

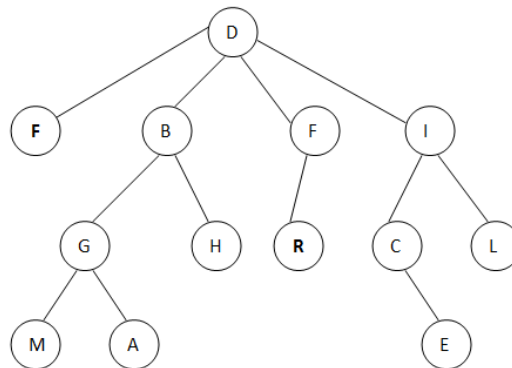
UNIT III

- Non Linear DataStructures–Trees,Tree ADT,Tree traversals,Binary Tree ADT,Expression Trees Applications of Trees,Binary search tree ADT,Threaded Binary Trees AVL Trees,B-Tre, B+Tree ,Heap,Applications of heap

Non Linear Data Structures – Trees

Tree ADT

Definition: A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r , called root, and zero or more nonempty (sub)trees T_1, T_2, \dots, T_k , each of whose roots are connected by a direct edge from r . A tree structure is shown in the figure below:



Tree Terminologies

Node

Item or information.

Root

The first and top node in a tree is called the root node. i.e. A node which doesn't have a parent is called a root.

Parent

A node that has a child is called the child's **parent node** (or ancestor node)

Edge

An edge is a connection between two nodes.

Child

A [node](#) of a [tree](#) referred to by a [parent](#) node. Every node, except the [root](#), is the child of some parent. Here Node B is the child of node A, if A is the parent of B.

Sibling

Children of the same parent are said to be siblings. Here B, C, D, E are siblings of A. Similarly I, J, K, L are siblings.

Path

A **path** in a tree is a list of distinct nodes in which successive nodes are connected by edges in the tree. There is exactly only one path from each node to root.

Leaf

A node which doesn't have children is called leaf or terminal node. (Or) Nodes at the bottommost level of the tree are called leaf nodes. Here B,K,L,G,H,M,J are leaf nodes.

Ancestor

Node A is an ancestor of node B, if A is the parent of B.

Length

The length is defined as the number of edges on the path. The length for the path A to L is 3.

Degree

The number of subtrees of a node is called its degree.

Degree of A is 4

Degree of C is 2

Degree of D is 1

Degree of H is 0.

Level of a Node

Level of the root of a tree is 0, and the level of any other node in the tree is one more than the level of its parent.

Depth of a Tree

The depth of a tree is the maximum level of any leaf in the tree (also called the height of the tree).

For any node n, the depth of n is the length of the unique path from root to n.

The depth of the root is 0.

Depth of node L is 3.

Descendant

Node B is the descendant of node A, if A is an ancestor of node B

Subtree

Any node of a tree with all of its descendants is called its subtree.

Height

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree.

For any node n, the height of the node n is the length of the longest path from n to the leaf.

The height of the leaf is always 0.

Here - height of node F is 1.

- height of L is 0.

Tree Traversals

Traversing is the process of visiting every node in the tree exactly once. Therefore, a complete traversal of a binary tree implies visiting the nodes of the tree in some linear sequence.

In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. There are no. of ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are

- **Postorder** traversal strategy
- **Preorder** traversal strategy
- **Inorder** traversal strategy

Preorder Traversal (Depth-First Order)

1. Visit the root
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

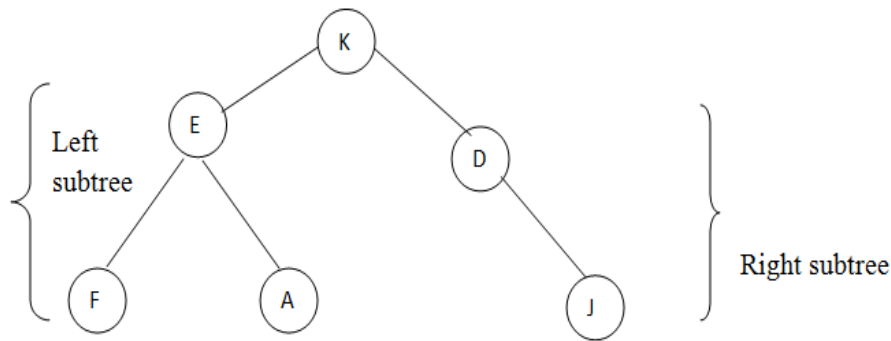
Postorder Traversal

1. Traverse the left subtree in postorder
2. Traverse the right subtree in preorder
3. Visit the root

Inorder Traversal (Symmetric Order)

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

Example



Preorder Traversal: ABDECF

- First, visit the Root **A**
- Now visit the left subtree in preorder .For left subtree, B is the root. So visit root B, then left of B and then visit right of B. It gives the traversing order **BDE**.
- Start visiting the right subtree of A in preorder. For right subtree, C is the root. So visit the root C, then left of C (In the given example there is no left of C), and then visit right of C. Now the order is **CF**.
- Finally, the preorder traversal of above tree is **ABDECF**.

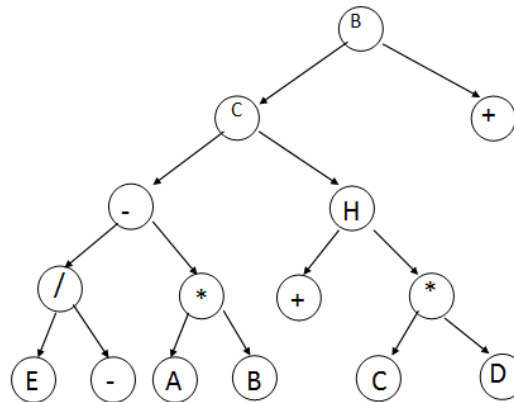
Inorder Traversal: DBEACF

- First visit the left subtree in inorder .For left subtree, B is the root. So visit left of B, then root and then visit right of B. It gives the traversing order **DBE**.
- Now visit the Root **A**.
- Start visiting the right subtree of A in inorder. For right subtree, C is the root. So visit left of C (In the given example there is no left of C), then root C and then visit right of C. Now the order is **CF**.
- Finally, the inorder traversal of above tree is **DBEACF**.

Postorder Traversal: DEBFCA

- First visit the left subtree in postorder .For left subtree, B is the root. So visit the left of B, then visit right of B and then visit root B. It gives the traversing order **DEB**.
- Start visiting the right subtree of A in preorder. For right subtree, C is the root. So left of C (In the given example there is no left of C), then visit right of C and then visit root C. Now the order is **FC**.
- Now visit the Root **A**
- Finally, the postorder traversal of above tree is **DEBFCA**.

Example



For the example given below, the traversals are given below:

Preorder

$+ - / + A B * C D * E - F G H$

Inorder :

$A + B / C * D - E * F - G + H$

Postorder:

$A B + C D * / E F G - * - H +$

Left Child Right Sibling Data Structure:

Implementation of Tree

Tree can be implemented using linked list concept. A structure can be declared with 3 elements. One element contains the data. The second element points to the first child of the tree. The third element points to the next sibling of the first child. The syntax is given below:

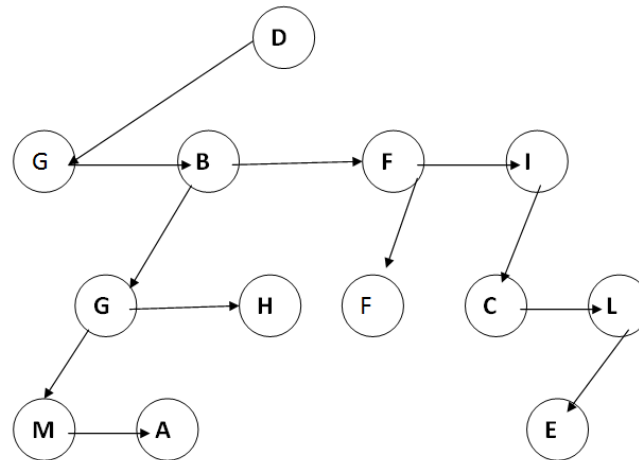
Node Declaration for Trees

```

typedef struct TreeNode * PtrToNode;

struct TreeNode
{
    ElementType element;
    PtrToNode FirstChild;
    PtrToNode NextSibling;
};
    
```

Example



Explanation

- A is the root
- B is the first child of A
- C, D & E are the next siblings of B.

Binary Tree ADT

- A binary tree is a tree in which no node can have more than two children.
- A binary tree is called strictly binary tree if every nonleaf node in the tree has nonempty left and right subtrees i.e., every nonleaf node has two children.
- A complete binary tree of depth d is a strictly binary tree with all leaf nodes at level d.

A tree is a finite set of nodes having a distinct node called root. **Binary Tree** is a tree which is either empty or has at most two subtrees, each of the subtrees also being a binary tree. It means each node in a binary tree can have 0, 1 or 2 subtrees. A left or right subtree can be empty.

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree.

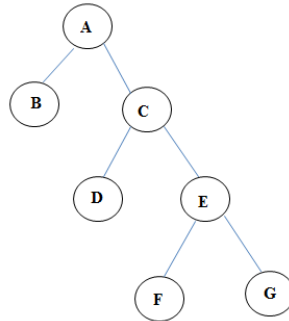
It has a distinct node called root i.e. 2. And every node has 0, 1 or 2 children. So it is a binary tree as every node has a maximum of 2 children.

If A is the root of a binary tree & B the root of its left or right subtree, then A is the parent or father of B and B is the left or right child of A. Those nodes having no children are leaf nodes. Any node say, A is the ancestor of node B and B is the descendant of A if A is either the father of

B or the father of some ancestor of B. Two nodes having same father are called brothers or Siblings.

Going from leaves to root is called climbing the tree & going from root to leaves is called descending the tree.

A binary tree in which every non leaf node has non empty left & right subtrees is called a strictly binary tree. The tree shown below is a strictly binary tree.



The no. of children a node has is called its degree. The level of root is 0 & the level of any node is one more than its father. In the strictly binary tree shown above A is at level 0, B & C at level 1, D & E at level 2 & F & g at level. The depth of a binary tree is the length of the longest path from the root to any leaf. In the above tree, depth is 3.

1.1. Linked List Representation of Binary Tree

The structure of each node of a binary tree contains one data field and two pointers, each for the right & left child. Each child being a node has also the same structure. The structure of a node is shown below.

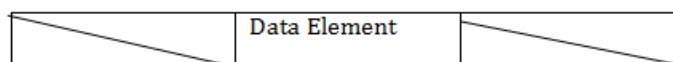
Value	
Left	Right

Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

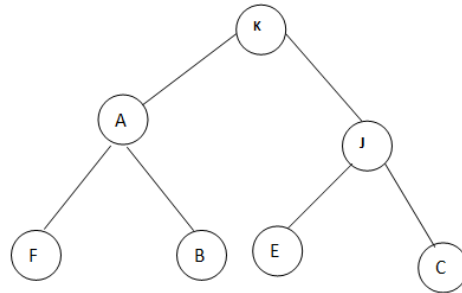
Pictorial Representation of a Binary Tree

Address of the Left Child	Data Element	Address of the Right Child
---------------------------	--------------	----------------------------

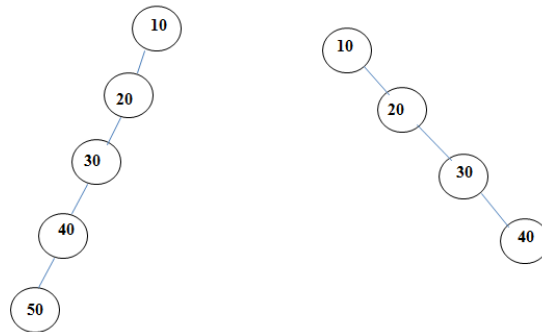
Representation of a Leaf



Note: Addresses of the Left Child and Right child are NULL.



Left and Right Skewed Trees



The tree in which each node is attached as a left child of its parent is called a left skewed tree. The tree in which each node is attached as a right child of its parent is called a right skewed tree.

Representation of Trees

There are two ways of representing the binary tree.

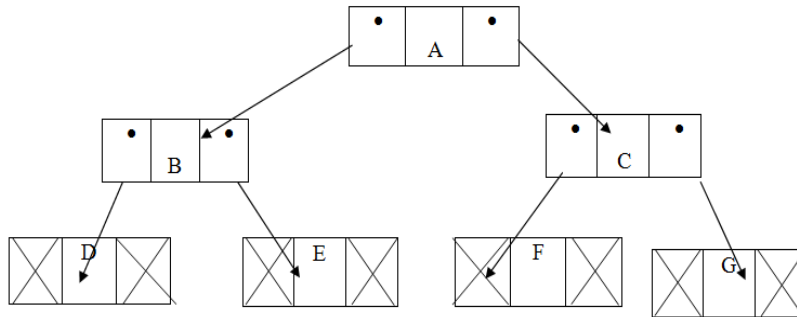
1. Sequential representation
2. Linked representation.

Let us see these representations one by one.

1. Sequential Representation of Binary Trees or Array Representation

Each node is sequentially arranged from top to bottom and understand this matter by numbering each node. The numbering of root node and then remaining nodes will give an ever increasing number direction. The nodes on the same level will be numbered from left to right. The numbering will be as shown below.

Linked List Implementation



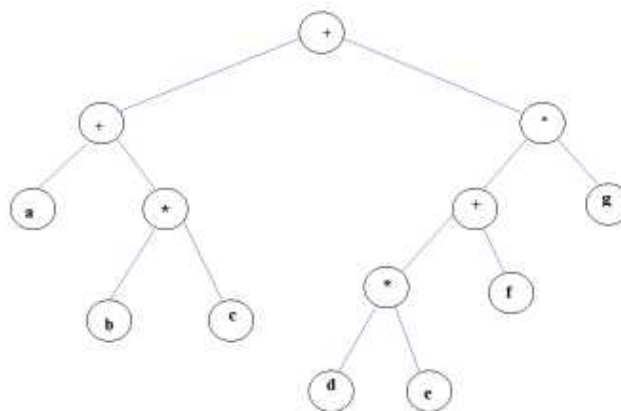
Binary Tree: Array Implementation

Binary tree can be implemented using array. The structure is given below.

```
typedef struct TreeNode *PtrToNode;
typedef struct TreeNode *Tree;
struct TreeNode
{
    ElementType Element;
    Tree Left;
    Tree Right;
};
```

Expression Tree

Expression tree is also a binary tree in which the leaf or terminal nodes are operands and non-terminal or intermediate nodes are operators used for traversal.



To construct an Expression Tree the Following Steps are to be Followed

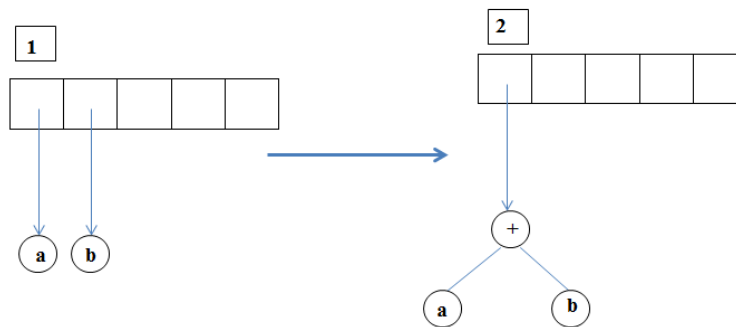
- Read out the given expression one symbol at a time.
- If the symbol is an operand, create a one-node tree and push a pointer to it onto a stack.

- If the symbol is an operator, pop pointers to two trees T1 and T2 from the stack and form a new whose root is the operator and whose left and right children point to T2 and T1, respectively.
- A pointer to this new tree is then pushed onto the stack.

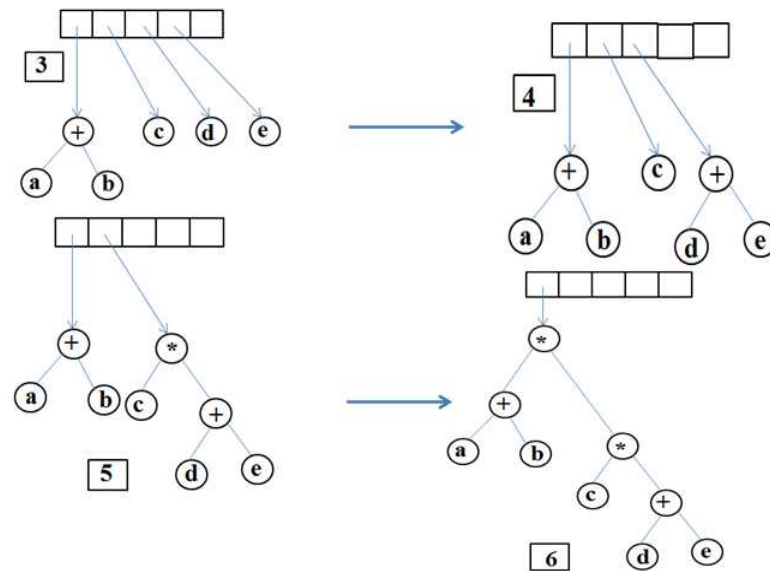
Example

Input: **ab+cde+****

1. The two symbols are operands, so create one-node trees and push pointers to them onto a stack.



2. Next + is read , so two pointers to trees are popped, a new tree is formed, a new tree is formed, and a pointer to it is pushed onto the stack.
3. Now, c, d and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.
4. Now '+' is read, so two trees are merged.
5. Continuing, '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.
6. Finally, the last symbol '*' is read, two trees are merged, and a pointer to the final tree is left on the stack.



Applications of Trees

There are many applications for trees. One of the popular uses is the directory structure in many common operating systems, including UNIX and DOS.

- The root of this directory is /usr.
- /usr has 3 children namely mark, alex and bill, which are directories.
- ./usr contains 3 directories and no regular files.
- The filename /usr/mark/book/ch1.r is obtained by following the leftmost child 3 times.
- Each / after the first name indicates an edge; the result is the full pathname.

Advantages of Hierarchical File System

1. It allows users to organize the data logically.
2. Two files in different directories can share the same name, because their path is different from the root.

Routine to List a Directory in a Hierarchical File System

Static void listdir(directoryorfile D, int depth)

```
{
    if(D is a legitimate entry)
    {
        printname(D, depth);
    }
}
```

```
    if(D is a directory)
        for each child, c, of D
            listdir(C, depth+1);
    }
```

Explanation

- This is executed by a recursive function call listdir.
- This procedure starts with the directory name and the depth 0. (Note: The depth of the root is 0).
- D can be either a directory or a file.
- It prints entry of D.
- If D is a Directory, it prints the children C recursively.
- This procedure terminates when the parameter for listdir is a not a valid parameter.

Routine which Call the Listdir Routine

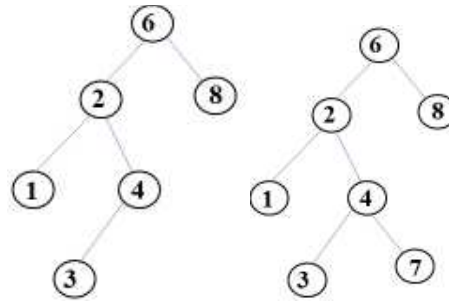
```
void listdirectory(directoryorfile D)
{
    listdir(D,0);
}
```

Explanation

- This routine call listdir with two parameters D and 0, where 0 denotes the depth of the root.

Binary Search Tree ADT

For every node, X, in the tree, the values of all the keys in its left subtree are smaller than the key value of X, and the values of all the keys in its right subtree are larger than the key value of X.



BST: Implementation

```

Struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;
SearchTree MakeEmpty( SearchTree T );
Position Find (ElementType X, SearchTree T);
Position FindMin( SearchTree T );
Position FindMax( SearchTree T );
SearchTree Insert (ElementType X, SearchTree T );
SearchTree Delete (ElementType X, SearchTree T );
ElementType Retrieve (Position P);

```

Binary Search Tree Declaration

```

struct TreeNode
{
    ElementType Element;
    SearchTree Left;
    SearchTree Right;
};

```

BST Implementation: MakeEmpty

This operation is mainly for initialization. The implementation follows recursive function call technique.

Procedure

```

SearchTree MakeEmpty( SearchTree T )
{
    if( T != NULL )
    {

```

```

        MakeEmpty( T→Left );
        MakeEmpty( T→ Right );
        free( T );
    }
    return NULL;
}

```

- This function uses recursive function call.
- This function initially makes the left subtrees empty.
- Then the function makes the right subtrees empty.

BST Implementation: Find

This operation generally requires returning a pointer to the node in tree T that has key X, or NULL if there is no such node.

- If T is NULL, then we can just return NULL.
- Otherwise, if the key stored at T is X, return T.
- Otherwise make a recursive call on a subtree of T, either left or right.
- If X is less than the key stored at T, then search Left subtree, otherwise search the right tree recursively.

Procedure

```

Position Find( ElementType X, SearchTree T )
{
    if( T == NULL )
        return NULL;
    if ( X < T → Element )
        return Find( X, T → Left );
    else if ( X > T → Element )
        return Find( X, T → Right );
    else
        return T;
}

```

Note: Here, key denotes the data element.

- The function compares the searching element with the element in the root.
- If the element is less than the root, it searches the left subtree recursively.
- Otherwise it searches the right subtree recursively.
- This function returns the element if it is found in the tree.
- Otherwise it returns NULL

BST Implementation: FindMin

This routine will return the position of the smallest elements in the tree. To perform the **FindMin** start at the root and go left as long as there is a left child. The stopping point is the smallest element, which will be always the left most child of the tree.

Procedure

```
Position FindMin( SearchTree T )
{
    if ( T == NULL )
        return NULL;
    else if( T → Left == NULL )
        return T;
    else
        return FindMin( T → Left );
}
```

- In a Binary Search Tree, The minimum element is available in the left subtree.
- Hence, this function searches the left subtree of the tree.
- Searching takes place recursively.
- The last node in the left subtree contains the minimum element. If T->left is NULL, then it returns the element in that node.
- If no tree is available, i.e T=NULL, then it returns NULL.

BST Implementation: FindMax

This routine will return the position of the largest elements in the tree. To perform the **FindMax** start at the root and go right as long as there is a right child. The stopping point is the largest element.

```
Position FindMax( SearchTree T )
{
```

```

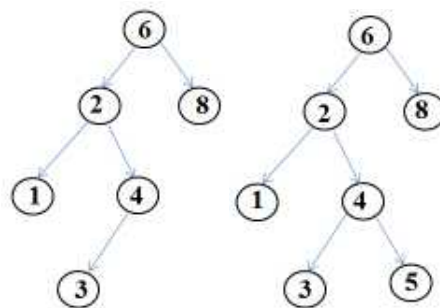
        if ( T != NULL )
            while ( T → Right != NULL )
                T = T → Right;
            return T;
        }

```

- In a Binary Search Tree, The maximum element is available in the right subtree.
- Hence, this function searches the right subtree of the tree.
- If T->right is not equal to NULL, T moves to T->right. This is done till T->right is equal to NULL.
- The last node in the right subtree contains the maximum element. If T->right is NULL, then it returns the element in that node.
- If no tree is available, i.e T=NULL, then it returns NULL.

BST Implementation: Insert

- To insert X in to tree T, proceed down the tree with a **Find**. If X is found do nothing because BST will not have duplicate values. Otherwise, insert X at the appropriate spot on the path traversed.



BST Implementation: Insert

In this routine T points to the root of the tree, and the root changes on the first insertion.

Insert is written as a function that returns a pointer to the root of the new tree.

```

SearchTree Insert( ElementType X, SearchTree T )
{
    if ( T == NULL )
    {
        T = malloc( sizeof( struct TreeNode ) );
        if ( T == NULL )
            FatalError( "Out of space!!!" );
        else

```



```

    {
        T → Element = X;
        T → Left = T → Right = NULL;
    }
}

else if ( X < T → Element )
    T → Left = Insert( X, T → Left );
else if( X > T → Element )
    T → Right = Insert( X, T → Right );
/* Else X is in the tree already; we'll do nothing */
return T; /* Do not forget this line!! */
}

```

Explanation

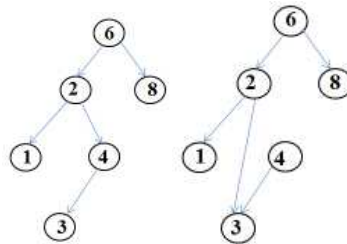
1. This function has two parameters namely, the element X to be inserted and T, the address of the root of the tree.
2. If T is NULL, a new node is created and the address is stored in T, else go to 5
3. If T is NULL, it implies that memory allocation is not done and terminates.
4. Otherwise the new element X is inserted into the node T. The left and right pointer is made NULL.
5. The element X is compared with the element present in T. If $X < T \rightarrow \text{element}$, then the element is to be inserted in the left. A recursive function call is made with two parameters X and $T \rightarrow \text{left}$.
6. Otherwise if $X > T \rightarrow \text{element}$, then the element is to be inserted in the right. A recursive function call is made with two parameters X and $T \rightarrow \text{right}$.
7. Otherwise, it implies that X is already present in the BST. Hence the element cannot be inserted.
8. After the element is inserted in the appropriate position, the address of the new tree is returned to the main function.

BST Implementation: Delete

If the node is a leaf it can be deleted immediately.

If the node has one child, the node can be deleted after its **parents** adjust a pointer to bypass the node.

The following figure shows an initial tree and the result of a deletion; the key value is 2. It is replaced smallest data in its right subtree(3), and then that node is deleted as before.



BST Implementation: Delete

```

SearchTree Delete( ElementType X, SearchTree T )
{
    Position TmpCell;
    if ( T == NULL )
        Error( "Element not found" );
    else if ( X < T → Element ) /* Go left */
        T → Left = Delete( X, T → Left );
    else if ( X > T → Element ) /* Go right */
        T → Right = Delete( X, T → Right );
    else /* Found element to be deleted */

    if ( T → Left && T → Right ) /* Two children */
    {
        TmpCell = FindMin( T → Right );
        T → Element = TmpCell → Element;
        T → Right = Delete( T → Element, T → Right );
    }
    else /* One or zero children */
    {
        TmpCell = T;
        if ( T → Left == NULL ) /* Also handles 0 children */
            T = T → Right;
    }
}

```

```

        else if ( T → Right == NULL )
            T = T → Left;
        free( TmpCell );
    }
    return T;
}

```

Explanation

1. This function has two parameters namely, the element X to be deleted and T, the address of the root of the tree.
2. If T is NULL, it implies that the tree is not present and the function terminates.
3. Otherwise, the element X is compared with the element present in T. If $X < T \rightarrow \text{element}$, then the element to be deleted is present in the left. A recursive function call is made with two parameters X and $T \rightarrow \text{left}$.
4. Otherwise if $X > T \rightarrow \text{element}$, then the element to be deleted is present in the right. A recursive function call is made with two parameters X and $T \rightarrow \text{right}$.
5. If the element is found, it checks where the node contains two children. If yes, It finds the minimum element from the two children and the minimum element is inserted into the node where the element is deleted.
6. If the node contains one or zero children, the left or children node value is inserted into the node where the element is deleted.
7. The function returns the address of the root of the newly formed tree after deletion to the main function.

Program

```

// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int key;
    struct node *left, *right;
};

```

// A utility function to create a new BST node

```
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

// A utility function to do inorder traversal of BST

```
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}
```

/* A utility function to insert a new node with given key in BST */

```
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
```

```

        node->right = insert(node->right, key);

        /* return the (unchanged) node pointer */
        return node;
    }

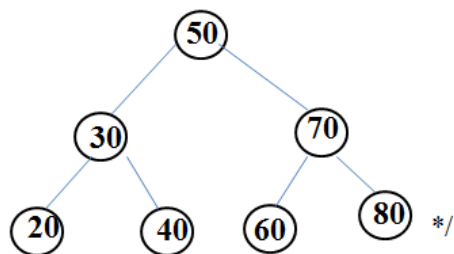
```

// Driver Program to test above functions

```
int main()
```

```
{
```

```
    /* Let us create following BST
```



```

struct node *root = NULL;

    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    (root);
    return 0;
}

```

Output:

20

30

40

50

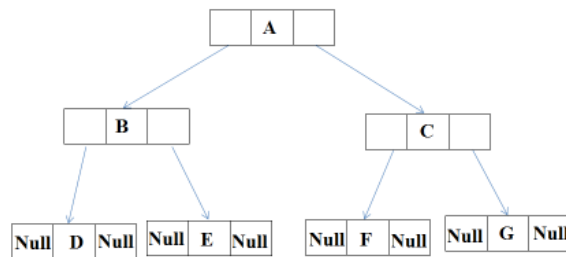
60

70

80

Threaded Binary Tree

A binary search tree in which each node uses an otherwise-empty left child link to refer to the node's in-order predecessor and an empty right child link to refer to its in-Order Successor.



In above binary tree, there are 8 null pointers & actual 6 pointers.

In all there are 14 pointers.

We can generalize it that for any binary tree with n nodes there will be $(n+1)$ null pointers and $2n$ total pointers.

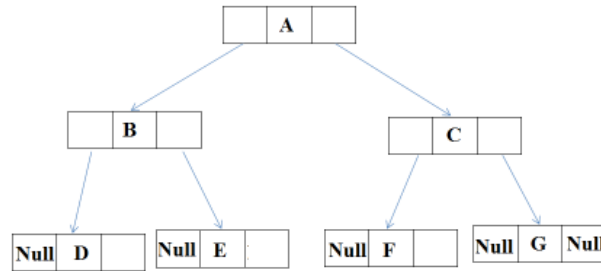
The objective here to make effective use of these null pointers. J. perils & C. Thornton jointly proposed idea to make effective use of these null pointers.

According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.

And binary tree with such pointers are called threaded tree. In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

Threaded Binary Tree: One-Way

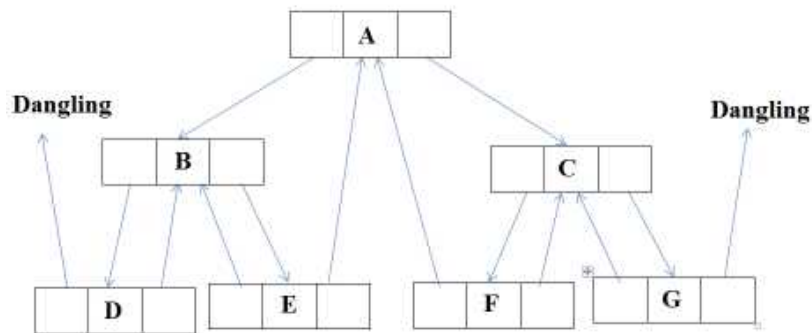
We will use the right thread only in this case. To implement threads we need to use in-order successor of the tree.



Inorder Traversal of The tree: D, B, E, A, F, C, G

Two Way Threaded Tree/Double Threads

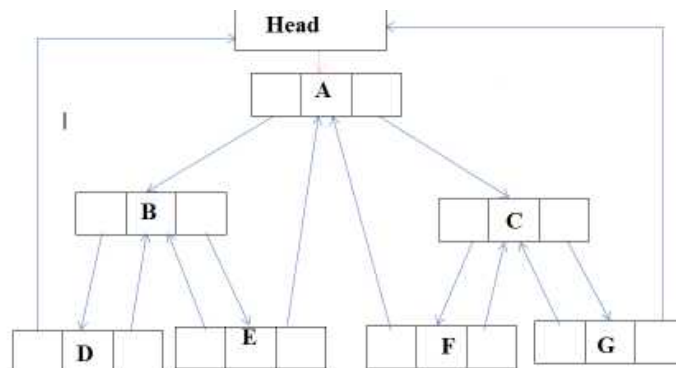
Again two-way threading has left pointer of the first node and right pointer of the last node will contain the null value. The header nodes is called two-way threading with header node threaded binary tree.



Inorder Traversal of The tree: D, B, E, A, F, C, G E C F G

Dangling can be Solved as Follows

Introduce a header node. The left and right pointer of the header node are treated as normal links and are initialized to point to header node itself.



Inorder Traversal of The tree: D, B, E, A, F, C, G E C F G

Node Structure

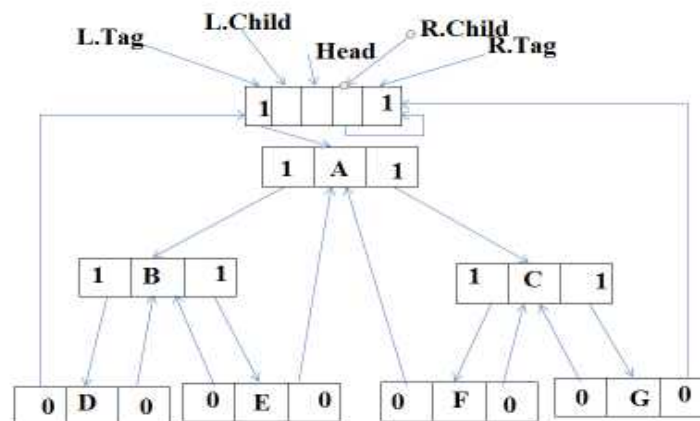
For the purpose of our evaluation algorithm, we assume each node has five fields:

LTag	Left	data	Right	RTag
------	------	------	-------	------

We define this node structure in C as:

```

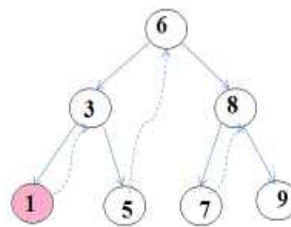
Struct Node{
    int data;
    struct Node *Left,*Right;
    bool Ltag;
    bool Rtag;
};
    
```



Inorder Traversal of The tree: D B E A F C G

Threaded Tree Traversal

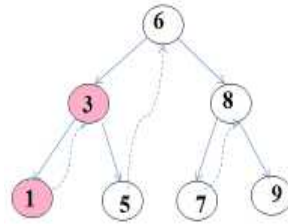
- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right.
- If we follow a link to the right, we go to the leftmost node, print it, and continue.



Start at leftmost node, print it.

Output

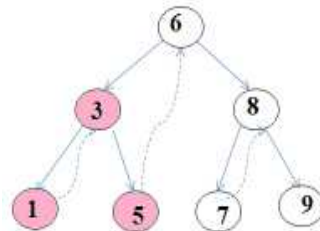
1



Follow thread to right, print it.

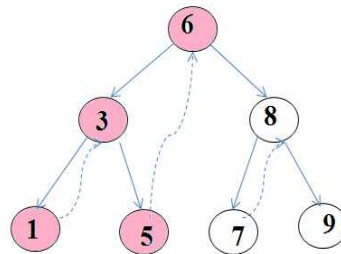
Output

13



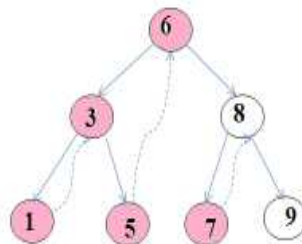
Follow link to right, go to leftmost node and print

Output 1 3 5



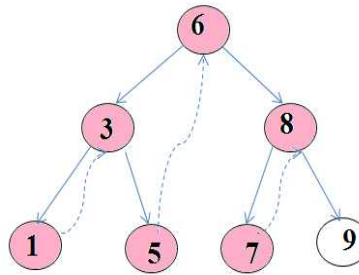
Follow thread to right, print node

Output 1 3 5 6



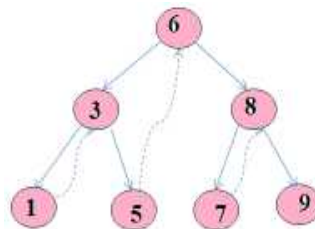
Follow link to right, go to leftmost node and print

Output 1 3 5 6 7



Follow thread to right, print node

Output 1 3 5 6 7 8



Follow link to right, go to leftmost node and print

Output 1 3 5 6 7 8 9

```
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);
        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else
            // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    } }
```

Comparison of Threaded Binary Tree with Normal Binary Tree

Threaded Binary Tree	Normal Binary Tree
----------------------	--------------------

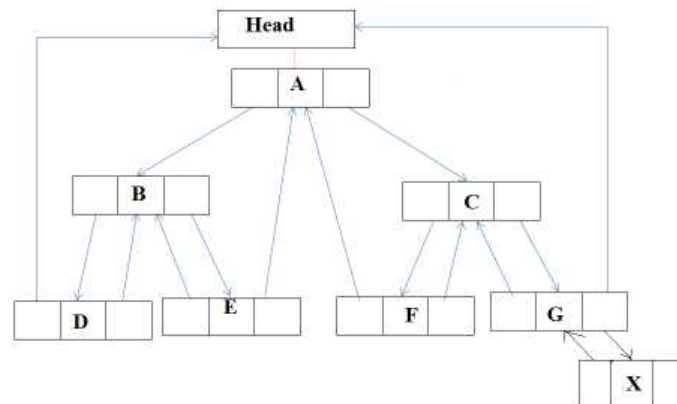
- | | |
|---|--|
| <ul style="list-style-type: none"> • In threaded binary trees, The null pointers are used as thread. • We can use the null pointers which is a efficient way to use computers memory. • Traversal is easy. • Completed without using stack or recursive function. • Structure is complex. • Insertion and deletion takes more time. | <ul style="list-style-type: none"> • In a normal binary trees, the null pointers remains null. • We can't use null pointers so it is a wastage of memory. • Traverse is not easy and not memory efficient. • Less complex than Threaded binary tree. • Less Time consuming than Threaded Binary tree. |
|---|--|

Inserting a node to Threaded Binary Tree

Inserting a node X as the right child of a nodes.

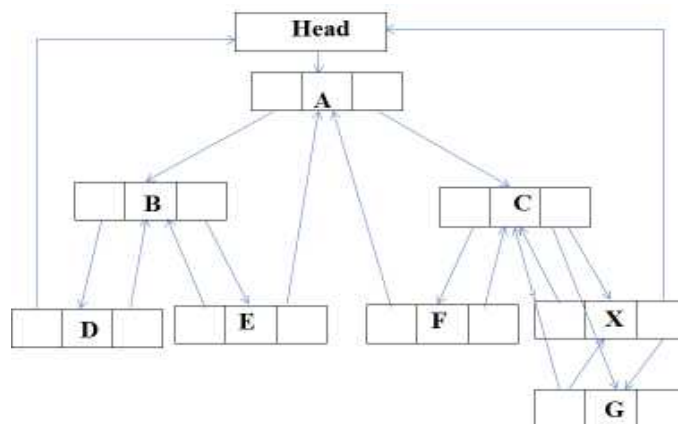
1st Case

If G has an empty right subtree, then the insertion is simple



Inserting X as a right child of G. New inorder traversal is: D,B,E,A,F,C,G,X

2nd Case: If the right subtree of C is not empty, then this right child is made the right child of X after insertion.



New Inorder Traversal of The tree: D, B, E, A, F, C, X, G

Advantages	Disadvantages
<ol style="list-style-type: none"> 1. By doing threading we avoid the recursive method of traversing a Tree , which doesn't use of stack and consumes a lot of memory and time . 2. The node can keep record of its root . 3. Backward Traverse is possible. 4. Applicable in most types of the binary tree. 	<ol style="list-style-type: none"> 1. This makes the Tree more complex . 2. More prone to errors when both the child are not present & both values of nodes pointer to their ancestors. 3. Lot of time consumes when deletion or insertion is performed.

Applications

Same as any kind of Binary Tree.

Used in search and Traverse based work.

AVL Tree

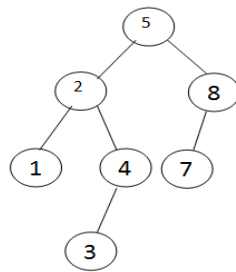
An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition. For every node in the AVL tree, the height of left subtree and the height of the right subtree can differ by at most one.

Height of left subtree- Height of right subtree ≤ 1

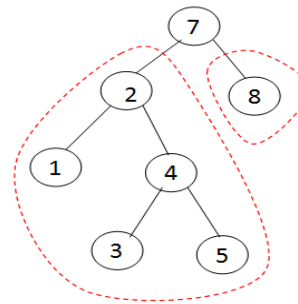
The height of the left subtree minus the height of the right subtree of a node is called the balance factor of the node. For an AVL tree, the balances of the nodes are always -1, 0 or 1.

- The height of an empty tree is defined to be -1.

- Given an AVL tree, if insertions or deletions are performed, the AVL tree may not remain height balanced.



An AVL Tree



Not an AVL Tree

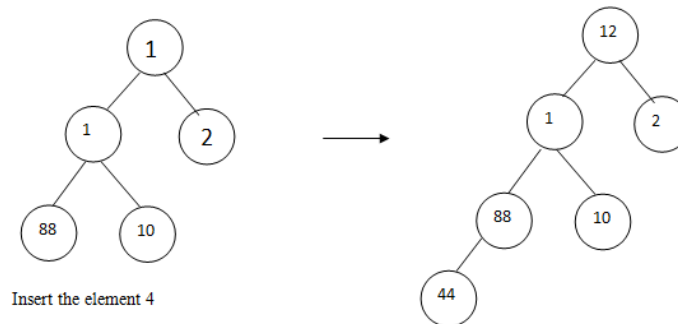
Balancing Trees

The tree becomes unbalanced whenever a node is inserted or deleted. The unbalanced tree is balanced by rotating the tree to the left or right.

There are four cases that require rebalancing. All unbalanced trees fall into one of these four cases:

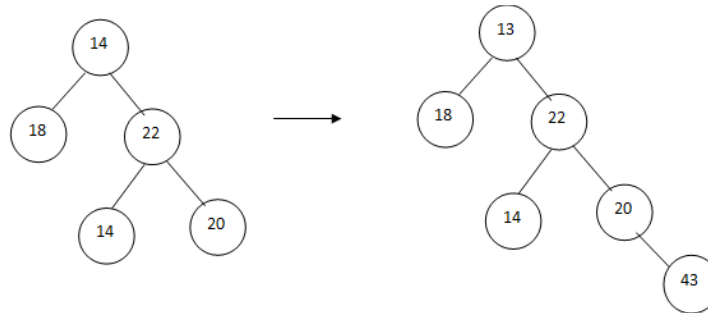
- Left of Left** - A sub tree of a tree that is left high has become left high after the insertion of an element.

Diagram



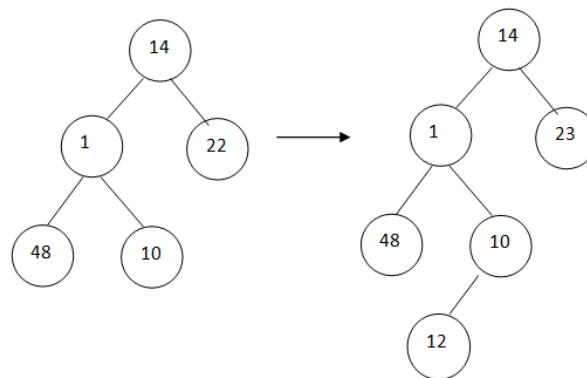
- Right of right** - A sub tree of a tree that is right high has become right high after inserting an element.

Diagram



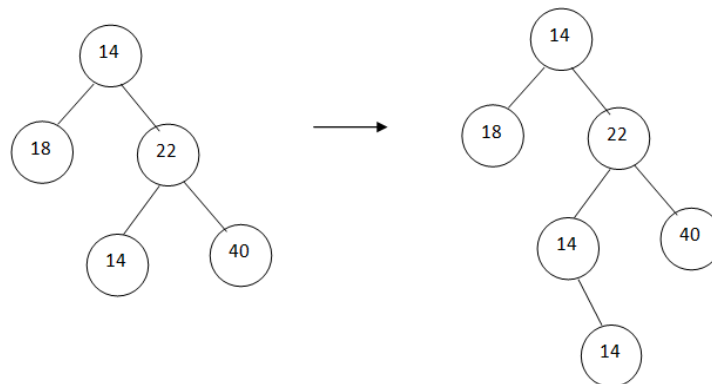
3. **Right of Left** - A sub tree of a tree that is left high has become right high after inserting an element.

Diagram



4. **Left of Right** - A sub tree of a tree that is right high has become left high after inserting an element.

Diagram



Rotation

An AVL tree becomes unbalanced when there is an insertion or deletion of a node from an existing AVL tree.

To balance the tree, transformation of the tree is performed. This transformation is called Rotation.

Rotation is a technique in which an unbalanced AVL tree is rotated either left or right side, so that the tree becomes balanced again. i.e

$$|H_L - H_R| \leq 1$$

There are two types of Rotation. They are:

- Single Rotation
- Double Rotation

Single Rotation: Rotation of the tree is performed only once either left or right side of the unbalanced tree. Single Rotation is performed for the two cases namely left of left and right of right

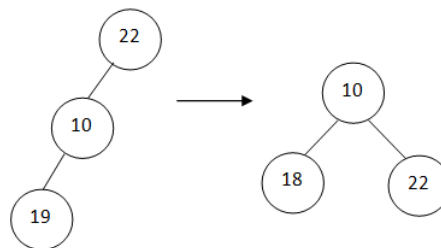
Double Rotation: Rotation of the tree is performed twice, one on the left side and the other one on the right side and vice versa of the unbalanced tree. Double Rotation is performed for the two cases namely left of right and right of left.

Single Rotation

Case 1: Left of Left: When the out-of-balance condition has been created by a left high subtree of a left high tree, the tree is balanced by rotating the out-of-balance node to the right.

Diagram

Simple Right Rotation



Case 2: Right of Right: When the out-of-balance condition has been created by a right high subtree of a right high tree, the tree is balanced by rotating the out-of-balance node to the left.

Double Rotation

Case 3: Right of Left: Double rotation is needed for balancing the AVL tree in this case. The tree is out of balance in which the root is left high and the left subtree is right high.

- The left subtree which is right high is rotated first in the left side.
- The root of the subtree which is left high is rotated second in the right side.

Basic Operations performed in an AVL Tree:

The basic operations performed in an AVL tree are as follows:

- Inserting an element.
- Deleting an element

Whenever these two operations are done, the balance condition has to be checked after the operation. If the balance condition is not satisfied then rotation is performed. Single or double rotation is performed depending upon the violation in the balance factor.

AVL Trees: Implementation

A VL tree is implemented using linked list concept. The node declaration for A VL trees is given below:

```
struct AvlNode;
typedef struct AvlNode *Position;
typedef struct AvlNode * AvlTree;
AvlTree MakeEmpty( AvlTree T);
Position Find(ElementType X, AvlTree T);
Position FindMin( AvlTree T );
Position FindMax( AvlTree T);
AvlTree Insert( ElementType X, AvlTree T);
AvlTree Delete( ElementType X, AvlTree T);
ElementType Retrieve( Position P );
struct AvlNode
{
    ElementType Element; AvlTree Left; AvlTree Right;
    int Height;
};
```

A structure AvlNode is created which contains four data elements. They are:

- The data to be stored in the node.
- Address of the left child
- Address of the right child
- Height of the node

Procedure to Compute the Height of a Node


```
static int Height( Position P )
{
    if( P == NULL)
        return -1 ; else
        return P->Height;
}
```

Explanation

- This function has a parameter P, which contains the address of the node.
- If P is NULL, it implies that there is no node, hence the height is returned as -1.
- Otherwise it returns the height of the node stored in the structure.

Procedure to Insert an Element Into the Tree

```
AvlTree Insert( ElementType X, AvlTree T)
{
    if ( T = NULL)
    {
        /* Create and return a one-node tree */
        T = malloc( sizeof( struct AvlNode ) ); if( T == NULL)
            FatalError( "Out ofspace!!!" );
        else
        { T->Element = X; T->Height= 0;
          T->Left = T->Right = NULL;
        }
    }
    else
    if ( X < T->Element )
        { T->Left = Insert( X, T->Left)
          if ( Height( T->Left ) - Height( T->Right) = 2 )
              if ( X < T->Left->Element )
                  T = SingleRotateWithLeft( T)
              else
                  T = DoubleRotate WithLeft( T );
```

```

}
else
if (X> T->Element)
    { T->Right = Insert( X, T->Right);
    if( Height( T->Right ) - Height( T->Left ) == 2 )
        if( X > T->Right->Element)
            T = SingleRotateWithRight( T);
        else
            T = DoubleRotatcWithRight( T);
    }
/* Else X is in the tree already; we'll do nothing */
T->Height = Max( Height( T->Left), Height{ T>Right» + 1;
return T;
}

```

Explanation

1. This function has two parameters namely, the clement X to be inserted and T, the address of the root of the tree.
2. If T is NULL, a new node is create p and the address is stored in T, else go to 5.
3. If T is NULL, it implies that memory allocation is not done and terminates.
4. Otherwise the new element X is inserted into the node T. The left and right pointer are made NULL. The height of the node is initialized to 0. Go to step 8.
5. The element X is compared with the element present in T. If $X < T \rightarrow \text{element}$, then the element is to be inserted in the left. A recursive function call is made with two parameters X and T->left.
6. Otherwise if $X > T \rightarrow \text{element}$, then the element is to be inserted in the right. A recursive function call is made with two parameters X and T ->right.
7. Otherwise, itimplies that X is already present in the A VL. Hence the element cannot be inserted.
8. After the element is inserted in the appropriate position, the balance factor is checked. If the Balance factor is equal to 2, then it implies that the tree is not balanced. Hence, rotation is to be performed.

9. The element is compared with T->left->element or T->right->element. If X is less, then single rotation is performed. Otherwise, double rotation is performed.
10. After the rotation is performed, the height of each node is calculated again and the address of the new tree is returned back to the main function.

Procedure for SingleRotation with Left

```
static Position SingleRotateWithLeft( Position K2 )
{
    Position KI;
    KI = K2->Left;
    K2->Left = KI->Right;
    KI->Right = K2;
    K2->Height = Max( Height(K2->Left), Height(K2->Right) )+1;
    KI->Height = Max( Height(KI->Left), K2->Height) + 1;
    return KI; /* New root */
}
```

Explanation

1. This function takes a parameter K2, which is the address of the node where rotation is to be done.
2. The rotation takes place on the right side.
3. K1 becomes the root node and K2 becomes the right child of K1.
4. The height of K1 and K2 are calculated and the address of K1 is returned to the calling function.

Procedure for Single Rotation with Right:

```
static Position SingleRotateWithRight( Position K2 )
{
    Position K1;
    KI = K2->right;
    K2->right = KI->left;
    KI->left = K2;
    K2->Height = Max( Height(K2->Left), Height(K2->Right) ) + 1;
    KI->Height = Max( Height(KI->Left), K2->Height) + 1;
    return KI; /* New root */
}
```

Explanation

1. This function takes a parameter K2, which is the address of the node where rotation is to be done.
2. The rotation takes place on the left side.
3. K1 becomes the root node and K2 becomes the left child of K1.
4. The height of K1 and K2 are calculated and the address of K1 is returned to the calling function.

Procedure for Double Rotation with Left

```
static Position DoubleRotateWithLeft( Position K3 )
{
    /* Rotate between K1 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left );
    /* Rotate between K3 and K2 */
    return SingleRotateWithLeft( K3 );
}
```

Explanation

1. This function takes a parameter K3, which is the address of the node
2. The first rotation takes place on the left side with K3->Left (K1) as parameter.
3. The rotation takes place between K1 and K2.
4. K2 becomes the root node and K1 becomes the left child of K2.
5. The height of K1 and K2 are calculated and the address of K2 is returned to the calling function.
6. The address of K2 is assigned to K3->left.
7. The second rotation takes between K3 and K2. The rotation takes place on the right side.
8. K2 becomes the root node and K3 becomes the right child of K2.
9. The height of K3 and K2 are calculated and the address of K2 is returned to the calling function.

Procedure for Double Rotation with Right

```
static Position DoubleRotateWithRight( Position K3 )
{
    /* Rotate between K1 and K2 */
    K3->Right = SingleRotateWithLeft( K3->Right );
```

```

/* Rotate between K3 and K2 >I< /
return SingleRotatcWithRight( K3 );
}

```

Explanation

1. This function takes a parameter K3, which is the address of the node.
2. The first rotation takes place on the right side with K3->right (K1) as the parameter.
3. The rotation takes place between K1 and K2.
4. K2 becomes the root node and K1 becomes the right child of K2.
5. The height of K1 and K2 are calculated and the address of K2 is returned to the calling function.
6. The address of K2 is assigned to K3->right.
7. The second rotation takes between K3 and K2. The rotation takes place on the left side.
8. K2 becomes the root node and K3 becomes the left child of K2.

The height of K3 and K2 are calculated and the address of K2 is returned to the calling function.

B-Tree

B-Tree is a self-balancing search tree. B Trees are multi-way trees. That is each node contains a set of keys and pointers. A B Tree with four keys and five pointers represents the minimum size of a B Tree node. B Trees are dynamic. That is, the height of the tree grows and contracts as records are added and deleted.

In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree

All leaves are at same level.

A B-Tree is defined by the term minimum degree 't'. The value of t depends upon disk block size.

Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

All nodes (including root) may contain at most $2t - 1$ keys.

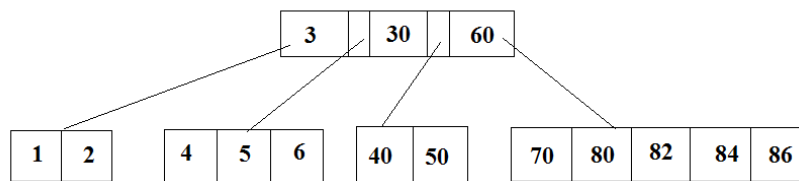
Number of children of a node is equal to the number of keys in it plus 1.

All keys of a node are sorted in increasing order. The child between two keys k_1 and k_2 contains all keys in range from k_1 and k_2 .

B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

Like other balanced Binary Search Trees, time complexity to search, insert and delete is $O(\log n)$.

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k . We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

1.2. B+ Trees

A B+ Tree combines features of B Trees. It contains index pages and data pages. The data pages always appear as leaf nodes in the tree. The root node and intermediate nodes are always

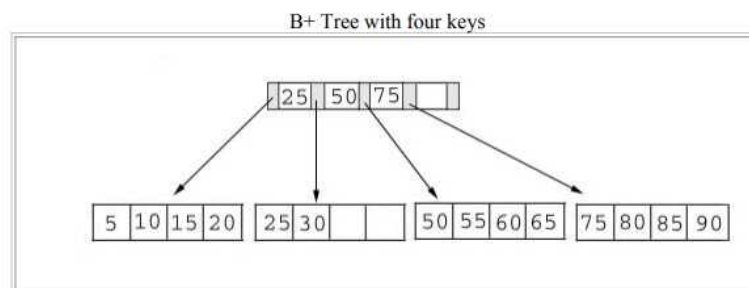
index pages. The index pages in a B+ tree are constructed through the process of inserting and deleting records.

Thus, B+ trees grow and contract like their B Tree counterparts. The contents and the number of index pages reflects this growth and shrinkage. B+ Trees and B Trees use a "fill factor" to control the growth and the shrinkage. A 50% fill factor would be the minimum for any B+ or B tree.

As our example we use the smallest page structure. This means that our B+ tree conforms to the following guidelines.

Number of Keys/page	4
Number of Pointers/page	5
Fill Factor	50%
Minimum Keys in each page	2

As this table indicates each page must have a minimum of two keys. The root page may violate this rule. The following table shows a B+ tree. As the example illustrates this tree does not have a full index page. (We have room for one more key and pointer in the root page.) In addition, one of the data pages contains empty slots.



Adding Records to a B+ Tree

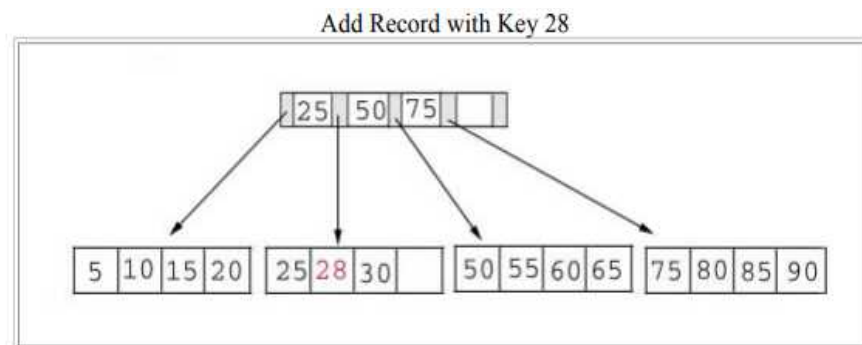
The key value determines a record's placement in a B+ tree. The leaf pages are maintained in sequential order AND a doubly linked list (not shown) connects each leaf page with its sibling page(s). This doubly linked list speeds data movement as the pages grow and contract. We must consider three scenarios when we add a record to a B+ tree. Each scenario causes a different action in the insert algorithm. The scenarios are:

The insert algorithm for B+ Trees

Leaf Page Full	Index Page FULL	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	<ol style="list-style-type: none"> 1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	<ol style="list-style-type: none"> 1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys \geq middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. <p>IF the next level index page is full, continue splitting the index pages.</p>

Illustrations of the Insert Algorithm

The following examples illustrate each of the insert scenarios. We begin with the simplest scenario: inserting a record into a leaf page that is not full. Since only the leaf node containing 25 and 30 contains expansion room, we're going to insert a record with a key value of 28 into the B+ tree. The following figures shows the result of this addition.

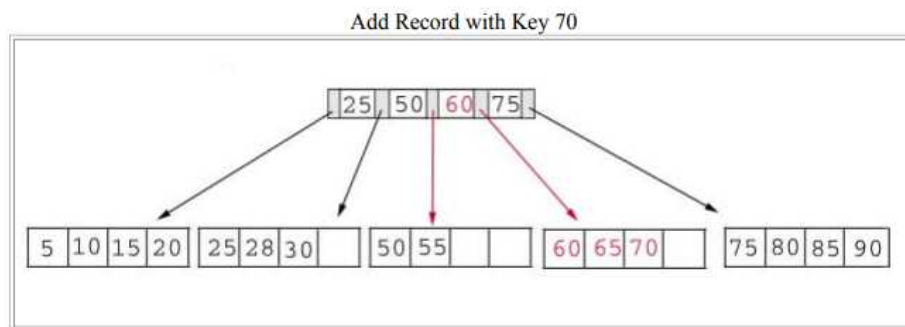


Adding a record when the leaf page is full but the index page is not.

Next, we're going to insert a record with a key value of 70 into our B+ tree. This record should go in the leaf page containing 50, 55, 60, and 65. Unfortunately this page is full. This means that we must split the page as follows:

Left Leaf Page	Right Leaf Page
50 55	60 65 70

The middle key of 60 is placed in the index page between 50 and 75. The following table shows the B+ tree after the addition of 70.



Adding a record when both the leaf page and the index page are full.

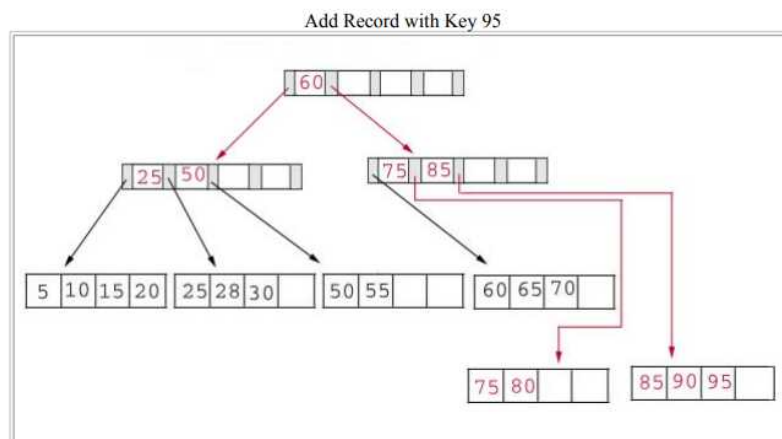
As our last example, we're going to add a record containing a key value of 95 to our B+ tree. This record belongs in the page containing 75, 80, 85, and 90. Since this page is full we split it into two pages:

Left Leaf Page	Right Leaf Page
75 80	85 90 95

The middle key, 85, rises to the index page. Unfortunately, the index page is also full, so we split the index page:

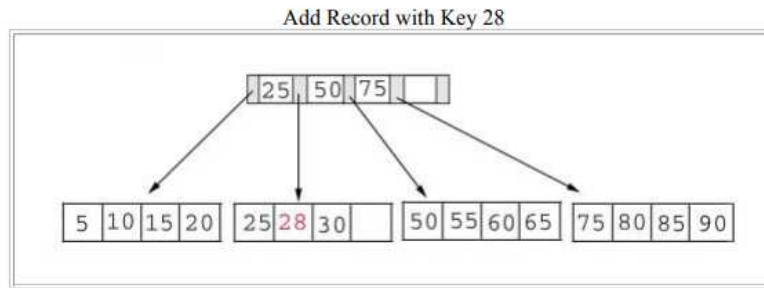
Left Index Page	Right Index Page	New Index Page
25 50	75 85	60

The following table illustrates the addition of the record containing 95 to the B+ tree.

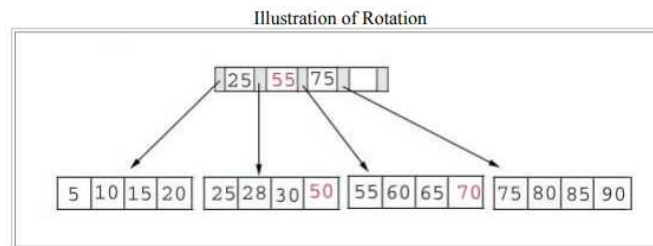


Rotation

B+ trees can incorporate rotation to reduce the number of page splits. A rotation occurs when a leaf page is full, but one of its sibling pages is not full. Rather than splitting the leaf page, we move a record to its sibling, adjusting the indices as necessary. Typically, the left sibling is checked first (if it exists) and then the right sibling. As an example, consider the B+ tree before the addition of the record containing a key of 70. As previously stated this record belongs in the leaf node containing 50 55 60 65. Notice that this node is full, but its left sibling is not.



Using rotation we shift the record with the lowest key to its sibling. Since this key appeared in the index page we also modify the index page. The new B+ tree appears in the following table.

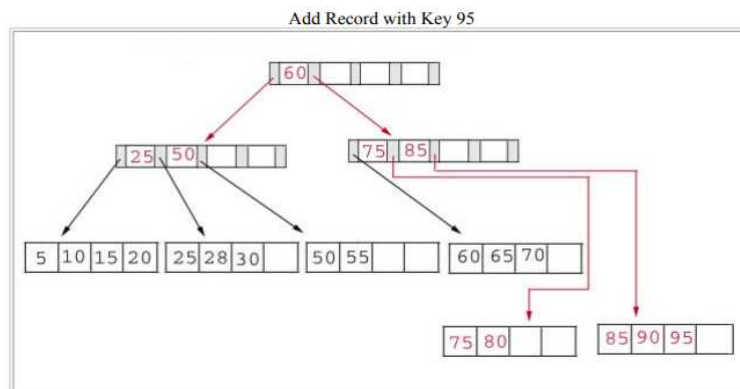


Deleting Keys from a B+ Tree

We must consider three scenarios when we delete a record from a B+ tree. Each scenario causes a different action in the delete algorithm. The scenarios are:

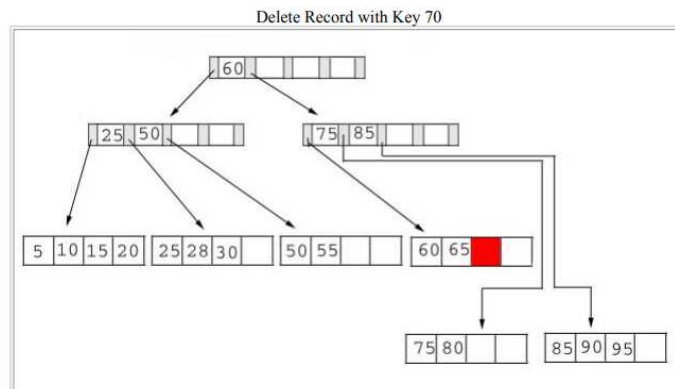
The delete algorithm for B+ Trees		
Leaf Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	<ol style="list-style-type: none"> 1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. <p>Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.</p>

As our example, we consider the B+ tree after we added 95 as a key. As a refresher this tree is printed in the following table.



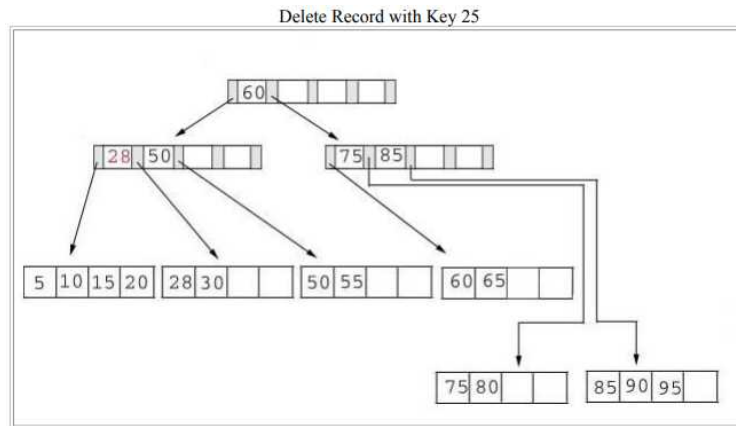
Delete 70 from the B+ Tree

We begin by deleting the record with key 70 from the B+ tree. This record is in a leaf page containing 60, 65 and 70. This page will contain 2 records after the deletion. Since our fill factor is 50% or (2 records) we simply delete 70 from the leaf node. The following table shows the B+ tree after the deletion.



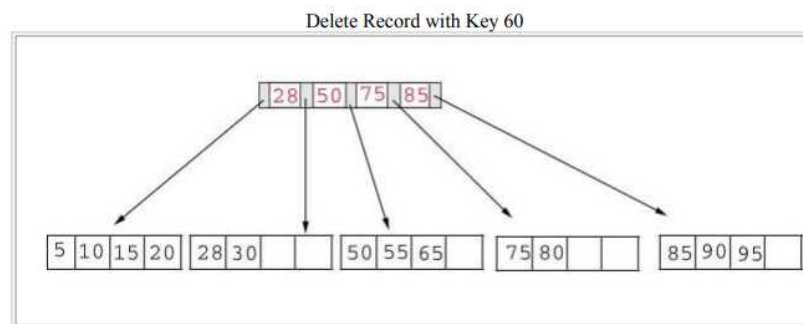
Delete 25 from the B+ Tree

Next, we delete the record containing 25 from the B+ tree. This record is found in the leaf node containing 25, 28, and 30. The fill factor will be 50% after the deletion; however, 25 appears in the index page. Thus, when we delete 25 we must replace it with 28 in the index page. The following table shows the B+ tree after this deletion.



Delete 60 from the B+ Tree

As our last example, we're going to delete 60 from the B+ tree. This deletion is interesting for several reasons: The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages. 1. With recombined pages, the index page will be reduced by one key. Hence, it will also fall below the fill factor. Thus, we must combine index pages. 2. 3. Sixty appears as the only key in the root index page. Obviously, it will be removed with the deletion. The following table shows the B+ tree after the deletion of 60. Notice that the tree contains a single index page.



Heap

Heap data structure is a specialized binary tree based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their value. A heap data structure, some time called as Binary Heap.

There are two types of heap data structures and they are as follows.

- Max Heap
- Min Heap

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in a order according to values based on Max heap or Min heap.

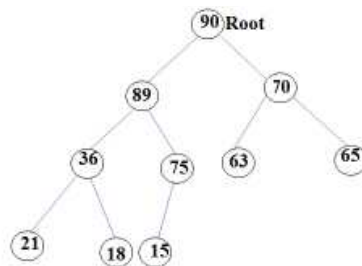
Property #2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

Max heap data structure is a specialized full binary tree data structure except last leaf node can be alone. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes. And last leaf node can be alone.

Example



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

- Finding Maximum
- Insertion
- Deletion

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In max heap, the root node has the maximum value than all other nodes in the max heap. So, directly we can display root node value as maximum value in max heap.

Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows.

Step 1: Insert the newNode as last leaf from left to right.

Step 2: Compare newNode value with its Parent node.

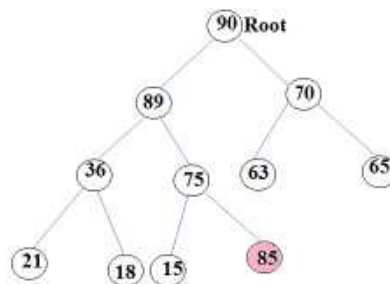
Step 3: If newNode value is greater than its parent, then swap both of them.

Step 4: Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reached to root.

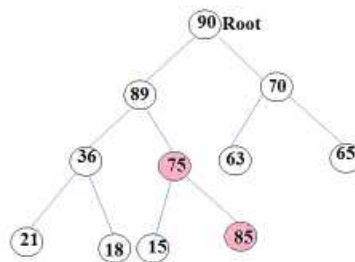
Example

Consider the above max heap. Insert a new node with value 85.

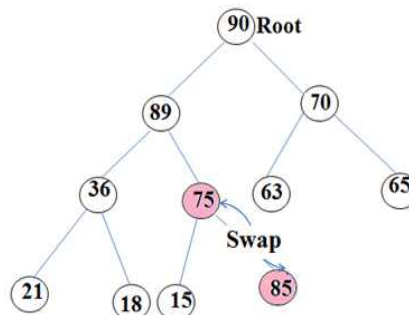
Step 1: Insert the newNode with value 85 as last leaf from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows:

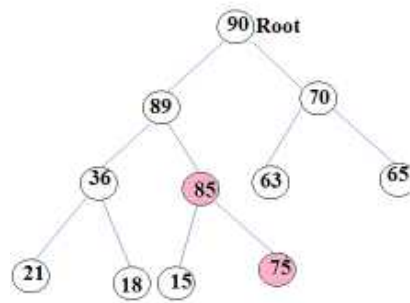


Step 2: Compare newNode value (85) with its Parent node value (75). That means $85 > 75$.

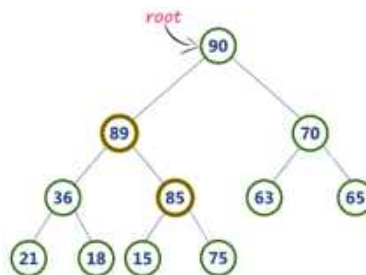


Step 3: Here new Node value (85) is greater than its parent value (75), then swap both of them. After swapping, max heap is as follows.

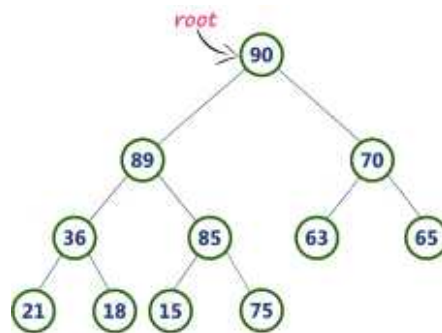




Step 4: Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows



Deletion Operation in Max Heap

In a max heap, deleting last node is very simple as it is not disturbing max heap properties.

Deleting root node from a max heap is quite difficult as it disturbs the max heap properties.

We use the following steps to delete root node from a max heap...

Step 1: Swap the root node with last node in max heap

Step 2: Delete last node.

Step 3: Now, compare root value with its left child value.

Step 4: If root value is smaller than its left child, then compare left child with its right sibling.

Else goto Step 6

Step 5: If left child value is larger than its right sibling, then swap root with left child. otherwise swap root with its right child.

Step 6: If root value is larger than its left child, then compare root value with its right child value.

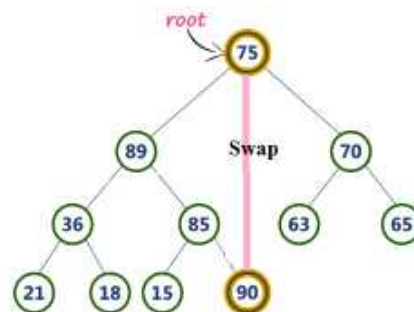
Step 7: If root value is smaller than its right child, then swap root with rith child. otherwise stop the process.

Step 8: Repeat the same until root node is fixed at its exact position.

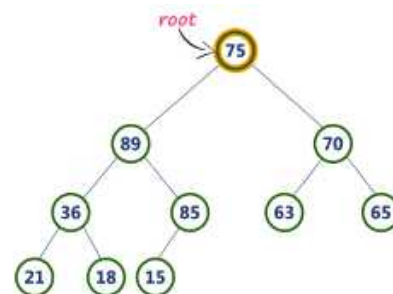
Example

Consider the above max heap. Delete root node (90) from the max heap.

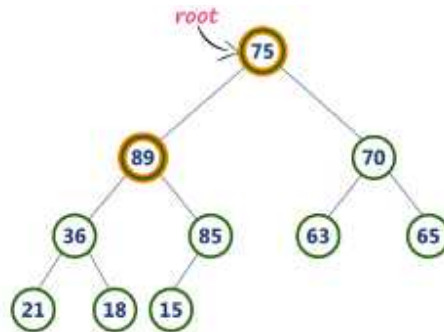
Step 1: Swap the root node (90) with last node 75 in max heap After swapping max heap is as follows...



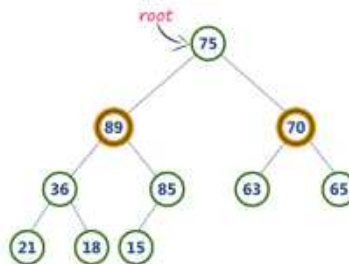
Step 2: Delete last node. Here node with value 90. After deleting node with value 90 from heap, max heap is as follows.



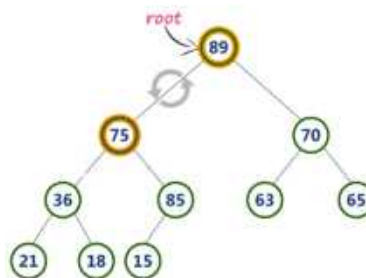
Step 3: Compare root node (75) with its left child (89).



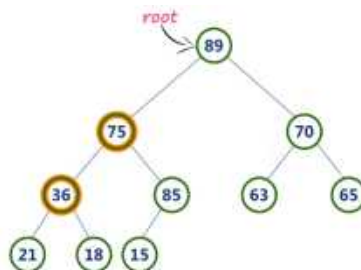
Here, root value (75) is smaller than its left child value (89). So, compare left child (89) with its right sibling (70).



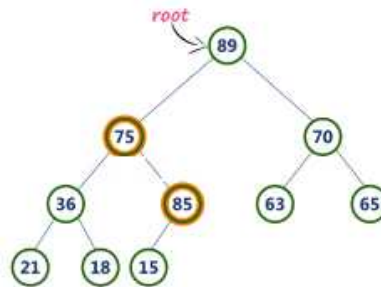
Step 4: Here, left child value (89) is larger than its right sibling (70), So, swap root (75) with left child (89).



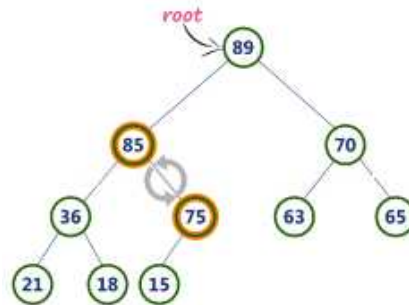
Step 5: Now, again compare 75 with its left child (36).



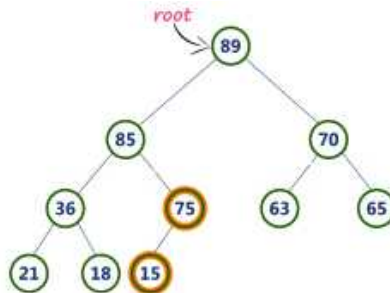
Here, node with value 75 is larger than its left child. So, we compare node with value 75 is compared with its right child 85.



Step 6: Here, node with value 75 is smaller than its right child (85). So, we swap both of them. After swapping max heap is as follows.

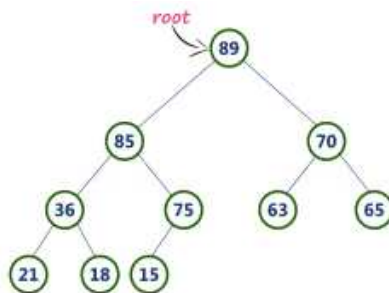


Step 7: Now, compare node with value 75 with its left child (15).



Here, node with value 75 is larger than its left child (15) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (90) is as follows...



UNIT IV

Non Linear Data Structures – Graphs

Definition – Representation of Graph – Types of graph - Breadth-first traversal - Depth-first traversal – Topological Sort – Bi-connectivity – Cut vertex – Euler circuits – Applications of graphs.

Definition

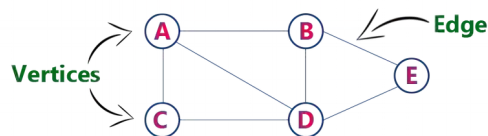
- Graph is a nonlinear data structure.
- It contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.
- Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as $G = (V, E)$

Where $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$.



Graph Terminology

We use the following terms in graph data structure...

Vertex

An individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A, B). In above example graph, there are 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E)).

Edges are three types:

Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).

Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

Weighted Edge - A weighted edge is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel Edges or Multiple Edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-Loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

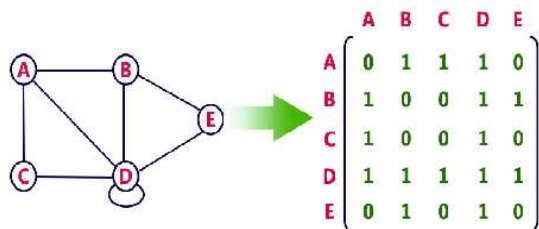
Graph data structure is represented using following representations...

- Adjacency Matrix
- Adjacency List

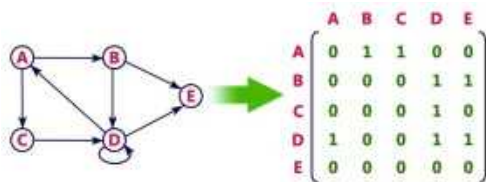
Adjacency Matrix

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

For example, consider the following undirected graph representation...

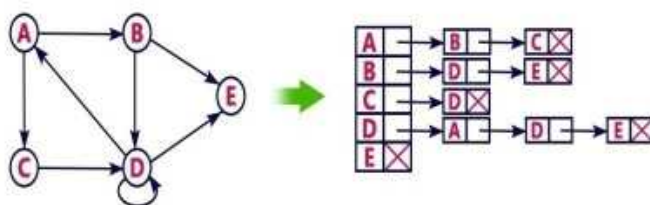


Directed graph representation...



Adjacency List

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



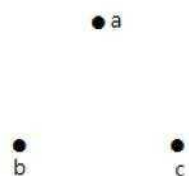
Types of Graphs

There are various types of graphs depending upon the number of vertices, number of edges, interconnectivity, and their overall structure. We will discuss only a certain few important types of graphs in this chapter.

Null Graph

A **graph having no edges** is called a Null Graph.

Example



In the above graph, there are three vertices named 'a', 'b', and 'c', but there are no edges among them. Hence it is a Null Graph.

Trivial Graph

A **graph with only one vertex** is called a Trivial Graph.

Example

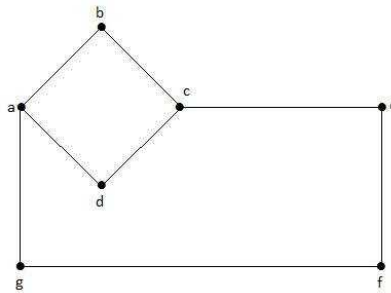


In the above shown graph, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.

Non-Directed Graph

A non-directed graph contains edges but the edges are not directed ones.

Example

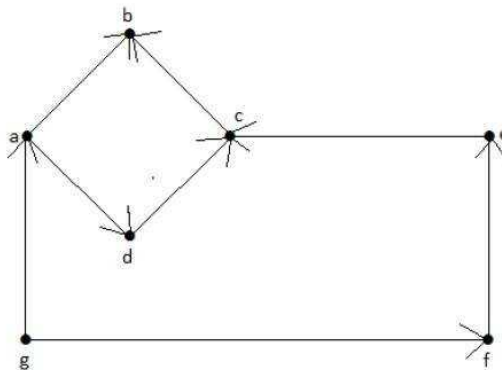


In this graph, 'a', 'b', 'c', 'd', 'e', 'f', 'g' are the vertices, and 'ab', 'bc', 'cd', 'da', 'ag', 'gf', 'ef' are the edges of the graph. Since it is a non-directed graph, the edges 'ab' and 'ba' are same. Similarly other edges also considered in the same way.

Directed Graph

In a directed graph, each edge has a direction.

Example



In the above graph, we have seven vertices 'a', 'b', 'c', 'd', 'e', 'f', and 'g', and eight edges 'ab', 'cb', 'dc', 'ad', 'ec', 'fe', 'gf', and 'ga'. As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, 'ab' is different from 'ba'.

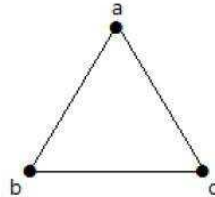
Simple Graph

A graph with **no loops** and **no parallel edges** is called a simple graph.

- The maximum number of edges possible in a single graph with 'n' vertices is nC_2 where ${}^nC_2 = n(n-1)/2$.
- The number of simple graphs possible with 'n' vertices $= 2^{nC_2} = 2^{n(n-1)/2}$.

Example

In the following graph, there are 3 vertices with 3 edges which is maximum excluding the parallel edges and loops. This can be proved by using the above formulae.



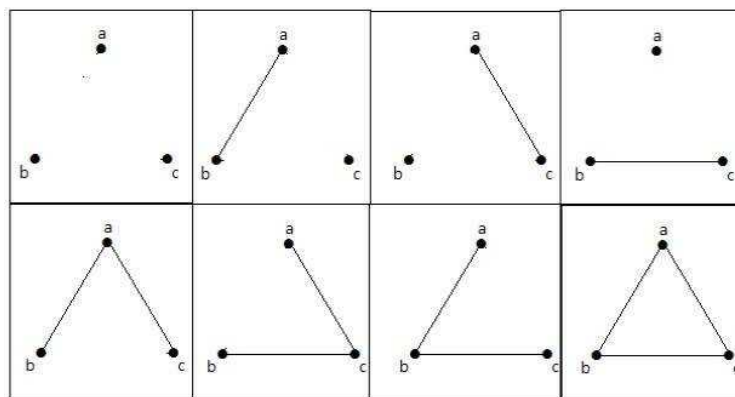
The maximum number of edges with $n=3$ vertices:

$$\begin{aligned} {}^nC_2 &= n(n-1)/2 \\ &= 3(3-1)/2 \\ &= 6/2 \\ &= 3 \text{ edges} \end{aligned}$$

The maximum number of simple graphs with $n=3$ vertices:

$$\begin{aligned} 2^{nC_2} &= 2^{n(n-1)/2} \\ &= 2^{3(3-1)/2} \\ &= 2^3 \\ &= 8 \end{aligned}$$

These 8 graphs are as shown below:

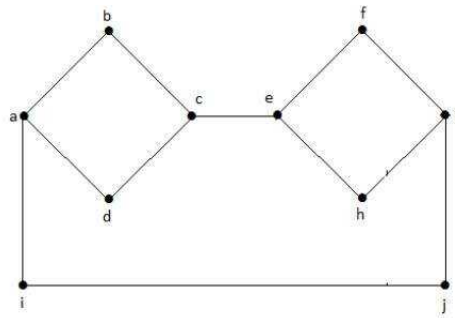


Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Example

In the following graph, each vertex has its own edge connected to other edge. Hence it is a connected graph.

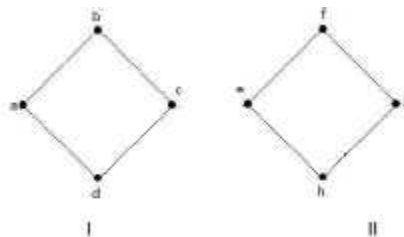


Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

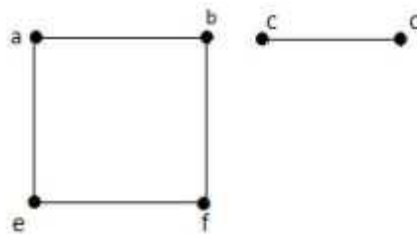
Example 1

The following graph is an example of a Disconnected Graph, where there are two components, one with 'a', 'b', 'c', 'd' vertices and another with 'e', 'f', 'g', 'h' vertices.



The two components are independent and not connected to each other. Hence it is called disconnected graph.

Example 2



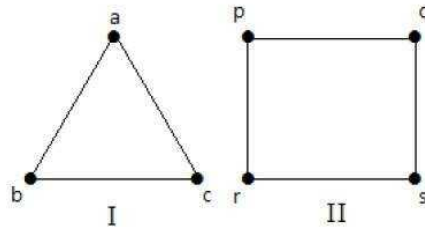
In this example, there are two independent components, a-b-f-e and c-d, which are not connected to each other. Hence this is a disconnected graph.

Regular Graph

A graph G is said to be regular, **if all its vertices have the same degree**. In a graph, if the degree of each vertex is ' k ', then the graph is called a ' k -regular graph'.

Example

In the following graphs, all the vertices have the same degree. So these graphs are called regular graphs.



In both the graphs, all the vertices have degree 2. They are called 2-Regular Graphs.

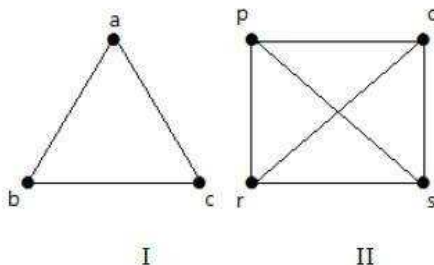
Complete Graph

A simple graph with ' n ' mutual vertices is called a complete graph and it is **denoted by ' K_n '**. In the graph, **a vertex should have edges with all other vertices**, then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

Example

In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.



In graph I,

	a	b	c
a	Not Connected	Connected	Connected
b	Connected	Not Connected	Connected
c	Connected	Connected	Not Connected

In graph II,

	p	q	r	s
p	Not Connected	Connected	Connected	Connected
q	Connected	Not Connected	Connected	Connected
r	Connected	Connected	Not Connected	Connected
s	Connected	Connected	Connected	Not Connected

Cycle Graph

A simple graph with 'n' vertices ($n \geq 3$) and 'n' edges is called a cycle graph if all its edges form a cycle of length 'n'.

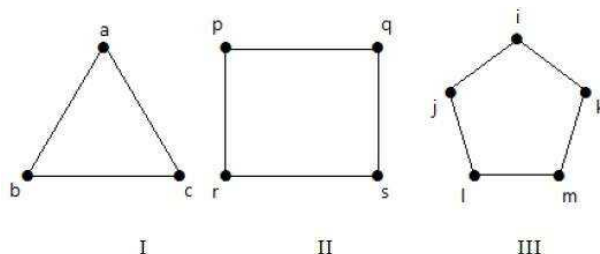
If the **degree of each vertex in the graph is two**, then it is called a Cycle Graph.

Notation – C_n

Example

Take a look at the following graphs:

- Graph I has 3 vertices with 3 edges which is forming a cycle 'ab-bc-ca'.
- Graph II has 4 vertices with 4 edges which is forming a cycle 'pq-qs-sr-rp'.
- Graph III has 5 vertices with 5 edges which is forming a cycle 'ik-km-ml-lj-ji'.



Hence all the given graphs are cycle graphs.

Wheel Graph

A wheel graph is obtained from a cycle graph C_{n-1} by adding a new vertex. That new vertex is called a **Hub** which is connected to all the vertices of C_n .

Notation – W_n

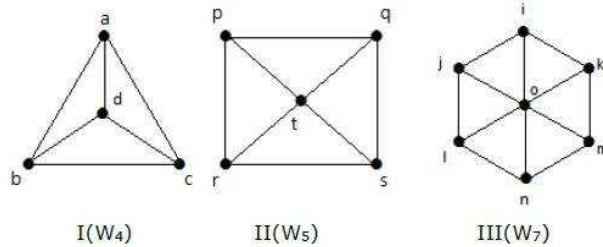
No. of edges in W_n = No. of edges from hub to all other vertices + No. of edges from all other nodes in cycle graph without a hub.

$$= (n-1) + (n-1)$$

$$= 2(n-1)$$

Example

Take a look at the following graphs. They are all wheel graphs.



In graph I, it is obtained from C_3 by adding an vertex at the middle named as 'd'. It is denoted as W_4 .

$$\text{Number of edges in } W_4 = 2(n-1) = 2(3) = 6$$

In graph II, it is obtained from C_4 by adding a vertex at the middle named as 't'. It is denoted as W_5 .

$$\text{Number of edges in } W_5 = 2(n-1) = 2(4) = 8$$

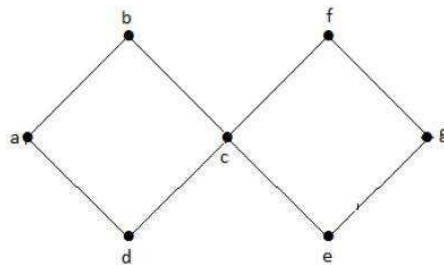
In graph III, it is obtained from C_6 by adding a vertex at the middle named as 'o'. It is denoted as W_7 .

$$\text{Number of edges in } W_4 = 2(n-1) = 2(6) = 12$$

Cyclic Graph

A graph **with at least one** cycle is called a cyclic graph.

Example

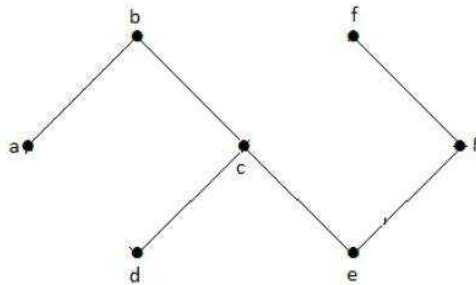


In the above example graph, we have two cycles a-b-c-d-a and c-f-g-e-c. Hence it is called a cyclic graph.

Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

Example



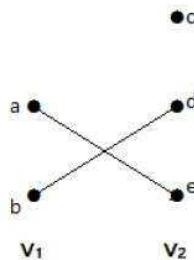
In the above example graph, we do not have any cycles. Hence it is a non-cyclic graph.

Bipartite Graph

A simple graph $G = (V, E)$ with vertex partition $V = \{V_1, V_2\}$ is called a bipartite graph if every edge of E joins a vertex in V_1 to a vertex in V_2 .

In general, a Bipartite graph has two sets of vertices, let us say, V_1 and V_2 , and if an edge is drawn, it should connect any vertex in set V_1 to any vertex in set V_2 .

Example



In this graph, you can observe two sets of vertices – V_1 and V_2 . Here, two edges named ‘ae’ and ‘bd’ are connecting the vertices of two sets V_1 and V_2 .

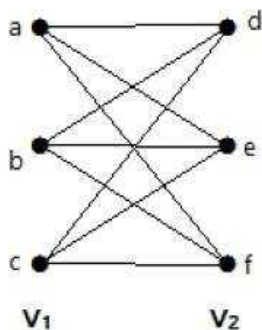
Complete Bipartite Graph

A bipartite graph ‘G’, $G = (V, E)$ with partition $V = \{V_1, V_2\}$ is said to be a complete bipartite graph if every vertex in V_1 is connected to every vertex of V_2 .

In general, a complete bipartite graph connects each vertex from set V_1 to each vertex from set V_2 .

Example

The following graph is a complete bipartite graph because it has edges connecting each vertex from set V_1 to each vertex from set V_2 .



If $|V_1| = m$ and $|V_2| = n$, then the complete bipartite graph is denoted by $K_{m, n}$.

- $K_{m,n}$ has $(m+n)$ vertices and (mn) edges.
- $K_{m,n}$ is a regular graph if $m=n$.

In general, **a complete bipartite graph is not a complete graph.**

$K_{m,n}$ is a complete graph if $m=n=1$.

The maximum number of edges in a bipartite graph with n vertices is

$$\left[\frac{n^2}{4} \right]$$

If $n=10, k=5, 5 = \left[\frac{10^2}{4} \right] = \left[\frac{100}{4} \right] = 25$

Similarly $K_{6, 4} = 24$

$K_{7, 3} = 21$

$K_{8, 2} = 16$

$K_{9, 1} = 9$

If $n=9, k=5, 4 = \left[\frac{9^2}{4} \right] = \left[\frac{81}{4} \right] = 20$

Similarly $K_{6, 3} = 18$

$K_{7, 2} = 14$

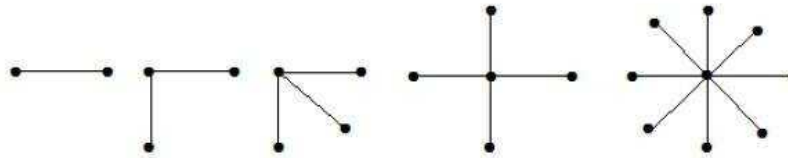
$K_{8, 1} = 8$

‘G’ is a bipartite graph if ‘G’ has no cycles of odd length. A special case of bipartite graph is a **star graph**.

Star Graph

A complete bipartite graph of the form $K_{1, n-1}$ is a star graph with n -vertices. A star graph is a complete bipartite graph if a single vertex belongs to one set and all the remaining vertices belong to the other set.

Example



In the above graphs, out of 'n' vertices, all the 'n-1' vertices are connected to a single vertex. Hence it is in the form of $K_{1, n-1}$ which are star graphs.

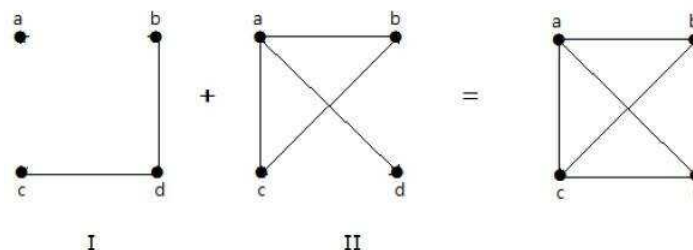
Complement of a Graph

Let ' G -' be a simple graph with some vertices as that of ' G ' and an edge $\{U, V\}$ is present in ' G -' if the edge is not present in G . It means, two vertices are adjacent in ' G -' if the two vertices are not adjacent in G .

If the edges that exist in graph I are absent in another graph II, and if both graph I and graph II are combined together to form a complete graph, then graph I and graph II are called complements of each other.

Example

In the following example, graph-I has two edges 'cd' and 'bd'. Its complement graph-II has four edges.



Note that the edges in graph-I are not present in graph-II and vice versa. Hence, the combination of both the graphs gives a complete graph of 'n' vertices.

Note – A combination of two complementary graphs gives a complete graph.

If ' G ' is any simple graph, then

$$|E(G)| + |E(G-)| = |E(K_n)|, \text{ where } n = \text{number of vertices in the graph.}$$

Example

Let ' G ' be a simple graph with nine vertices and twelve edges, find the number of edges in ' G -'.

$$\text{You have, } |E(G)| + |E(G-)| = |E(K_n)|$$

$$12 + |E(G-)| =$$

$$9(9-1) / 2 = {}^9C_2$$

$$12 + |E(G-)| = 36$$

$$|E(G-)| = 24$$

'G' is a simple graph with 40 edges and its complement 'G-' has 38 edges. Find the number of vertices in the graph G or 'G-'.

Let the number of vertices in the graph be 'n'.

We have, $|E(G)| + |E(G-)| = |E(K_n)|$

$$40 + 38 = n(n-1)/2$$

$$156 = n(n-1)$$

$$13(12) = n(n-1)$$

$$n = 13$$

Graph Traversals

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process.

A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

Depth First Search

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once.

If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

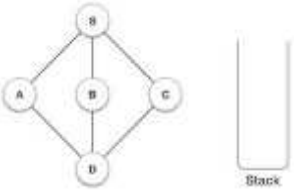
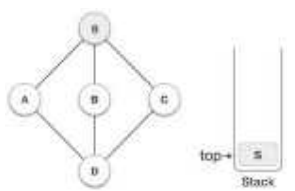
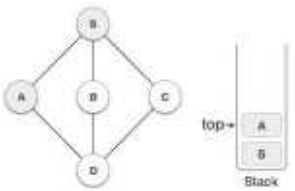
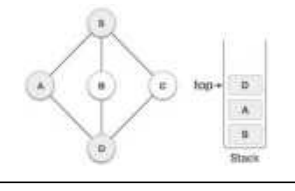
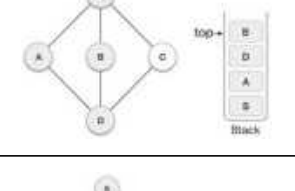
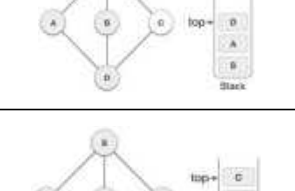
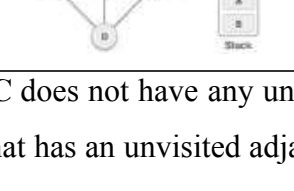
Pseudocode

```

DFS-iterative (G, s):                                //Where G is graph and s is source vertex
    let S be stack
    S.push( s )                                       //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
                                                //Pop a vertex from stack to visit next
        v = S.top( )
        S.pop( )
                                                //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                S.push( w )
                mark w as visited
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
    
```

Example

Step	Traversal	Description
------	-----------	-------------

1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose B , mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7		Only unvisited adjacent node is from D is C now. So we visit C , mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Implementation in C

```
#include <stdio.h>
```

```
#include <stdlib.h>

/*      ADJACENCY MATRIX      */

int source,V,E,time,visited[20],G[20][20];

void DFS(int i)
{
    int j;
    visited[i]=1;
    printf(" %d->",i+1);
    for(j=0;j<V;j++)
    {
        if(G[i][j]==1&&visited[j]==0)
            DFS(j);
    }
}

int main()
{
    int i,j,v1,v2;
    printf("\t\t\tGraphs\n");
    printf("Enter the no of edges:");
    scanf("%d",&E);
    printf("Enter the no of vertices:");
    scanf("%d",&V);
    for(i=0;i<V;i++)
    {
        for(j=0;j<V;j++)
            G[i][j]=0;
    }
    /*      creating edges :P      */
    for(i=0;i<E;i++)
    {
        printf("Enter the edges (format: V1 V2) : ");
```

```

scanf("%d%d",&v1,&v2);
G[v1-1][v2-1]=1;
}

for(i=0;i<V;i++)
{
    for(j=0;j<V;j++)
        printf(" %d ",G[i][j]);
    printf("\n");
}

printf("Enter the source: ");
scanf("%d",&source);
DFS(source-1);
return 0;
}

```

```

E:\2018-2019\Winston Raja\DS Lab\graph.exe
Graphs
Enter the no of edges:11
Enter the no of vertices:10
Enter the edges <format: U1 U2> : 1 2
Enter the edges <format: U1 U2> : 1 3
Enter the edges <format: U1 U2> : 2 4
Enter the edges <format: U1 U2> : 2 5
Enter the edges <format: U1 U2> : 3 6
Enter the edges <format: U1 U2> : 3 7
Enter the edges <format: U1 U2> : 4 8
Enter the edges <format: U1 U2> : 5 9
Enter the edges <format: U1 U2> : 6 10
Enter the edges <format: U1 U2> : 8 9
Enter the edges <format: U1 U2> : 9 10
0 1 1 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
Enter the source: 1
1-> 2-> 4-> 8-> 9-> 10-> 5-> 3-> 6-> 7->Press any key to continue . . .

```

Breadth First Search

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where we start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node).

We must then move towards the next-level neighbour nodes.

As the name BFS suggests,

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked.

The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Pseudocode

```

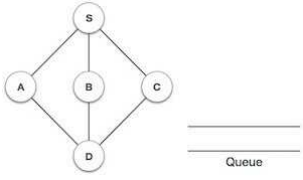
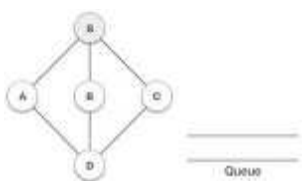
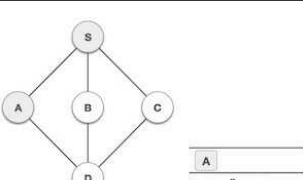
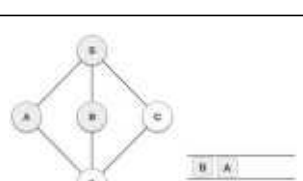
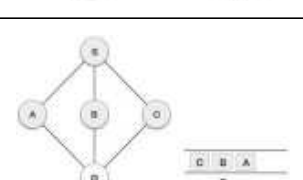
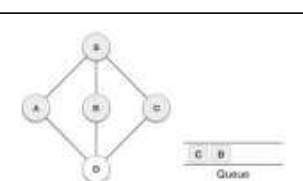
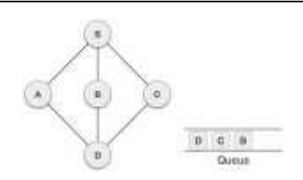
BFS (G, s)           //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s )    //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited
        now
        v = Q.dequeue( )

        //processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w )    //Stores w in Q to further visit its neighbour
                mark w as visited.
    
```

Example

Step	Traversal	Description
------	-----------	-------------

1		Initialize the queue.
2		We start from visiting S(starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Implementation in C

```
#include<stdio.h>
```

```
int G[20][20],q[20],visited[20],n,front = 1, rear = 0 ;
```

```
void bfs(int v)
{
    int i;
    visited[v] = 1;
    for(i=1;i<=n;i++)
        if(G[v][i] && !visited[i])
            q[++rear]=i;
    if(front <= rear)
        bfs(q[front++]);
}

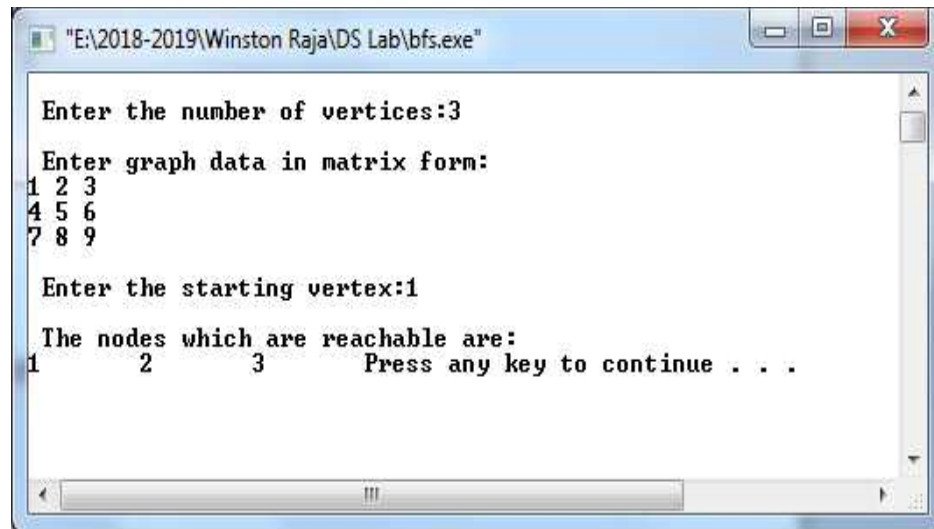
int main()
{
    int v,i,j;
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        q[i]=0;
        visited[i]=0;
    }
    printf("\n Enter graph data in matrix form:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&G[i][j]);
    printf("\n Enter the starting vertex:");
    scanf("%d",&v);
    bfs(v);
    printf("\n The nodes which are reachable are:\n");
    for(i=1;i<=n;i++)
        if(visited[i])
```

```

        printf("%d\t",i);
    else
        printf("\n %d is not reachable",i);

return 0;
}

```

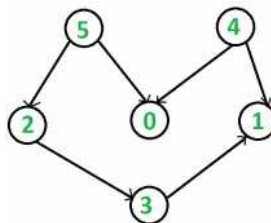


Topological Sorting

Definition

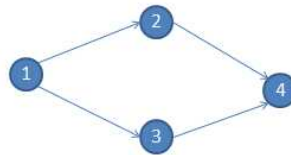
Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



Topological Sort Example

Consider the following graph “D”. Find the topological ordering of this graph “G”.



Step 1: Write in-degree of all vertices:

Vertex	in-degree
1	0
2	1
3	1
4	2

Step 2: Write the vertex which has in-degree 0 (zero) in solution. Here vertex 1 has in-degree 0.

So, solution is: 1 -> (not yet completed)

Step 3: Decrease in-degree count of vertices who are adjacent to the vertex which recently added to the solution. Here vertex 1 is recently added to the solution. Vertices 2 and 3 are adjacent to vertex 1. So decrease the in-degree count of those and update.

Updated result is:

Vertex	in-degree
1	Already added to solution
2	0
3	0
4	2

Step 4: Again repeat the same thing which we have done in step1 that is, write the vertices which have in-degree 0 in solution. Here we can observe that two vertices (2 and 3) have in-degree 0 (zero). Add any one vertex into the solution.

Note that, you may add vertex 2 into solution, and I may add vertex 3 to solution. That means the solution to topological sorting is not unique. Now add vertex 3.

Solution is: 1->3->

Step 5: Again decrease the in-degree count of vertices which are adjacent to vertex 3.

Updated result is:

Vertex	in-degree
1	Already added to solution
2	0
3	Already added to solution
4	1

Now add vertex 2 to solution because it only has in-degree 0.

Solution is: 1->3->2->

Updated result is:

Vertex	in-degree
1	Already added to solution
2	Already added to solution
3	Already added to solution
4	0

Finally add 4 to solution.

Final solution is: 1->3->2->4

Program for Topological Sort in C

```
#include <stdio.h>
```

```
int main(){
```

```
    int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;
```

```
    printf("Enter the no of vertices:\n");
```

```
    scanf("%d",&n);
```

```
    printf("Enter the adjacency matrix:\n");
```

```
    for(i=0;i<n;i++){
```

```
        printf("Enter row %d\n",i+1);
```

```
        for(j=0;j<n;j++){
```

```
            scanf("%d",&a[i][j]);
```

```
        }
```

```
    for(i=0;i<n;i++){
```

```

    indeg[i]=0;
    flag[i]=0;
}

for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        indeg[i]=indeg[i]+a[j][i];

printf("\nThe topological order is:");

while(count<n){
    for(k=0;k<n;k++){
        if((indeg[k]==0) && (flag[k]==0)){
            printf("%d ",(k+1));
            flag [k]=1;
        }

        for(i=0;i<n;i++){
            if(a[i][k]==1)
                indeg[k]--;
        }
    }

    count++;
}

return 0;
}

```

Output

Enter the no of vertices:

4

Enter the adjacency matrix:

Enter row 1

0 1 1 0

Enter row 2

0 0 0 1

Enter row 3

0 0 0 1

Enter row 4

0 0 0 0

The topological order is: 1 2 3 4

Bi-Connectivity

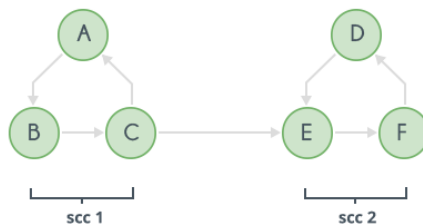
Connectivity

Connectivity in an undirected graph means that every vertex can reach every other vertex via any path. If the graph is not connected the graph can be broken down into **Connected**

Components.

Strong Connectivity

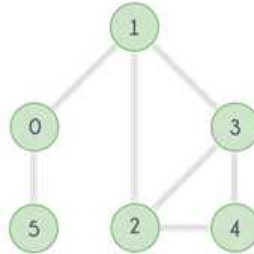
Strong Connectivity applies only to directed graphs. A directed graph is strongly connected if there is a **directed path** from any vertex to every other vertex. This is same as connectivity in an undirected graph, the only difference being strong connectivity applies to directed graphs and there should be directed paths instead of just paths. Similar to connected components, a directed graph can be broken down into **Strongly Connected Components**.



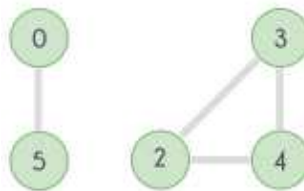
Articulation Point or Cut Vertex

A cut-vertex is a single vertex whose removal disconnects a graph.

In a graph, a vertex is called an **articulation point** if removing it and all the edges associated with it results in the increase of the number of connected components in the graph. For example consider the graph given in following figure.



If in the above graph, vertex 1 and all the edges associated with it, i.e. the edges 1-0, 1-2 and 1-3 are removed, there will be no path to reach any of the vertices 2, 3 or 4 from the vertices 0 and 5, that means the graph will split into two separate components. One consisting of the vertices 0 and 5 and another one consisting of the vertices 2, 3 and 4 as shown in the following figure.



Likewise removing the vertex 0 will disconnect the vertex 5 from all other vertices. Hence the given graph has two articulation points: 0 and 1.

Euler Circuits

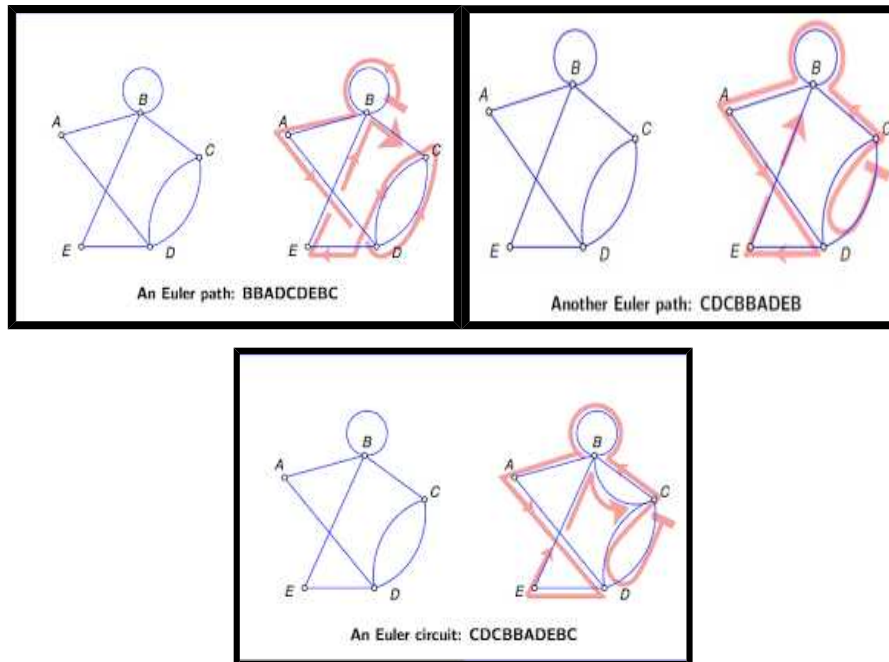
Euler Path

An Euler path is a path that uses every edge of a graph exactly once.

Euler Circuit

An Euler circuit is a circuit that uses every edge of a graph exactly once.

An Euler path starts and ends at different vertices. An Euler circuit starts and ends at the same vertex.



Applications of Graphs

Graphs are nothing but connected nodes(vertex). So any network related, routing, finding relation, path etc related real life applications use graphs.

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Utility graphs. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
5. Protein-protein interactions graphs. Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.
6. Network packet traffic graphs. Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
7. Scene graphs. In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.
8. Finite element meshes. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.
9. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

10. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.
11. Graphs in quantum field theory. Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).
12. Semantic networks. Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.
13. Graphs in epidemiology. Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.
14. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
15. Constraint graphs. Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.
16. Dependence graphs. Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.