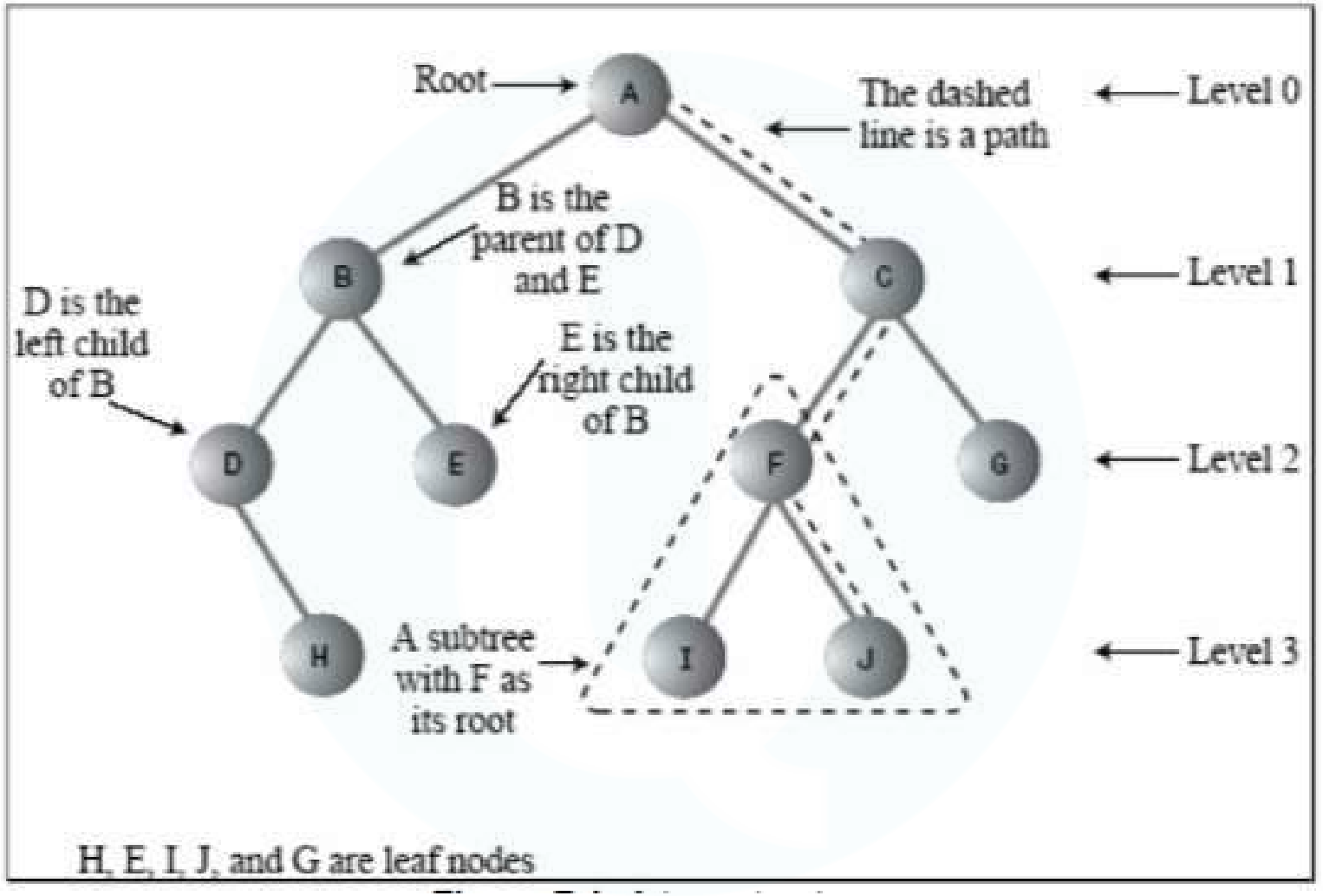
Fig. 8.1 *A Hypothetical Family Tree*

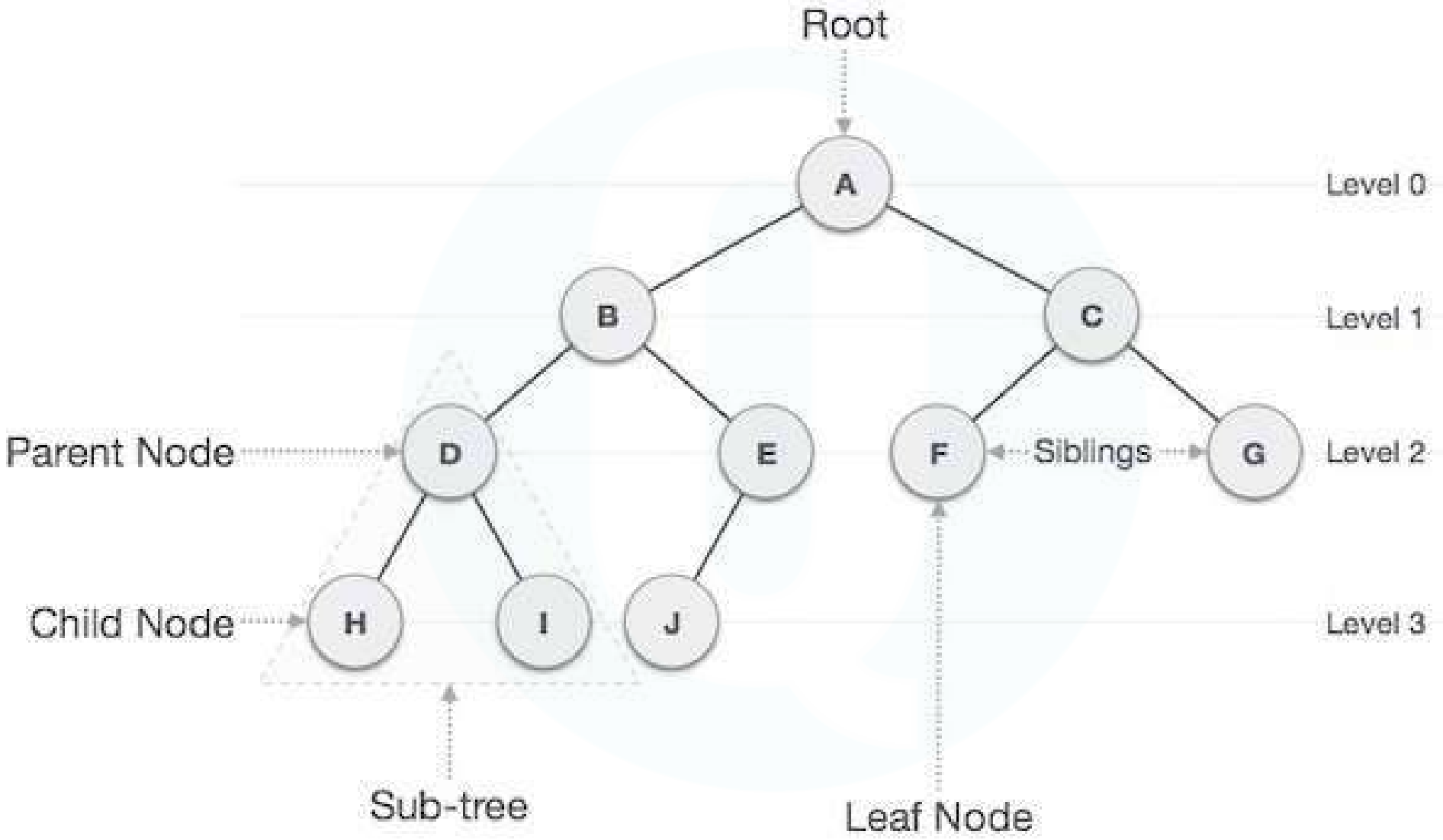
Tree

- A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
  - Tree is a sequence of nodes
  - There is a starting node known as a root node
  - Every node other than the root has a parent node.
  - Nodes may have any number of children



A has 3 children, B, C, D
A is parent of B



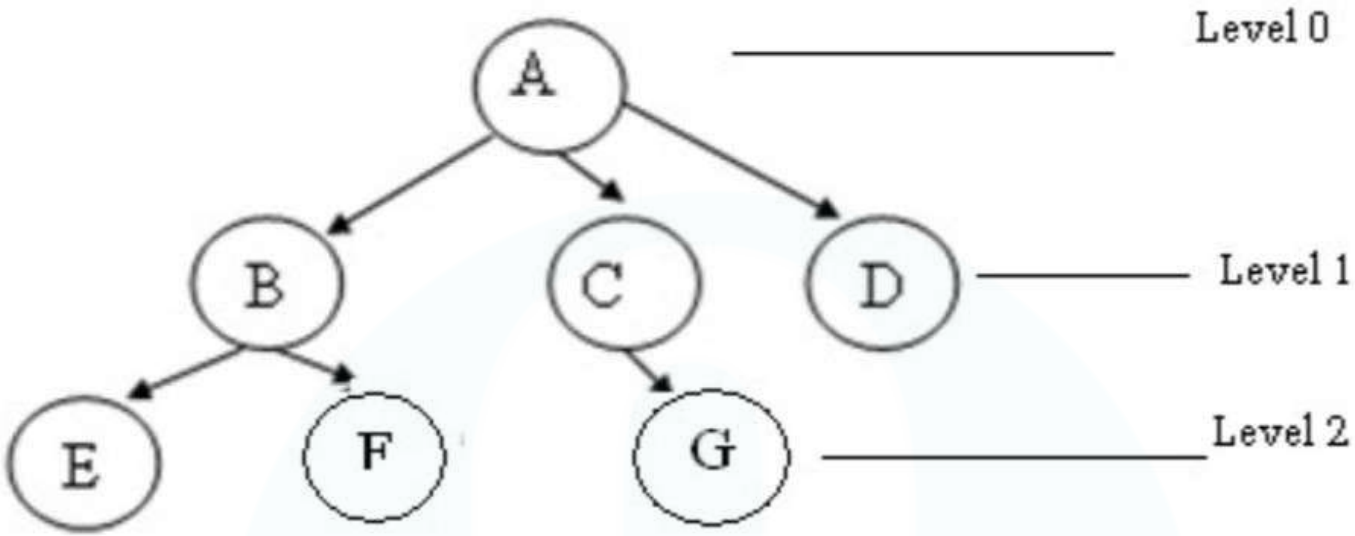


Some Key Terms:

- Root − Node at the top of the tree is called root.
- Parent − Any node except root node has one edge upward to a node called parent.
- Child − Node below a given node connected by its edge downward is called its child node.
- Sibling – Child of same node are called siblings
- Leaf − Node which does not have any child node is called leaf node.
- Sub tree − Sub tree represents descendants of a node.
- Levels − Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

Some Key Terms:

- Degree of a node:
  - The degree of a node is the number of children of that node
- Degree of a Tree:
  - The degree of a tree is the maximum degree of nodes in a given tree
- Path:
  - It is the sequence of consecutive edges from source node to destination node.
- Height of a node:
  - The height of a node is the max path length form that node to a leaf node.
- Height of a tree:
  - The height of a tree is the height of the root
- Depth of a tree:
  - Depth of a tree is the max level of any leaf in the tree



- ✔ A is the root node
- ✔ B is the parent of E and F
- ✔ D is the sibling of B and C
- ✔ E and F are children of B
- ✔ E, F, G, D are external nodes or leaves
- ✔ A, B, C are internal nodes
- ✔ Depth of F is 2
- ✔ the height of tree is 2
- ✔ the degree of node A is 3
- ✔ The degree of tree is 3

Characteristics of trees

- Non-linear data structure
- Combines advantages of an ordered array
- Searching as fast as in ordered array
- Insertion and deletion as fast as in linked list
- Simple and fast

Application

- Directory structure of a file store
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- used in compression algorithms, such as those used by the .jpeg and .mp3 file- formats.

Introduction To Binary Trees

- A binary tree, is a tree in which no node can have more than two children.
- Consider a binary tree T, here ‘A’ is the root node of the binary tree T.
- ‘B’ is the left child of ‘A’ and ‘C’ is the right child of ‘A’
  - i.e A is a father of B and C.
  - The node B and C are called siblings.
- Nodes D,H,I,F,J are leaf node

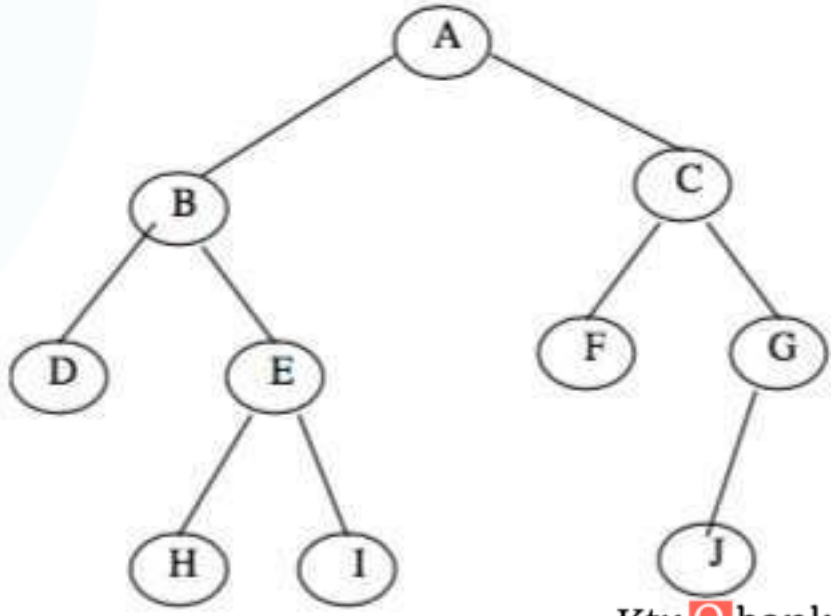
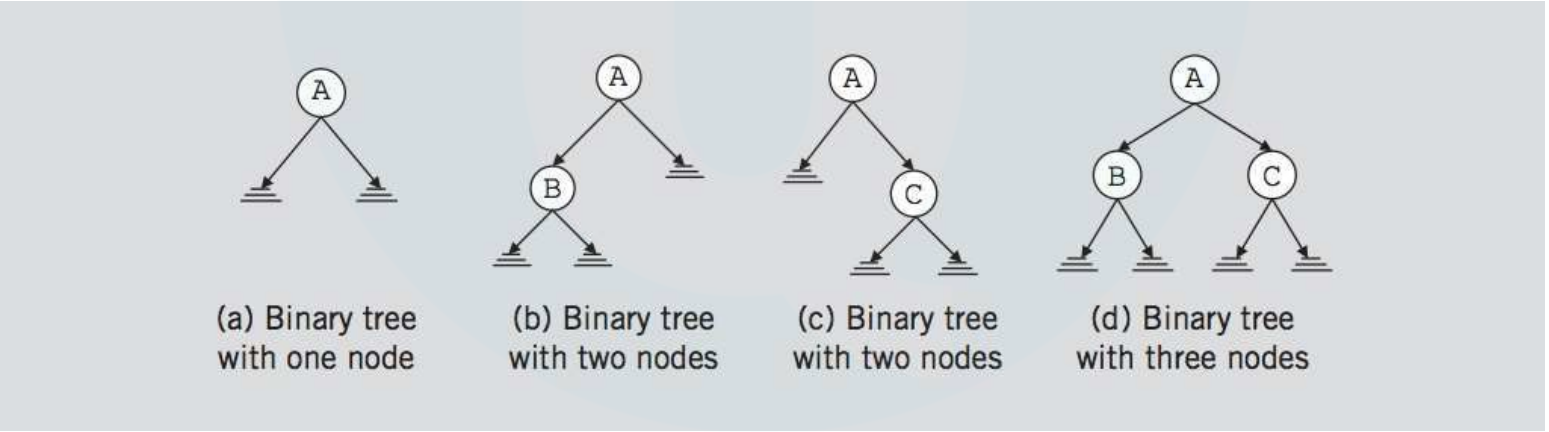


Fig. 8.3. Binary tree

Binary Trees

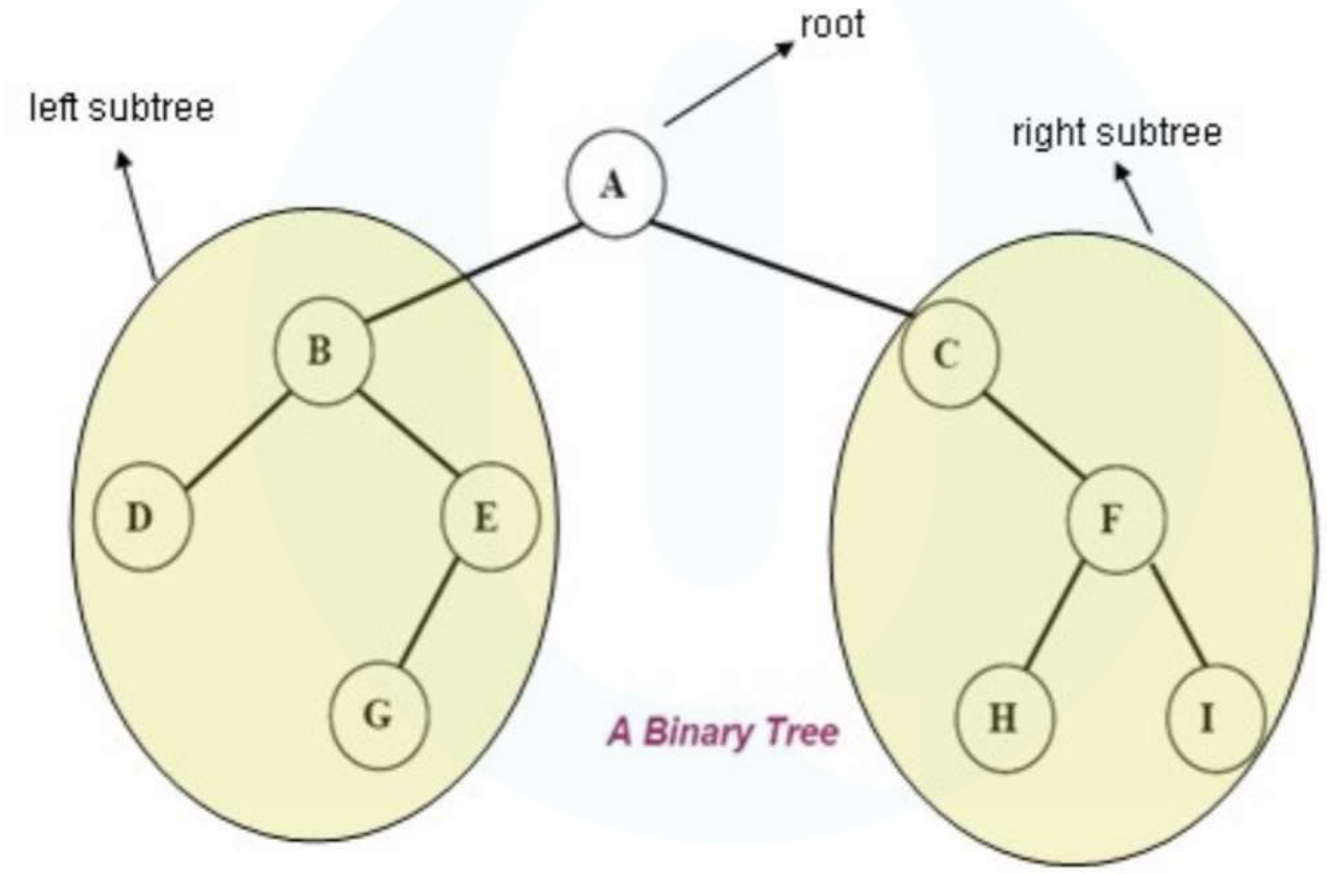
- A binary tree, *T*, is either empty or such that
  - I. *T* has a special node called the root node
  - II. *T* has two sets of nodes L_T and R_T, called the left subtree and right subtree of
  - III. *T*, respectively.
  - . L_T and R_T are binary trees.



Binary Tree

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

The following figure shows a binary tree with 9 nodes where A is the root

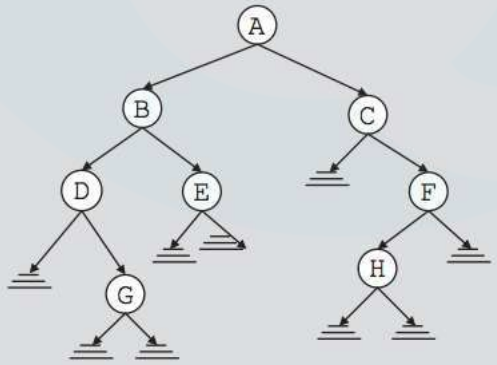


Binary Tree

- The root node of this binary tree is A.

- The left sub tree of the root node, which we denoted by L_A, is the set $L_A = \{B,D,E,G\}$ and the right sub tree of the root node, R_A is the set $R_A = \{C,F,H\}$

- The root node of L_A is node B, the root node of R_A is C and so on



Binary Tree Properties

- If a binary tree contains m nodes at level L, it contains at most 2m nodes at level L+1

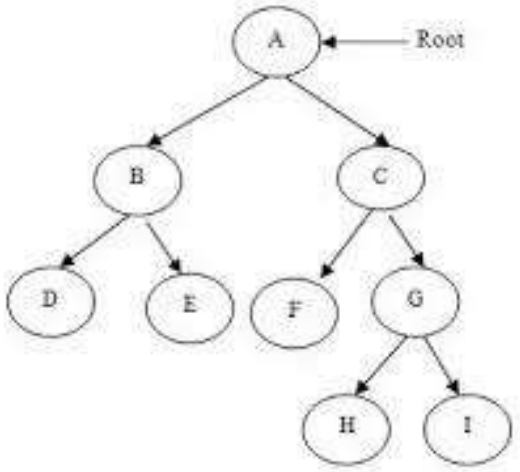
- Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most 2L nodes at level L.

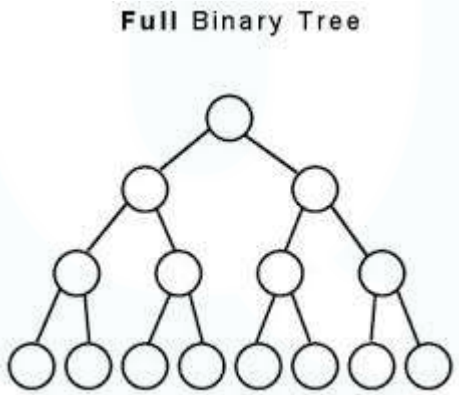
Types of Binary Tree

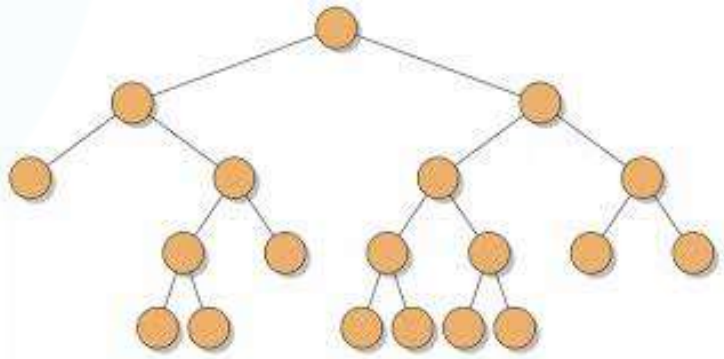
- Full binary tree
- Complete binary tree
- Perfect binary tree

Full binary tree

- If every node has 0 or 2 children.
- A full binary tree is a binary tree in which all nodes except leaf nodes have two children.
- In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1
- L = I + 1 , Where L = Number of leaf nodes, I = Number of internal nodes



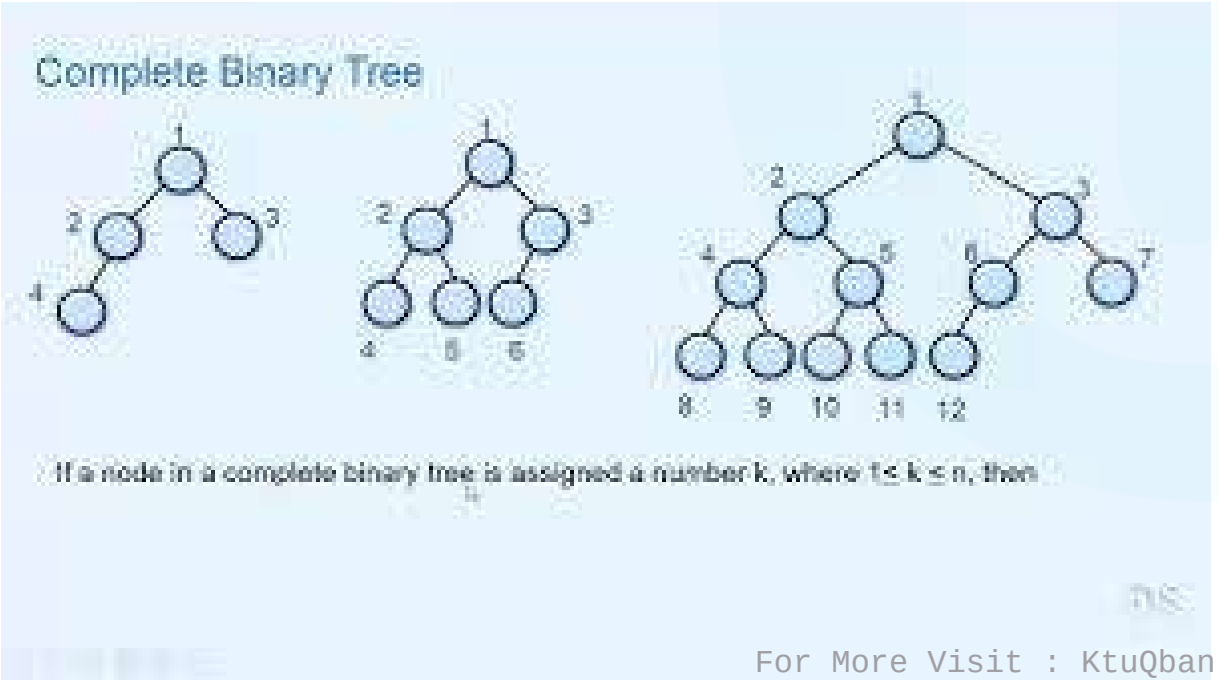


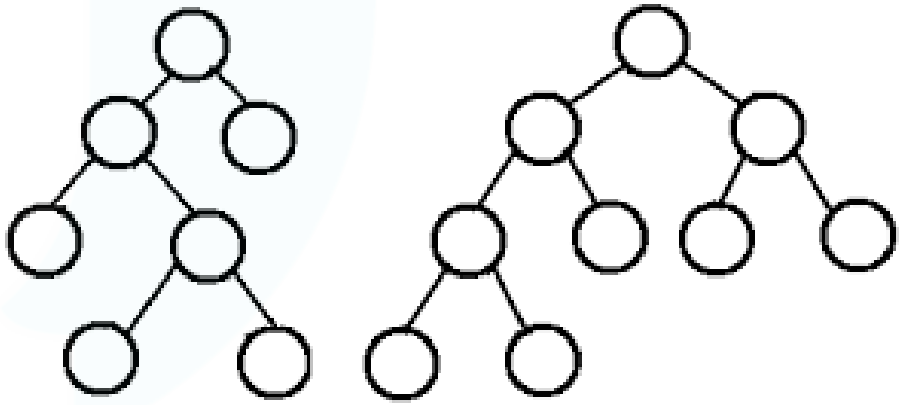


Complete binary tree

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- A *complete binary tree* of depth d is called strictly binary tree if all of whose leaves are at level d.



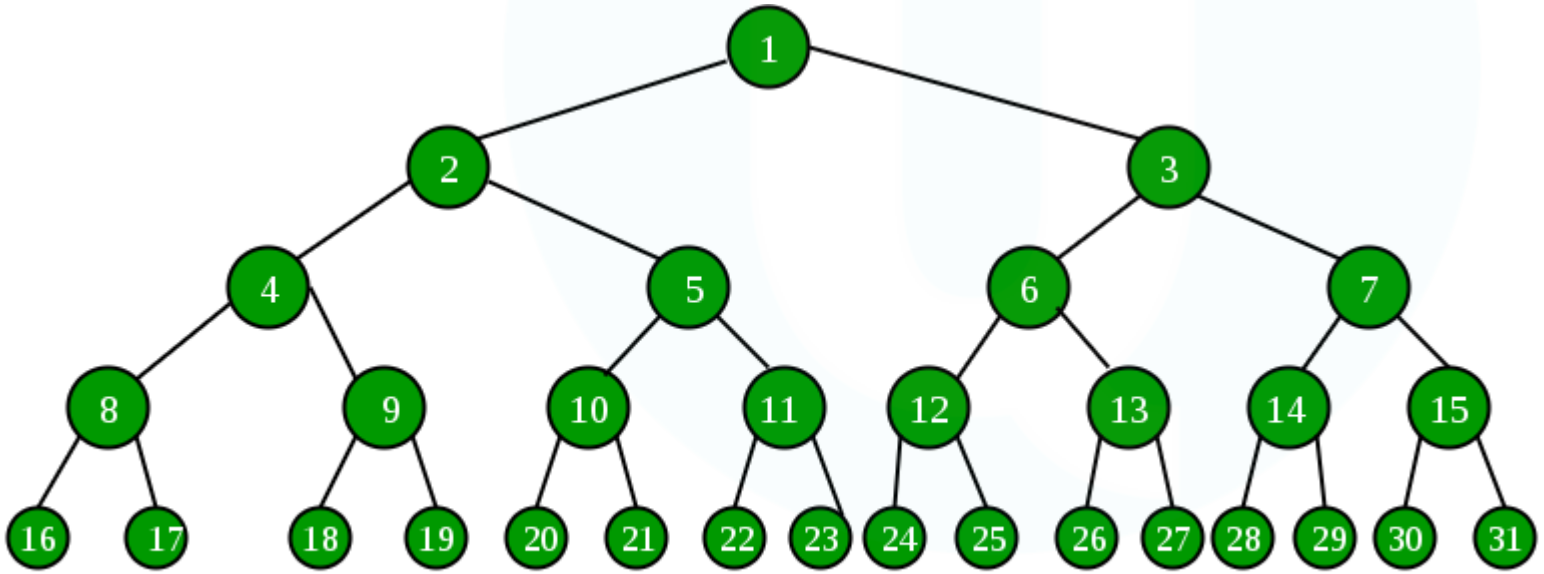


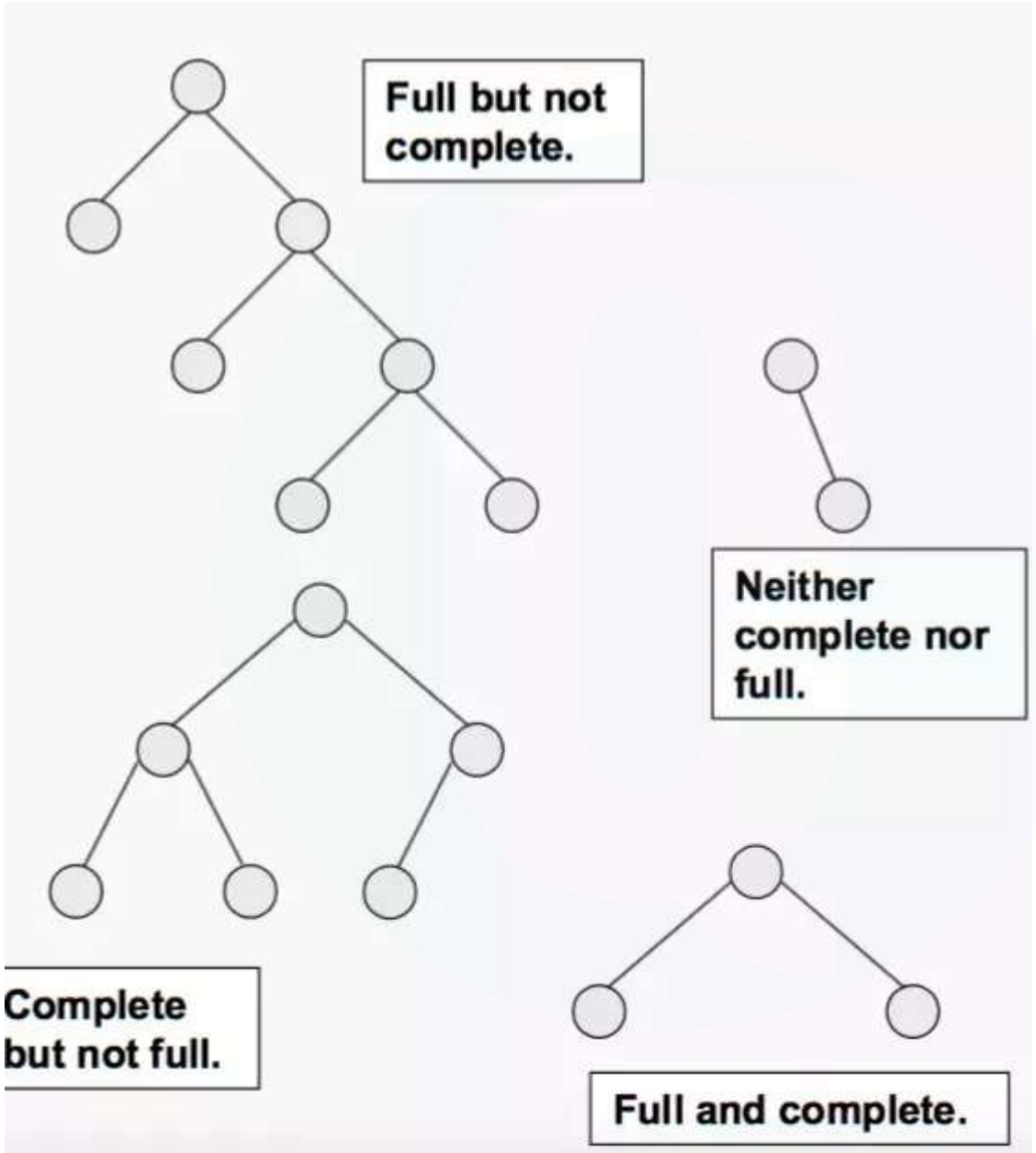
full tree

complete tree

Perfect binary tree

- A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.





Operations on Binary tree:

- **father(n,T):**Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- **LeftChild(n,T):**Return the left child of node n in tree T. Return NULL if n does not have a left child.
- **RightChild(n,T)**:Return the right child of node n in tree T. Return NULL if n does not have a right child.
- **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- **Sibling(n,T):** return the sibling node of node **n** in tree T. Return NULL if n has no sibling.
- **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- **Size(T):** Return the number of nodes in tree T
- **MakeEmpty(T):** Create an empty tree T
- **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- **Preorder(T):** Traverses all the nodes of tree T in preorder.
- **postorder(T):** Traverses all the nodes of tree T in postorder
- **Inorder(T):** Traverses all the nodes of tree T in inorder.

C representation for Binary tree:

```
struct bnode
{
        int info;
        struct bnode  *left;
        struct bnode  *right;
};
struct bnode *root=NULL;
```

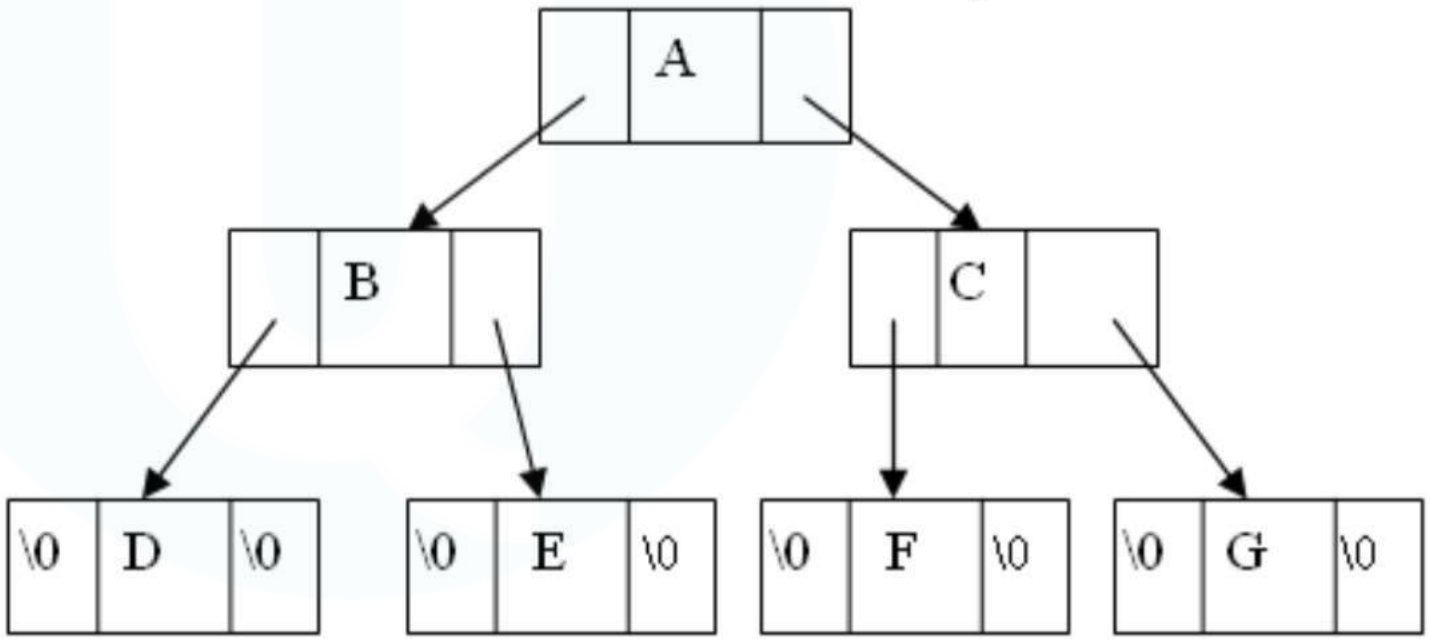


Fig: Structure of Binary tree

Tree traversal

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- All nodes are connected via edges (links) we always start from the root (head) node.
- There are three ways which we use to traverse a tree
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal
- Generally we traverse a tree to search or locate given item or key in the tree or to print all the values it contains.

Pre-order, In-order, Post-order

- Pre-order

<root><left><right>

- In-order

<left><root><right>

- Post-order

<left><right><root>

Ashim Lamichhane

Pre-order Traversal

- The preorder traversal of a nonempty binary tree is defined as follows:
  - Visit the root node
  - Traverse the left sub-tree in preorder
  - Traverse the right sub-tree in preorder

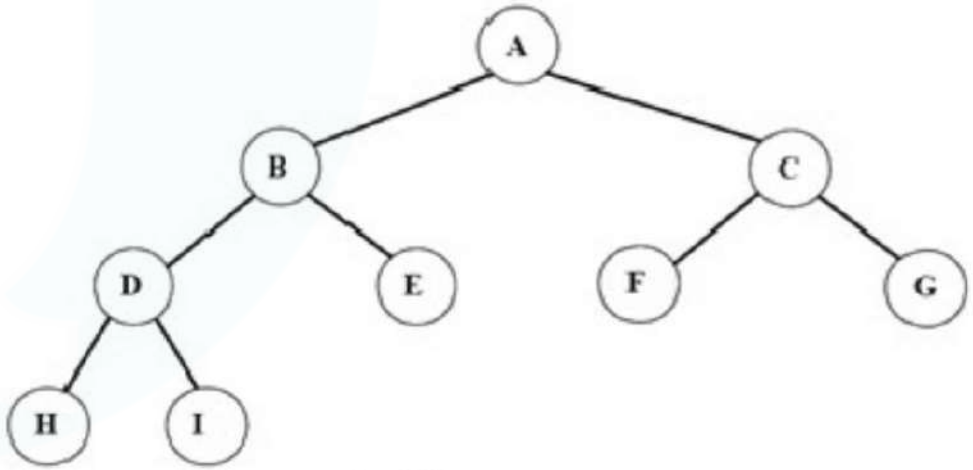


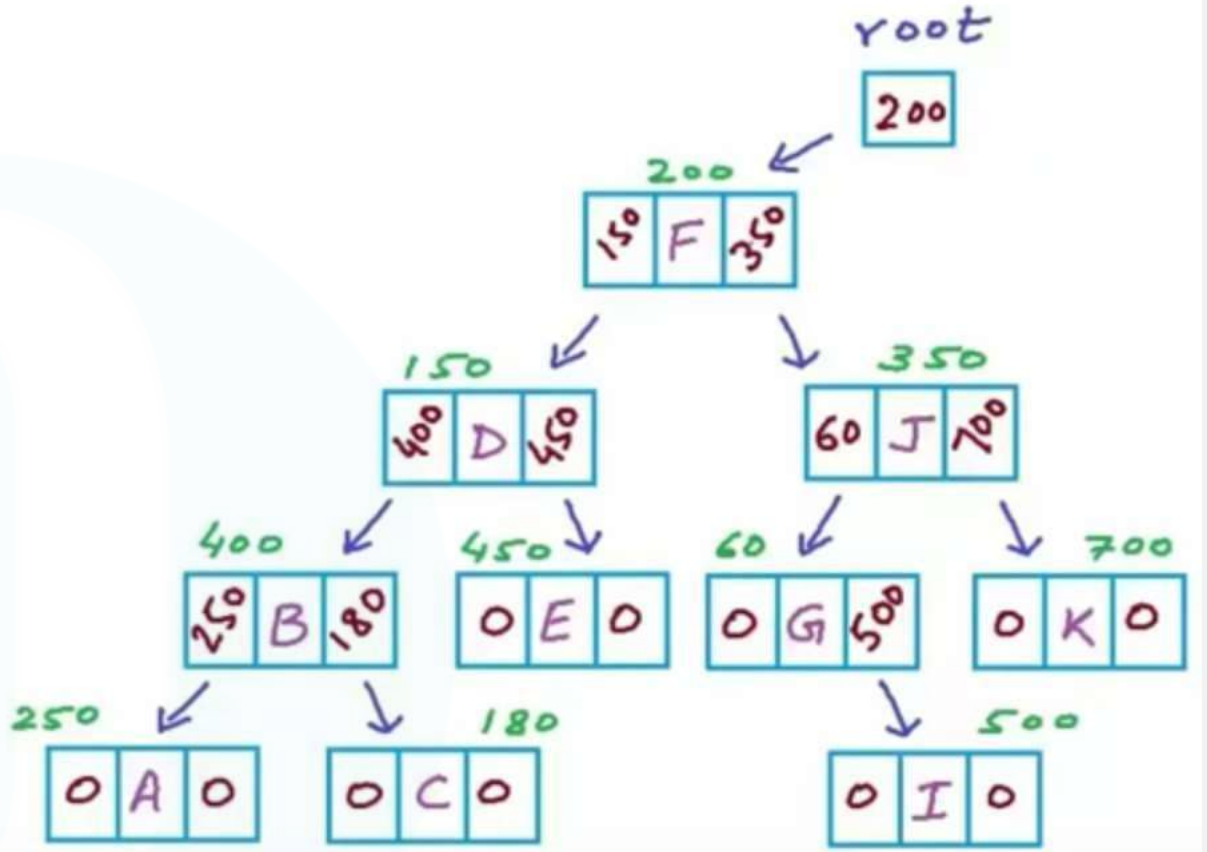
fig Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

Pre-order Pseudocode

```
struct Node{
 char data;
Node *left;  Node*right;
}
void Preorder(Node *root)
{
if (root==NULL) return;
printf (“%c”, root->data);
Preorder(root->left); Preorder(root-
   >right);
}
```



Preorder traversal- non recursive algorithm using stack

```
Preorder (Root)
{
 if (Root == NULL)
          {Return}
PUSH(Root)
While ( stack not empty) do
     ptr = POP()
     If (ptr != NULL) then
     Visit (ptr-> Data)
     PUSH (ptr-> right child)
     PUSH (ptr-> left child)
     End if
End While
}
Stop
```

In-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in in-order
  - Visit the root node
  - Traverse the right sub-tree in inorder

- The in-order traversal output of the given tree is
  H D I B E A F C G

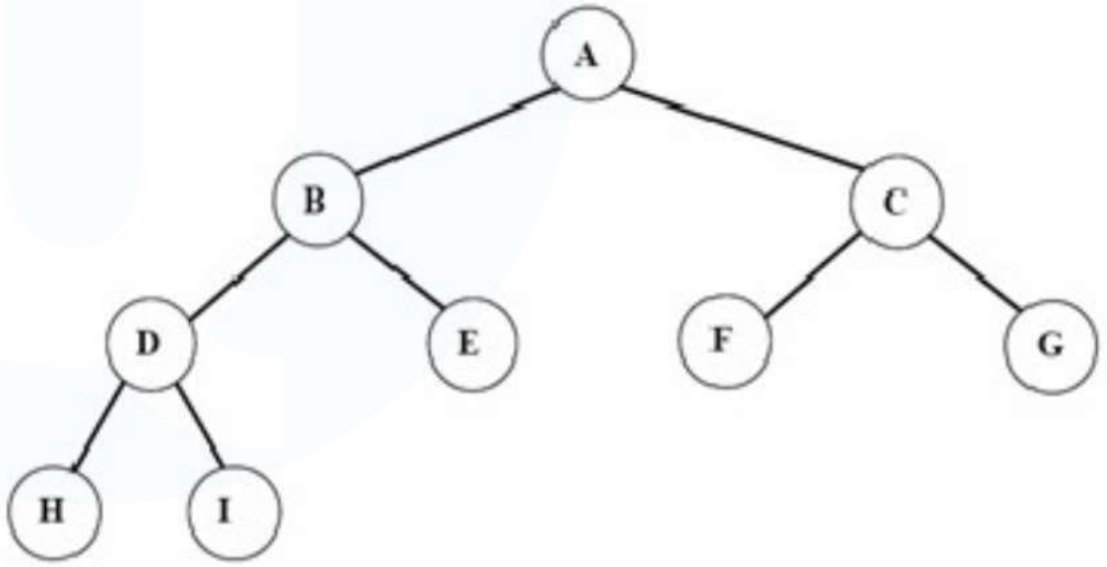
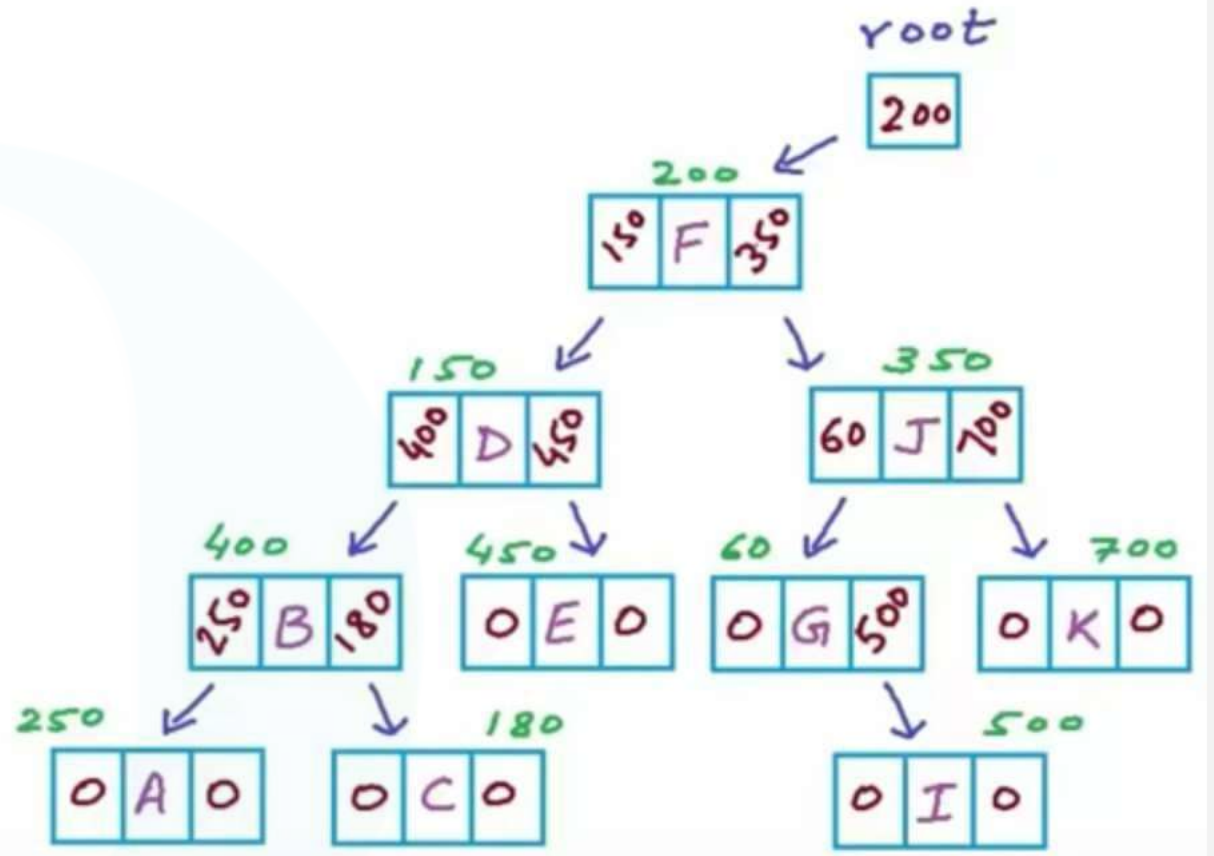


fig Binary tree

In-order Pseudocode

```
struct Node{  char data;  Node
    *left;  Node *right;
}
void Inorder(Node *root)
{
if (root==NULL) return;
 Inorder(root->left);
printf (“%c”, root->data);
Inorder(root->right);
}
```



In order- non recursive algorithm using stack

```
Inorder (Root)
  {
   Ptr = Root
    While ( (Stack not empty) OR (ptr != NULL))   do
       If (ptr != NULL ) then
           PUSH (ptr)
           Ptr = ptr ->left child
         Else
           Ptr = POP()
           Visit (ptr -> Data)
           Ptr = ptr -> right child
     End If
    End While
 }
Stop
```

Post-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in post-order
  - Traverse the right sub-tree in post-order
  - Visit the root node

- The in-order traversal output of the given tree is H I D E B F G C A

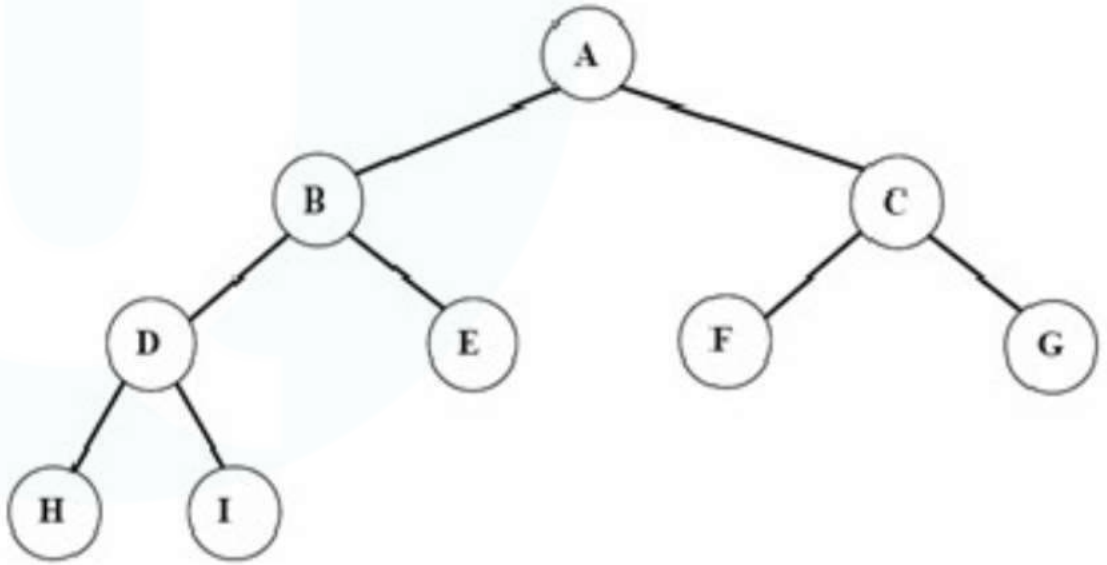
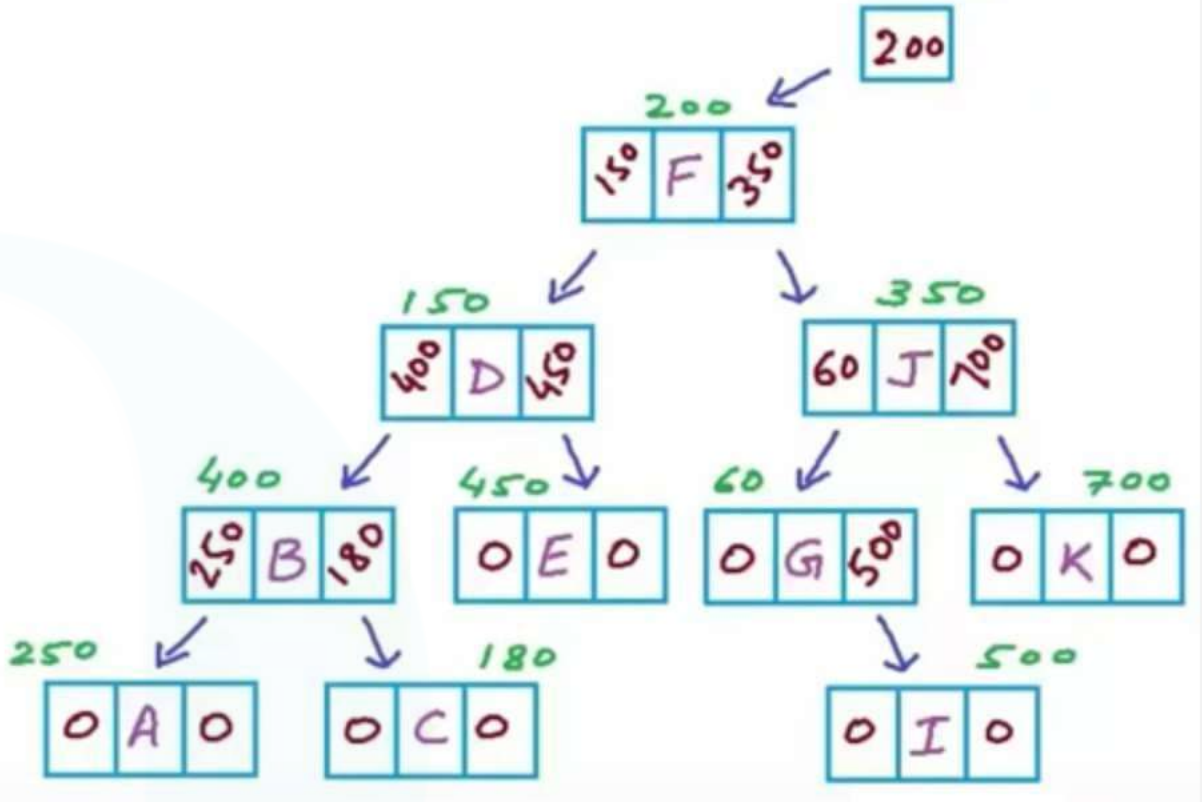


fig Binary tree

Post-order Pseudocode

```
struct Node{  char data;  Node
   *left;  Node *right;
}
void Postorder(Node *root)
{
if (root==NULL) return;
Postorder(root->left);
Postorder(root->right);
printf (“%c”, root->data);
}
```



Post order- non recursive algorithm using **2 stacks**

```
Post order (Root)
{
PUSH Stack1 (Root)
    While (Stack1 not empty) do
    Ptr = POP Stack1()
    PUSH Stack2(ptr)
        If (ptr-> left child != NULL)
    PUSH Stack1(ptr->left child)
        If (ptr-> right child != NULL)
    PUSH Stack1(ptr->right child)
    End While
    While (Stack 2 not empty) do
    Ptr = POP Stack2()
    Visit (ptr)
    End While }
Stop
```

Binary Search Tree(BST)

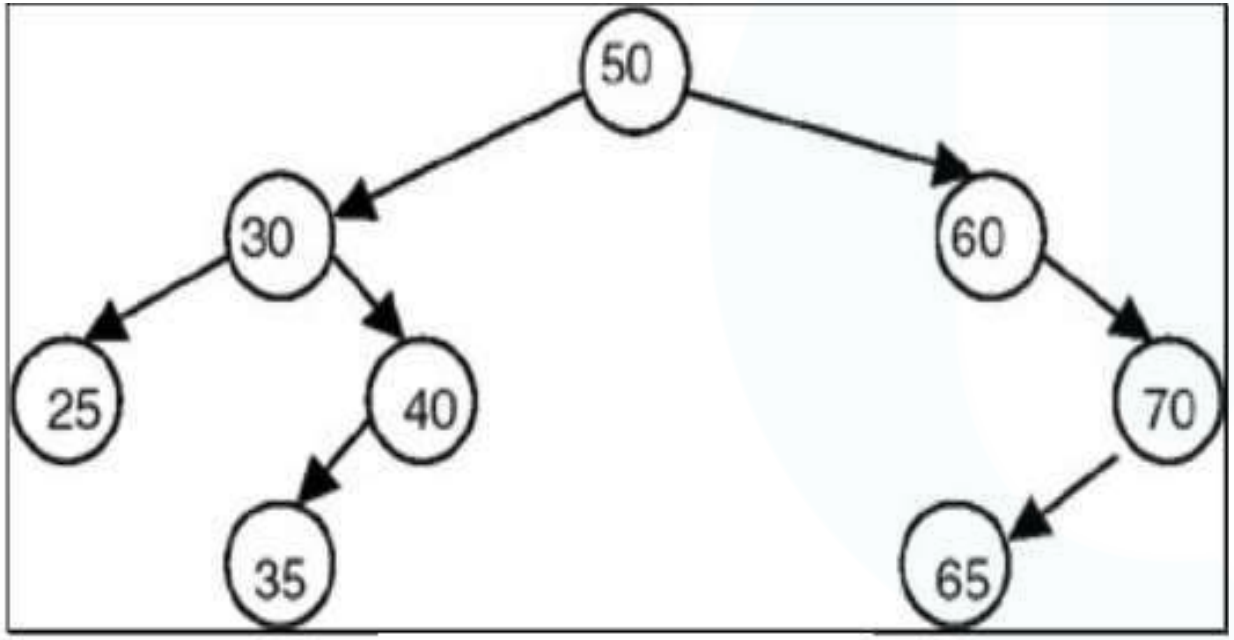
- A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:
  - All keys in the left sub-tree of the root are smaller than the key in the root

    node
  - All keys in the right sub-tree of the root are greater than the key in the root

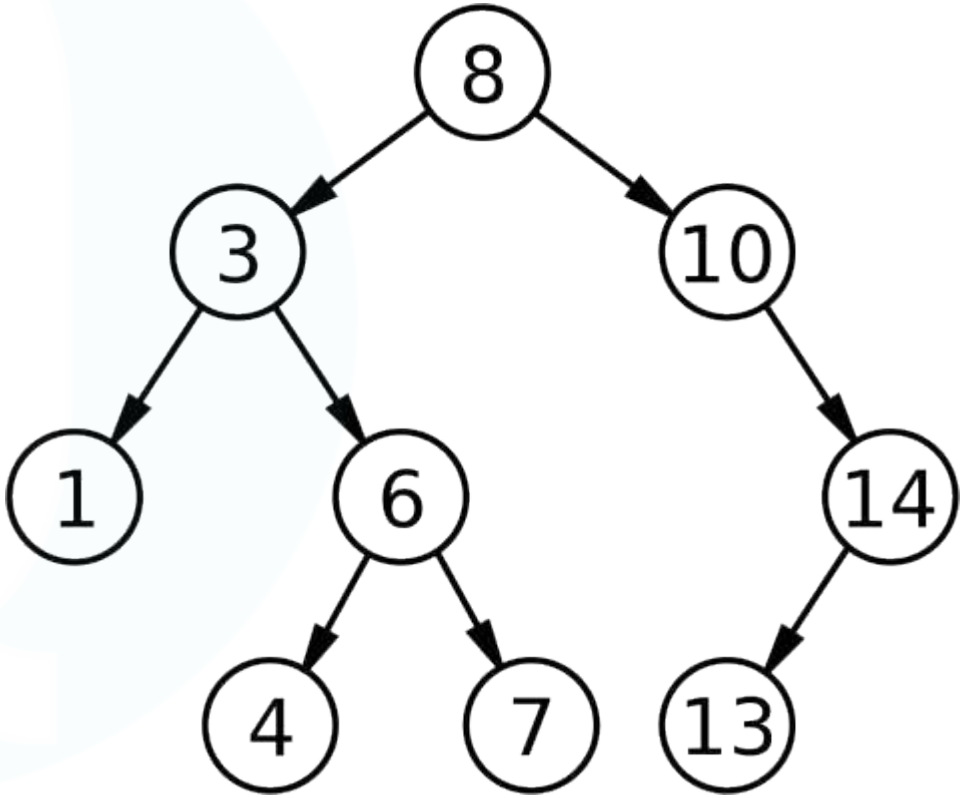
    node
  - The left and right sub-trees of the root are again binary search trees

Ashwin Lamichhane

Binary Search Tree(BST)



The binary search tree.



Ashim Lamichhane

Binary Search Tree(BST)

- A binary search tree is basically a binary tree, and therefore it can be traversed in inorder, preorder and postorder.

- If we traverse a binary search tree in inorder and print the identifiers contained in the nodes of the tree, we get a sorted list of identifiers in ascending order.

Why Binary Search Tree?

- Let us consider a problem of searching a list.
- If a list is ordered searching becomes faster if we use contiguous list(array).
- But if we need to make changes in the list, such as inserting new entries or deleting old entries, (SLOWER!!!!) because insertion and deletion in a contiguous list requires moving many of the entries every time.

Why Binary Search Tree?

- So we may think of using a linked list because it permits insertion and deletion to be carried out by adjusting only few pointers.

- But in an n-linked list, there is no way to move through the list other than one node at a time, permitting only sequential access.

- Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary search tree, we find that we can search for a key in O(logn)

Binary Search Tree(BST)

Time Complexity			
	Array	**Linked List**	**BST**
Search	O(n)	O(n)	O(logn)
Insert	O(1)	O(1)	O(logn)
Remove	O(n)	O(n)	O(logn)

Operations on Binary Search Tree (BST)

- Following operations can be done in BST:

  - Search(k, T): Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
  - Insert(k, T): Insert a new node with value k in the info field in the tree T such
    that the property of BST is maintained.
  - Delete(k, T):Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
  - FindMin(T), FindMax(T): Find minimum and maximum element from the given nonempty BST.

Searching Through The BST

- Compare the target value with the element in the root node
  - ✓ If the target value is equal, the search is successful.
  - ✓ If target value is less, search the left subtree.
  - ✓ If target value is greater, search the right subtree.
  - ✓ If the subtree is empty, the search is unsuccessful.

C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
        if(root == NULL)
        {
                printf("The number does not exist");
                exit(1);
        }
        else  if (key == root->info)
        {
                printf("The searched item is found"):
        }
        else if(key < root->info)
                return BinSearch(root->left, key);
        else
                return BinSearch(root->right, key);
}
```

Insertion of a node in BST

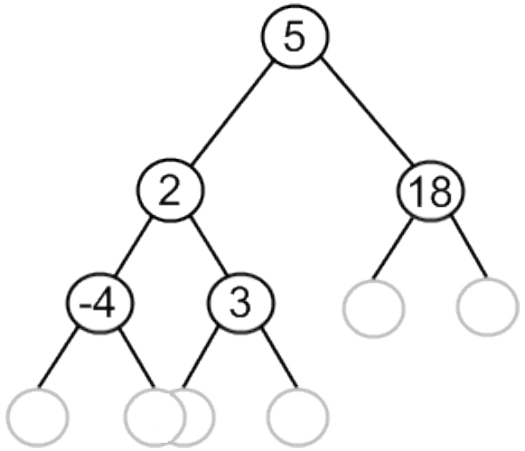
- To insert a new item in a tree, we must first verify that its key is different from those of existing elements.
- If a new value is less, than the current node's value, go to the left subtree, else go to the right subtree.
- Following this simple rule, the algorithm reaches a node, which has no left or right subtree.
- By the moment a place for insertion is found, we can say for sure, that a new value has no duplicate in the tree.

Algorithm for insertion in BST

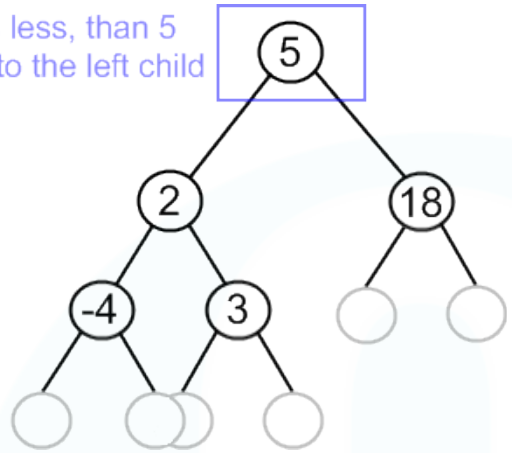
- Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,

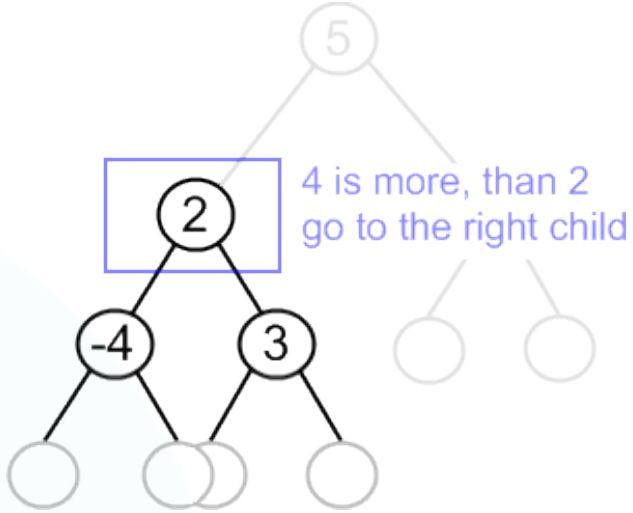
- if a new value is less, than the node's value:
  - if a current node has no left child, place for insertion has been found;
  - otherwise, handle the left child with the same algorithm.

- if a new value is greater, than the node's value:
  - if a current node has no right child, place for insertion has been found;
  - otherwise, handle the right child with the same algorithm.

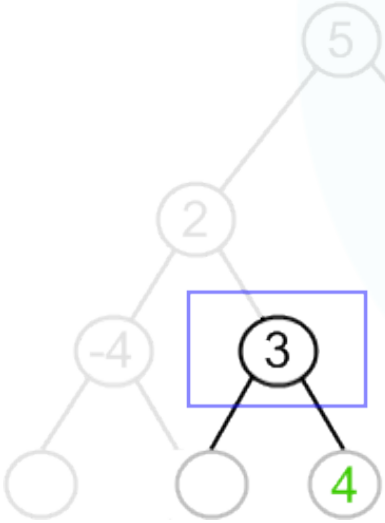


4 is less, than 5
go to the left child

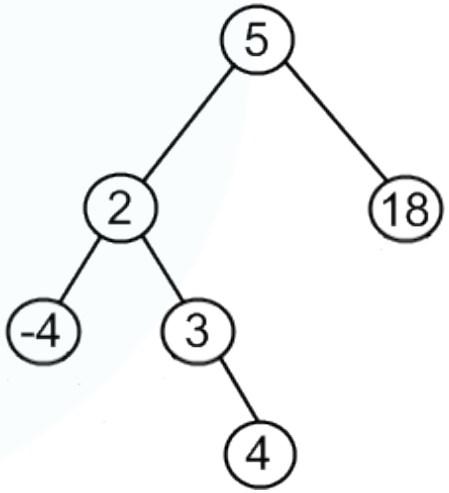




4 is more, than 2
go to the right child



4 is more, than 3
node has no right child
place for insertion has
been found



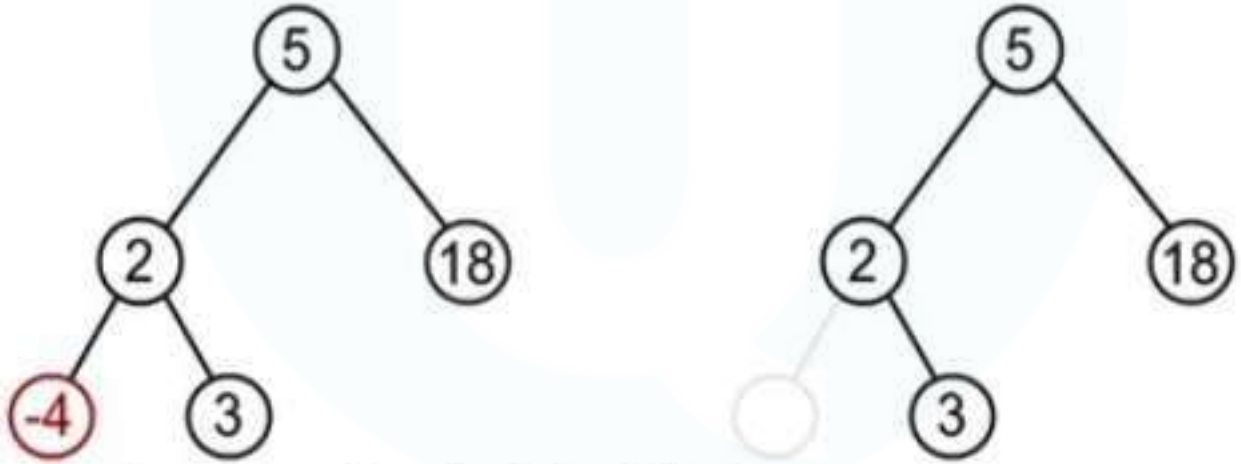
C function for BST insertion:

```
void insert(struct bnode *root, int item)
{
        if(root=NULL)
        {
                root=(struct bnode*)malloc (sizeof(struct bnode));
                root->left=root->right=NULL;
                root->info=item;
        }
        else
        {
                if(item<root->info)
                        root->left=insert(root->left, item);
                else
                        root->right=insert(root->right, item);
        }
}
```

Deleting a node from the BST

- While deleting a node from BST, there may be three cases:

1. The node to be deleted may be a leaf node:
   - In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.



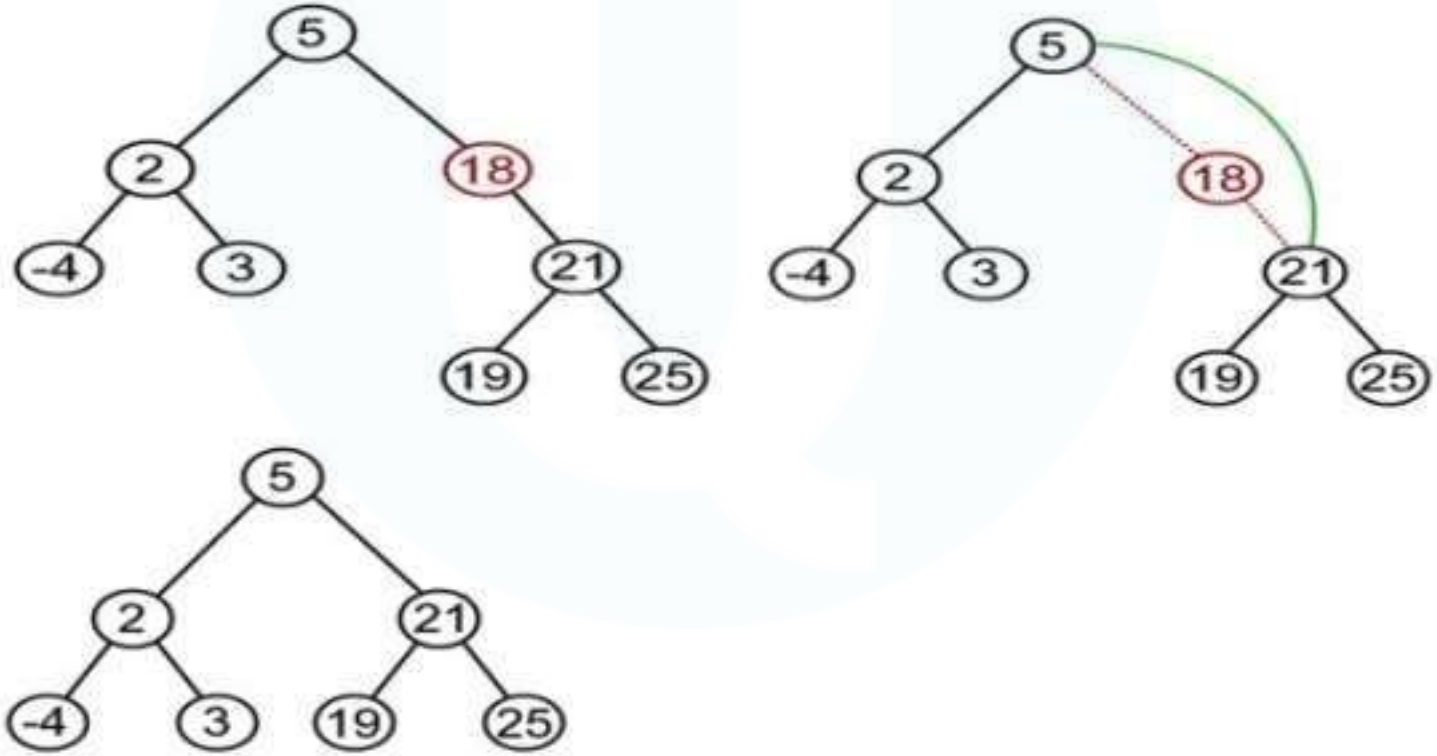
Suppose node to be deleted is -4

Deleting a node from the BST

2. The node to be deleted has one child

- In this case the child of the node to be deleted is appended to its parent node.

  Suppose node to be deleted is 18



Ashim Lamichhane

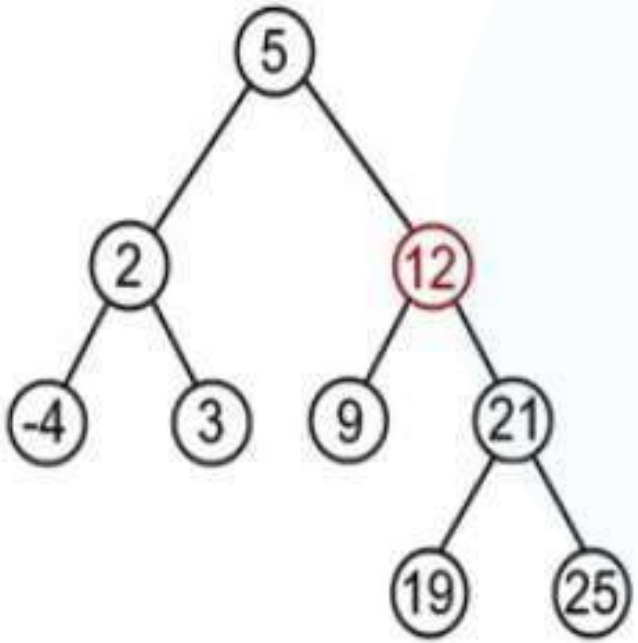
Deleting a node from the BST

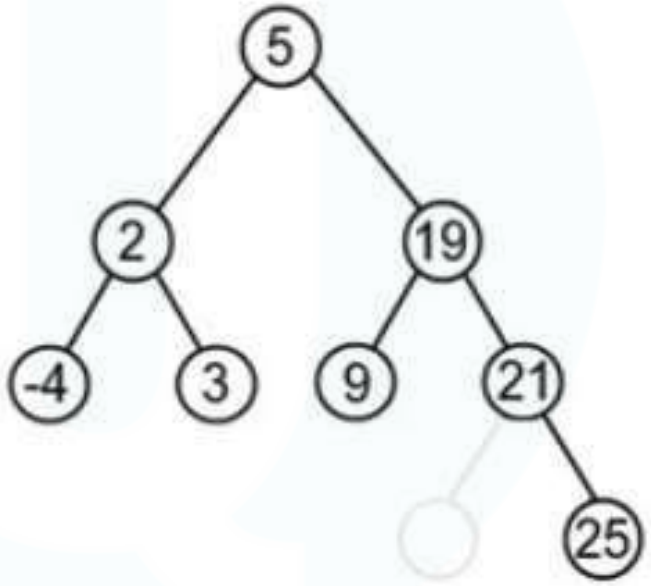
3. the node to be deleted has two children:

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.





Suppose node to deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

Ashim Lamichhane

General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has one child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
   a. right most node of it's left sub-tree or
   b. left most node of it's right sub-tree.
6. End

Ashim Lamichhane

The deleteBST function:

```
struct bnode *delete(struct bnode *root, int item)
{
    struct bnode *temp;
    if(root==NULL)
    {
        printf("Empty tree");
        return;
    }
    else if(item<root->info)
        root->left=delete(root->left, item);
    else if(item>root->info)
        root->right=delete(root->right, item);
    else if(root->left!=NULL &&root->right!=NULL)  //node has two child
    {
        temp=find_min(root->right);
        root->info=temp->info;
        root->right=delete(root->right, root->info);
    }
    else
    {
        temp=root;
        if(root->left==NULL)
            root=root->right;
        else if(root->right==NULL)
            root=root->left;
        free(temp);
    }
    return(temp);
}
/**********find minimum element function**********/
struct bnode *find_min(struct bnode *root)
{
        if(root==NULL)
            return0;
        else if(root->left==NULL)
            return root;
        else
            return(find_min(root->left));
}
```


Ashwin Lamichhane

Huffman Algorithm

- Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights.
- This is a method for the construction of minimum redundancy codes.
- Applicable to many forms of data transmission
- multimedia codecs such as JPEG and MP3

Huffman Algorithm

- 1951, David Huffman found the “most efficient method of representing numbers, letters, and other symbols using binary code”. Now standard method used for data compression.

- In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm. Initially 2 nodes are considered and their sum forms their parent node.

- When a new element is considered, it can be added to the tree.

- Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.

Huffman Algorithm

- Let us take any four characters and their frequencies, and sort this list by increasing frequency.

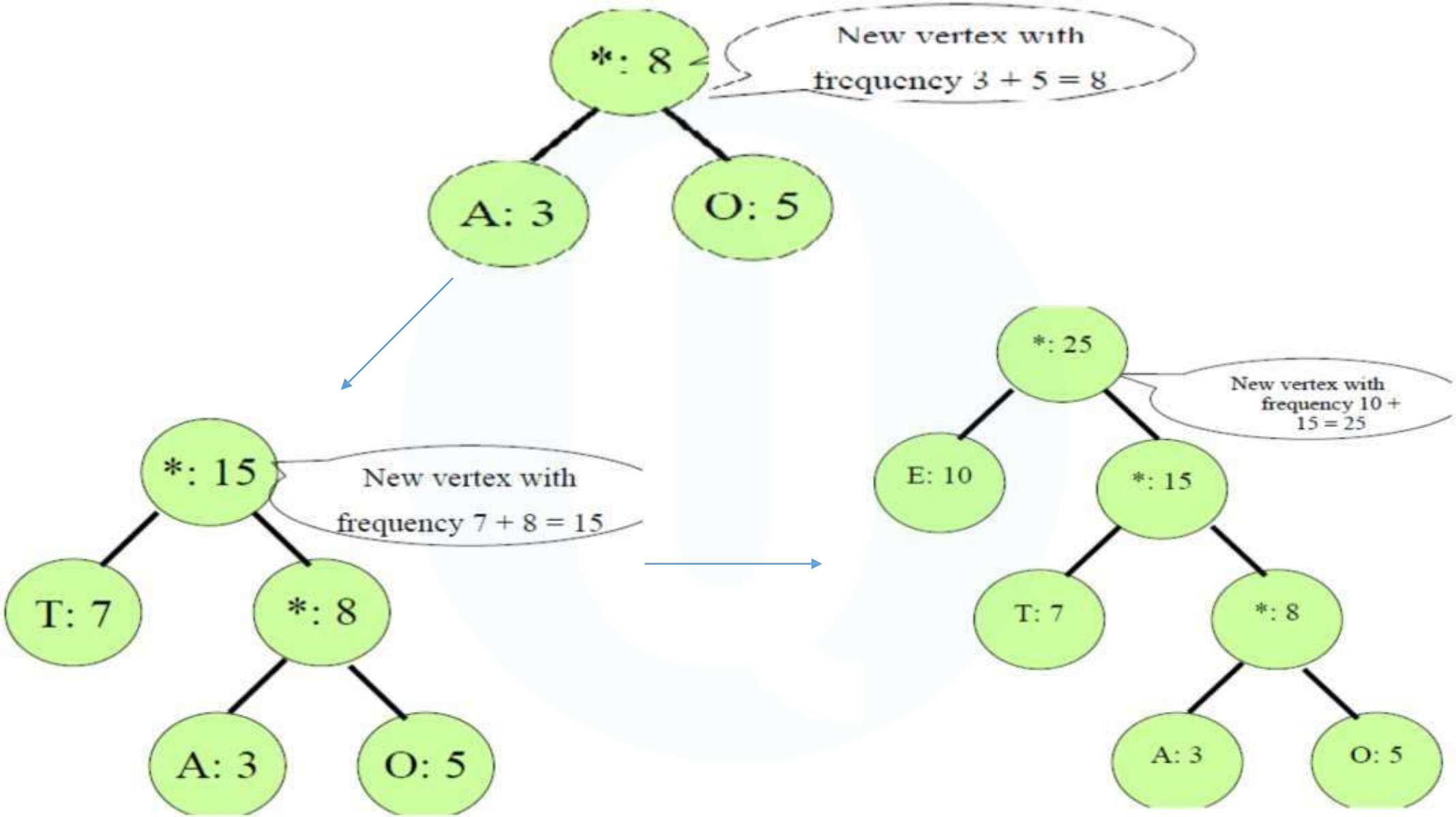
- Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

character	frequencies
E	10
T	7
O	5
A	3

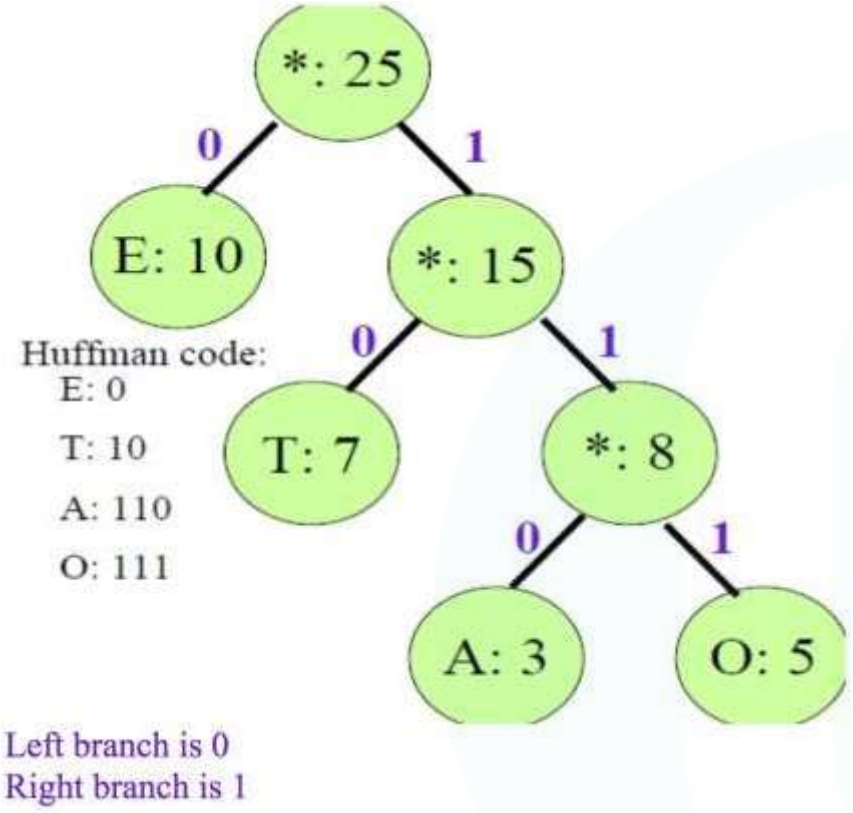
sort →

Character	frequencies	code
A	3	00
O	5	01
T	7	10
E	10	11

- Here before using Huffman algorithm the total number of bits required is: nb=3*2+5*2+7*2+10*2 =06+10+14+20 =50bits



Ashim Lamichhane



Character	frequencies	code
A	3	110
O	5	111
T	7	10
E	10	0

- Thus after using Huffman algorithm the total number of bits required is

nb=3*3+5*3+7*2+10*1 =09+15+14+10 =48bits

i.e

(50-48)/50*100%=4%

Since in this small example we save about 4% space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies we can save a lot of space

Huffman Algorithm

- Lets say you have a set of numbers and their frequency of use and want to create a huffman encoding for them

Value	Frequencies
1	5
2	7
3	10
4	15
5	20
6	45

Huffman Algorithm

- Creating a Huffman tree is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies:

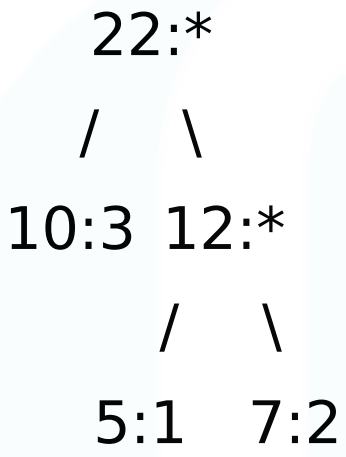
```
      12:*
     /    \
  5:1      7:2
```

- The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

```
10:3
12:*
15:4
20:5
45:6
```

Huffman Algorithm

- You then repeat the loop, combining the two lowest elements. This results in:



- The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

15:4

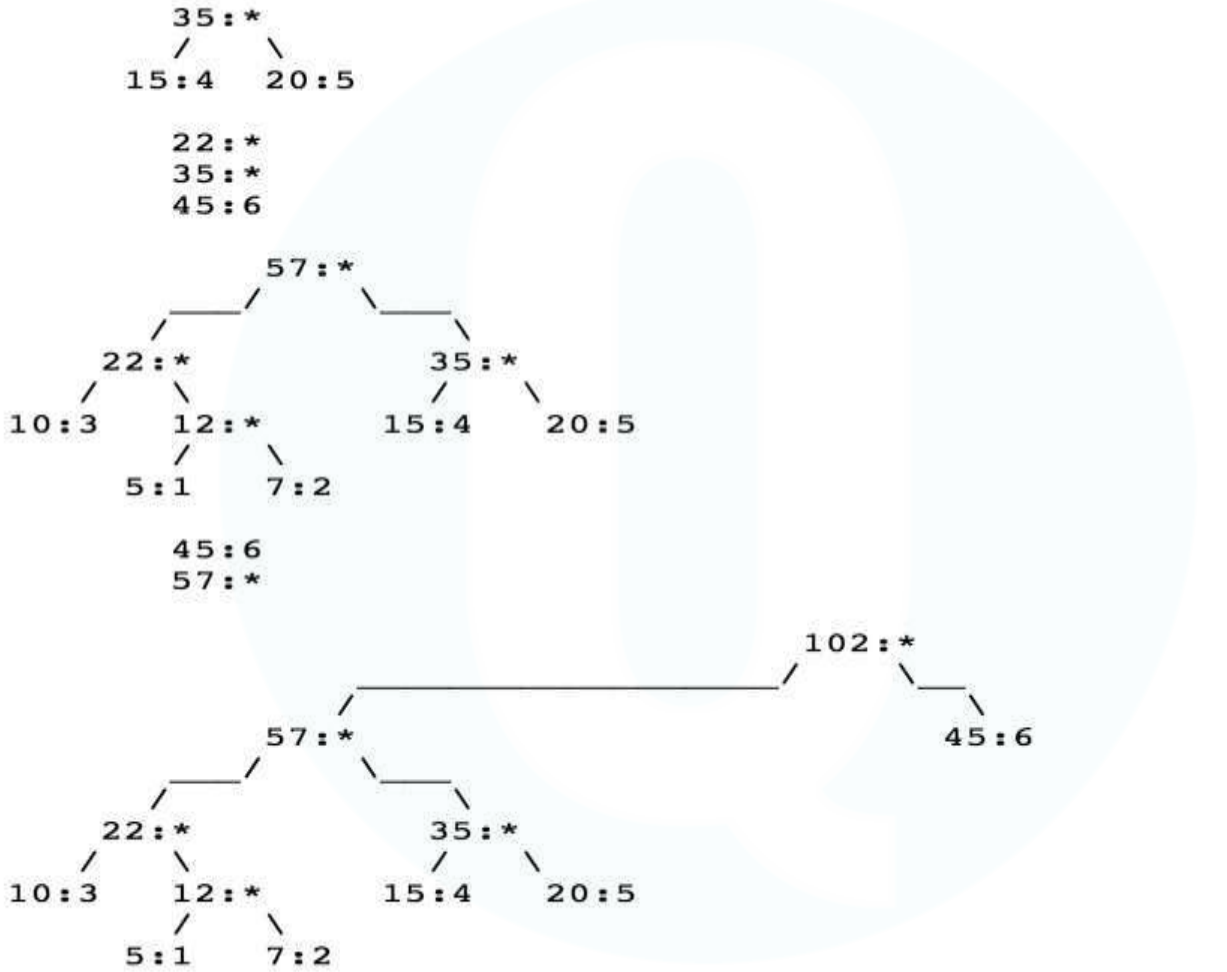
20:5

22: *

45:6

Huffman Algorithm

You repeat until there is only one element left in the list.



Now the list is just one element containing 102:*, you are done.

Huffman Algorithm

Value	C	D	E	K	L	M	U	Z
Frequency	32	42	120	7	42	24	37	2

After sorted

Value	Z	K	M	C	L	D	U	E
Frequency	2	7	24	32	42	42	37	120

We can represent it using 3 bit

Huffman Algorithm

- Find code for
  - DEED
  - MUCK

- Try to decode the bit string 1011001110111101

Assignments

- Slides at myblog

http://www.ashimlamichhane.com.np/2016/07/tree-slide-for-data-structure-and-algorithm/

- Assignments at github

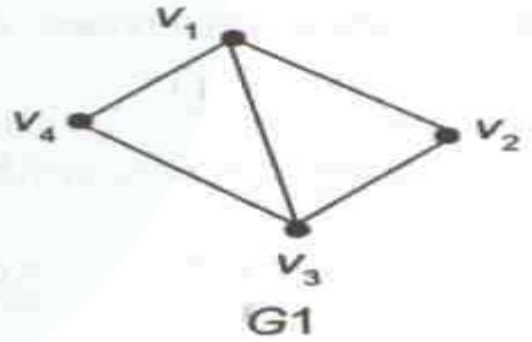
https://github.com/ashim888/dataStructureAndAlgorithm/tree/dev/Assignments/assignment_7

Reference

- https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html
- https://en.wikipedia.org/wiki/Binary_search_tree
- https://www.cs.swarthmore.edu/~newhall/unixhelp/Java_bst.pdf
- https://www.cs.usfca.edu/~galles/visualization/BST.html
- https://www.cs.rochester.edu/~gildea/csc282/slides/C12-bst.pdf
- http://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

Graph

A graph G is a collection of two sets V and E where V is the collection of vertices v0,v1,v2……vn-1 also called nodes and E is the collection of edges e1,e2….en where an edge is an arch which connects two vertices.

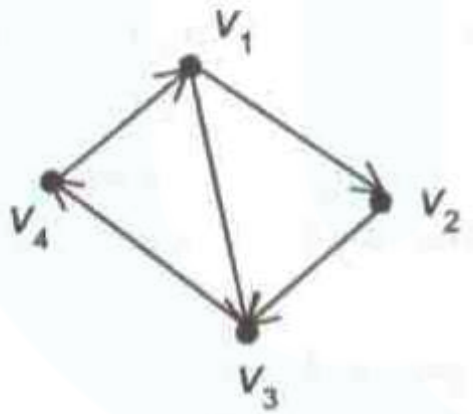


G1

G=(V,E)
V(G)=(v1,v2-------vn)
E(G)=(e1,e2……..en)
E(G)=((v1,v2),(v1,v3),(v1,v4)(v2,v3)(v3,v4))

Digraph

A digraph is also called directed graph. It is a graph G, such that G=<V,E>, where V is a set of all vertices and E is a set of ordered pair of elements from V.



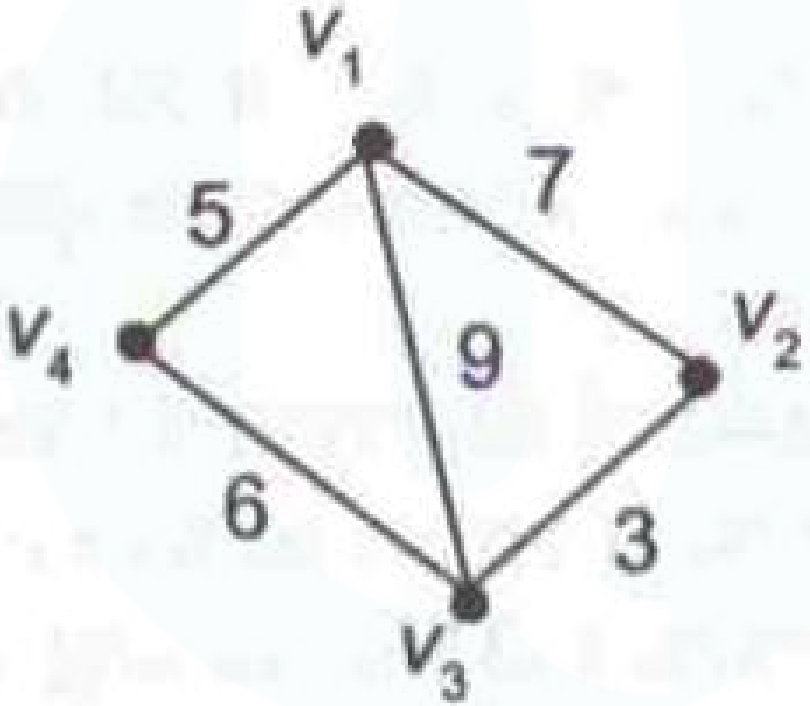
G2

V={v1,v2,v3,v4}

E={(v1,v2),(v1,v3),(v2,v3),(v3,v4),(v4,v1)}

Weighted Graph

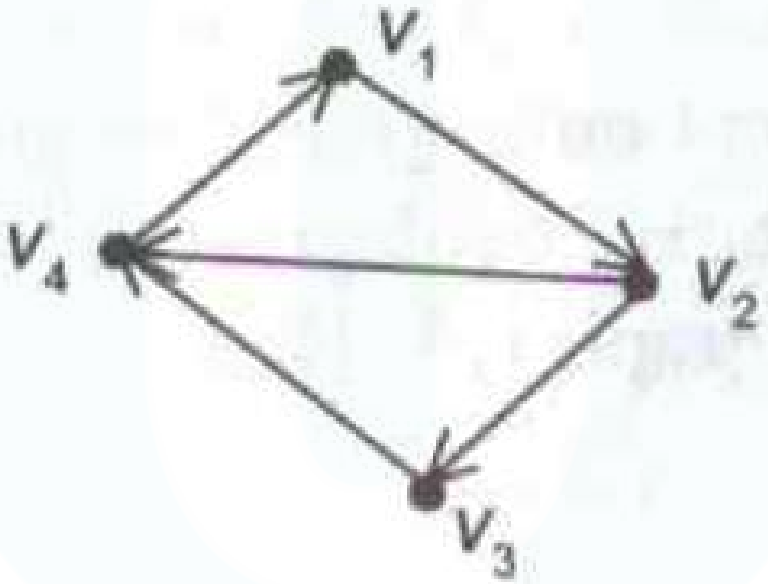
A graph (or diagraph) is termed as weighted graph if all the edges in it are labeled with some weights.



G3

Adjacent vertices

A vertex vi is adjacent to another vertex vj, if there is an edge from vi to vj.

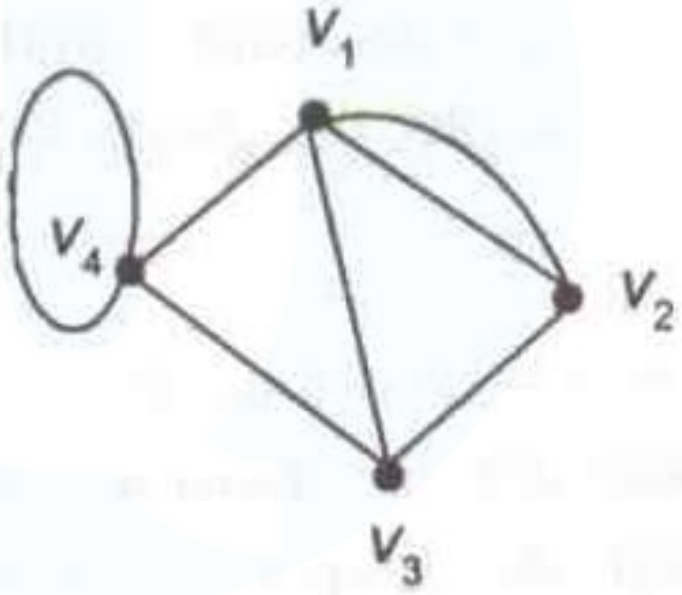


G11

v2 is adjacent to v3 and v4, v1 is not adjacent to v4 but to v2

Self Loop

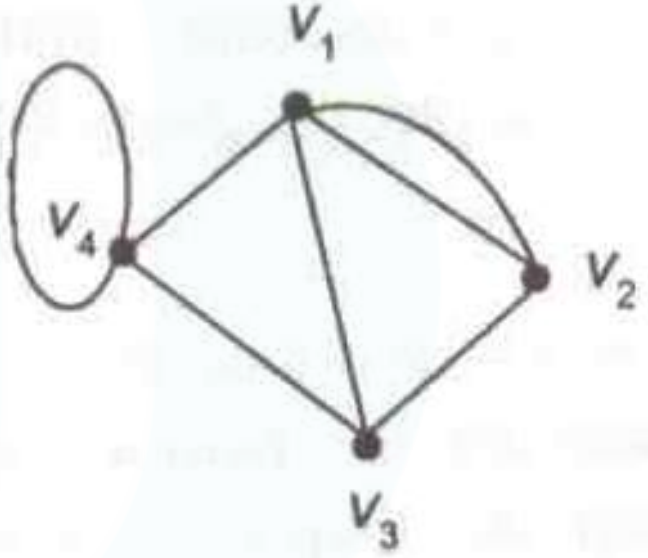
If there is an edge whose starting and end vertices are same, that is(vi,vi) is an edge, then it is called a self loop.



G5

Parallel Edges

If there are more than one edges between the same pair of vertices, then they known as .



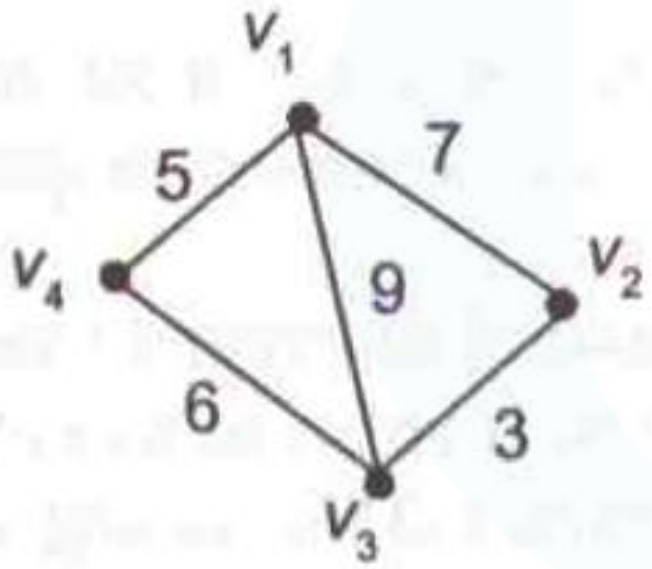
G5

There are edges between v1 and v2

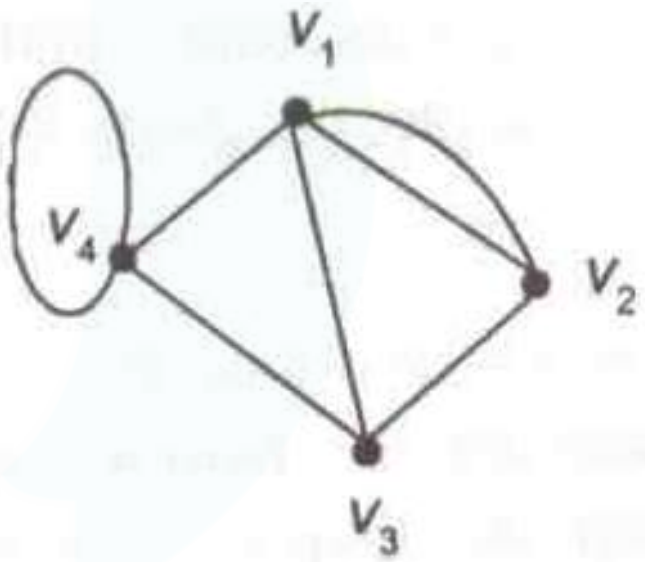
A graph has either self loop or parallel edges or both is called multigraph

Simple Graph

A graph does not have any self loop or parallel edges is called a simple graph.



G3

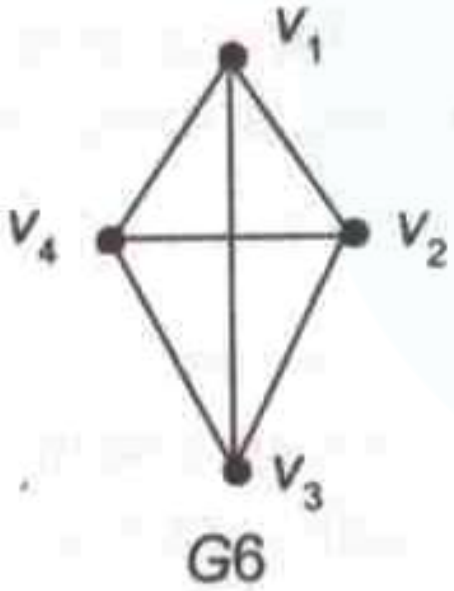


G5

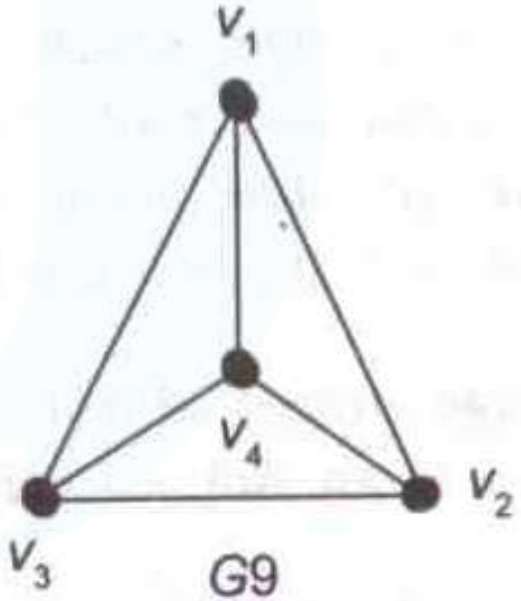
Not a simple graph

Complete Graph

A graph said to be complete, if each vertex vi is adjacent to every other vertex in vj in G. In other words there are edges from any vertex to all other vertices



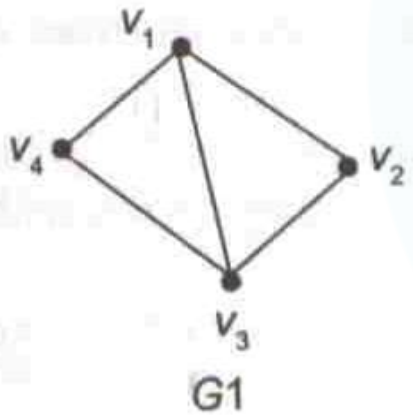
G6



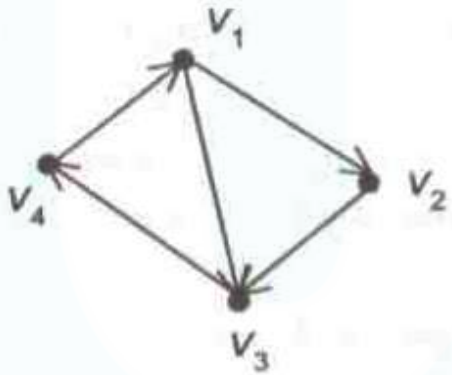
G9

Acyclic Graph

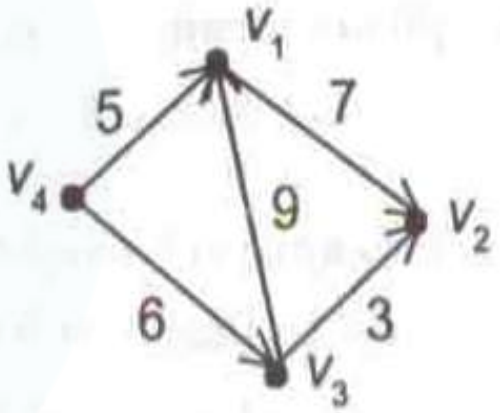
If there is a path containing one or more edges which start from the vertex vi and terminates into the same vertex then the path is known as cycle.



G1



G2

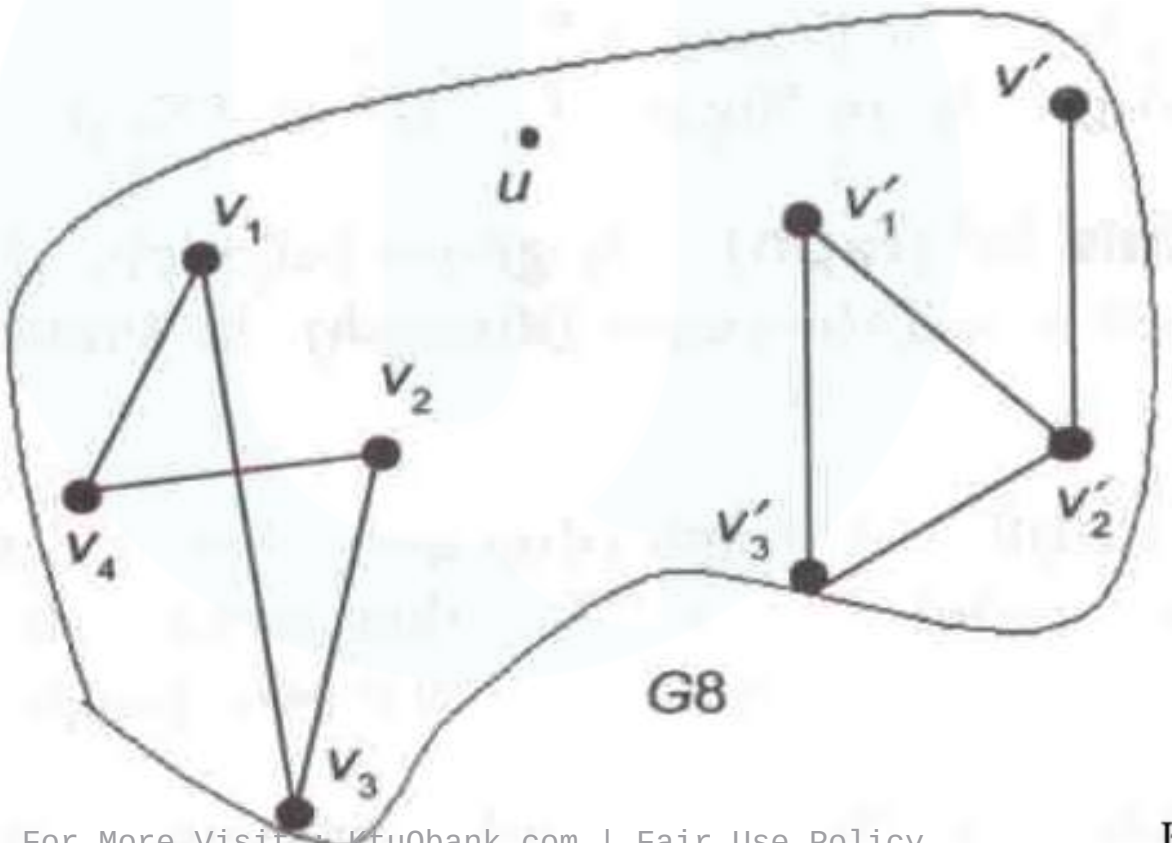


G4

If a graph does not have any cycle is called acyclic graph.(G1 and G2 are cyclic, G4 is acyclic)

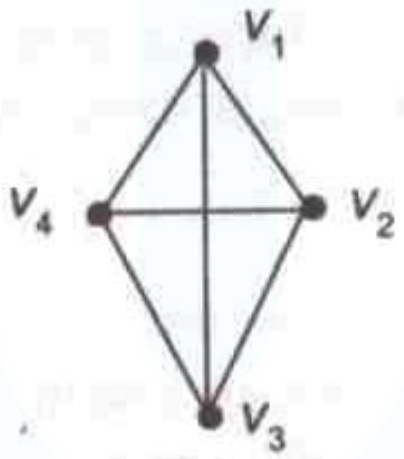
Isolated vertex

A vertex is isolated if there is no edge connected from any other vertex to the vertex.



Degree of vertex

The number of edges connected with vertex vi is called the degree of vertex vi and is denoted by degree(vi).



G6

Degree(vi)=3

For digraph, there are two degrees

1. Indegree: indegree of vertex vi denoted as indegree(vi) is the number of edges incident into vi.

2. Outdegree: is the number of edges emanating from vi.

Consider G4

Indegree(v1)=2

outdegree(v1)=1

Indegree(v2)=2

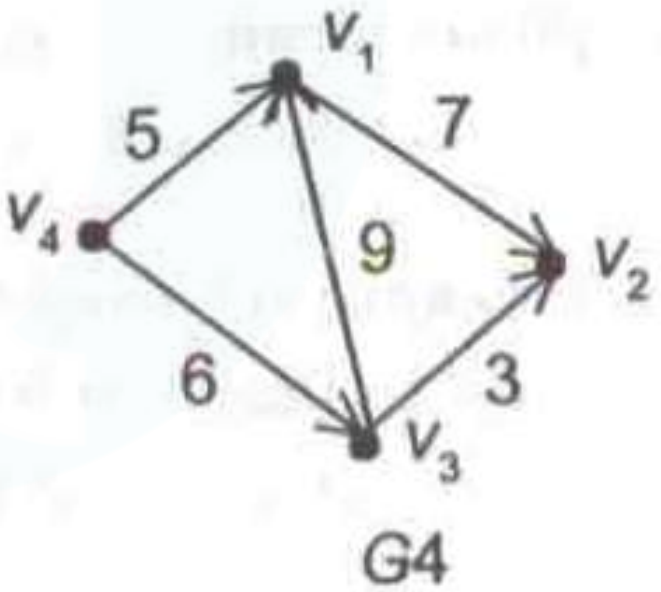
outdegree(v2)=0

Indegree(v3)=1

outdegree(v3)=2

Indegree(v4)=0

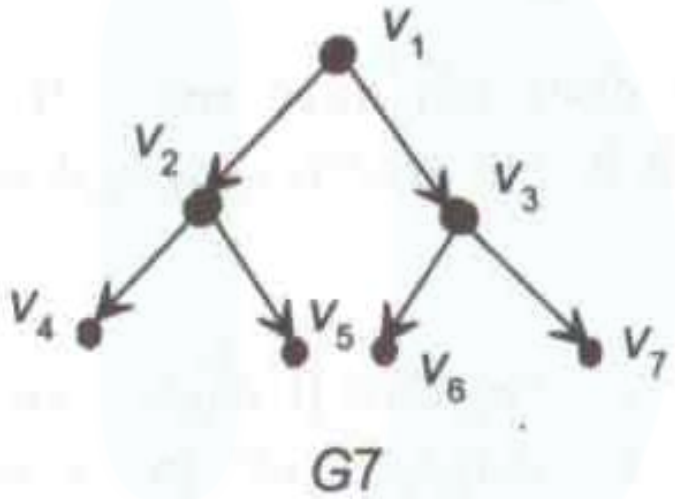
outdegree(v4)=2



G4

Pendant vertex

A vertex vi is pendant if its indegree(vi)=1 and outdegree(vi)=0.

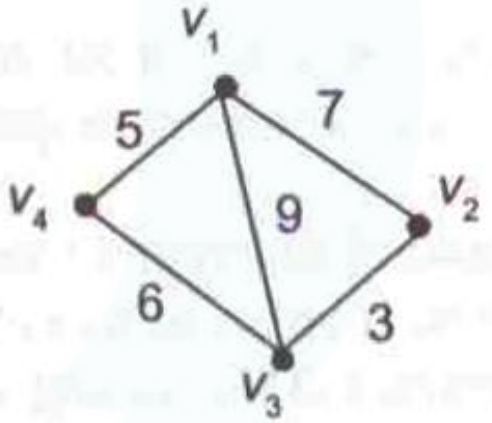


G7

There are 4 pendant vertices v4,v5, v6 and v7.

Connected Graph

In a graph(not digraph) G, two vertices vi and vj are said to be connected if there is a path in G from vi to vj. A graph said to be connected if for every pair of distinct vertices vi,vj in G, there is a path.



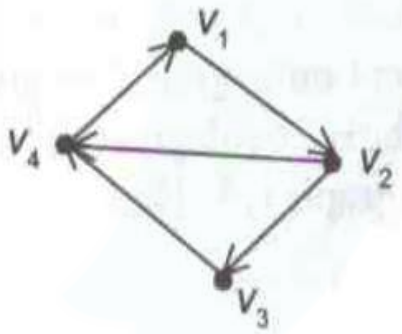
G3

A directed graph (digraph) with this property is called strongly connected graph.

Strongly Connected Graph

A directed graph (digraph) with this property is called strongly connected graph.

A digraph G is said to be strongly connected if for every pair of distinct vertices vi,vj in G, there is a path from vi to vj and also from vj to vi.



G11

A digraph is not strongly connected is known as weakly connected graph.

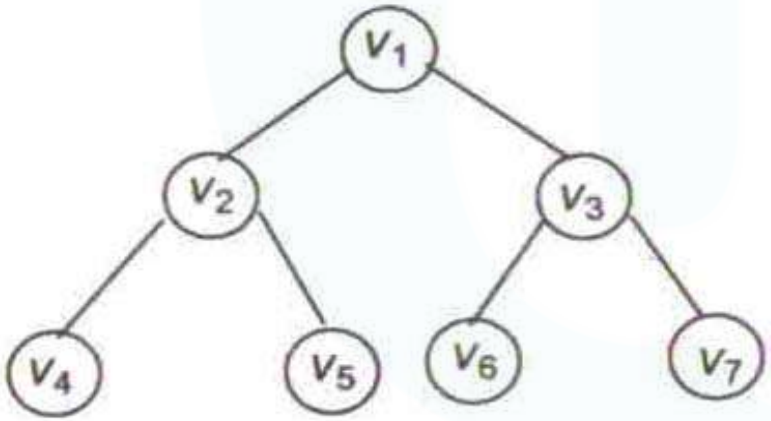
- To implement a graph, you need to first represent the given information in the form of a graph.
- The most commonly used ways of representing a graph are as follows:
  - Set Representation
  - Adjacency Matrix
  - Adjacency List

Set Representation

Two sets are maintained 1) V the set of vertices 2) E the set of Edges (V * V)

If the graph is weighted, then the set E is the ordered collection of 3 tuples.

E=W*V*V where W is the weight

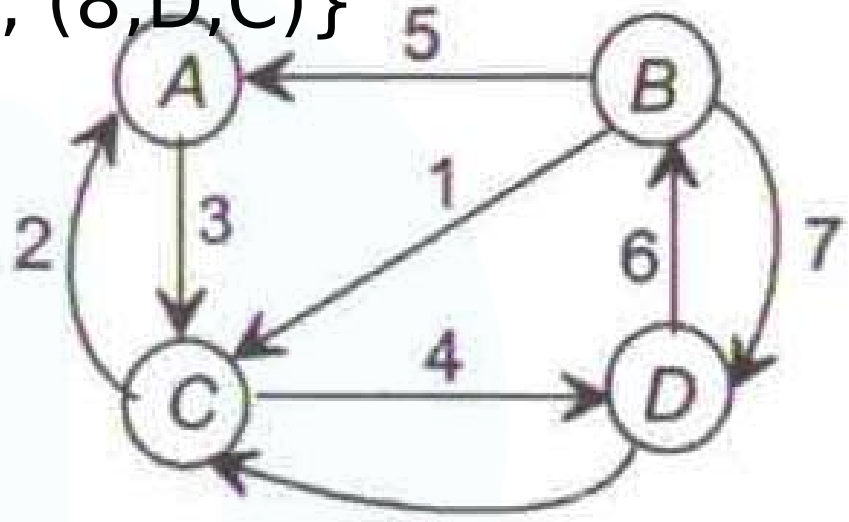


V(G1)={ v1,v2,v3,v4, v5,v6,v7}

E(G1)={(v1,v2),(v1,v3),(v2,v4),(v2,v5),(v3,v6),(v3,v7)}

V(G4)={A,B,C,D,E}

E(G4)={(3,A,C), (5,B,A), (1,B,C), (7,B,D), (2,C,A), (4,C,D), (6,D,B), (8,D,C)}



G4

Advantages

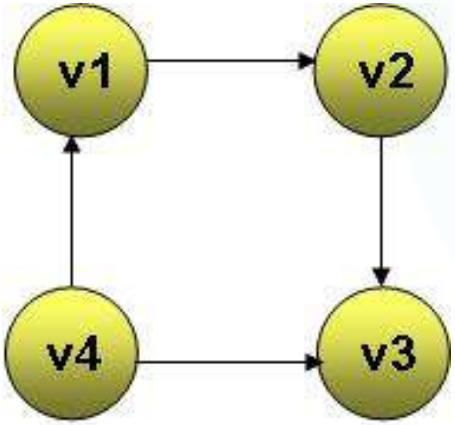
1. Efficient utilization of memory
2. Easy representation

Disadvantages

1. We can’t represent parallel , loop edges
2. Graph manipulation is not possible with set

Adjacency Matrix Representation

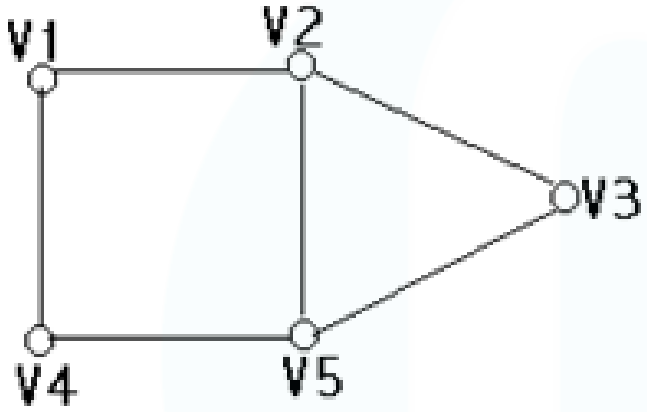
Consider the following graph:



	v1	v2	v3	v4
V1	0	1	0	0
v2	0	0	1	0
v3	0	0	0	0
v4	1	0	1	0

Adjacency Matrix for a Simple Graph:

Example: Given a graph G as follows



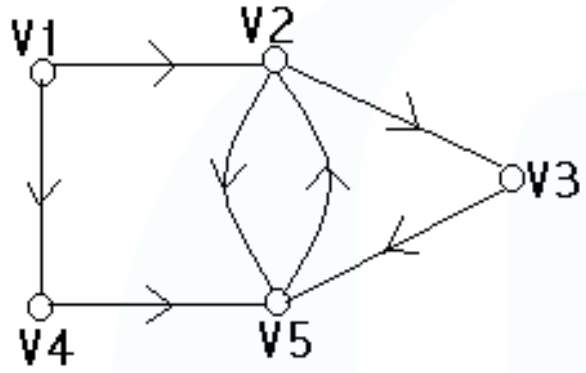
	V1	V2	V3	V4	V5
V1	0	1	0	1	0
V2	1	0	1	0	1
V3	0	1	0	0	1
V4	1	0	0	0	1
V5	0	1	1	1	0

From the chart above, the adjacency matrix for the graph G is:

0	1	0	1	0
1	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0

Adjacency Matrix for a Directed Graph:

Example: Given a directed graph G as follows



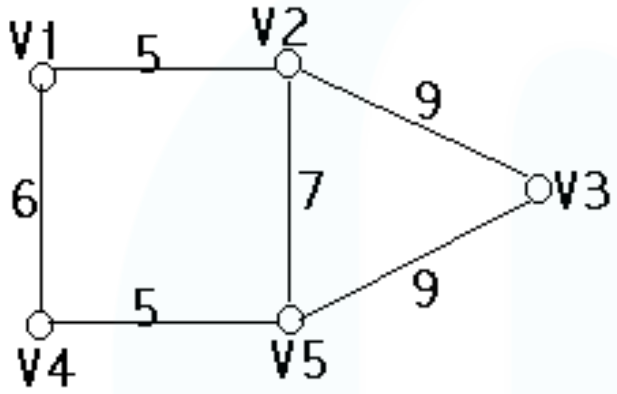
	V1	V2	V3	V4	V5
V1	0	1	0	1	0
V2	0	0	1	0	1
V3	0	0	0	0	1
V4	0	0	0	0	1
V5	0	1	0	0	0

From the chart above, the adjacency matrix for the directed graph is:

0	1	0	1	0
0	0	1	0	1
0	0	0	0	1
0	0	0	0	1
0	1	0	0	0

Adjacency Matrix for a Weighted Graph:

Example: Given a weighted graph W as follows.



	V1	V2	V3	V4	V5
V1	0	5	0	6	0
V2	5	0	9	0	7
V3	0	9	0	0	9
V4	6	0	0	0	5
V5	0	7	9	5	0

From the chart above, the adjacency matrix for the graph G is:

0	5	0	6	0
5	0	9	0	7
0	9	0	0	9
6	0	0	0	5
0	7	9	5	0

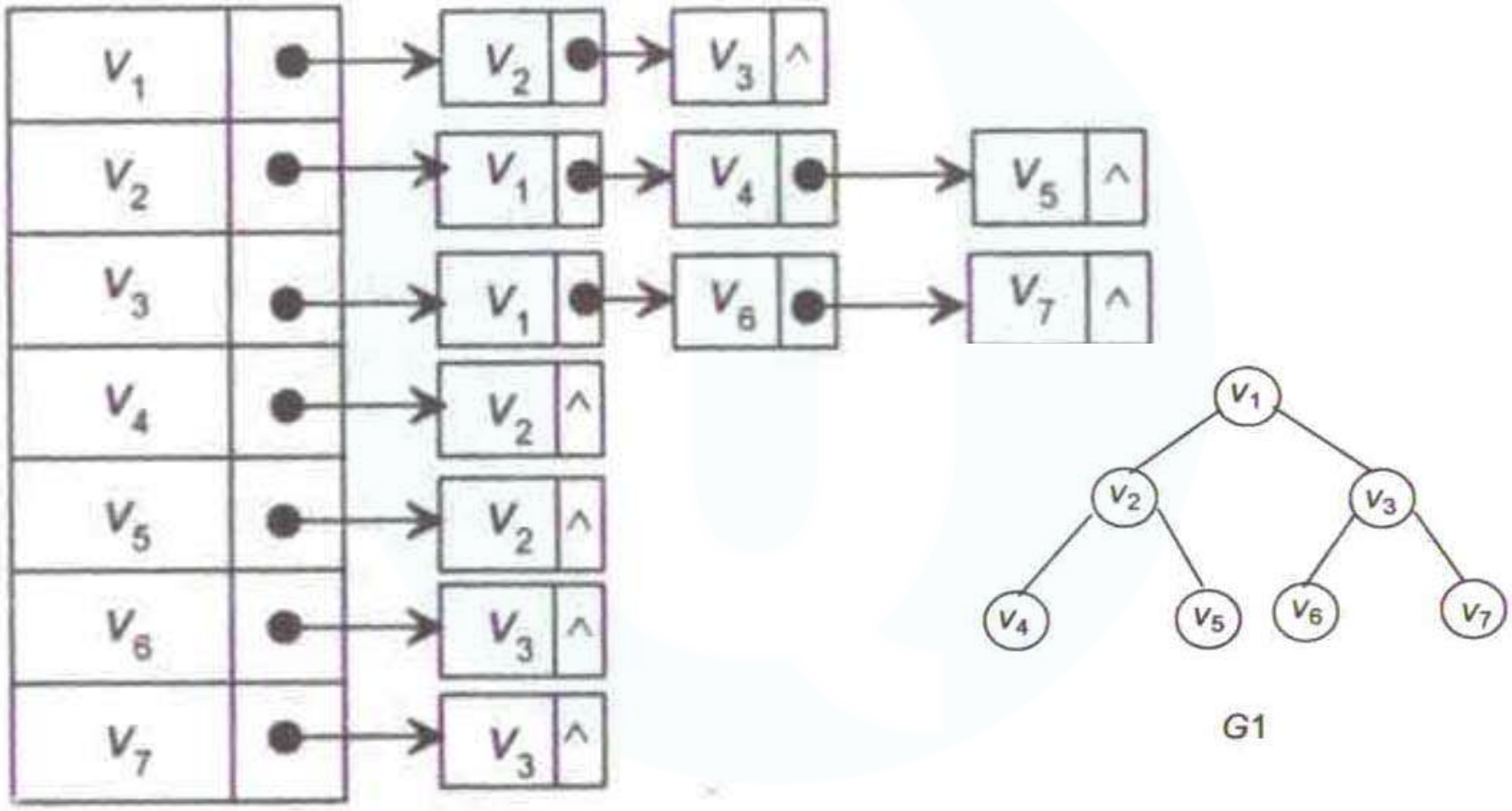
Linked representation(Adjacency List)

linked representation is another space – saving way of graph representation

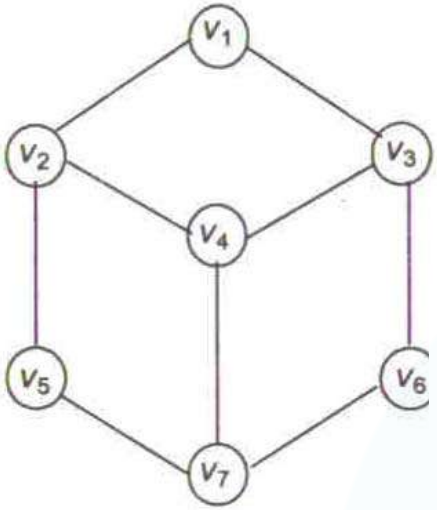
Node structure for non – weighted graph

Node structure for weighted graph

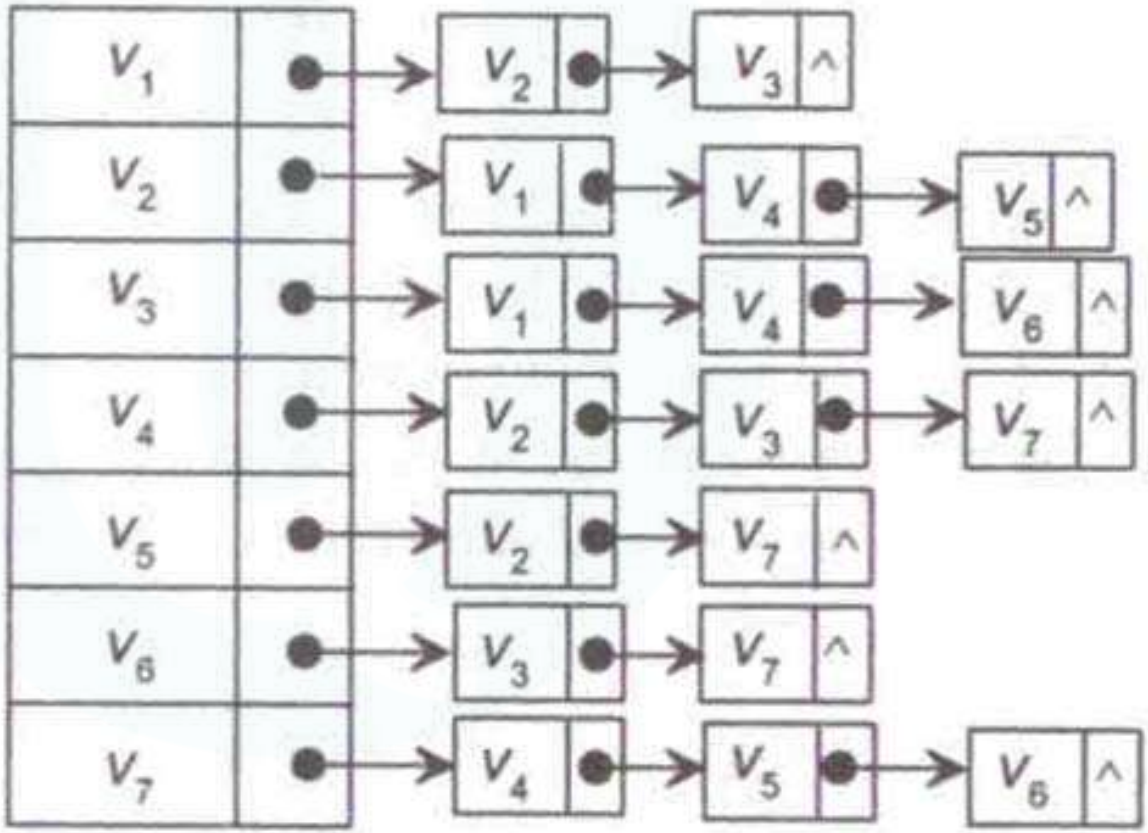
Header node in each list maintains a list of all adjacent vertices of a node.



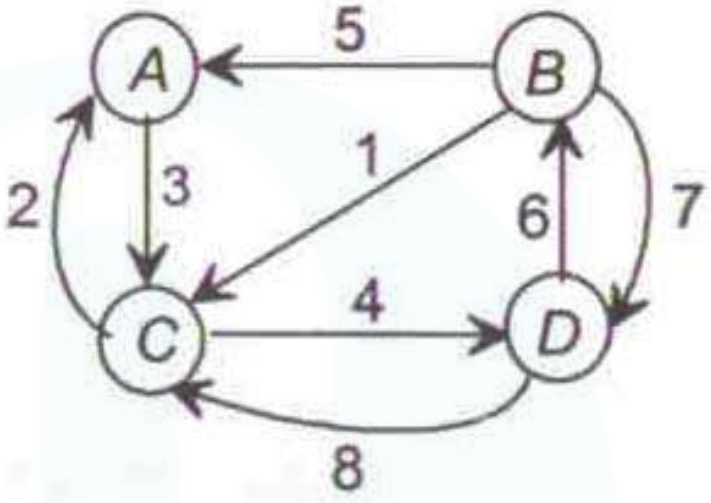
(a) Representation of graph G1



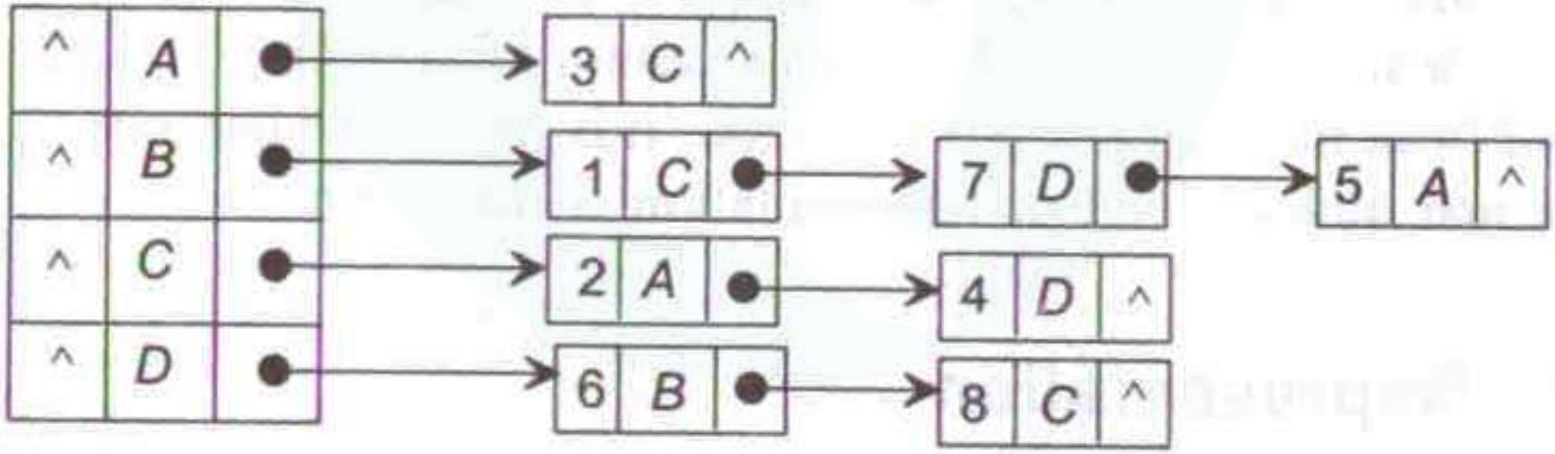
G2



(b) Representation of graph G2



G4



(d) Representation of weighted digraph G4

Adjacency List Vs Adjacency Matrix

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.
We can also have quick insertions and deletions of edges.

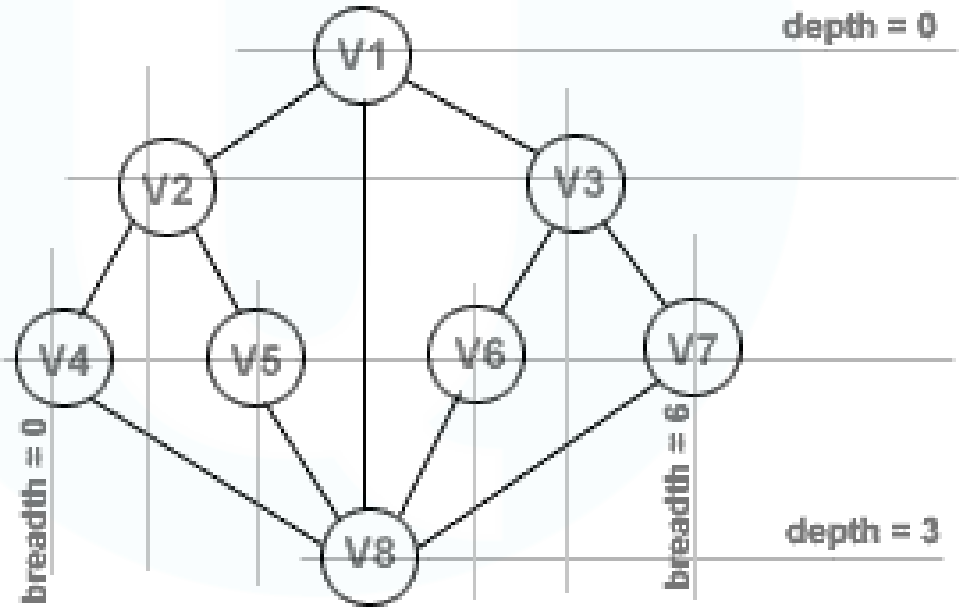
Adjacency lists use memory in proportion to the number edges, which might save a lot of memory if the adjacency matrix is sparse. It is fast to iterate over all edges, but finding the presence or absence specific edge is slightly slower than with the matrix.
Adjacency lists are the right data structure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

Graph Traversal

- Traversing a graph means visiting all the vertices in a graph.
- You can traverse a graph with the help of the following two methods:
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

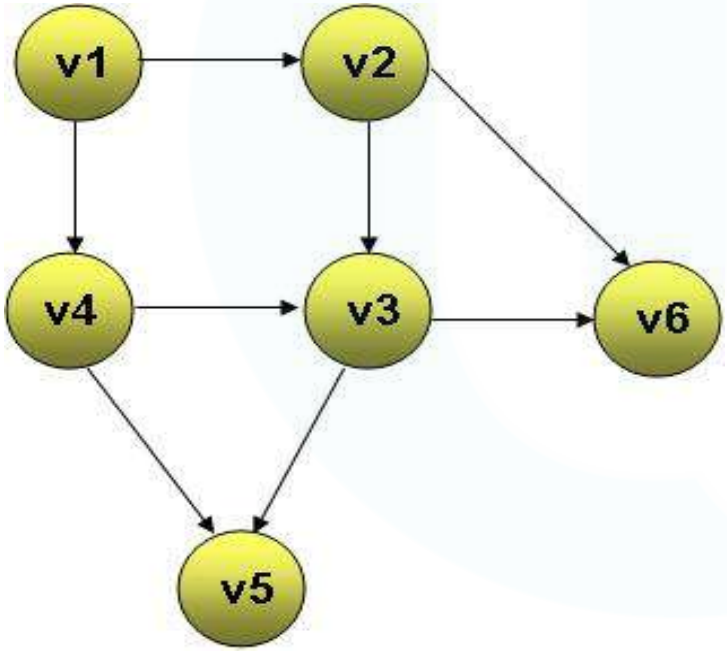
Algorithm: DFS(v)

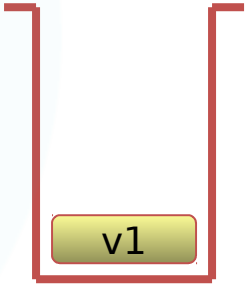
1. Push the starting vertex, v into the stack.
2. Repeat until the stack becomes empty:
   a. Pop a vertex from the stack.
   b. Visit the popped vertex.
   c. Push all the unvisited vertices adjacent to the popped vertex into the stack.



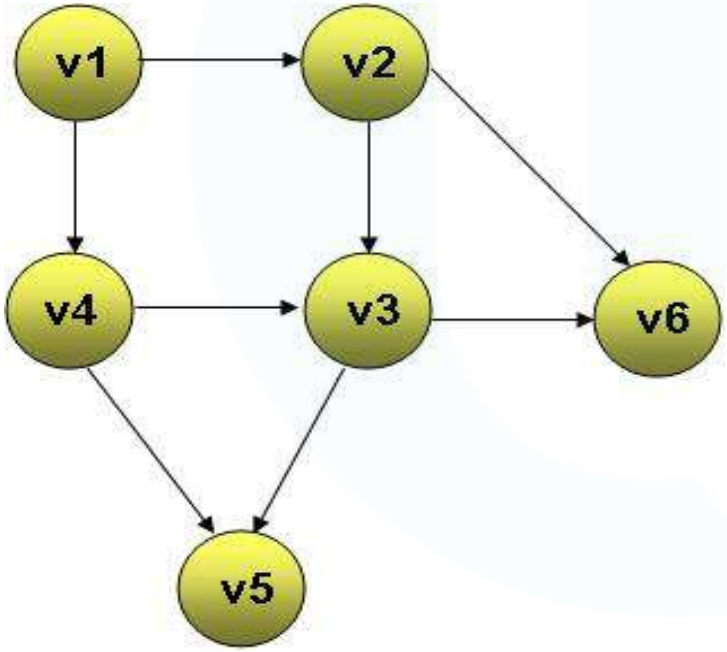
Stack is used in the implementation of the depth first search.

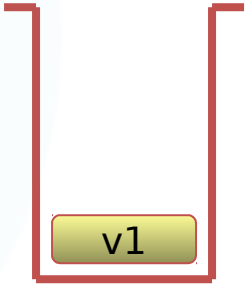
- Push the starting vertex, v1 into the stack





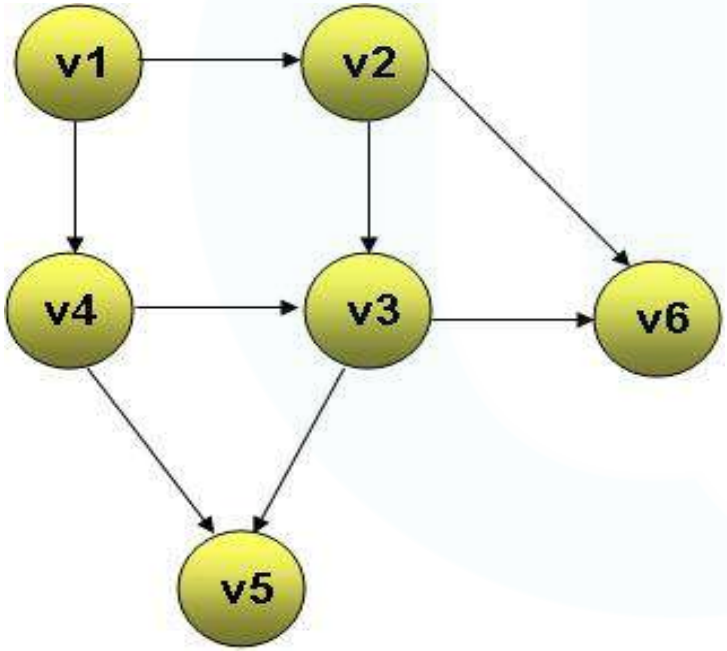
- Pop a vertex, v1 from the stack
- Visit v1
- Push all unvisited vertices adjacent to v1 into the stack

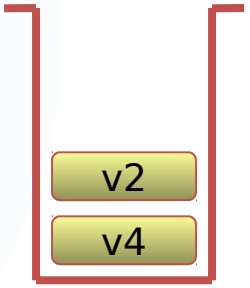




v
1

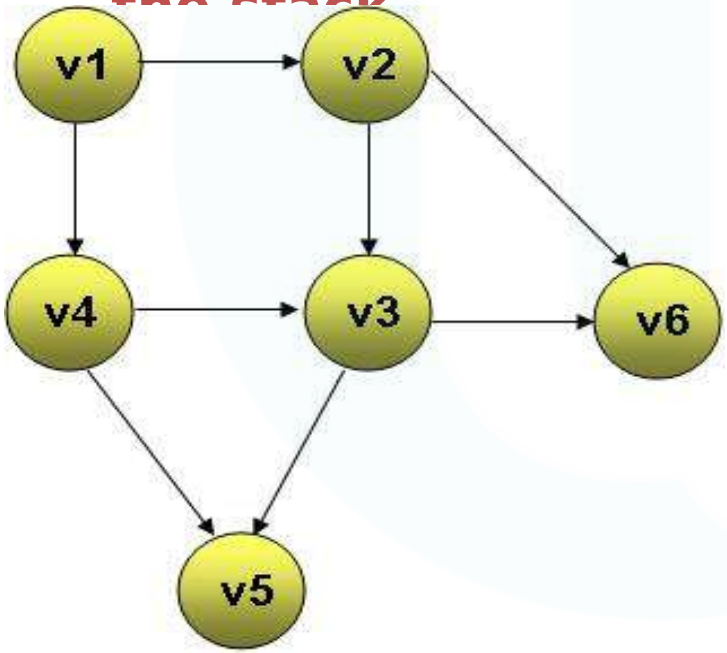
- Pop a vertex, v1 from the stack
- Visit v1
- Push all unvisited vertices adjacent to v1 into the stack

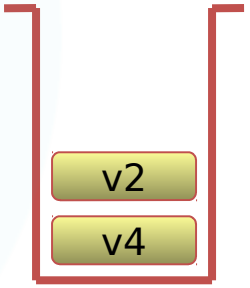




v
1

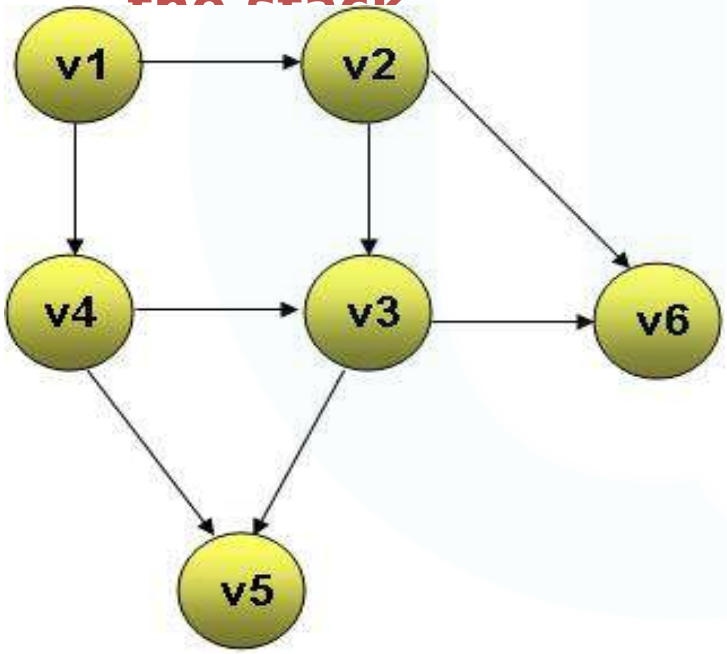
- Pop a vertex, v2 from the stack
- Visit v2
- Push all unvisited vertices adjacent to v2 into the stack

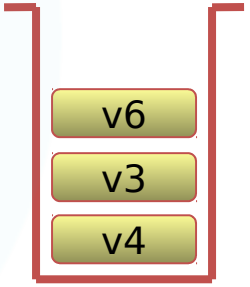




v v
1 2

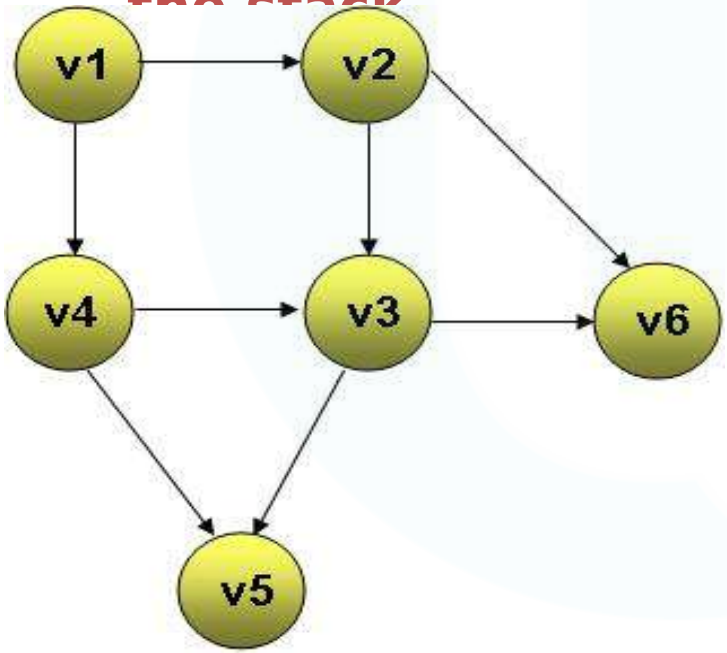
- Pop a vertex, v2 from the stack
- Visit v2
- Push all unvisited vertices adjacent to v2 into the stack

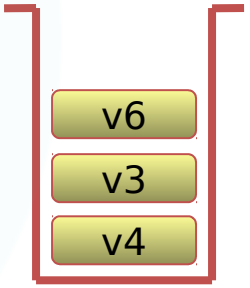




v v
1 2

- Pop a vertex, v6 from the stack
- Visit v6
- Push all unvisited vertices adjacent to v6 into the stack

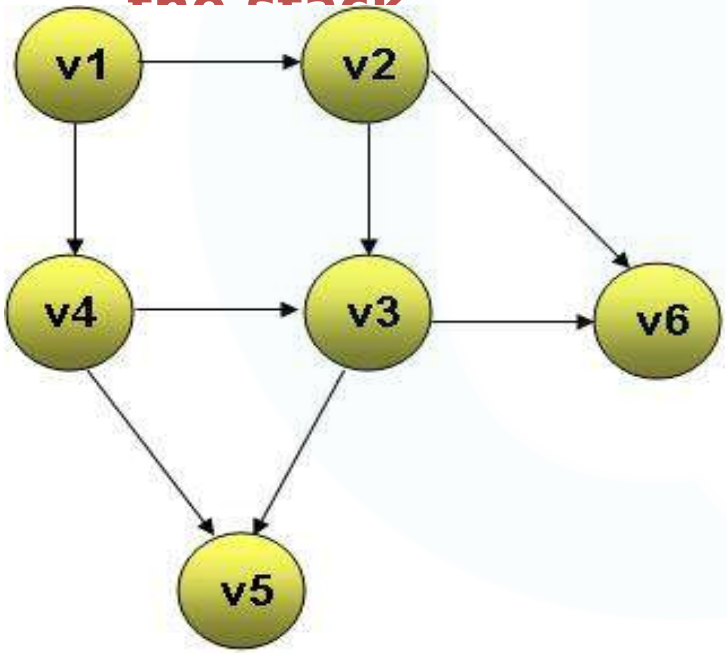


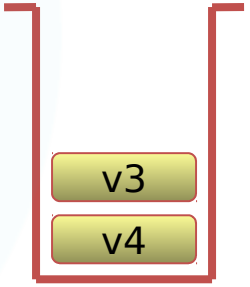


There are no unvisited vertices adjacent to v6

v1 v2 v6

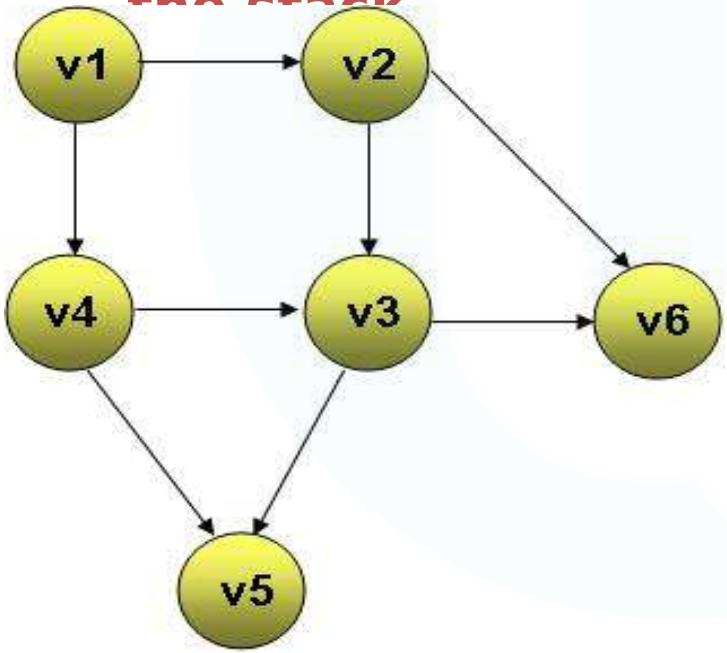
- Pop a vertex, v3 from the stack
- Visit v3
- Push all unvisited vertices adjacent to v3 into the stack

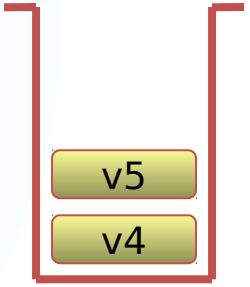




v v v v
1 2 6 3

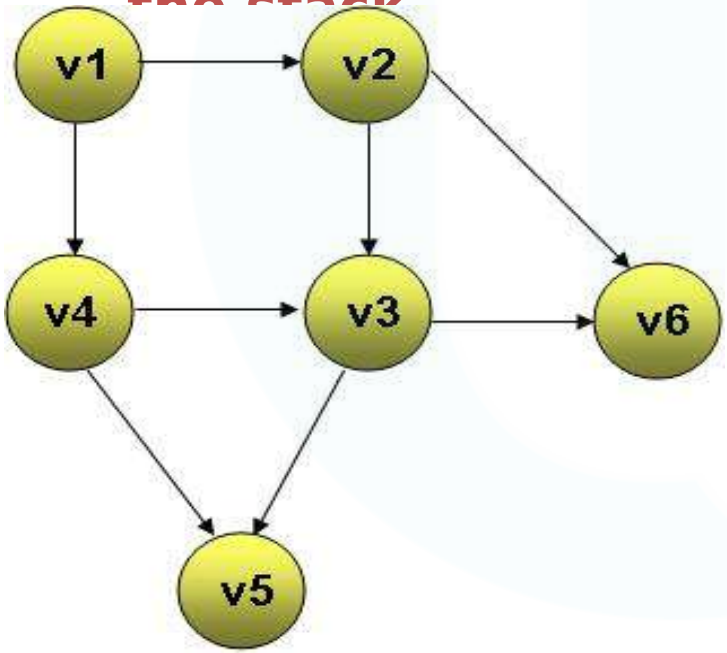
- Pop a vertex, v3 from the stack
- Visit v3
- Push all unvisited vertices adjacent to v3 into the stack

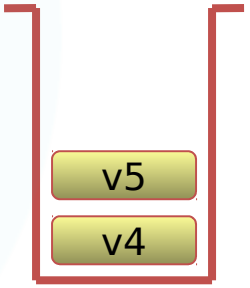




v1 v2 v6 v3

- Pop a vertex, v5 from the stack
- Visit v5
- Push all unvisited vertices adjacent to v5 into the stack

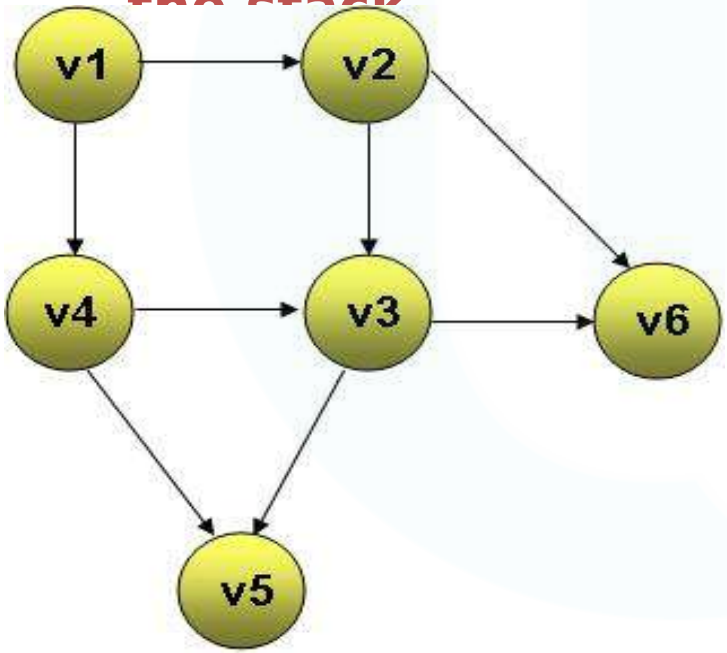


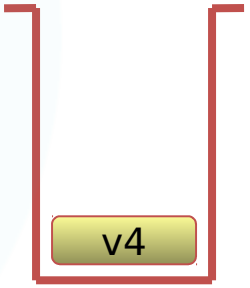


There are no unvisited vertices adjacent to v5

v1 v2 v6 v3 v5

- Pop a vertex, v4 from the stack
- Visit v4
- Push all unvisited vertices adjacent to v4 into the stack

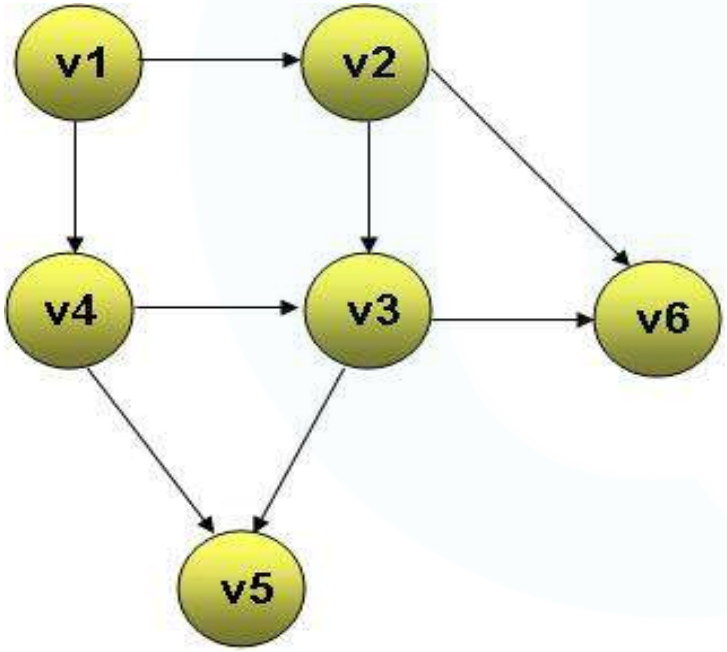


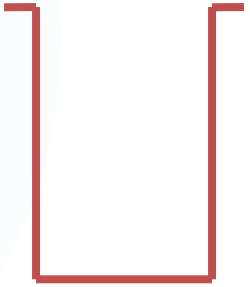


There are no unvisited vertices adjacent to v4

: v v v v v v
1 2 6 3 5 4

- The stack is now empty
- Therefore, traversal is complete





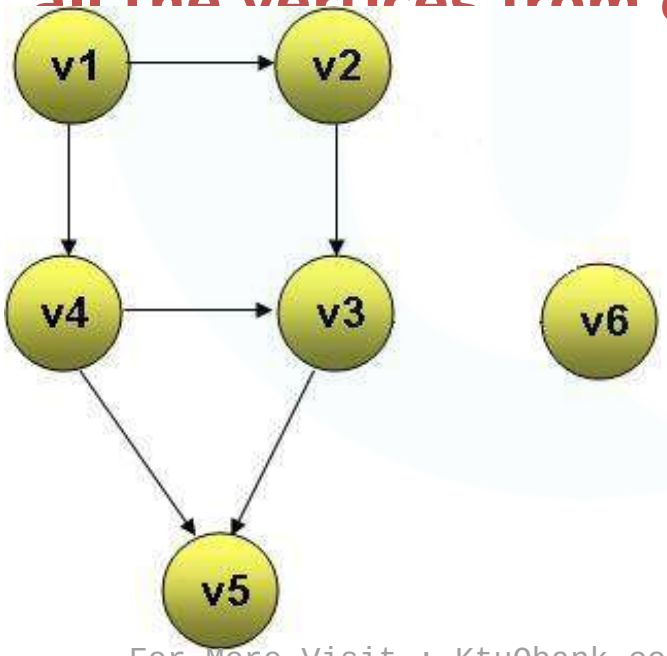
: v v v v v v
1 2 6 3 5 4

Simple Algorithm - DFS

Algorithm DFS

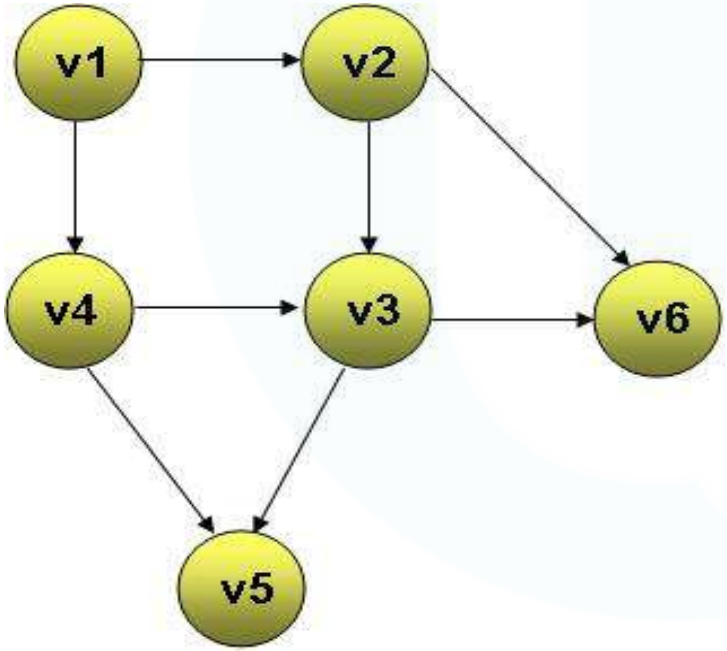
1. Push the starting vertex in to the stack STACK
2. While STACK is not empty
    1. POP a vertex V
    2. If Vis not in the VISIT
        1. Visit the vertex V
        2. Store V in VISIT_LIST
        3. PUSH all adjacent vertex of V on to STACK
    3. EndIf
3. EndWhile
4. Stop

- Although the preceding algorithm provides a simple and convenient method to traverse a graph, the algorithm will not work correctly if the graph is not connected.
- In such a case, you will not be able to traverse all the vertices from one single starting



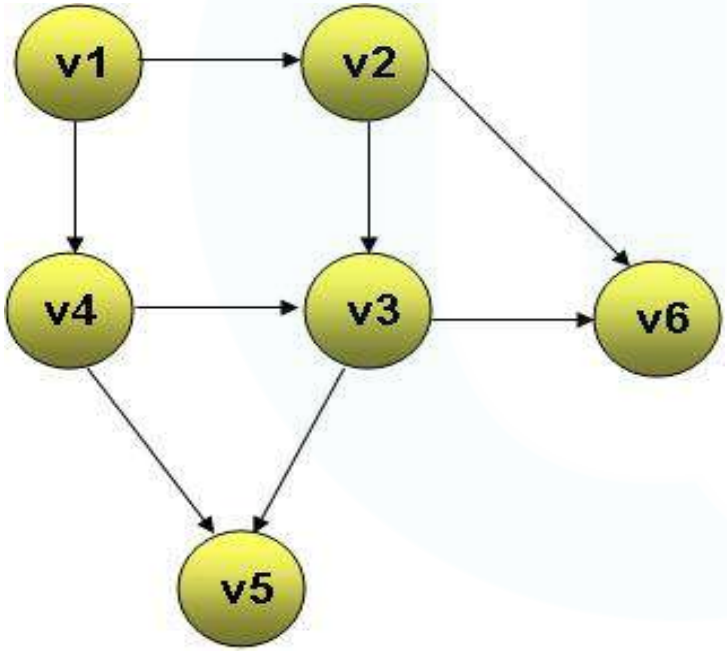
- **Algorithm: BFS(v)**
  1. **Visit the starting vertex, v and insert it into a queue.**
  2. **Repeat step 3 until the queue becomes empty.**
  3. **Delete the front vertex from the queue, visit all its unvisited adjacent vertices, and insert them into the queue.**

- **Insert v1 into the queue**



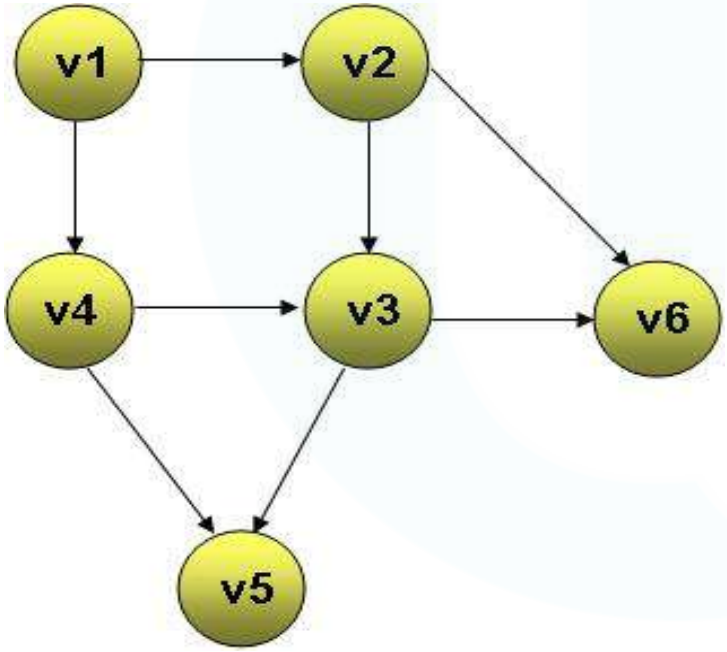
- Remove a vertex v1 from the queue
- Visit v1

- Visit all unvisited vertices adjacent to v1 and insert them in the queue



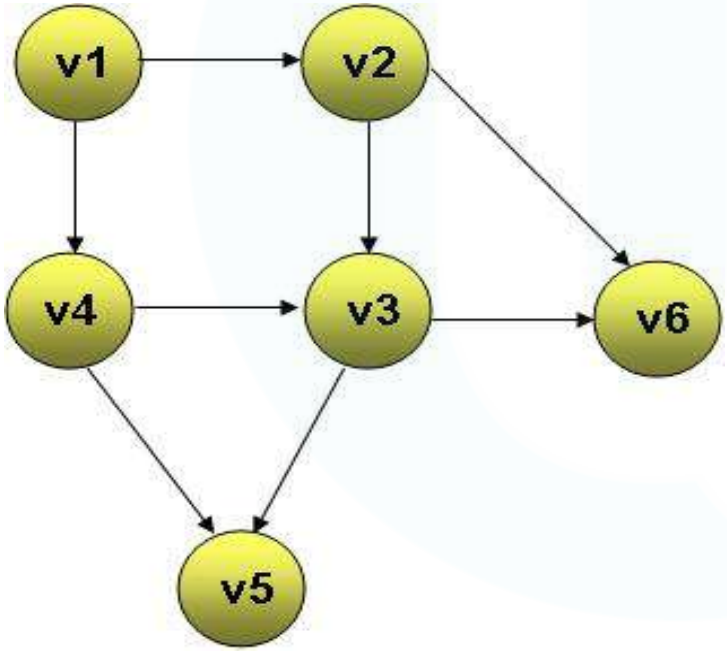
v
1

- **Visit all unvisited vertices adjacent to v1 and insert them in the queue**



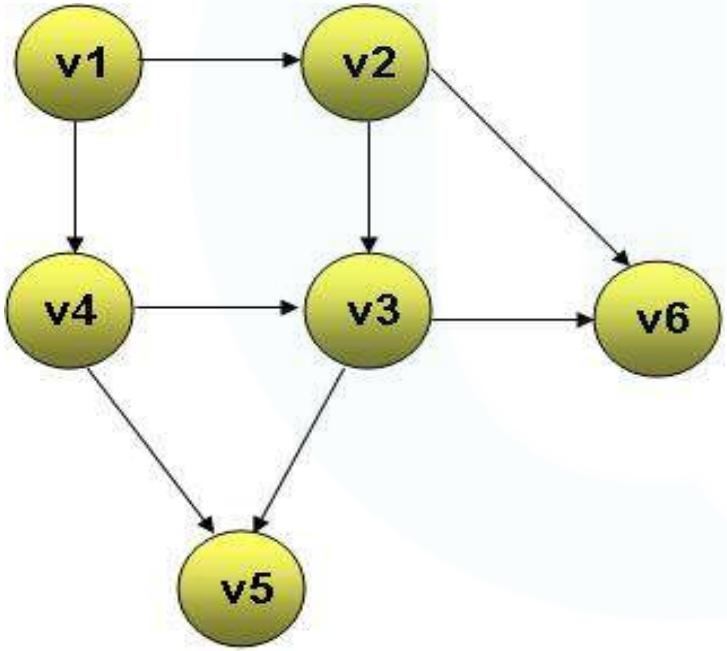
V
1

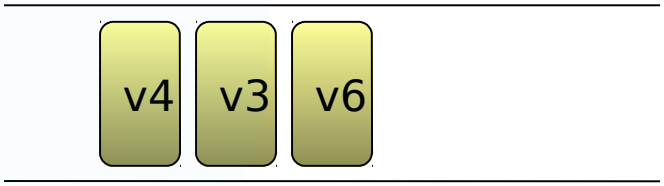
- Remove a vertex v2 from the queue
- Visit v2
- Visit all unvisited vertices adjacent to v2 and insert them in the queue



v1 v2

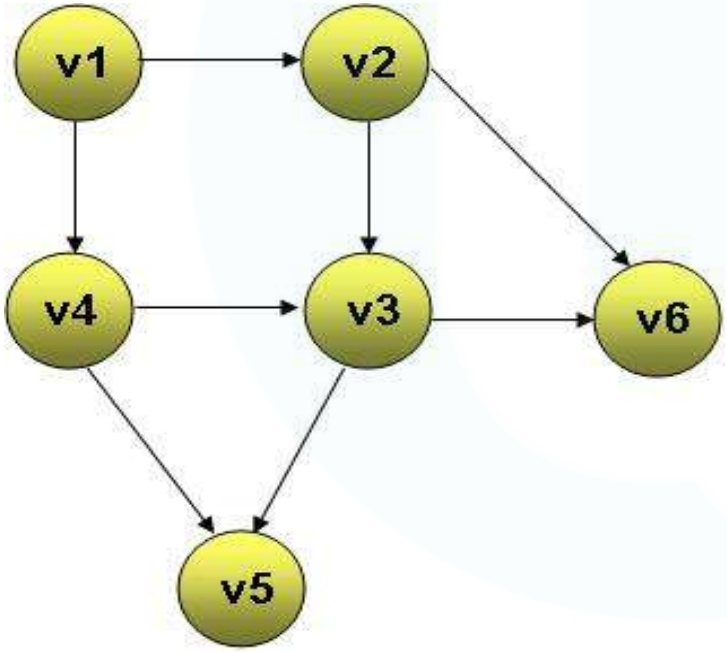
- Remove a vertex v4 from the queue
- Visit v4
- Visit all unvisited vertices adjacent to v4 and insert them in the queue

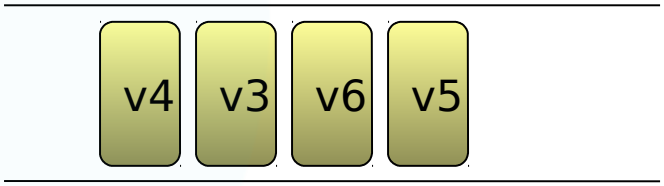




v1 v2

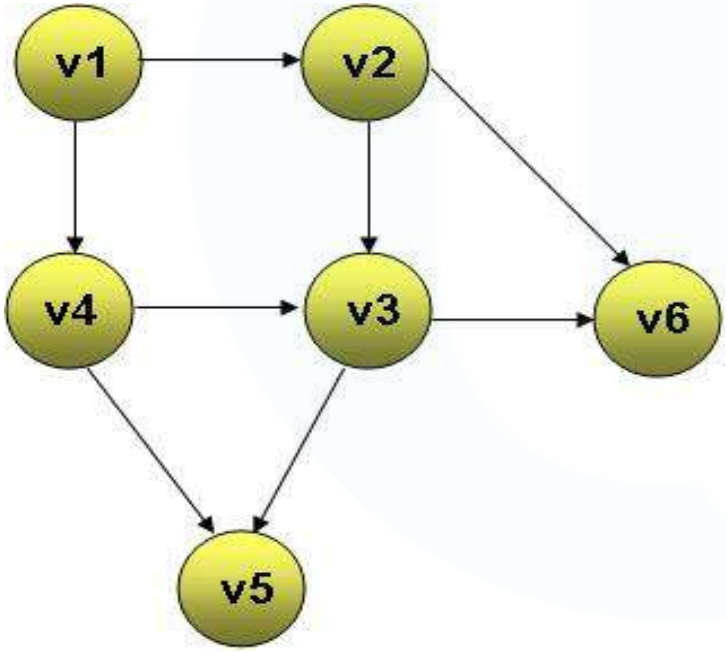
- **Visit all unvisited vertices adjacent to v4 and insert them in the queue**

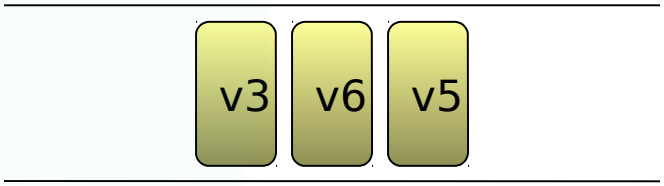




v1 v2 v4

- **Remove a vertex v3 from the queue**
- **Visit v3**
- **Visit all unvisited vertices adjacent to v3 and insert them in the queue**

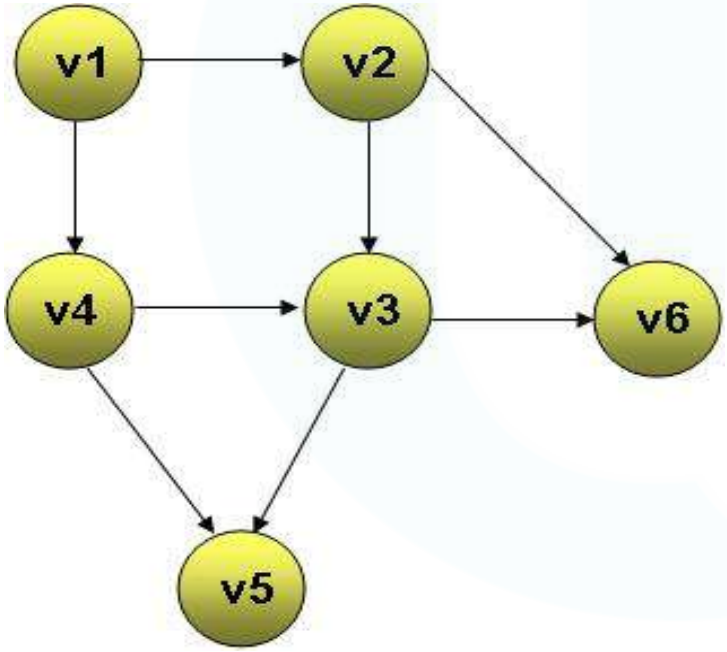


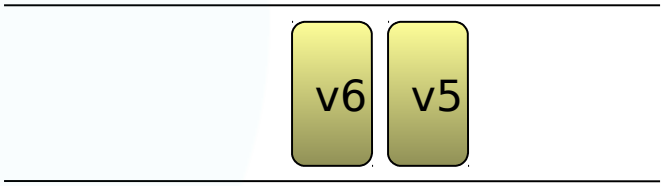


v3 does not have any unvisited adjacent vertices

v v v v
1 2 4 3

- **Remove a vertex v6 from the queue**
- **Visit v6**
- **Visit all unvisited vertices adjacent to v6 and insert them in the queue**

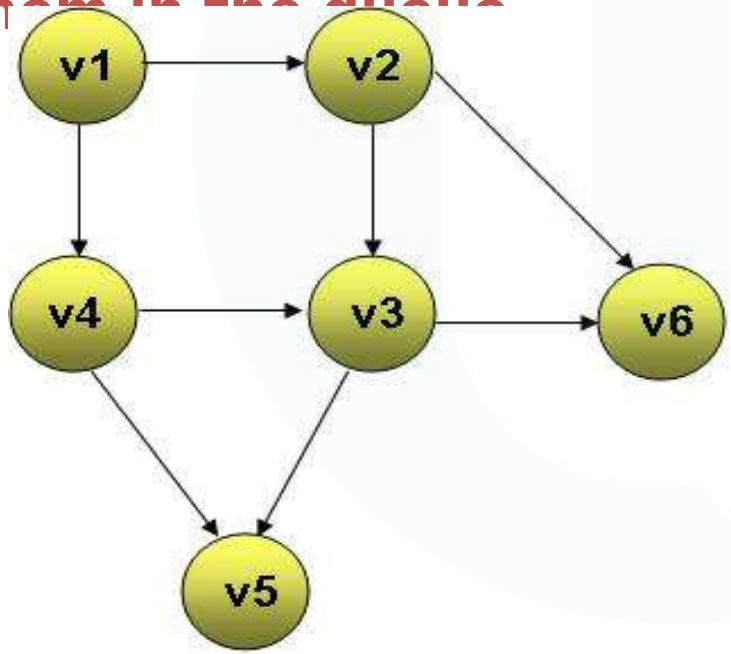


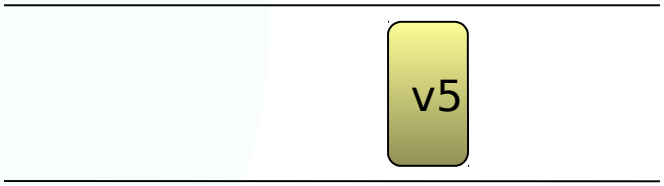


v3 does not have any unvisited adjacent vertices

v1 v2 v4 v3 v6

- Remove a vertex v5 from the queue
- Visit v5
- Visit all unvisited vertices adjacent to v5 and insert them in the queue

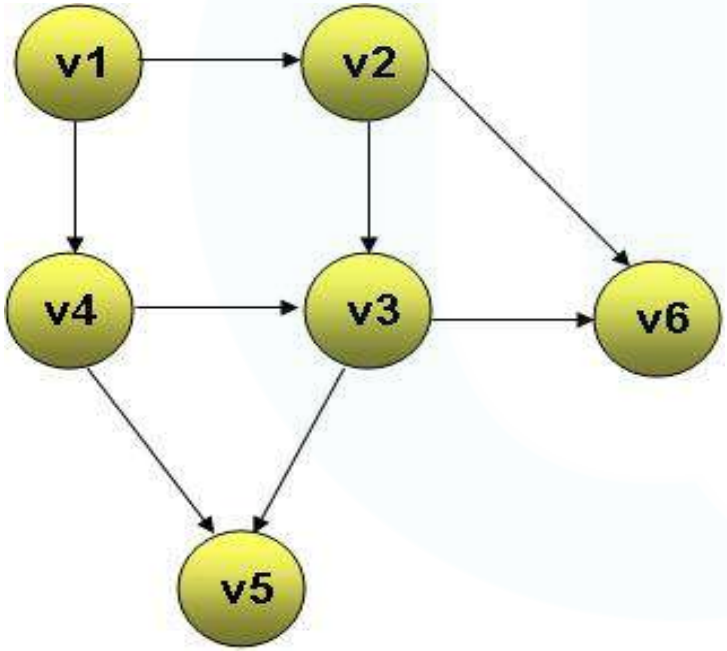




v5 does not have any unvisited adjacent vertices

v1 v2 v4 v3 v6 v5

- The queue is now empty
- Therefore, traversal is complete



v5 does not have any unvisited adjacent vertices

v1	v2	v4	v3	v6	v5

Simple Algorithm - BFS

Algorithm BFS

1. Enqueue the starting vertex in to the queue QUEUE
2. While QUEUE not empty
    1. Dequeue a vertex V
    2. If Vis not in the VISIT
        1. Visit the vertex V
        2. Store V in VISIT_LIST
        3. Enqueue all adjacent vertex of V on to QUEUE
    3. EndIf
3. EndWhile
4. Stop

- In this session, you learned that:
  - The two most commonly used ways of representing a graph are as follows:
    - Adjacency matrix
    - Adjacency list
  - Traversing a graph means visiting all the vertices in the graph.
  - In a graph, there is no special vertex designated as the starting vertex. Therefore, traversal of the graph may start from any vertex.
  - You can traverse a graph with the help of the following two methods:
    - DFS
    - BFS

- Graph theory has been instrumental in analyzing and solving problems in areas as diverse as computer network design, urban planning, finding shortest paths and molecular biology.