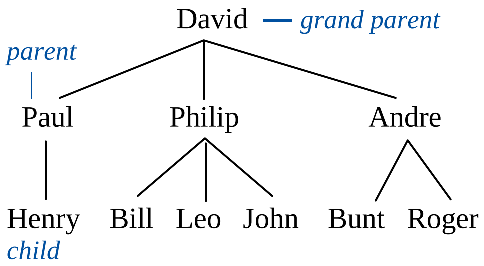
Tree

Data sometimes contain hierarchical relationship (ancestor-descendant, superior-subordinate or similar relationship) between its various elements. Data structure that reflects such a relationship is called a tree (or general tree).

A tree is a non-linear data structure which represents a hierarchical relationship between various elements of the data.

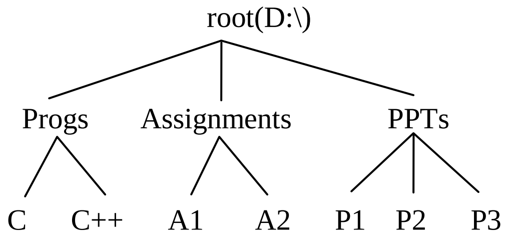
For example, if we want to represent a person and all his descendants then we will have to use the tree data structure (see Family tree structure given below):



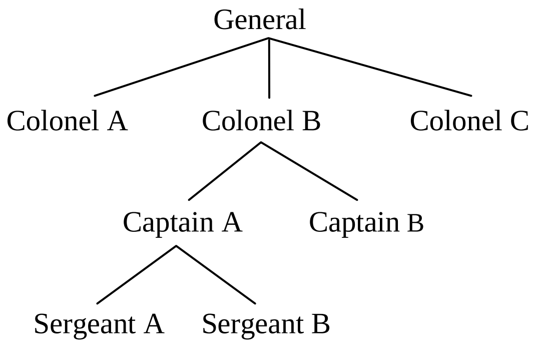
(i) Family tree

Above figure shows descendants of David arranged in hierarchical manner, beginning with David at the top of the hierarchy. David's children (Paul, Philip, Andre) are shown next in the hierarchy, and a line (edge) joins David and his children. Paul has one child, Philip has three and Andre has two children. From this hierarchical representation, it is easy to identify Paul's siblings, David's descendants, John's ancestors and so on.

Similarly following figures show hierarchical folder structure on some storage media and hierarchical structure of positions in a military organization respectively:



(ii) Directory/Folder structure on a storage media



(iii) Hierarchical positions in an organization

A tree is an ideal data structure for representing such kind of hierarchical data. There can be many types of trees in data structures – (general) trees, binary trees, expression trees, binary search trees, threaded binary trees, AVL trees and B-trees.

Definition of Tree

A (general) tree T may be defined as a **finite nonempty set of one or more nodes,** such that:

1. There is a specially designated node called, the root of T
2. The remaining nodes of T are partitioned into m≥0 disjoint subsets T_1, T_2, T_3, … T_m, each of which in turn is a tree and are called subtrees of the root.

In natural trees, root is at the bottom and the branches/leaves grow upwards (from the ground into the air). Contrary to the natural trees, the root of the tree data structure is depicted at the top and branches/leaves grow downwards from top to bottom. In a tree, except root node every node has exactly one parent.

Example:

In Fig.1, a tree T is given in which A is the root of T and T_1, T_2, T_3 are disjoint subtrees of the root node A.

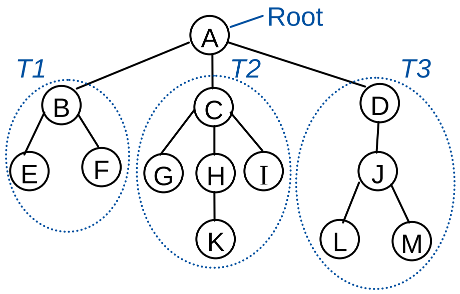


Fig.1: Tree T

Tree Terminology

Some special terms are used while dealing with trees. These are described below:

Parent and Children nodes. If any node N has S_1, S_2 ,… S_m nodes as its children then N is called parent of S_1, S_2, … S_m and nodes S_1, S_2, ... S_m are called children of N. They have 1:m parent-child relationship i.e. a parent can have many children but a child can have only one parent.

Edge. A line drawn from a node to its successor (child).

Ancestor and Descendant. Consider a node X in a tree. Its parent, grand parent and so on up to the root node are called ancestors of the node X. Similarly its children, grand children and so on nodes are called descendants of X.

Siblings. The **children of the same parent** are called siblings (or brothers). For example in Fig.2, nodes H, and I are siblings of G.

Terminal (or Leaf) Node. It is **a node which has no children**. Thus a node which has one or more successors is called a non-terminal or non-leaf node. For example in Fig.2, nodes E,F,G,K,I,L and M are terminal nodes. In a tree, root node is always called root, no matter it has any subtree or not.

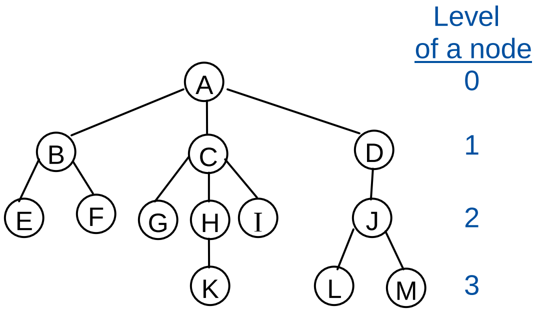


Fig.2

Path. A **sequence of consecutive edges** from the source node to the destination node is called a path. For example, ACH and ADJ in Fig.2. A path ending in a leaf is called a *branch.*

Note: In a tree, there must be one (and only one path) from the root to any other node.

Length of Path. It is **equal to the number of edges** in the path. For example, length of path ACH is 2.

Degree of a Node. It is **equal to the number of subtrees of that node**. For example in Fig.2, degree of node A is 3 and degree of node B is 2. Node with degree 0 is called *leaf* or *terminal* node.

Degree of a Tree. The degree of a tree is **equal to the maximum degree of a node in the tree**. For example for the tree given in Fig.2, the degree of tree is 3.

Level of a Node. The level of the root node is 0. The level of any other node is **1 more than the level number of its parent**. Thus in Fig.2, level of nodes B, C and D is 1, and level of K, L and M is 3. Furthermore, those nodes with the same level number are said to belong to the same generation.

Depth (or Height) of a Tree. It is **equal to the maximum level number of a leaf** in the tree (or maximum number of edges in a branch of the tree). For example, depth (or height) of tree in Fig.2 is 3.

Forest. If we delete the root node and its associated edges connecting the nodes at level 1, then the set of n≥0 disjoint trees so obtained is called a forest. Thus, it is a collection of disjoint trees (see Fig.3). Removing the root of a binary *tree* will produce a forest of two *trees*.

Conversely, if we add just one node to a forest at the top, we get a tree.

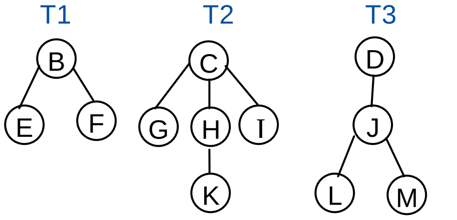


Fig.3: Forest

Binary Tree

- A binary tree T may be defined as **a finite set of zero or more nodes** such that:
  1. T is **empty** (called the null or empty tree)**, if number of nodes is zero**, or
  2. T **contains a specially designated node called the root of T which may have at most two disjoint binary subtrees T_1 and T_2,** called the left subtree and right subtree.
- The above definition of the binary tree T is recursive since T is defined in terms of the binary trees T_1 and T_2. Thus in a binary tree each node may have at most two children (i.e. the degree of each node is 0, 1 or 2) and the children of each node are distinguished as left and right child.
- For example, the diagram in Fig.4 represents a binary tree. Node A at the top of the diagram is the root of the binary tree. A left-downward slanted line from a node indicates a left successor and a right-downward slanted line from a node indicates a right successor of that node.
- Node B and all its descendants are collectively known as left subtree of the root node A. Similarly node C and all its descendants are collectively known as right subtree of the root.

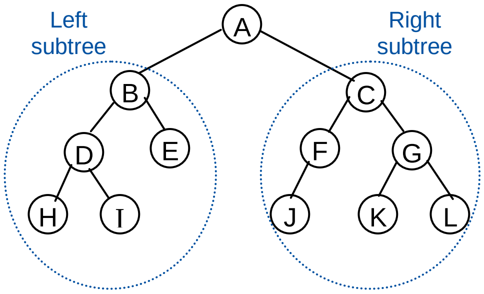


Fig.4: Binary tree

The essential differences between a Binary Tree and a (General) Tree:

- A binary tree may be empty while a (general) tree can’t be empty.
- In a binary tree, a node can have at most two children; but in a (general) tree, a node can have any number of children.
- In a binary tree, the children are distinguished as left or right child (i.e. subtrees are ordered); whereas in a (general) tree there is no such distinction (i.e. subtrees are unordered).

Full Binary Tree (Proper Binary Tree or 2-Tree)

- A binary tree is full **if every node has 0 or 2 children.**
- In a full binary tree, **no. of leaf nodes is equal to the no. of non-leaf nodes + 1**.
- Examples of full binary trees:

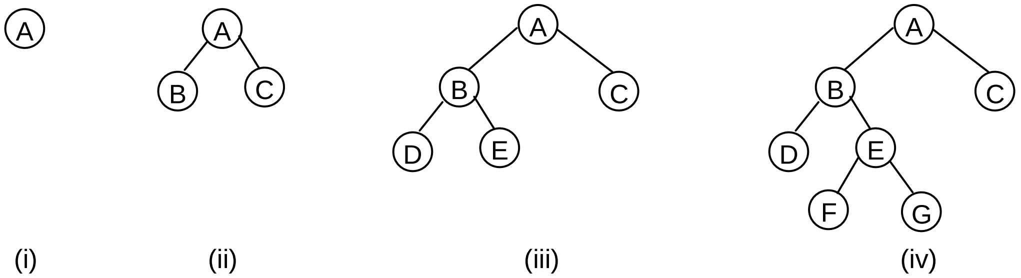


Fig.5: Full binary trees

Perfect Binary Tree

- A binary tree is called perfect binary tree **if all non-leaf nodes have two children and all leaves are at the same level**.
- Examples of perfect binary trees:

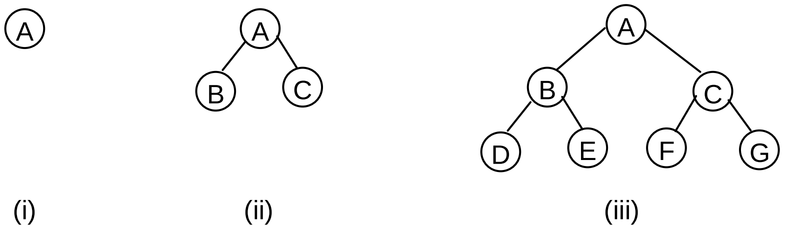
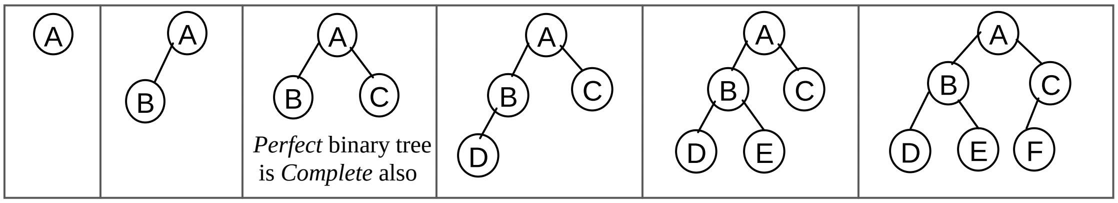


Fig.6: Perfect binary trees

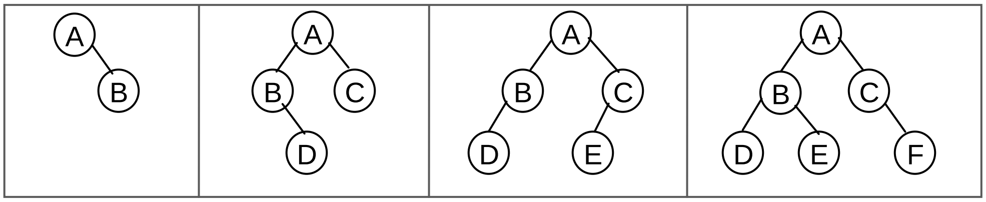
Complete Binary Tree

- A binary tree is said to be complete, if:
  - i). **All its levels (except possibly the deepest level) are fully filled** (i.e. they have maximum number of possible 2^{level} nodes), and
  - ii). **If the deepest level of the binary tree is not fully filled, the nodes of that level are filled strictly from left to right.**

Examples of Complete Binary Trees:



Examples of binary trees that are Not Complete:



Skewed Binary Tree: It can be a left-skewed binary tree or right-skewed binary tree:

1. **Left-skewed binary tree**

   A binary tree in which **every node has a non-empty left subtree and empty right subtree**.

2. **Right-skewed binary tree**

   A binary tree in which **every node has a non-empty right subtree and empty left subtree**.

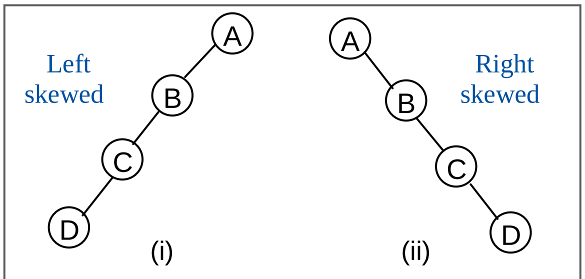


Fig.9: Skewed binary trees

Memory Representation of Binary Trees

A binary tree may be represented in memory in two ways:

1. Sequential representation
2. Linked representation

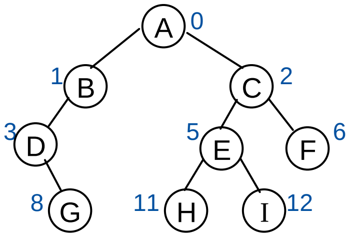


Fig.11

1. Sequential Representation

- In this method a 1-D array is used to represent a binary tree in memory.
- Let T[n] be the array that represents a binary tree. Then in the array T:

1. Root is stored in location T[0].
2. If a node is stored in location T[k], then
   a). its left child is stored in location T[2*k+1] and
   b). its right child is stored in location T[2*k+2].
3. A null entry is used to indicate an empty subtree.

Size of the array T

Approximate size of the required array T is equal to $\mathbf{2^{d+1}}$, where *d* is the depth of the binary tree. (However, if we include null entries for the non-existing successors of the terminal nodes also, then the size of T will be 2^{d+2}).

Example:

Consider the binary tree given in Fig.11. It can be represented in memory using a single array T[16] as below:

Array T[16]

A	B	C	D	–	E	F	–	G			H	I	–	–	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sequential representation

Value at the root (i.e. A) is stored in location T[0]. Left successor of A (i.e. B) is stored in T[1] and right successor (i.e. C) is stored in T[2]. Left successor of B (i.e. D) is stored in T[3]. Since B does not have a right successor, hence null entry is stored in T[4].

In the same way, other nodes of the binary tree are stored in T.

Main Objectives of any Binary Tree Representation Method

1. We must be able to access the root node
2. We must be able to access the left and right subtrees of a given node, and
3. We must be able to access the parent of any given node.

In sequential representation, one can easily determine that:

- Root node is in location T[0].
- The left child of any node k is in location T[2*k+1] and its right child is in T[2*k+2].
- Parent of a node k is in location T[(k-1)/2] *(integer division with truncation)*.

Advantages of Sequential Representation

1. Representation is easy to understand.
2. Best representation for complete and perfect binary tree.
3. Very easy to move from a parent to its children and vice-versa.
4. Programming is very easy.

Disadvantages of Sequential Representation

1. **Wastage of memory: If the binary tree is severely unbalanced or skewed binary** tree then a lot of storage may be wasted. For example, to represent the binary tree given in Fig.12 which have only four nodes, an array T[16] will be required.
2. **Insertion & deletion of a node requires a lot of data movement.**
3. **Insertion of a new node as the child of a leaf at the deepest level is not possible** due to the insufficient size of the array.

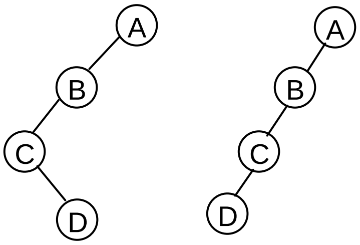


Fig.12: Severely unbalanced binary trees

Prob: Sequential representation of a binary tree is as below:

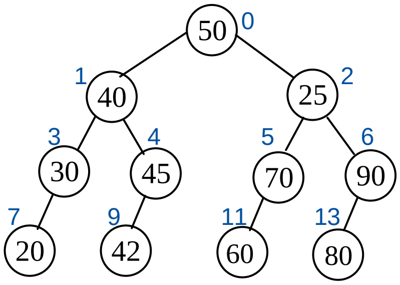
Array T[16]

50	40	25	30	45	70	90	20	–	42	–	60	–	80	–	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sequential representation

Draw the binary tree.

Solu: The binary tree represented by the above linear array T[16] is given below:



Binary tree

2. Linked Representation

- This representation is analogous to the method in which linked lists are represented in memory. In linked representation of binary tree, each node consists of three fields: (i) LEFT pointer (ii) INFO field and (iii) RIGHT pointer.
- For example, consider the binary tree T given in Fig.13. A schematic diagram of the linked representation of T is given in Fig.14. Observe that each node is depicted with its three fields and that the empty subtrees are depicted by using X that indicate Null entries.

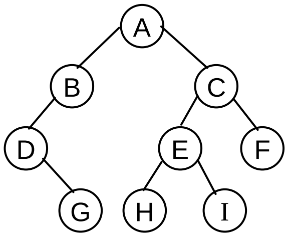


Fig.13

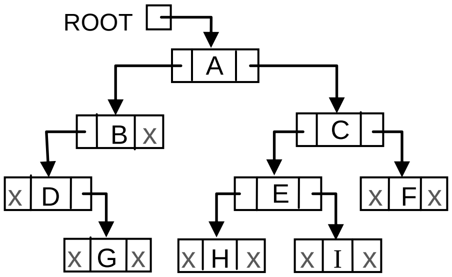


Fig.14: Linked representation

- In linked representation, we need to define the (i) node structure of the binary tree as a C++ structure, say ***node*** and (ii) binary tree as a C++ class named ***bintree*** which contains only one data member (ROOT pointer) to hold the address of the root node of the binary tree (or NULL if binary tree is empty) as below:

```
struct node
{ node *LEFT;     // LEFT pointer to hold the address of the left child
  char INFO;      //INFO field to store data associated with the node
  node *RIGHT;    // RIGHT pointer to hold the address of the right child
};

class bintree
{ private:
    node *ROOT;          //pointer to hold the address of root node
  public:
    bintree()            //constructor
    { ROOT=NULL; }
    . . .
    . . .
};
```

Advantages of Linked Representation

1. Insertions & deletions can be made directly without data movements.
2. Best representation for any type of binary trees.
3. Dynamic size – no predefined limit for the number of nodes.
4. New nodes can be easily inserted anywhere.

Disadvantages of Linked Representation

1. Additional memory is required for storing pointers.
2. Random access to a particular node is not possible.

Binary Tree Traversal

- In many applications we need to systematically traverse a binary tree. Visiting each node of a binary tree exactly once is called traversing a binary tree.
- There are three standard ways in which binary trees can be traversed. These are known as:
  1. **Preorder traversal**
  2. **Inorder traversal**
  3. **Postorder traversal**
- It is easier to define each traversal order by using recursion. This is because a binary tree is recursive in that each subtree is really a binary tree itself. These traversal methods are defined in the following manner:

1. Preorder Traversal (or N-L-R traversal)

To traverse a nonempty binary tree in preorder, following three operations are performed:

Recursive definition:

1. **Process the root R**
2. **Traverse the left subtree of R in preorder**
3. **Traverse the right subtree of R in preorder**

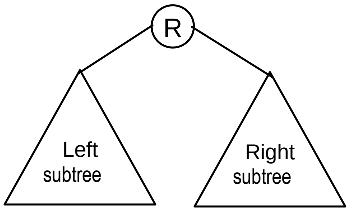


Fig.15

Consider the binary tree given in Fig.16:

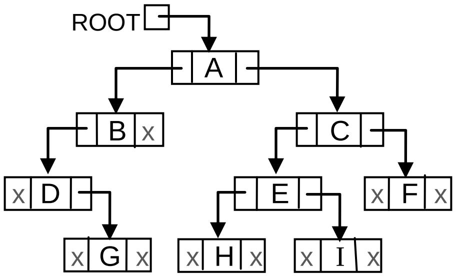
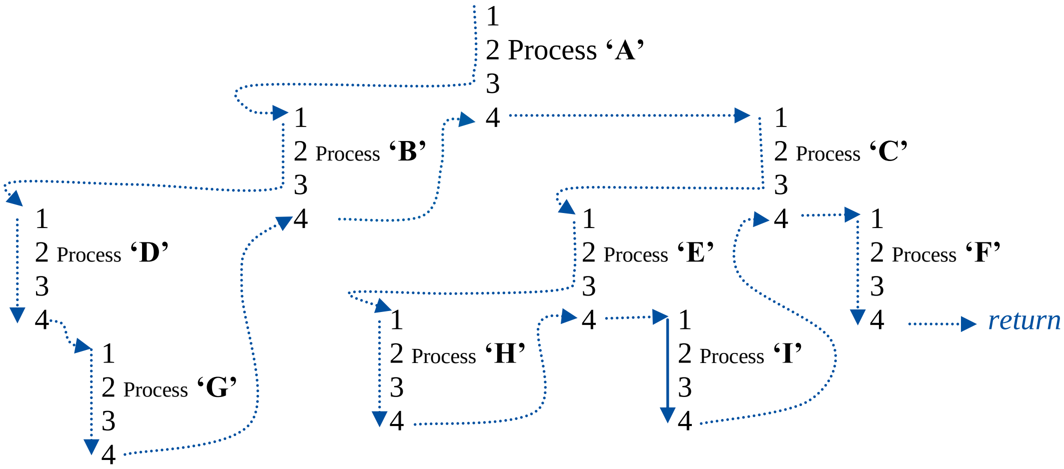


Fig.16: Binary tree

The recursive function will traverse the binary tree (Fig.16) in preorder in the following manner:



Thus, preorder traversal of the binary tree given in Fig.16: **A B D G C E H I F**.

Recursive C++ function for Preorder traversal

```
void bintree::preorder(node *R)
{ if (R==NULL) return;
  cout<<R->INFO<<endl;
  preorder(R->LEFT);
  preorder(R->RIGHT);
}
```

2. **Inorder Traversal** (or L-N-R traversal)

To traverse a nonempty binary tree in inorder, following three operations are performed:

Recursive definition:

1. **Traverse the left subtree of R in inorder**
2. **Process the root R**
3. **Traverse the right subtree of R in inorder**

Thus, inorder traversal of the binary tree given in Fig.16: **D G B A H E I C F**

3. **PostorderTraversal** (or L-R-N traversal)

To traverse a nonempty binary tree in postorder, following three operations are performed:

Recursive definition:

1. **Traverse the left subtree of R in postorder**
2. **Traverse the right subtree of R in postorder**
3. **Process the root R**

Thus, postorder traversal of the binary tree given in Fig.16: **G D B H I E F C A**

Note: Time complexity of traversal operation is **O(n)**.

Exercise: Give preorder, inorder and postorder traversals of the following binary trees:

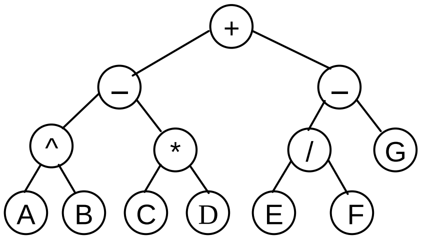


Fig.17

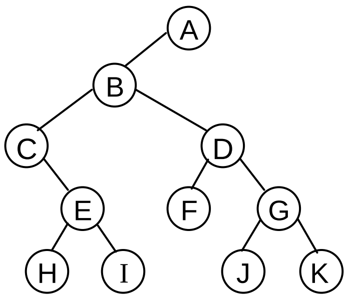


Fig.18

Solu:

Preorder:	**+ – ^ A B * C D – / E F G**	Preorder:	**A B C E H I D F G J K**
Inorder:	**A ^ B – C * D + E / F – G**	Inorder:	**C H E I B F D J G K A**
Postorder:	**A B ^ C D * – E F / G – +**	Postorder:	**H I E C F J K G D B A**

Expression Trees

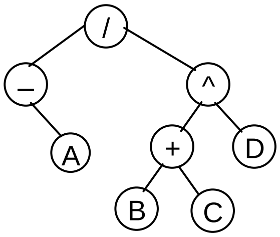


Fig.19

- are binary trees that represent algebraic expressions in a tree-like structure. In expression trees, internal (non-leaf) nodes contain operators and external (leaf) nodes contain operands.
- In case of unary operator, the left child is absent and the right child contains the operand.
- For example, the binary tree given in Fig.19 represents the algebraic expression:

`-A/(B+C)^D`

Representing Algebraic Expressions as Binary Trees

This is an important application of binary trees. Algebraic expressions can be represented as binary trees, called expression trees.

Technique of Converting an Algebraic Expression into Expression Tree

1. Note down the order in which the operators in the expression will be evaluated based on the precedence and associativity of operators.
2. The operator that will be evaluated in the last will be the root of the tree. The subexpression to the left of the root will be in the left subtree of the root and the subexpression to the right of the root will be in the right subtree of the root.
3. The same process is then applied to each subexpression repeatedly until we get the desired binary tree.

Constructing the Expression Trees corresponding to the given Algebraic Expressions

Example: Draw the binary tree corresponding to the following algebraic expression. Is this a unique binary tree?

```
A^B-C*D+E/(F-G)
```

Solu: Note down the order in which the operators in the expression will be evaluated based on their precedence and associativity as below:

First parentheses **( )** will be evaluated:

$$A\,{}^\wedge B-C*D+E/(F\overset{1}{-}G)$$

Then ^ :

$$A\overset{2}{{}^\wedge}B-C*D+E/(F\overset{1}{-}G)$$

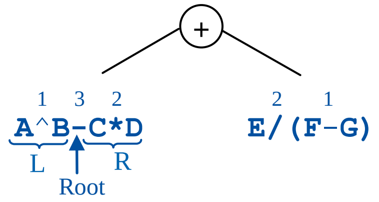
Then *****, / :

$$A\overset{2}{{}^\wedge}B-C\overset{3}{*}D+E\overset{4}{/}(F\overset{1}{-}G)$$

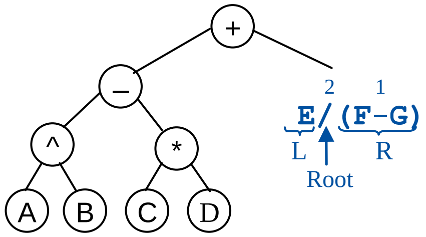
Then **–, +** :

$$\underbrace{A\overset{2}{{}^\wedge}B\overset{5}{-}C\overset{3}{*}D}_{L}\underset{\uparrow\atop\text{Root}}{\overset{6}{+}}\underbrace{E\overset{4}{/}(F\overset{1}{-}G)}_{R}$$

The addition operation (labeled as 6) is the last operation to be performed. Therefore this operator will be the root of the tree and the left and right subexpressions will be in the left and right subtrees of the root node respectively.



In the left subexpression, subtraction operation (labeled as 3) is the last operation to be performed. Therefore this operator will be the root of the left subtree and the left and right subexpressions will be in the left and right subtrees of the root node respectively.



In the right subtree of the root, division (labeled as 2) is the last operation to be performed. Therefore this operator will be the root of the right subtree and the left and right subexpressions will be in the left and right subtrees of the root node respectively.

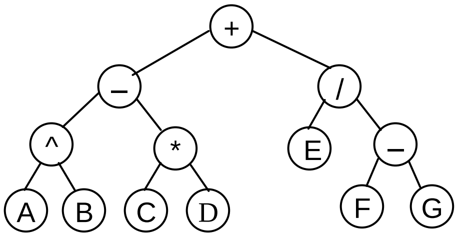


Fig.20

Thus, the binary tree given in Fig.20 is the desired expression tree corresponding to the given algebraic expression.

Obviously**, every algebraic expression will correspond to a unique Binary Tree and vice-versa**.

Inorder, Preorder and Postorder Traversals of the Expression Tree

Inorder, preorder and postorder traversal of the expression tree gives the infix, prefix and postfix forms of the algebraic expression respectively, e.g. traversals of expression tree given in Fig.20:

Inorder traversal: **A ^ B – C * D + E / F – G** (Infix expression)

Preorder traversal: **+ – ^ A B * C D / E – F G** (Prefix expression)

Postorder traversal: **A B ^ C D * – E F G – / +** (Postfix expression)

Exercise-1: Draw the binary tree corresponding to the following algebraic expression:

A+B*C+(D^E-F)*G

Solu: First parentheses will be evaluated:

$$A+B*C+(D\overset{1}{\hat{}}E\overset{2}{-}F)*G$$

Then ***,*** :

$$A+B\overset{3}{*}C+(D\overset{1}{\hat{}}E\overset{2}{-}F)\overset{4}{*}G$$

Then **+,+** :

$$\underbrace{A\overset{5}{+}B\overset{3}{*}C}_{L}\underset{\text{Root}}{\overset{6}{+}}\underbrace{(D\overset{1}{\hat{}}E\overset{2}{-}F)\overset{4}{*}G}_{R}$$

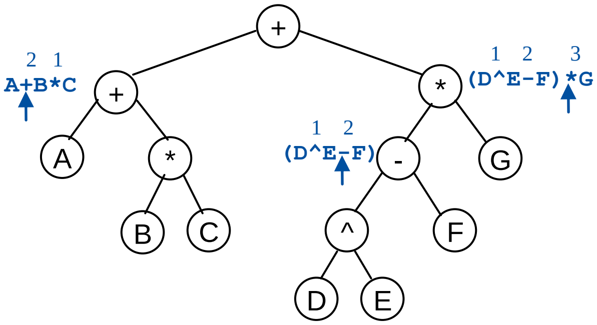


Fig.21

The binary tree corresponding to the above expression is given in Fig.21.

Exercise-2: Draw the binary tree corresponding to the following algebraic expression:

A^B-C*(A-B/D)+E*C^2

Solu: First parentheses **( )** will be evaluated:

$$A\hat{}B-C*(A\overset{2}{-}B\overset{1}{/}D)+E*C\hat{}2$$

Then **^, ^**:

$$A\overset{3}{\hat{}}B-C*(A\overset{2}{-}B\overset{1}{/}D)+E*C\overset{4}{\hat{}}2$$

Then ***, ***:

$$A\overset{3}{\hat{}}B-C\overset{5}{*}(A\overset{2}{-}B\overset{1}{/}D)+E\overset{6}{*}C\overset{4}{\hat{}}2$$

Then **–, +** :

$$\underbrace{A\overset{3}{\hat{}}B\overset{7}{-}C\overset{5}{*}(A\overset{2}{-}B\overset{1}{/}D)}_{L}\underset{\text{Root}}{\overset{8}{+}}\underbrace{E\overset{6}{*}C\overset{4}{\hat{}}2}_{R}$$

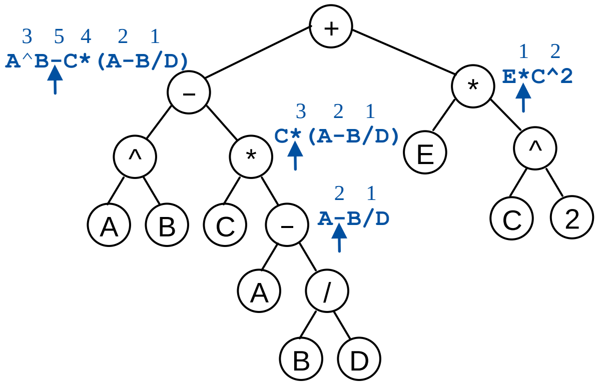


Fig.22

The binary tree corresponding to the above expression is given in Fig.22.

Constructing Binary Tree Whose Preorder and Inorder Traversals are Given

Prob1: Suppose a binary tree has 9 nodes. The preorder and inorder traversals of the binary tree yield the following sequence of nodes:

Preorder:	**A B D E G H C F I**	(N-L-R traversal)
Inorder:	**D B G E H A C I F**	(L-N-R traversal)

Draw the binary tree. Is it unique binary tree?

Solu:

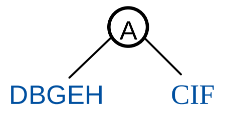
As you know the processing order of nodes in preorder and inorder traversals is **N-L-R** and **L-N-R** respectively. Therefore:

(i) from the given preorder traversal, it is clear that **A** is the root node of the binary tree, and

(ii) from the given inorder traversal, it is clear that the nodes **DBGEH** will be in the left subtree of the root and **CIF** will be in the right subtree of the root as given below:

Preorder:	$\boxed{A}\ \underline{B\ D\ E\ G\ H}\ \underline{C\ F\ I}$
Inorder:	$\underbrace{\underline{D\ B\ G\ E\ H}}_{\text{Left}}\ \underbrace{\boxed{A}}_{\text{Root}}\ \underbrace{\underline{C\ I\ F}}_{\text{Right}}$

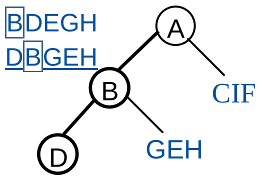
Now let's draw the desired binary tree step-by-step. A is the root node and nodes **DBGEH** will be in its left subtree and **CIF** in the right subtree.



Constructing Left subtree of root A

The traversal orders of nodes in the Left subtree of A are:

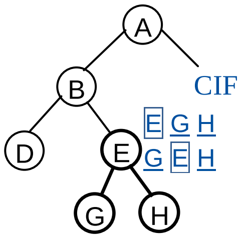
Preorder:	$\boxed{B}\ \underline{D}\ \underline{E\ G\ H}$
Inorder:	$\underline{D}\ \boxed{B}\ \underline{G\ E\ H}$



From the above, it is clear that the root of the Left subtree of A is **B** and (i) **D** will be in in its Left subtree and (ii) **GEH** in its Right subtree.

The traversal orders of nodes in the Right subtree of B are:

Preorder:	$\boxed{E}\ \underline{G}\ \underline{H}$
Inorder:	$\underline{G}\ \boxed{E}\ \underline{H}$

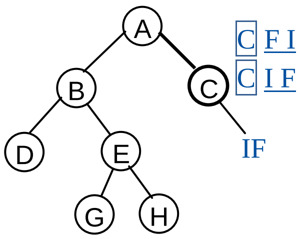


From the above, it is clear that the root of the Right subtree of B is **E** and (i) **G** will be in in its Left subtree and (ii) **H** in its Right subtree.

Constructing Right subtree of root A

The traversal orders of nodes in the Right subtree of A are:

Preorder:	$\boxed{C}\ \underline{F\ I}$
Inorder:	$\boxed{C}\ \underline{I\ F}$



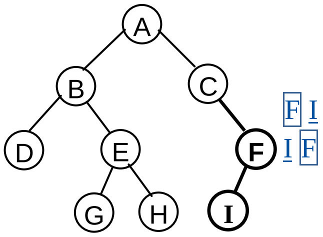
From the above, it is clear that the root of the Right subtree of A is **C** and (i) its Left subtree is empty and (ii) nodes **IF** will be in its Right subtree.

The traversal orders of nodes in the Right subtree of C are:

Preorder: $\boxed{F}\ \underline{I}$

Inorder: $\boxed{I}\ \underline{F}$

From the above, it is clear that the root of the Right subtree of C is **F** and (i) I will be in its Left subtree and (ii) its Right subtree is empty.



Desired binary tree

Fig.23

Thus, the binary tree given in Fig.23 is the binary tree whose preorder traversal is **ABDEGHCFI** and inorder traversal is **DBGEHACIF**.

It is **unique binary** tree.

Exercise-1: Suppose a binary tree has 9 nodes. The preorder and inorder traversals of the binary tree yield the following sequence of nodes:

Preorder: **1 2 3 4 6 7 5 8 9**
Inorder: **2 6 4 7 3 8 5 9 1**

Draw the binary tree.

Ans: The binary tree is given in Fig.24.

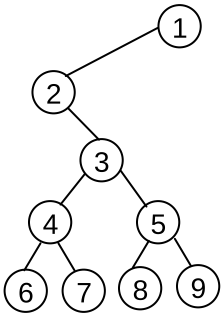


Fig.24

Exercise-2: Suppose a binary tree has 7 nodes. The preorder and inorder traversals of the binary tree yield the following sequence of nodes:

Preorder: **E X A M F U N**
Inorder: **M A F X U E N**

Draw the binary tree.

Ans: The binary tree is given in Fig.25.

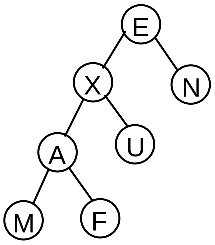


Fig.25

Binary Search Tree (or Binary Sorted Tree)

It is defined as **a binary tree in which at each node N:**

1. **the value of its left subtree is < the value of N,** and
2. **the value of its right subtree is ≥ the value of N.**

For example, the binary tree given in Fig.26 is a binary search tree.

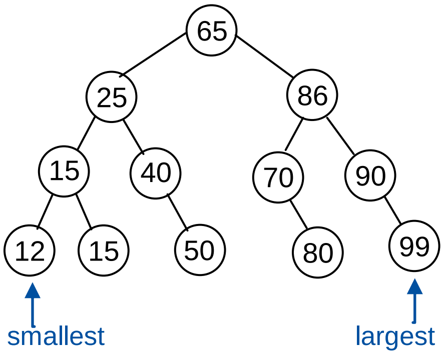


Fig.26: Binary search tree

Operations on Binary Search Tree (BST)

Operations that are commonly performed on BST:

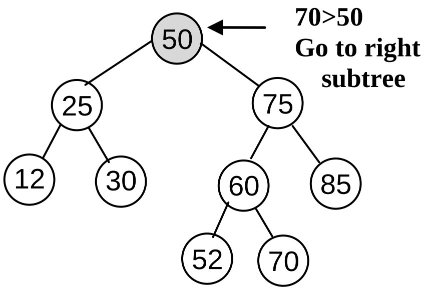
1. Searching
2. Insertion
3. Deletion
4. Traversal

Searching in BST

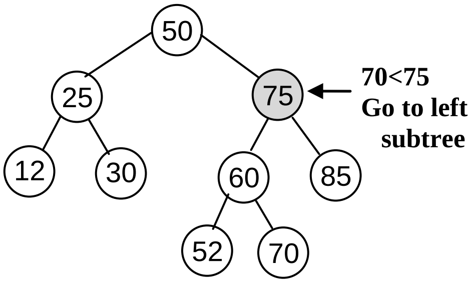
Whenever an element is to be searched, we start **searching** from the root node. Steps to search an element, say *item*, in the binary search tree are as below:

1. compare the item with the key value at the root of the tree.
2. if item=key value then return the location of the node.
3. else if item<key value then move down to the left subtree.
4. else move down to the right subtree.
5. repeat steps 1 to 2 recursively until match is found or a NULL pointer is encountered.
6. if element is not found then return NULL.

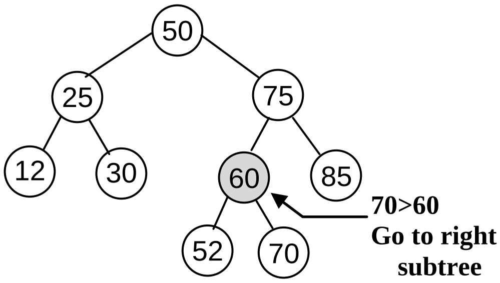
Example: Searching for item=70:



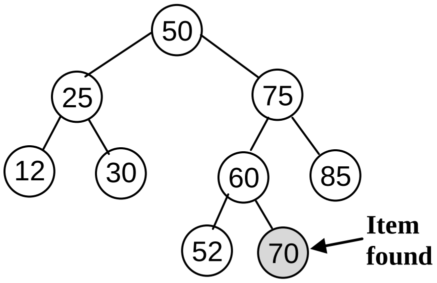
Step 1



Step 2



Step 3



Step 4

C++ recursive function to find the node in a BST that contains a given element, say *item*, is given below:

```
NODE* BST::search(node *R,int item)
{ if(R==NULL||item==R->INFO)
      return R;
  else if(item<R->INFO)
      return(search(R->LEFT,item));
  else
      return(search(R->RIGHT,item));
}
```

```
struct node
{ node *LEFT;
  int INFO;
  node *RIGHT;
};
class BST
{  node *ROOT;
  public:
   BST() { ROOT=NULL; }
   ……
   ……
};
```

Insertion in BST

To insert an element (item), we search for the item in the BST until a NULL pointer is encountered. Steps to insert item into a BST are as below:

```
1. if the BST is empty then
2.       insert item as root node and return
3. ▷search the value to be inserted in the BST until an empty subtree is encountered
4. if item < key value then
5.        move down to the left subtree
6. else move down to the right subtree.
7. repeat steps 4 to 6 until an empty subtree is encountered.
8. insert item as the appropriate (left or right) child of the current node (where empty
   subtree is encountered).
9. return
```

Example: Suppose we want to insert an element item=65 in the binary search tree given in Fig.26(c).

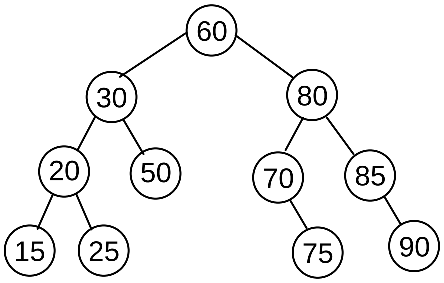


Fig.26(c): Binary search tree

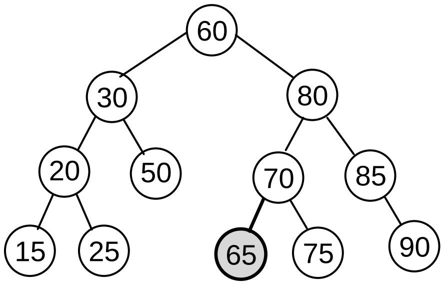


Fig.26(d): BST (after insertion)

In order to insert an element 65 in the BST, first we need to search the element to be inserted in the BST until an empty subtree (NULL pointer) is encountered. So we will compare 65 with 60. Since 65>60, hence we will move down to the right subtree. Now compare 65 with 80. Since 65<80, hence we will move down to the left subtree. Now compare 65 with 70. Since 65<70, hence we need to move down to the left subtree but it is empty. Now insert 65 as the left child of 70. Binary search tree after insertion operation is given in Fig.26(d).

Building a BST From Given Data

Suppose we want to construct a Binary search tree that represents the following names:

DROVER, BOB, ALAN, GREEN, FRY, LEO, CRAY, HARIS, DIXON, BOON, BUNT.

Binary search tree may be constructed from the given data in the following manner:

- Since initially the BST is empty, hence first element i.e. DROVER is inserted as root node of the BST (refer Fig.27).
- Next element to be inserted is BOB. It is compared with root node i.e. DROVER. Since BOB is lexically < DROVER, hence BOB is inserted as left successor of DROVER.
- Next element to be inserted is ALAN. It is compared with root node i.e. DROVER. Since ALAN is lexically < DROVER, hence we move down to the left subtree. Now ALAN is compared with BOB. Since ALAN is lexically < BOB, hence ALAN is inserted as left successor of BOB.
- Next element to be inserted is GREEN. It is compared with root node i.e. DROVER. Since GREEN is lexically > DROVER, hence GREEN is inserted as right successor of DROVER.
- Next element to be inserted is FRY. It is compared with root node i.e. DROVER. Since FRY is lexically > DROVER, hence we move down to the right subtree. Now FRY is compared with GREEN. Since FRY is lexically < GREEN, hence FRY is inserted as left successor of GREEN.
- Similarly, the remaining names are inserted one-by-one and finally we get the BST given in Fig.27.

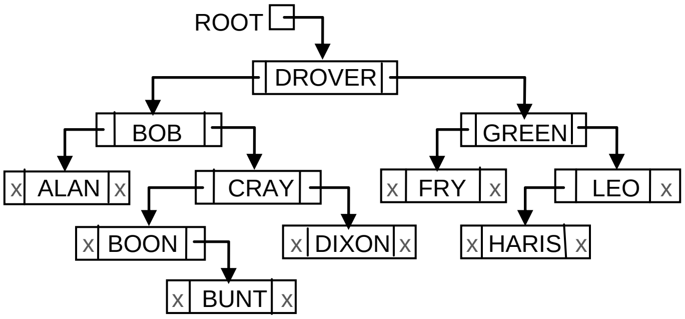


Fig.27: Lexically ordered BST

Inorder Traversal of a BST Gives Data in Sorted order

An important property of BST is that if it is traversed in inorder we get the data represented by the BST in ascending order. For example, inorder traversal of the BST given in Fig.27 is:

ALAN, BOB, BOON, BUNT, CRAY, DIXON, DROVER, FRY, GREEN, HARIS, LEO.

This is known as *tree sort* method which is described below:

Tree Sort

Tree sort method is based on binary search tree. In this method, data can be sorted in ascending order as below:

1. Create a BST from the data to be sorted and
2. Traverse the BST in inorder.

To sort the data in descending order, any one of the following approaches can be used:

Approach-I: Create a BST and perform **reverse inorder** traversal i.e. use **R-N-L** traversal order (refer fig.27(a)) or

Approach-II: Create a BST such that **smaller elements go right** and **larger elements go left** (refer Fig.27(b)). Then traverse it in inorder.

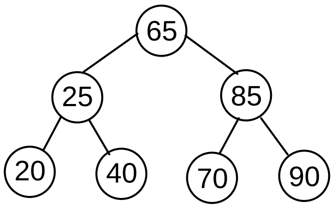


Fig.27(a): R-N-L traversal gives
90, 85, 70, 65, 40, 25, 20

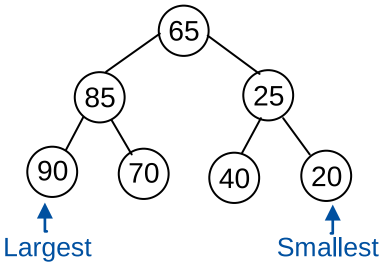


Fig.27(b)

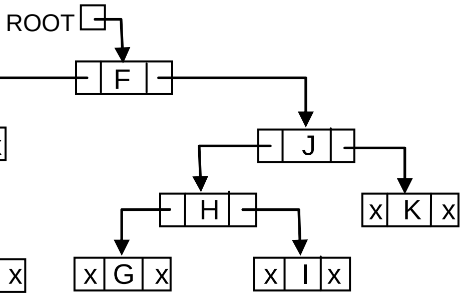
Deletion of a Node in the BST

In a BST, a node is deleted in such a way that the resultant binary tree is still a BST. To achieve this, a node of a BST is deleted based on how many subtrees it has in the following manner:

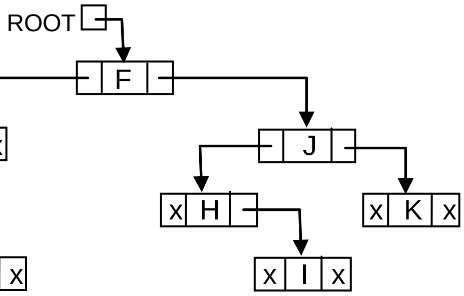
(i). Deleting a Node which is Leaf

To delete it, replace its address in its parent node by NULL.

Example: Suppose we want to delete node G from the following BST:



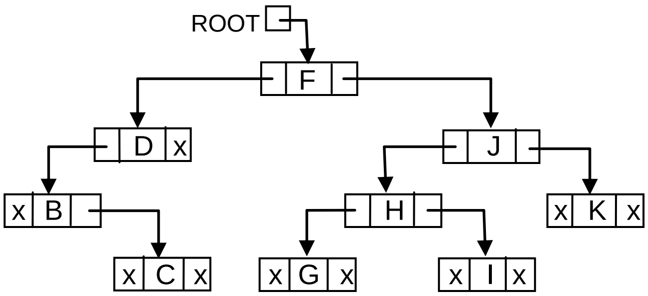
BST after node G is deleted:



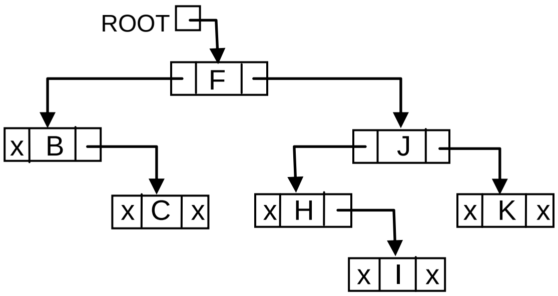
(ii). Deleting a Node which has only One Subtree

To delete it, replace its address in its parent node by the address of its only subtree.

Example: Suppose we want to delete node D from the following BST:



BST after node D is deleted:

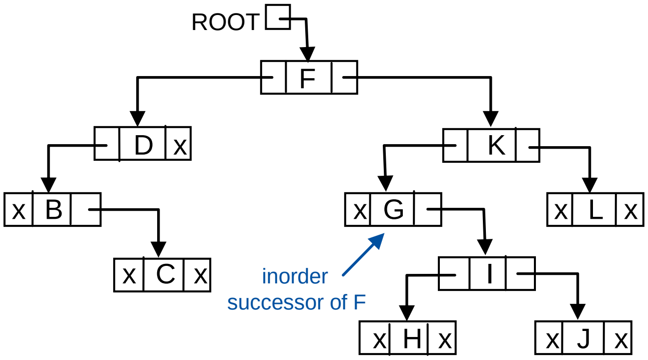


(iii). Deleting a Node which has Two Subtrees

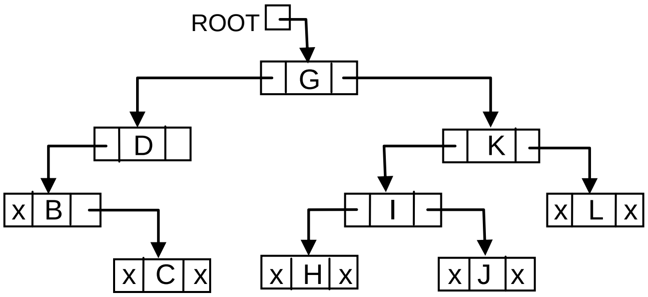
It is deleted in the following manner:

1. Find inorder successor of the node to be deleted (it is the next node in inorder traversal of the BST).
2. Replace the node to be deleted by its inorder successor.
3. The position vacated by inorder successor is filled by its right subtree (if it exists).

Example: Suppose we want to delete node F from the following BST:



BST after node F is deleted:



Prob: Given following data:

500, 510, 605, 460, 570, 485, 495, 475, 700, 470, 435, 200, 580, 472, 474

Perform following operations:

1. Construct a BST from the given data.
2. Delete the node containing 510 and draw the resultant BST.
3. Delete the node containing value 460 and redraw the resultant BST.
4. Give inorder traversal of the final BST.

Solu:

1. The BST built from the above data is given in Fig.29:

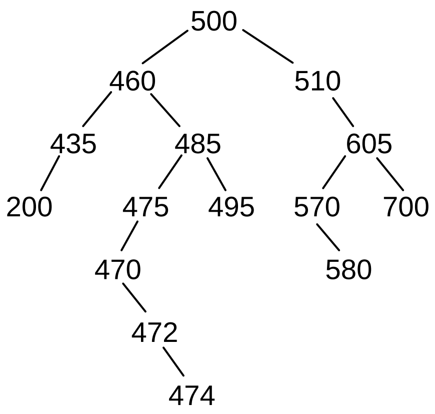
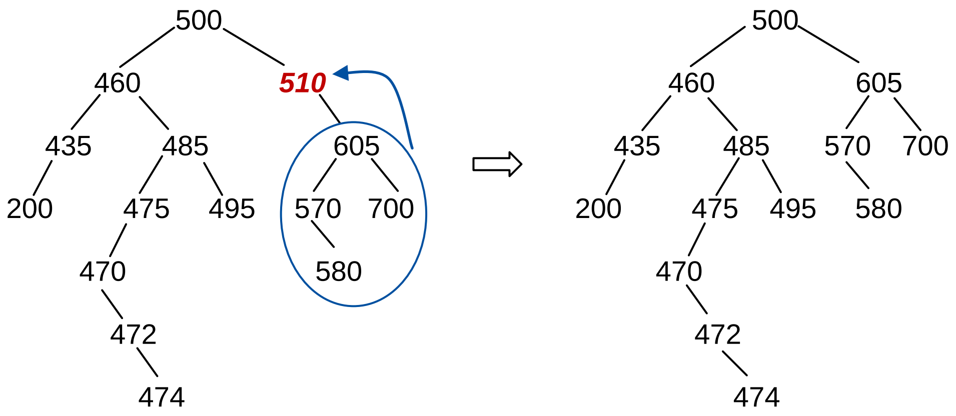


Fig.29: BST

2. Since the containing value 510 has only one subtree, hence to delete it, its address in its parent node is replaced by the address of its only subtree. The resultant BST is given in Fig.30:

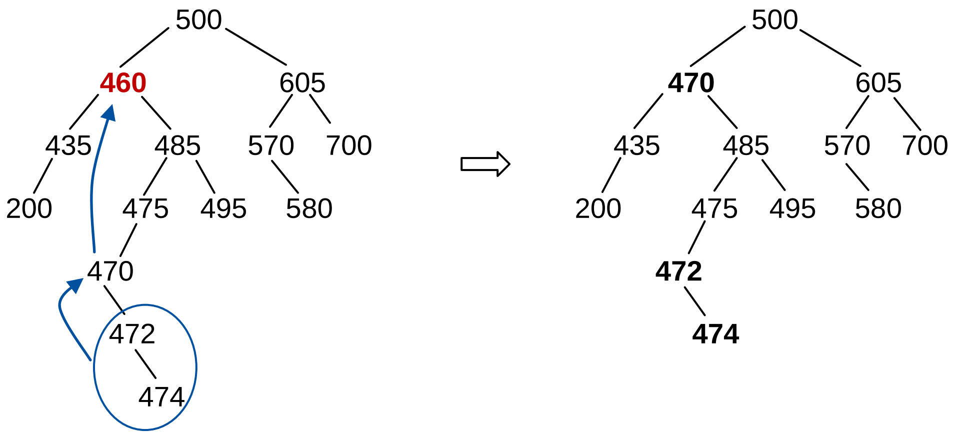


Deletion of node with value 510

Resultant BST

Fig.30

3. The node containing 460 will be deleted in the following manner (see Fig.31):



Deletion of node with value 460

Resultant BST

Fig.31

4. Inorder traversal of BST given in Fig.31 is:

200, 435, 470, 472, 474, 475, 485, 495, 500, 570, 580, 605, 700.

Time Complexity of Various Operations in BSTs

Performance of insertion, deletion, retrieval and searching in binary search trees is O(h). Binary Search Trees allow approximate binary search because after each comparison we select the appropriate left or right subtree for further searching and the other subtree is completely discarded. A BST has a minimum height = $\lfloor \log n \rfloor$ when it is completely balanced and has a maximum height = n-1 when it is severely unbalanced (refer Fig.32(a) and 32(b)). Therefore:

Best case: If the **BST is completely balanced** (see Fig.32(a)), then performance of insertion deletion and searching is **O(log n)** which is equivalent to binary search.

Worst case: If the BST is **severely unbalanced** (see Fig.32(b) (i) and 32(ii)), then the performance of insertion deletion and searching is **O(n)** which is equivalent to linear search. For example, if we search for 70 in the BST then n comparisons will be made during the searching operation.

Average case: It is approximately equal to that of binary search i.e. **O(log n)**.

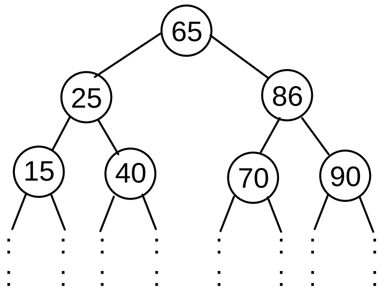
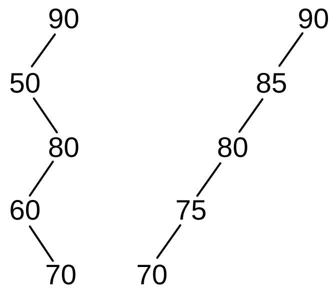


Fig.32(a): Completely balanced BST



(i) (ii)

Fig.32(b): Severely unbalanced BSTs

Advantages of Binary Search Trees

1. Insertion and deletion operations can be performed efficiently.
2. Allow binary search.
3. Data sorting (using Tree sort method).

Note: A C++ program "bst1.cpp" that implements and demonstrates insertion, deletion, searching, and traversal operations in a Binary Search Tree is given in *Programs* folder.

Applications of Binary Trees

There are many applications/uses of binary trees:

1. Whenever we want to take two-way decisions, binary trees are the best option. One such application is a BST which we have already discussed.
2. To represent algebraic expressions as binary trees, called expression trees. The subtrees represent subexpressions.

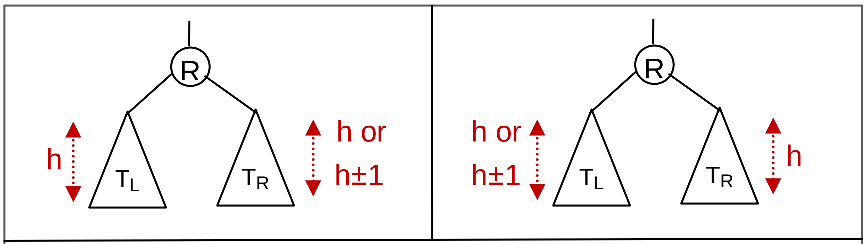
   Further, inorder traversal yields the infix form of the expression. Preorder traversal yields the prefix form of the expression and postorder traversal yields the postfix form of the expression.
3. To convert an infix expression into prefix or postfix form. For this create the expression tree for the given infix expression and traverse it in preorder or postorder respectively.
4. Representing a general tree by an equivalent binary tree. When a linked structure is used, it is more space-efficient representation because it has fewer wasted null pointers.
5. Data sorting using Tree sort.
6. An efficient sorting method, called heap sort is based on a special kind of binary tree, called heap.

Height-Balanced Binary Search Tree (or AVL Tree)

AVL tree is a height-balanced bainary search tree which was introduced by the Russian Mathematicians: G.M. **A**delson, **V**elskii and E.M. **L**andis who first defined and studied this form of tree in 1962.

An AVL tree is a binary search tree T in which for each node N, the difference in the heights of its two subtrees (**T_L** and **T_R**) **is not more than 1.**

This constraint placed on the height of subtrees is known as **AVL tree property**.



If for each node with root R in a BST, the height of its two subtrees is like above - then it is an AVL tree.

The **difference in the heights h(T_L)-h(T_R)** is known as **balance factor** (BF) of that node. Thus the **balance factor of a node** in an AVL tree is:

-1: if the height of its left subtree is one less than the height of its right subtree,

0: if the height of its left subtree is same as the height of its right subtree, or

+1: if the height of its left subtree is one more than the height of its right subtree.

A node with any other balance factor is considered unbalanced and requires rebalancing the tree.

An example of AVL tree is given in Fig.45(a) with the balance factor of each node given beside it.

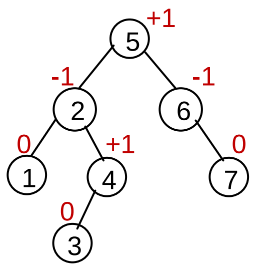


Fig.45(a): AVL tree

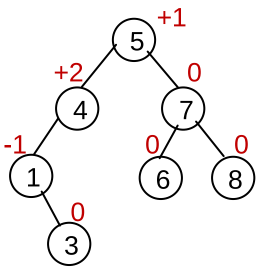


Fig.45(b): Not an AVL tree

Note: *An empty binary tree is a BST which is an AVL tree. The height of an empty AVL tree is taken as -1.*

Memory Representation of AVL Tree

AVL trees are represented in memory in the same way as binary search trees are represented. The only difference is that at every node its balance factor is also stored as below:

LEFT	INFO	BAL_FAC	RIGHT

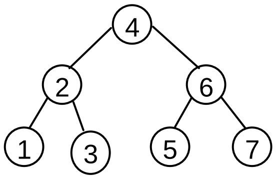
Node structure of AVL tree

Searching in AVL Tree

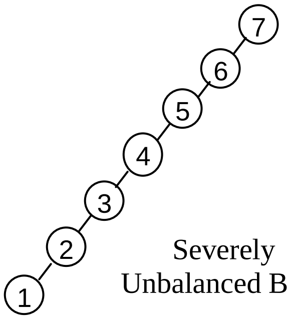
Searching in an AVL tree is exactly similar to the method used in binary search trees.

Advantage of AVL trees

Since AVL trees are height-balanced, they guarantee to perform search, insertion and deletion operations in **O(log n)** time in both the average and worst case. Whereas, in case of unbalanced binary search trees, the efficiency of these operations is O(n).



AVL Tree



Severely Unbalanced BST

Note: If an insertion or deletion causes an AVL tree to become unbalanced, then the tree must be restructured to return it to a balanced state.

m-Way (or Multi-Way) Search Tree

A binary search tree (BST) is a **2-way search tree** because in a BST, each node may have at most 2 subtrees (or branches) i.e. the degree of each node (or order of tree or branching factor) is 2 (refer Fig.46).

Similarly, if we store at most two values at each node and each node has at most 3 subtrees, then it will be a **3-way search tree**, also called **2-3 tree**. A 3-way search tree is given in Fig.47.

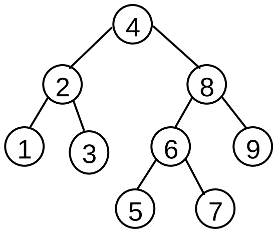


Fig.46: BST

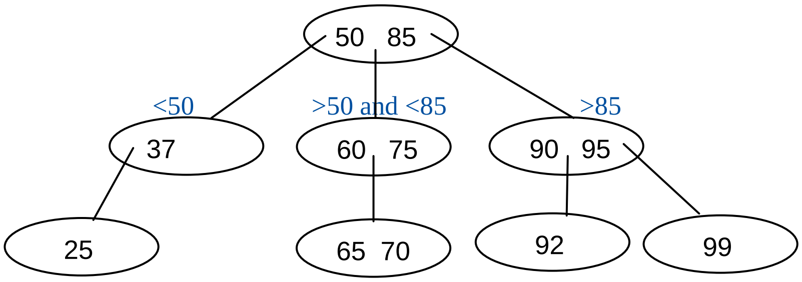
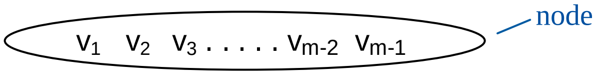


Fig.47: 3-way search tree

BST can be generalized by storing more values at each node and such a search tree is called an m-way search tree. Thus m-way search trees are generalized version of binary search trees that can be used for storing the large set of data. m is called the degree of the tree.

Definition: An **m-way search tree** may be empty. If it is non-empty, it satisfies the following properties:

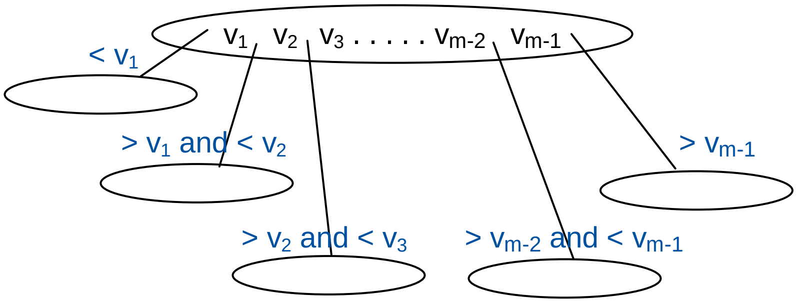
i). Each node can have at most *m-1* keys which are stored in increasing order in the following manner:



where, $v_1 < v_2 < v_3 < \ldots. < v_{m-2} < v_{m-1}$

ii). Each non-leaf node can have at most *m* child nodes. A node with k keys, where k≤(m-1), can have at most k+1 child nodes (subtrees).

iii). The keys in the subtrees are partitioned as a search tree as below:


v_1 v_2 v_3 . . . . . v_{m-2} v_{m-1}
$< v_1$
$> v_1$ and $< v_2$
$> v_{m-1}$
$> v_2$ and $< v_3$
$> v_{m-2}$ and $< v_{m-1}$

The left-most (or first) subtree of a node has values < v1,
The second subtree has values > v1 and < v2 and so on.
The second right-most subtree has values $> v_{m-2}$ and $< v_{m-1}$, and
The right-most subtree has values $> v_{m-1}$

iv). Each of the subtrees is also an m-way search tree.

Example:

An example of a 5-way search tree is given in Fig.48. Observe how each node has at most 5 children and therefore has at most 4 values (keys) contained in it. It also satisfies the rest of the properties of an m-way search tree:

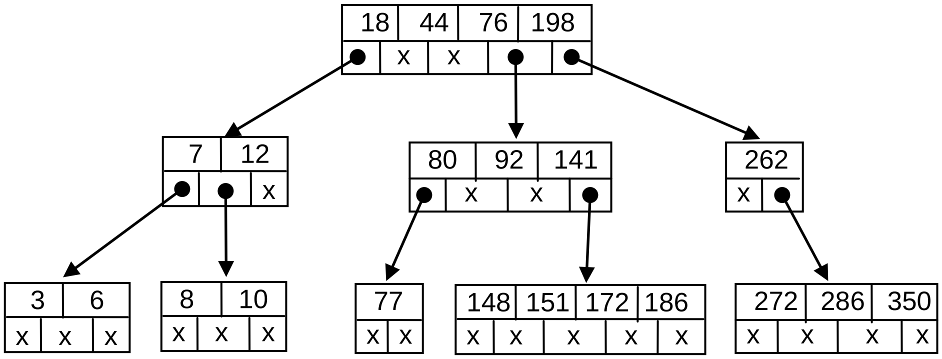


Fig.48: 5-way search tree

Note:

- A node containing m-1 keys in an m-way search tree is known as ***full node***.
- A non-full node may also have one/more subtrees.
- In the above example, for simplicity we have taken a small value of m. But bear in mind that, in practice, m is usually very large.

- Each node corresponds to a physical block on disk, and m represents the maximum number of data items that can be stored in a single block. m is maximized in order to speedup processing: to move from one node to another involves reading a block from disk which is a very slow operation compared to moving around a data structure stored in internal memory.

Searching an m-Way Search Tree

The process of searching in an m-way search tree is similar to that of searching in BST. The searching starts from the root node. For example, suppose we want **to search for a value (key) V=148** in the 5-way search tree given in Fig.48. Then the searching operation proceeds as below:

1. V is first compared with the first value at the root i.e. 18. Since V>18, hence it is compared with the second value i.e. 44. Since V>44, it is compared with the third value i.e. 76. Since V>76 and V<198, hence searching operation proceeds in the fourth subtree of the root node.
2. Now V is compared with the first value contained in the current node (root node of the fourth subtree) i.e. 80. Since V>80, hence it is compared with second value i.e. 92. Since V>92, hence it is compared with third value i.e. 141. Since V>141, hence searching operation proceeds in the fourth subtree of the current node.
3. Now V is compared with the first value contained in the current node i.e. 148. Since V is equal to this first value, hence searching operation is successful.
4. Similarly, **to search for V=13**, we begin at the root and as 13<18, we move down to the first subtree. Again since V>7, hence it is compared with the second value i.e. 12. Since 13>12, we need to move down the third subtree of the current node but it is empty. This indicates that the search for V=13 is unsuccessful.

In m-way search tree, searching, insertion and deletion operations need O(h) disk accesses and maximum height of an m-way search tree can be n.

m-Way Search Trees are Useful for Storing Large Data in External Memory

- All the data structures discussed so far (array, stack, queue, linked list, binary tree, binary search tree, height-balanced (AVL) tree etc.) are used to store, manipulate and retrieve data in internal memory. These structures are well suited to applications in which the data is small enough to be accommodated in the internal memory.
- However, when the data is too large (that can not fit into the main memory), it has to be stored in external memory. If BST is used to store the large data set on disk, search time will be more due to large number of disk access. This is due to the greater height of binary tree as each node can only have two child at most. Therefore there is a need for some special type of data structures that enable us to minimize disk accesses. Since the time required for a disk access is significantly more than that for an internal memory access, we need structures that would reduce the number of disk accesses due to their restricted (smaller) height. The examples of such data structures are m-way search trees, B trees, and B^+ trees. The **limitation** of m-way search tree is that it is not balanced. This limitation is overcome by B trees and B+ trees, where B stands for balanced m-way search trees.

B-Tree

- We have already defined m-way search tree. A B-tree is a balanced m-way search tree (also known as balanced sorted tree).
- To reduce disk accesses, it is essential that the height of the B-tree must be kept to a minimum. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a AVL tree, or B-tree must be used.

Definition: A B-tree of order m, if non-empty, is an m-way search tree in which:

i). Each node (except the root and leaves) has at least $\lceil m/2 \rceil$ children and at most m children.

ii). The root node has at least two children if it is not a leaf, or none if the tree consists of the root only.

iii). All the leaf nodes are on the same level.

The first condition ensures that each node of the tree is at least half full, the second one forces the tree to branch early while the third one keeps the tree balanced.

Example of B-Tree:

A B-tree of order 5 is given in Fig.49. All the properties of the B-tree can be verified on the tree.

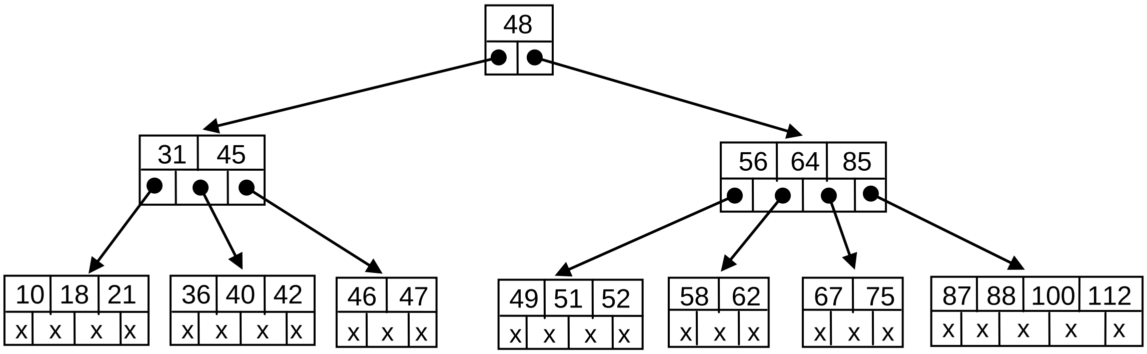


Fig.49: B-tree of order 5

Several variations of the basic B-tree have been developed, two of them, that have become popular are:

B^+ Tree: It is a B-tree in which the leaf nodes are linked to each other from left to right to form a linked list of the keys in sequential order.

B^* Tree: It is a B-tree in which each internal node is at least two-third full rather than just half-full.

Graph

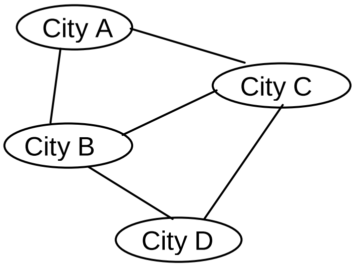


Fig.1: Graph

- Data sometimes contain a relationship between pairs of elements, which is not necessarily hierarchical in nature. The data structure that reflects this type of relationship is called a graph.
- **A graph is a non-linear data structure which is used to represent data containing relationships between pairs of elements that are not necessarily hierarchical in nature.**
- For example, a graph showing major highways connecting nearby cities (or showing air routes between important cities) is given in Fig.1. Note that there is no highway that directly connects city A to D.

Definition

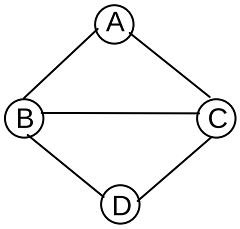


Fig.2

- **A graph G is defined as G=(V,E), where:**

  V: is a finite, non-empty set of vertices (or nodes) of G and
  E: is a set of edges connecting certain pairs of vertices. E may be empty.

- Fig.2 shows a sample graph, for which:

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

Edge: A **line drawn to connect one node to another node** is called an edge. It can be directed or undirected (see Fig.3).

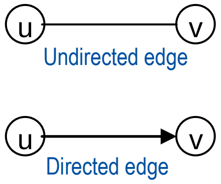


Fig.3

- An **undirected edge** has no specific direction associated with the edge. It is represented by an unordered pair of vertices e=(u,v) i.e. ordering of vertices is not significant here and the edges e=(u,v) and e=(v,u) represent the same edge and can be traversed from u to v or vice versa.
- A **directed edge** has a specific direction associated with the edge. It is represented by an **ordered pair of vertices e=<u,v>** i.e. it can be traversed only from u to v. Thus the edges e=<u,v> and e=<v,u> represent two different edges.

Undirected Graph: A graph in which **all the edges are undirected**. For example, the graph given in Fig.2 is an undirected graph. This graph is represented as G=(V,E), where:

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

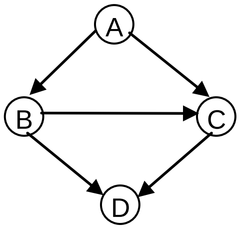


Fig.4

Directed Graph (or **Digraph)**: A graph in which **every edge is directed** from one node to another. For example, graph given in Fig.4 is a directed graph. This graph is represented as G=(V,E), where:

V={A,B,C,D}
E={ <A,B>, <A,C>, <B,C>, <B,D>, <C,D> }

<u>Mixed Graph</u>: A graph in which **some of the edges are directed and some are undirected**, For example, the graph given in Fig.5 is a mixed graph. A mixed graph can be used to represent 1-way and 2-way streets connecting some important places in a city.

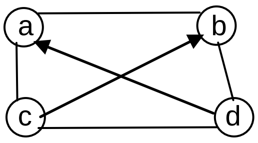


Fig.5: Mixed graph

Basic Graph Terminology

Adjacent Node

A node v is said to be adjacent of u, if there exists an edge from u to v. For example in the graph given in Fig.7, node B is adjacent of A and D, but not adjacent of C and E.

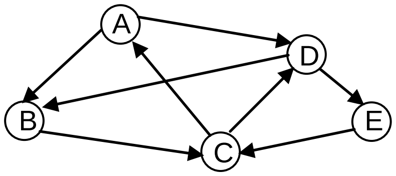


Fig.7

<u>Path</u>: It is a **sequence of consecutive edges** from a source node to destination node. For example, in Fig.7, ABCDE, ADEC are paths. First node in the path is called the *start node*; the last node in the path is called the *end node*. A path consisting of directed edges is called a *directed path*.

<u>Cycle</u>: It is a **closed simple path which begins and ends in the same node** . For example in Fig.7, BCDB and CADEC are cycles. In a digraph, it is called a directed cycle. If a graph has one/more cycles, then it is said to be *cyclic*; otherwise it is said to be *acyclic graph*.

<u>Simple Path</u>: A path is simple if **all the nodes in the path are distinct**, with the exception that first node and last node in the path are same if it is a cycle (called *closed simple path* or *simple cycle*). For example, paths ABCD, BCDE and ABCA in the Fig.7.

<u>Isolated Node</u>: A **node** which is **not connected to any other node by an edge.** For example in Fig.8, all nodes are isolated nodes.

<u>Null Graph</u>: A graph in which **E** (set of edges) **is empty** i.e. the graph contains only isolated nodes. For example the graph given in Fig.8 is a null graph.

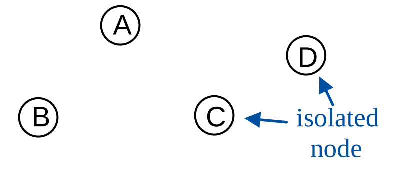
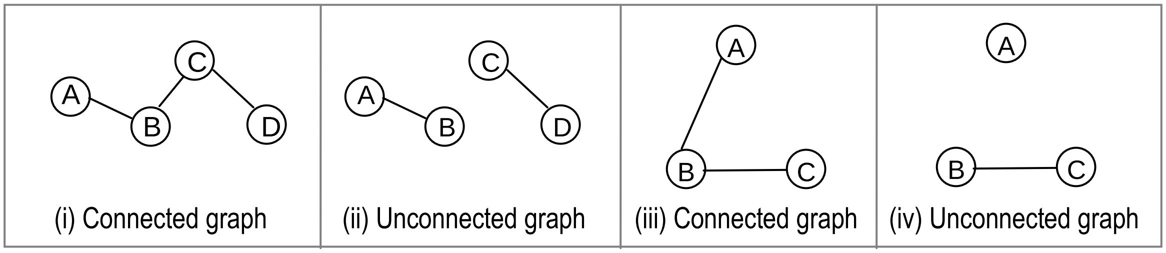


Fig.8: Null graph

<u>Connected Graph</u>: An undirected graph in which **there exists a path from every node to every other node** (or every node is reachable from every other node). For example, the graph given in Fig.9(a)(i) and (iii) are connected graphs.



(i) Connected graph (ii) Unconnected graph (iii) Connected graph (iv) Unconnected graph

Fig.9(a)

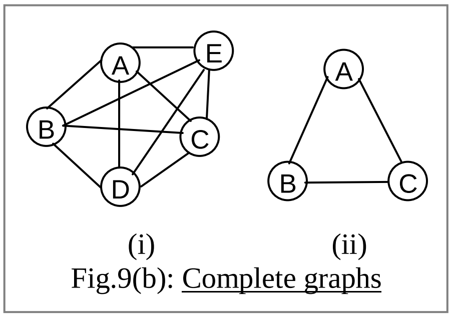


Fig.9(b): Complete graphs

Complete Graph: An undirected graph is said to be *complete* or *fully connected* if **there exists an edge between every pair of nodes**. For example graphs given in Fig.9(b) are complete graphs.
A complete undirected graph with n nodes has (nx(n-1))/2 edges. This is maximum number of edges in any n vertex undirected graph.

Strongly Connected Graph: A digraph in which **there exists a path in both directions between every pair of nodes**. For example, the graphs given in Fig.10 are strongly connected.

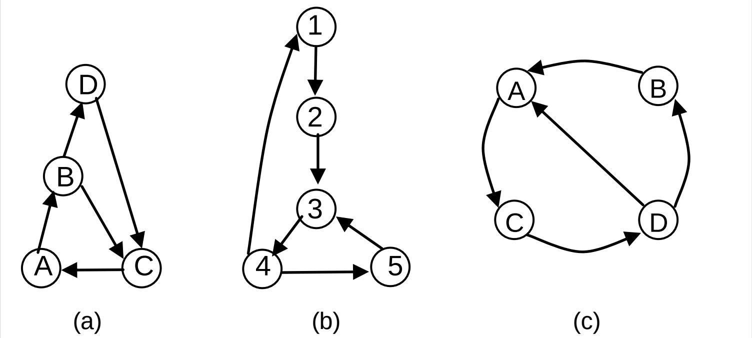


Fig.10: Strongly connected graphs

Unilaterally Connected Graph: A digraph in which **there exists a path only in one direction between every pair of nodes**. For example, the graphs given in Fig.11 are unilaterally connected.

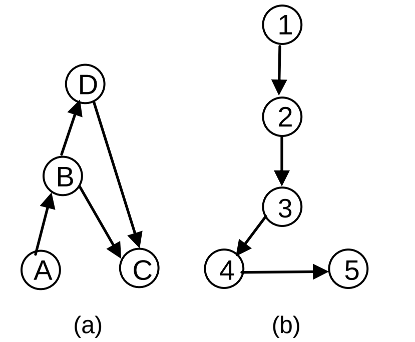


Fig.11: Unilaterally connected

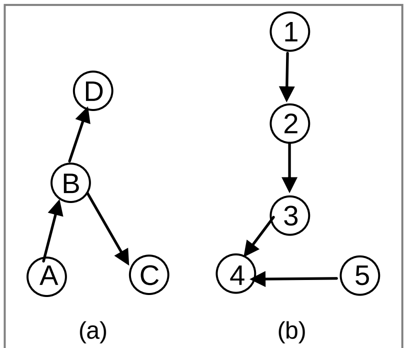


Fig.12: Weakly connected

Weakly Connected Graph: It can be thought of a digraph in which **there exists a path between every pair of nodes but not necessarily following the directions of the edges.** For example, the graphs given in Fig.12 are weakly connected.

Multigraph: A graph in which **there exists multiple occurrences of same edge** (also called parallel edges i.e. edges that have same end nodes). For example, graphs shown in Fig.13 are multigraphs. A graph is said to be *simple* if it doesn't have multiple edges between any two nodes.

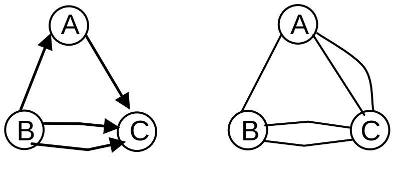


Fig.13: Multigraphs

Weighted Graph: A graph in which **some non-negative numerical value (called the weight) is assigned to every edge** in the graph is called a weighted (or labeled) graph.

The weight of an edge may represent:

- **distance** between the vertices or
- **traffic density** on city streets or
- **number of trains/flights** available between cities
- cost of moving along that edge etc.

For example, the weighted graph given in Fig.14 represents distance between some nearby cities.

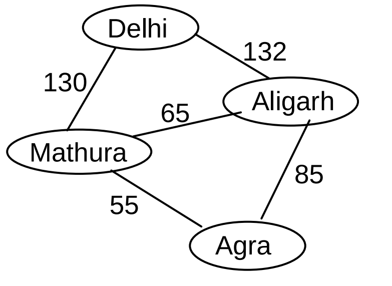


Fig.14 Weighted graph

Loop: An **edge that begins and ends in the same node**. For example, the edge <D,D> in Fig.15. Total degree of loop is 2 (because its indegree is 1 and outdegree is 1). The direction of loop is of no significance, hence it can be considered either a directed or undirected edge.

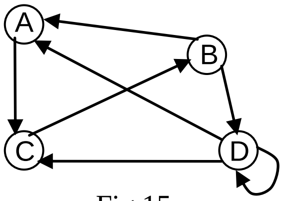


Fig.15

Length of Path

a). **in a weighted graph:**

It is equal to the **sum of the weights of the edges appearing in the path**. For example in Fig.14, length of the path Delhi-Mathura-Agra is 185.

b). **in an unweighted graph:**

It is equal to the **number of edges appearing in the path**. For example in Fig.15, length of the path ACBD is 3, ACBDD is 4.

Degree of a Node:

Degree (or total degree) of a node V is equal to the total number of edges incident at V i.e.

Degree(V)= total number of edges incident (occurring) **at V**.

For example, in the digraph given in Fig.15, Degree(A) = 3 and Degree(D) = 5. Note that the degree of an isolated node is 0.

Indegree and Outdegree of a Node in a Digraph

In a digraph, **indegree** of a node is the **number of edges ending** (terminating) at that node and **outdegree** of a node is the **number of edges beginning** (originating) from that node.

For example, in the digraph given in Fig.15:

Indegree(A)= 2, Outdegree(A)= 1
Indegree(D)= 2, Outdegree(D)= 3

Thus, degree of a node V in a digraph: **Degree(V) = Indegree(V) + Outdegree(V)**.

Source: In a digraph, a node whose **indegree is 0 and outdegree is positive**. For example, node B of the graph shown in Fig.16.

Sink: A node whose **outdegree is 0 and indegree is positive**. For example, node G of the graph shown in Fig.16.

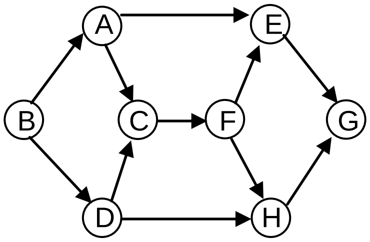


Fig.16

Tree Graph (or Free Tree)

It is an **undirected, unweighted, connected, acyclic** graph G in which:

- There is a unique simple path between any two nodes u and v
- No. of edges = No. of vertices - 1
- If any edge is removed, the resulting graph is disconnected
- If a new edge is added, the resulting graph contains a cycle (which is called a graph).

Examples of tree graph are given in Fig.17(a):

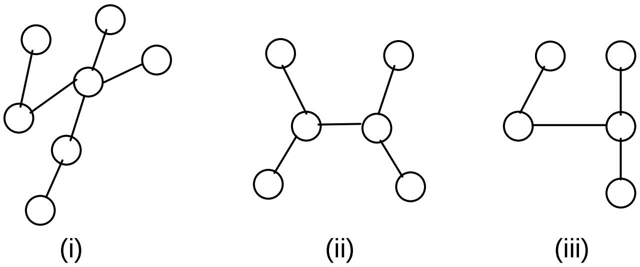


Fig.17(a): Tree graphs

Applications: Graphs are used to model real-life systems, such as:

- Airline flights (each node represents an airport and each edge a flight)
- Major highways between cities (each node represents a city and each edge represents a highway)
- Traffic density on different routes/streets (weight of each edge represents the traffic density)
- To show one-way and two-way streets connecting some important places in a city
- Network of computers (each node is a computer and each edge is a link)
- Used for computing distances and determining connectivity etc.

Representation of Graph in Memory

In order to solve problems which involve graphs using a computer, we must be able to represent them in memory. Graphs can be represented in memory in two ways:

1. Sequential (or Adjacency Matrix) representation
2. Linked (or Adjacency List) representation

1. Sequential (or Adjacency Matrix) Representation of Graph

In this representation, a graph is represented in memory by means of its *adjacency matrix*.

Adjacency Matrix

- Let G=(V,E) be the graph with n nodes in which $V=\{V_0,V_1,V_2,\ldots V_{n-1}\}$ and the nodes are assumed to be ordered from V_0 to V_{n-1}.

- Then the adjacency matrix $A=(a_{ij})$ of G is an nxn matrix whose elements are defined as:

$$a_{ij} = \begin{cases} 1, & \text{if the edge } (V_i,V_j) \in E \text{ (i.e. if there exists an edge from node } V_i \text{ to } V_j \text{)} \\ 0, & \text{otherwise.} \end{cases}$$

- Thus the value of each element of the adjacency matrix is either 0 or 1. Such a matrix is called a bit or Boolean matrix.

Example: Consider the mixed graph G=(V,E) given in Fig.18 in which:

V={A,B,C,D}

E={<A,B>,<B,A>,(B,C),<C,A>,<C,D>,<D,B>,<D,D>}.

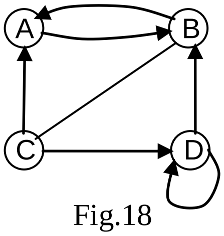


Fig.18

- To represent this mixed graph in memory using sequential representation, first we need to determine its adjacency matrix as below:

$$\begin{array}{c|cccc} i \diagdown j & A & B & C & D \\ \hline A & 0 & 1 & 0 & 0 \\ B & 1 & 0 & 1 & 0 \\ C & 1 & 1 & 0 & 1 \\ D & 0 & 1 & 0 & 1 \end{array}$$

- Now the graph can be represented in memory by means of two arrays - a 1D array, say V[n] and a 2D array, say A[n][n], where n represents number of vertices in the graph. Array V[n] represents the set of vertices and A[n][n] represents the set of edges (i.e. adjacency matrix).

- Thus, the adjacency matrix representation of this graph will be as below:

	V[n]	A[n][n]			
0	A	0	1	0	0
1	B	1	0	1	0
2	C	1	1	0	1
3	D	0	1	0	1

→ outdegree of V_i=number of 1^s in the i^{th} row

→ indegree of node V_j=number of 1^s in the j^{th} column

Sequential Representation of a Weighted Graph

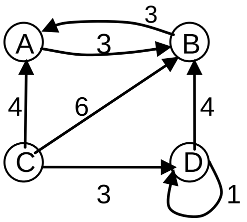


Fig.19

- A weighted graph is represented in memory by means of a **weight matrix** W of size nxn whose elements W=(w_{ij}) are defined as:

$$w_{ij} = \begin{cases} \text{Weight of the edge } (V_i,V_j), & \text{if the edge } (V_i,V_j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Example: Suppose we want to represent weighted graph given in Fig.19. The weight matrix of this graph is:

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 3 & 0 & 0 \\ B & 3 & 0 & 0 & 0 \\ C & 4 & 6 & 0 & 3 \\ D & 0 & 4 & 0 & 1 \end{array}$$

- Thus the sequential representation of this weighted graph in memory will be as below:

V[n]

A
B
C
D

W[n][n]

0	3	0	0
3	0	0	0
4	6	0	3
0	4	0	1

An adjacency (or **weight) matrix of a graph completely defines that graph** and the operations of matrix algebra can be performed on it to obtain the paths, cycles and other characteristics of the graph.

Advantages of Sequential Representation

- This is the simplest way to represent a graph in memory.
- This method is preferable when the graph is reasonably small (i.e. graph has small number of nodes).

Limitations of Sequential Representation

1. This is **static representation.** Due to the use of arrays, we **can't insert new nodes and their associated edges** in the graph as the size of the arrays is fixed and they can't grow or shrink once created in the memory.
2. Takes O(n^2) time to solve most of the graph problems.
3. Takes O(n^2) space to represent the graph (irrespective of the number of edges in the graph).

2. Linked (or Adjacency List) Representation of Graph

- To overcome the limitations of sequential representation of graph, graphs may be represented in memory using linked representation.
- In linked representation, a graph is represented in memory by means of adjacency list (linked list of neighbours) of each node of the graph. The adjacency list of a node is nothing but the listing of its adjacent nodes.
- Linked representation of graph consists of:
  1. A **Node list** - it is a linked list that contains information about all nodes.
  2. An **Edge list** - it is a collection of linked lists that represents adjacency list of each node in the graph i.e. for each node we have a linked list of all its adjacent nodes.

Example: Consider the digraph given in Fig.24 in which V={A,B,C,D,E} and
E={<A,B>,<A,C>,<B,C>,<B,D>,<B,E>,<C,A>,<C,D>,<E,D>}.

- To represent this graph in memory using linked representation, first we need to find the adjacency list of each node of the graph as below:

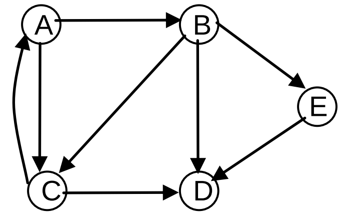
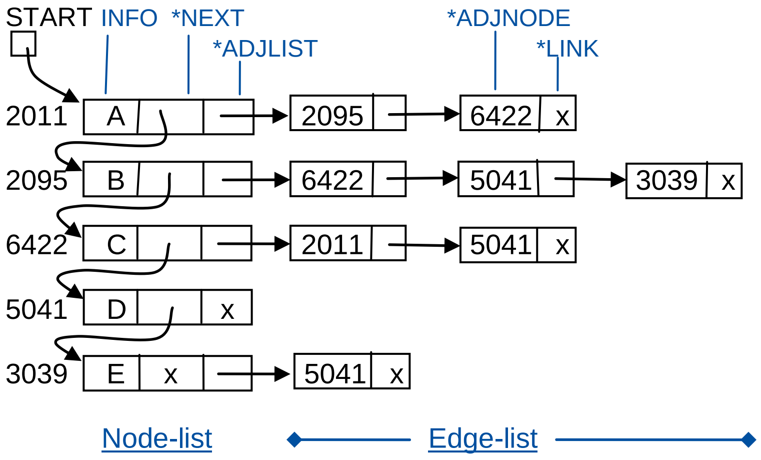


Fig.24

Node	Adjacency list
A	B, C
B	C, D, E
C	A, D
D	–
E	D

- Linked representation of this graph is given below:



- Node list has three parts. First part is the INFO part, second part is the NEXT pointer field which contains the address of next node in the list. Third part is the ADJLIST pointer which contains the address of the adjacency list of that node.
- In the edge list, addresses of adjacent nodes are stored instead of information to avoid data redundancy and wastage of storage.

Advantages: 1. A node or an edge can be added/removed easily.

2. Takes less memory if graph is sparse (large no. of nodes with few edges).

Node structures and a C++ *class* to represent unweighted graph in memory can be defined as below:

```
struct edge;                  //forward declaration
struct node
{   char INFO;
    node *NEXT;
    edge *ADJLIST;            //contains address of 1st adjacent node
};
struct edge
{   node *ADJNODE;            //address of adjacent node in the node list
    edge *LINK;               //contains address of the next adjacent node in the adjacency list
};
class graph
{   node *START;              //V(G) can't be empty.
  public:
     ….                       //member functions to perform various operations.
     ….
};
```

Prob: Give linked representation of the weighted graph given in Fig.25.

Solu: Adjacency list of each node of the weighted graph is:

Node	Adjacency list
A	B, C, E
B	D
C	B, D
D	C, E
E	–

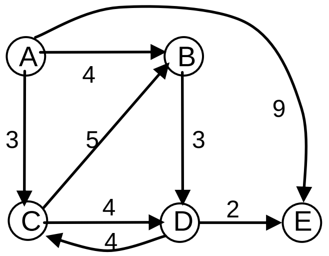
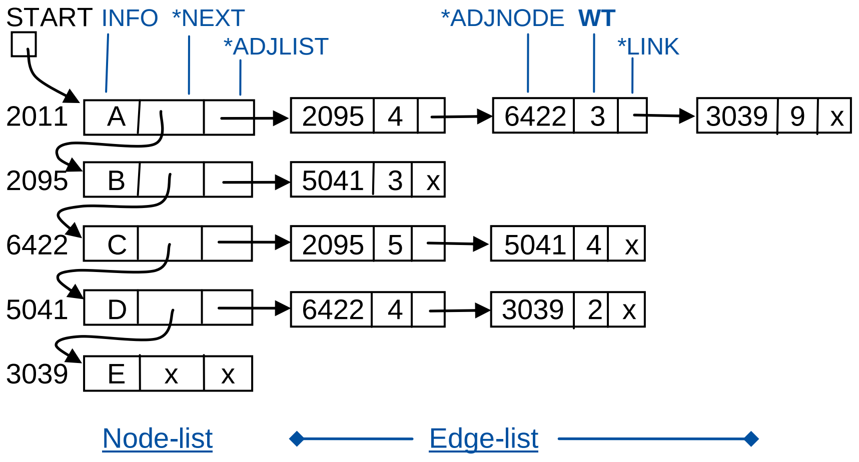


Fig.25

In linked representation, weight of each edge is stored in corresponding node in the edge list:



Graph Traversal

In many applications we need to systematically examine the nodes and edges of a graph. There are two standard ways that this is done. These are known as:

1. Breadth-First Search (uses a queue)
2. Depth-First Search (uses a stack)

Graph traversal is also known as graph search. The breadth-first search will use a queue as an auxiliary structure to hold nodes for future processing, whereas the depth-first search will use a stack.

1. Breadth-First Search

- In breadth-first search, first we examine the given starting node, say s. Then we examine all the neighbors of s. Then we examine all the neighbors of neighbors of s, and so on. This process continues until all the nodes reachable from the starting node are processed.
- Naturally we need to keep track of the neighbors of a node and we need to ensure that no node is processed more than once. This is accomplished by using a *queue* to hold nodes that are waiting to be processed, and by using a field *status* which gives the current status of any node. During the traversal operation, each node will be in one of the three states, called the status, indicating the visiting status of the node as follows:

  status=**1**: (Ready state) Initial state of a node.
  status=**2**: (Waiting state) Node inserted in the queue and waiting for processing.
  status=**3**: (Processed state) Node has been processed.

Following algorithm executes a breadth-first search on a graph G beginning at a starting node s.

Algorithm bfs(G,s)

```
initialize all nodes to the ready state (i.e. status=1).
insert the starting node s into the queue and change its status to waiting state (status=2).
while (queue is not empty) do
    delete the front node, say x, from the queue, process it and change its status to processed
    state (i.e. status=3).
    insert into the queue all the neighbors of x that are in ready state (status=1) and change
    their status to waiting state (i.e. status=2).
end.
```

Example: Find breadth-first traversal of the graph given in Fig.26 from starting node A.

Solu: From the graph, it is clear that the adjacent nodes (neighbors) of each node in the graph are:

Node	Adjacency list
A	B, C, D
B	F
C	E, F
D	E, G
E	I
F	E, H
G	I
H	I
I	–

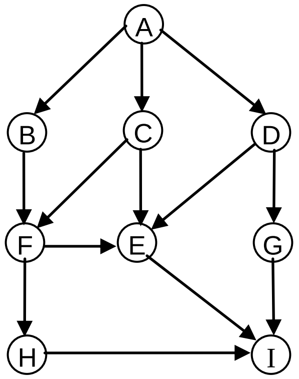
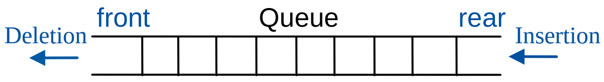


Fig.26

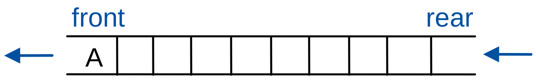
The steps of processing each node using BFS are as follows:

(i). Initialize all nodes to the ready state:



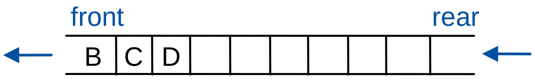
Ready state: A B C D E F G H I

(ii). Insert starting node A into the queue and change its status to waiting state:



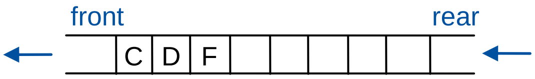
Ready state: B C D E F G H I
Waiting state: A

(iii). Delete the front node **A**, process it and change its status to processed state. Then insert into queue all the neighbors of A that are in ready state and change their status to waiting state:



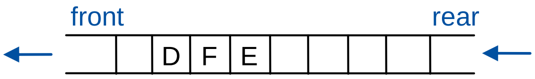
Ready state: E F G H I
Waiting state: B C D
Processed state: A

(iv). Delete the front node **B**, process it and change its status to processed state. Then insert into queue all the neighbors of B that are in ready state and change their status to waiting state:



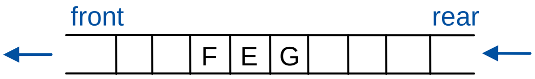
Ready state: E G H I
Waiting state: C D F
Processed state: A B

(v). Delete the front node **C**, process it and change its status to processed state. Then insert into queue all the neighbors of C that are in ready state and change their status to waiting state:



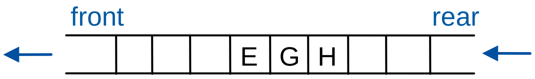
Ready state: G H I
Waiting state: D F E
Processed state: A B C

(vi). Delete the front node **D**, process it and change its status to processed state. Then insert into queue all the neighbors of D that are in ready state and change their status to waiting state:



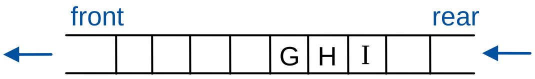
Ready state: H I
Waiting state: F E G
Processed state: A B C D

(vii). Delete the front node **F**, process it and change its status to processed state. Then insert into queue all the neighbors of F that are in ready state and change their status to waiting state:



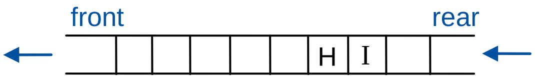
Ready state: I
Waiting state: E G H
Processed state: A B C D F

(viii). Delete the front node **E**, process it and change its status to processed state. Then insert into queue all the neighbors of E that are in ready state and change their status to waiting state:



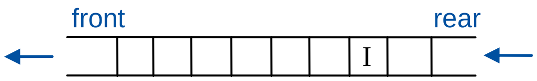
Ready state:
Waiting state: G H I
Processed state: A B C D F E

(ix). Delete the front node **G**, process it and change its status to processed state. Then insert into queue all the neighbors of G that are in ready state and change their status to waiting state:



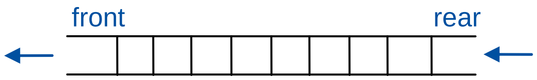
Ready state:
Waiting state: H I
Processed state: A B C D F E G

(x). Delete the front node **H**, process it and change its status to processed state. Then insert into queue all the neighbors of H that are in ready state and change their status to waiting state:



Ready state:
Waiting state: I
Processed state: A B C D F E G H

(xi). Delete the front node **I**, process it and change its status to processed state. Then insert into queue all the neighbors of I that are in ready state and change their status to waiting state:



Queue: Empty

Ready state:
Waiting state:
Processed state: **A B C D F E G H I**

The queue is now empty, so the breadth-first search of graph G starting at node A is now complete. The breadth-first traversal order of nodes is **A B C D F E G H I**.

<u>Note</u>: Above BFS algorithm will process only those nodes which are reachable from the starting node. To examine all the nodes of G that are reachable as well as not reachable from starting node, the algorithm must be modified so that it begins again with another node, say x, that is still in the ready state and so on until all the nodes are traversed.

<u>Exercise</u>: Find breadth-first traversal of the graph given in Fig.26(b) from starting node A.

<u>Solu</u>: Adjacent nodes (neighbors) of each node in the graph are:

Node	Adjacency list
A	B, D
B	C, E
C	A, F
D	C, G
E	F, H
F	H
G	F, H
H	-

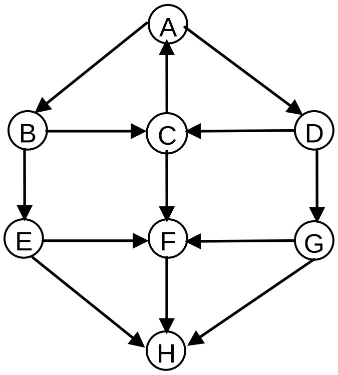
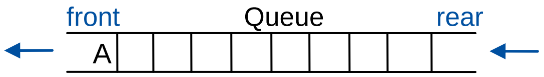


Fig.26(b)

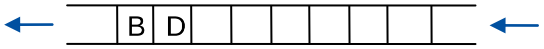
The steps of processing each node using BFS are as follows:

(i). Initialize all nodes to ready state and insert starting node A into the empty queue and change its status to waiting state.



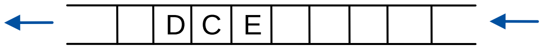
Ready state: B C D E F G H
Waiting state: A
Processed state:

(ii). Delete front node **A**, process it and change …. Then insert all the neighbors of A ….



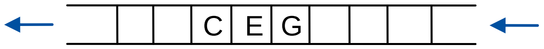
Ready state: C E F G H
Waiting state: B D
Processed state: A

(iii). Delete front node **B**, process it and change …. Then insert all the neighbors of B ….



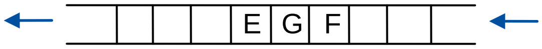
Ready state: F G H
Waiting state: D C E
Processed state: A B

(iv). Delete front node **D**, process it and change …. Then insert all the neighbors of D ….



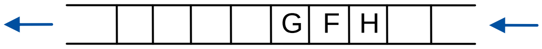
Ready state: F H
Waiting state: C E G
Processed state: A B D

(v). Delete front node **C**, process it and change …. Then insert all the neighbors of C ….



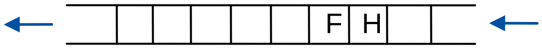
Ready state: H
Waiting state: E G F
Processed state: A B D C

(vi). Delete front node **E**, process it and change …. Then insert all the neighbors of E ….



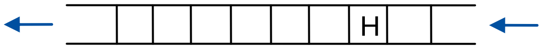
Ready state:
Waiting state: G F H
Processed state: A B D C E

(vii). Delete front node **G**, process it and change …. Then insert all the neighbors of G ….



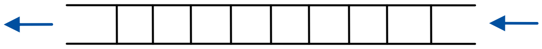
Ready state:
Waiting state: F H
Processed state: A B D C E G

(viii). Delete front node **F**, process it and change …. Then insert all the neighbors of F ….



Ready state:
Waiting state: H
Processed state: A B D C E G F

(ix). Delete front node **H**, process it and change …. Then insert all the neighbors of H ….



Queue: Empty.

Ready state:
Waiting state:
Processed state: **A B D C E G F H**

Thus breadth-first traversal order of nodes is: **A B D C E G F H**.

2. Depth-First Search

- In depth-first search, first we examine the given starting node, say s. Then we examine each node along a path P which begins at s; i.e. we process a unvisited neighbor of s (if there exists many unvisited neighbors then any one of them can be selected for visiting next), then an unvisited neighbor of neighbor of s, and so on until we reach to a "dead end" i.e. a node whose outdegree is 0 or whose all the neighbors have already been visited. Then we backtrack on P until we can continue along another path P' and so on. This process continues until all the nodes reachable from the starting node are processed.
- Naturally we need to keep track of the neighbors of a node and we need to ensure that no node is processed more than once. This is accomplished by using a *stack* to hold nodes that are waiting to be processed, and by using a field *status* which gives the current status of any node. During the traversal operation, each node will be in one of the three states, called the status, indicating the visiting status of the node as follows:

status=**1**:	(Ready state)	Initial state of a node.
status=**2**:	(Waiting state)	Node inserted in the stack and waiting for processing.
status=**3**:	(Processed state)	Node has been processed.

Following algorithm executes a depth-first search on a graph G beginning at a starting node s.

Algorithm dfs(G,s)

```
1.  initialize all nodes to the ready state (i.e. status=1).
2.  push the starting node s onto the stack and change its status to waiting state (status=2).
3.  while (stack is not empty) do
4.        pop out top element, say x, from the stack, process it and change its status to processed
          state (i.e. status=3).
5.        push all the neighbors of x that are in ready state (status=1) onto the stack and change their
          status to waiting state (i.e. status=2).
6.  end.
```

Example: Find depth-first traversal of the graph given in Fig.27 from starting node A.

Solu: From the graph, it is clear that the adjacent nodes (neighbors) of each node in the graph are:

Node	Neighbors
A	B, C, D
B	F
C	E, F
D	E, G
E	I
F	E, H
G	I
H	I
I	--

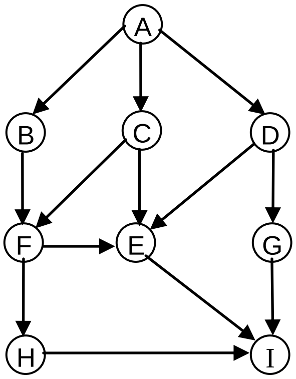


Fig.27

The steps of processing each node using DFS are as follows:

(i). Initialize all nodes to the ready state:

Stack

Ready state: A B C D E F G H I

(ii). Push starting node A into the stack and change its status to waiting state:

A									

Ready state: B C D E F G H I
Waiting state: A

(iii). Pop top node **A**, process it and change its status to processed state. Then push into stack all the neighbors of A that are in ready state and change their status to waiting state:

B	C	D							

Ready state: E F G H I
Waiting state: B C D
Processed state: A

(iv). Pop top node **D**, process it and change its status to processed state. Then push into stack all the neighbors of D that are in ready state and change their status to waiting state:

B	C	E	G						

Ready state: F H I
Waiting state: B C E G
Processed state: A D

(v). Pop top node **G**, process it and change its status to processed state. Then push into stack all the neighbors of G that are in ready state and change their status to waiting state:

B	C	E	I						

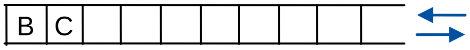
Ready state: F H
Waiting state: B C E I
Processed state: A D G

(vi). Pop top node **I**, process it and change its status to processed state. Then push into stack all the neighbors of I that are in ready state and change their status to waiting state:

B	C	E							

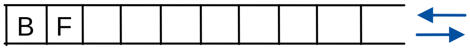
Ready state: F H
Waiting state: B C E
Processed state: A D G I

(vii). Pop top node **E**, process it and change its status to processed state. Then push into stack all the neighbors of E that are in ready state and change their status to waiting state:



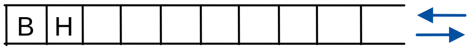
Ready state: F H
Waiting state: B C
Processed state: A D G I E

(viii). Pop top node **C**, process it and change its status to processed state. Then push into stack all the neighbors of C that are in ready state and change their status to waiting state:



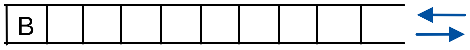
Ready state: H
Waiting state: B F
Processed state: A D G I E C

(ix). Pop top node **F**, process it and change its status to processed state. Then push into stack all the neighbors of F that are in ready state and change their status to waiting state:



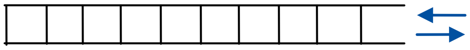
Ready state:
Waiting state: B H
Processed state: A D G I E C F

(x). Pop top node **H**, process it and change its status to processed state. Then push into stack all the neighbors of H that are in ready state and change their status to waiting state:



Ready state:
Waiting state: B
Processed state: A D G I E C F H

(xi). Pop top node **B**, process it and change its status to processed state. Then push into stack all the neighbors of B that are in ready state and change their status to waiting state:



Stack: Empty.

Ready state:
Waiting state:
Processed state: **A D G I E C F H B**

The stack is now empty, so the depth-first search of graph G starting at node A is now complete. The depth-first traversal order of nodes is **A D G I E C F H B**.

Note: Above DFS algorithm will process only those nodes which are reachable from the starting node. To examine all the nodes of G that are reachable as well as not reachable from starting node, the algorithm must be modified so that it begins again with another node, say x, that is still in the ready state and so on until all the nodes are traversed.

Time complexity of BFS and DFS is **O(V+E)**, where V and E denote the number of vertices and number of edges in the graph respectively.

Exercise: Find depth-first traversal of the graph given in Fig.28 from starting node A.

Solu: Adjacency list of each node of the graph:

Node	Neighbors
A	B, D
B	C, E
C	A, F
D	C, G
E	F, H
F	H
G	F, H
H	--

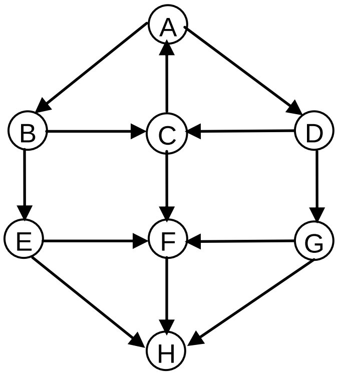


Fig.28

The steps of processing each node using DFS are as follows:

(i). Push starting node A into the empty stack and change its status to waiting state.

Stack

A									

⇆

Ready state: B C D E F G H
Waiting state: A
Processed state:

(ii). Pop top node **A**, process it and change …. Then push all the neighbors of A ….

B	D								

⇆

Ready state: C E F G H
Waiting state: B D
Processed state: A

(iii). Pop top node **D**, process it and change …. Then push all the neighbors of D ….

B	C	G							

⇆

Ready state: E F H
Waiting state: B C G
Processed state: A D

(iv). Pop top node **G**, process it and change …. Then push all the neighbors of G ….

B	C	F	H						

⇆

Ready state: E
Waiting state: B C F H
Processed state: A D G

(v). Pop top node **H**, process it and change …. Then push all the neighbors of H ….

B	C	F							

⇆

Ready state: E
Waiting state: B C F
Processed state: A D G H

(vi). Pop top node **F**, process it and change …. Then push all the neighbors of F ….

B	C								

⇆

Ready state: E
Waiting state: B C
Processed state: A D G H F

(vii). Pop top node **C**, process it and change …. Then push all the neighbors of C ….

B									

⇆

Ready state: E
Waiting state: B
Processed state: A D G H F C

(viii). Pop top node **B**, process it and change …. Then push all the neighbors of B ….

E									

⇆

Ready state:
Waiting state: E
Processed state: A D G H F C B

(ix).Pop top node **E**, process it and change …. Then push all the neighbors of E ….

⇆

Stack empty.

Ready state:
Waiting state:
Processed state: **A D G H F C B E**

Thus depth-first traversal order of nodes is: **A D G H F C B E.**

Applications of Graphs

Some important applications of graphs are discussed below:

Finding Shortest Paths

- In many applications that deal with weighted graphs (directed or undirected), we are often required to find shortest path (path having the minimum weight/length) between a source node and a destination node. The shortest paths are not necessarily unique.

- For example, suppose we have with us an airline map which gives the cities having direct flights and total time of flight. We may be interested to find that which route we should take so that we can reach the destination as quickly as possible.

- There can be following two types of problems in finding shortest paths:

  1. **Single-Source Shortest Paths**

     In order to find shortest paths from a source node to all other nodes in the graph, we will discuss **Dijkstra's algorithm**.

  2. **Shortest Paths Among All-Pair of Vertices**

     In order to find shortest paths from every node to every other node, **Floyd-Warshall's algorithm** is used.

Dijkstra’s Single Source Shortest Paths Algorithm

- Dijkstra’s algorithm is used to find shortest paths from a source node to all other nodes in a weighted graph in which all the weights are non-negative.
- Dijkstra’s algorithm finds the shortest paths from a source node to all other nodes one-by-one. The essential feature of this algorithm is the order in which the paths are determined. The paths are discovered in the order of their weighted lengths, starting with the shortest proceeding to the longest.
- For each node, the algorithm keeps track of three pieces of information – *distance*, *predecessor node* and *status* in the following manner:
    - **Label**: Each node has a label which consists of *distance* and *predecessor node*, where:
        - » **Distance**: represents the path length (distance) of that node from the source node.
        - » **Predecessor**: represents the *node which precedes* the node in the shortest path from source. It is the node from which the probe was made to compute this label. It helps in constructing the shortest path from a source node to a destination node later.
    - **Status**: represents the status of each label. The status of a label can be permanent or tentative. Initially the status of each label is *tentative* (it indicates the distance of that node from source node along the best known path). When it is discovered that a tentative label represents the shortest possible path from the source node to that node, it is made *permanent* and never changed thereafter.

Algorithm dijkstra(G,s)

```
label the source node s with a value $(0,s)$ and all other nodes with $(\infty,-)$.
▷∞ means a distance value which is larger than any possible path length of that node from source node
▷and ‘–‘ means that the predecessor is undefined
mark the label of the source node as permanent and that of all other nodes as tentative.
while (all nodes are not marked as permanent) do
    working node u ← new permanent node (i.e. the node whose label is most recently made permanent).
    compute new tentative labels of all the tentatively labeled adjacent nodes of the working node
    u in the following manner:
        $\mathbf{TL(x)} \overset{\text{new distance of adjacent node x}}{} = \mathbf{MIN(}\overset{\text{old distance of adjacent node x}}{\mathbf{TL(x)}}, \overset{\text{distance of working node u}}{\mathbf{PL(u)}} + \overset{\text{weight of edge (u,x)}}{\mathbf{W(u,x)}}\mathbf{)}$
    examine all the tentative labels in the whole graph and mark the smallest label as permanent
    (if more than one node have the same minimum label, then any one can be made permanent).
end.
```

Time complexity of Dijkstra’s shortest path algorithm is **O(E log V)**, where V and E denote the number of vertices and number of edges in the graph respectively.

Example: Find shortest paths from source node A to all other nodes in the weighted graph in Fig.29 using Dijkstra's algorithm.

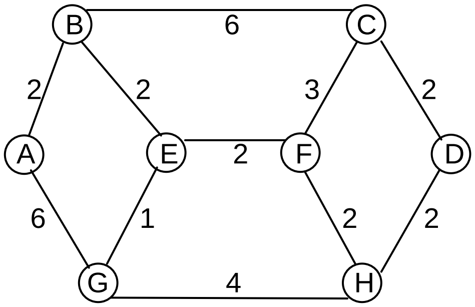


Fig.29

Solu:

1. Label the source node A with (0,A) and all other nodes with (∞,-):

2. Mark the label of the source node i.e. A as permanent and that of all other nodes as tentative:

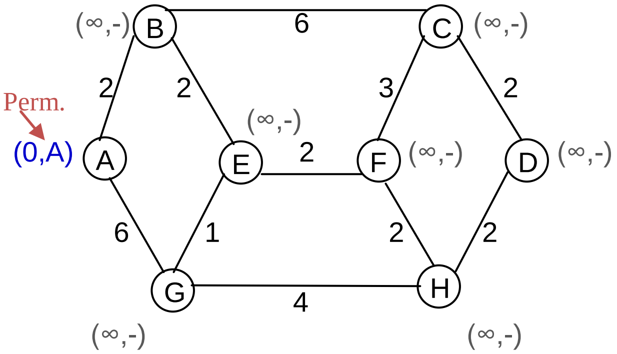


Fig.29(a)

3. Repeat steps 4 to 6 until all the labels are marked as permanent.

4. Working node ← A

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of the working node A:

(i) TL(B)=MIN(TL(B), PL(A)+W(A,B))

=MIN(∞ , 0+2) = 2 i.e. (2,A)

(ii) TL(G)=MIN(TL(G),PL(A)+W(A,G))

= MIN(∞ , 0+6) = 6 i.e. (6,A)

6. Examine all the tentative labels in the whole graph and mark the smallest label i.e. (2,A) as permanent.

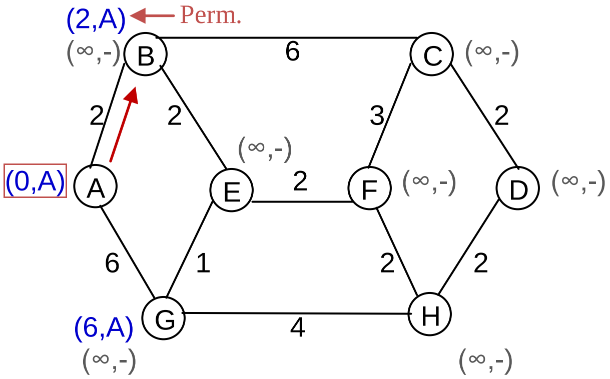


Fig.29(b)

4. Working node ← B

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of B:

(i) TL(C)=MIN(∞, 2+6) = 8 i.e. (8,B)

(ii) TL(E)=MIN(∞, 2+2) = 4 i.e. (4,B)

6. Mark the smallest tentative label i.e. (4,B) as permanent.

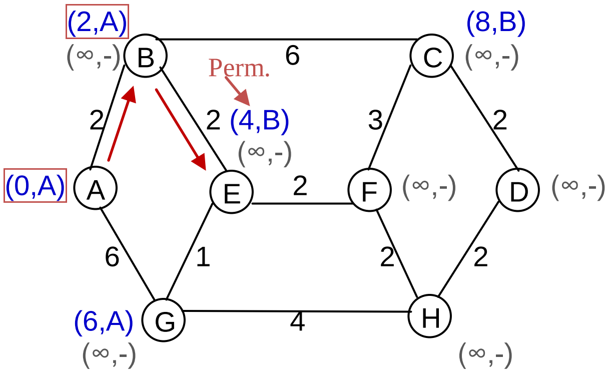


Fig.29(c)

4. Working node ← E

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of E:

   (i) TL(F)=MIN(∞, 4+2) = 6 i.e. (6,E)

   (ii) TL(G)=MIN(6, 4+1) = 5 i.e. (5,E)

6. Mark the smallest tentative label i.e. (5,E) as permanent.

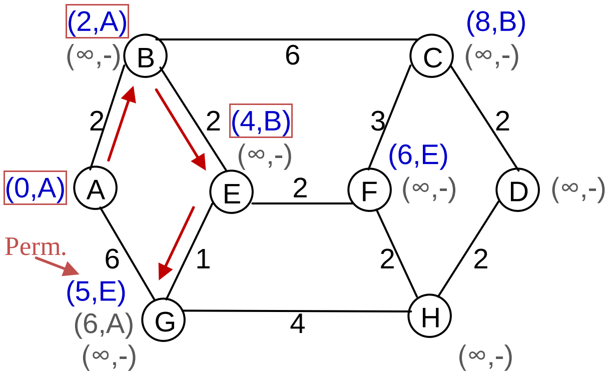


Fig.29(d)

4. Working node ← G

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of G:

   TL(H)=MIN(∞, 5+4) = 9 i.e. (9,G)

6. Mark the smallest tentative label i.e. (6,E) as permanent.

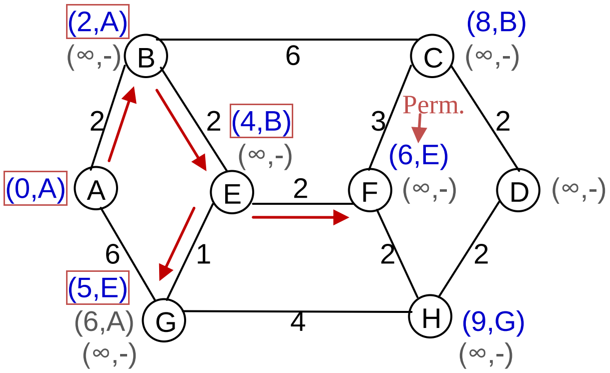


Fig.29(e)

4. Working node ← F

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of F:

   (i) TL(C)=MIN(8, 6+3) = 8 i.e. (8,F)

   Hence label of node C need bot be modified.

   (ii) TL(H)=MIN(9, 6+2) = 8 i.e. (8,F)

6. Mark the smallest tentative label i.e. (8,B) as permanent.

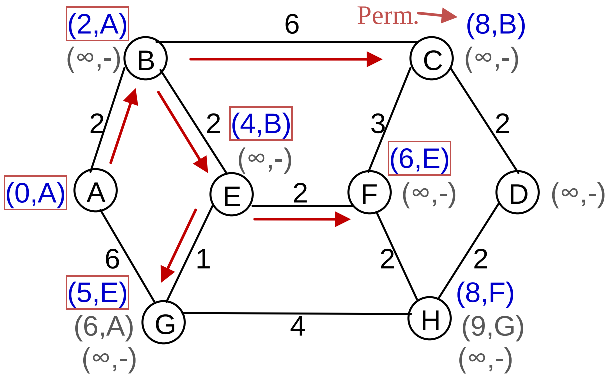


Fig.29(f)

4. Working node ← C

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of C:

   TL(D)=MIN(∞, 8+2) = 10 i.e. (10,C)

6. Mark the smallest tentative label i.e. (8,F) as permanent.

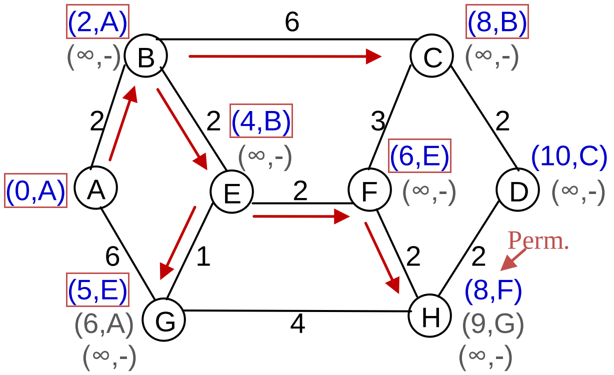


Fig.29(g)

4. Working node ← H

5. Calculate new tentative labels of all the tentatively labeled adjacent nodes of H:

   TL(D)=MIN(10 , 8+2) = 10 i.e. (10,H)

   Hence label of node D need not be modified.

6. Mark the smallest tentative label i.e. (10,C) as permanent.

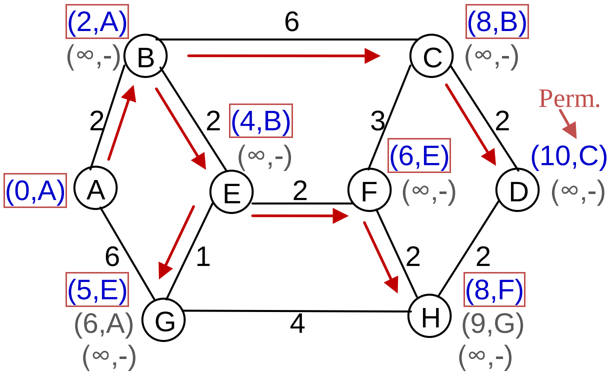


Fig.29(h)

Now all the labels have been marked as permanent, hence execution of the algorithm comes to an end.

After performing all the steps of the algorithm, the contents of labels of nodes are as follows:

	A	B	C	D	E	F	G	H
Distance	0	2	8	10	4	6	5	8
Predecessor	A	A	B	C	B	E	E	F

Shortest paths:

	Length	Path	
From A to B	2	A B	(because B's predecessor is A).
From A to C	8	A B C	(because C's predecessor is B and B's predecessor is A).
From A to D	10	A B C D	(because D's predecessor is C, C's predecessor is B and B's predecessor is A).
From A to E	4	A B E	
From A to F	6	A B E F	
From A to G	5	A B E G	
From A to H	8	A B E F H	

Prob-1: Find shortest paths from source node A to all other nodes in the weighted graph given in Fig.30 using Dijkstra's algorithm.

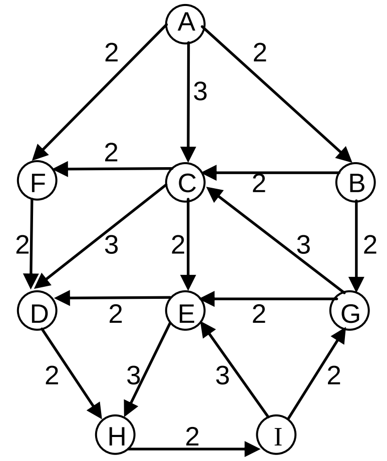


Fig.30

Solu:

Node	Adj.List
A	B, C, F
B	C, G
C	D, E, F
D	H
E	D, H
F	D
G	C, E
H	I
I	E, G

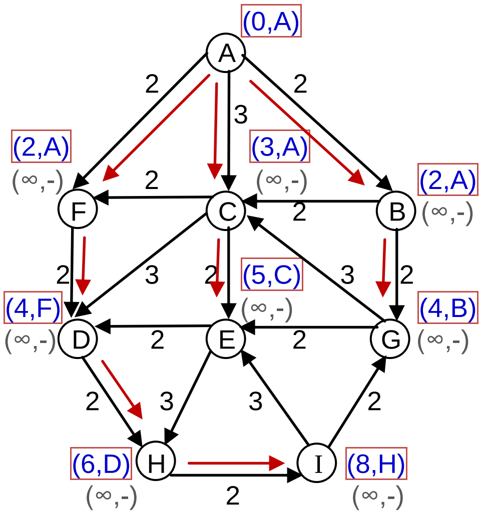


Fig.30(a)

After performing all the steps of the algorithm, the contents of labels of nodes are as follows:

	A	B	C	D	E	F	G	H	I
Distance	0	2	3	4	5	2	4	6	8
Predecessor	A	A	A	F	C	A	B	D	H

The order in which labels of the nodes are made permanent: A B F C D G E H I

Shortest paths:

	Length	Path
From A to B	2	A B
From A to C	3	A C
From A to D	4	A F D
From A to E	5	A C E
From A to F	2	A F
From A to G	4	A B G
From A to H	6	A F D H
From A to I	8	A F D H I

Prob-2: Find shortest paths from source node A to all other nodes in the weighted graph given in Fig.31 using Dijkstra's algorithm.

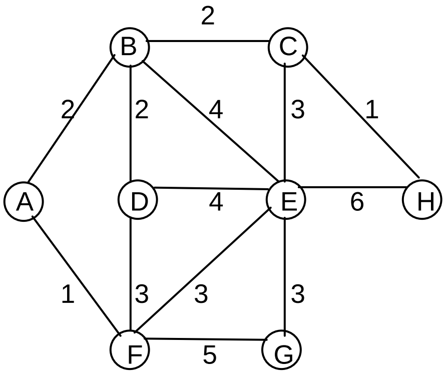


Fig.31

Solu:

Node	Adj.List
A	B, F
B	A, C, D, E
C	B, E, H
D	B, E, F
E	B, C, D, F, G, H
F	A, D, E, G
G	E, F
H	C, E

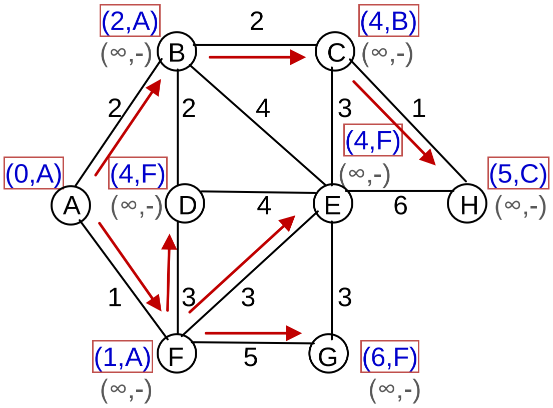


Fig.31(a)

After performing all the steps of the algorithm, the contents of labels of nodes are as follows:

	A	B	C	D	E	F	G	H
Distance	0	2	4	4	4	1	6	5
Predecessor	A	A	B	F	F	A	F	C

The order in which labels of the nodes are made permanent: A F B C D E H G

Shortest paths:

	Length	Path
From A to B	2	A B
From A to C	4	A B C
From A to D	4	A F D
From A to E	4	A F E
From A to F	1	A F
From A to G	6	A F G
From A to H	5	A B C H

Spanning Tree of Graph

A spanning tree of a connected, undirected graph G=(V,E) with n nodes, is a subgraph of G such that:

1. **It is connected and has no cycles**.
2. **The set of nodes is same as that of G**.
3. **The set of edges is a subset of E having n-1 edges that forms a free tree.**

Thus, spanning trees are free trees. There can be many spanning trees of a graph which contain different sets of n-1 edges.

Example1:

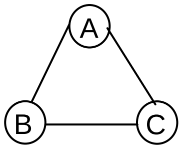


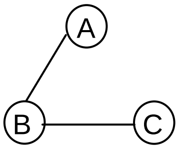
Fig.33

Consider the undirected graph G given in Fig.33 in which:

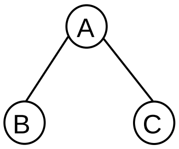
V = {A,B,C}

E = {(A,B),(A,C),(B,C)}

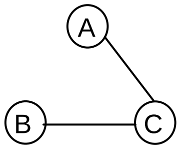
The spanning trees of this graph are:



Spanning tree-1



Spanning tree-2



Spanning tree-3

Example2:

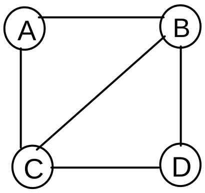


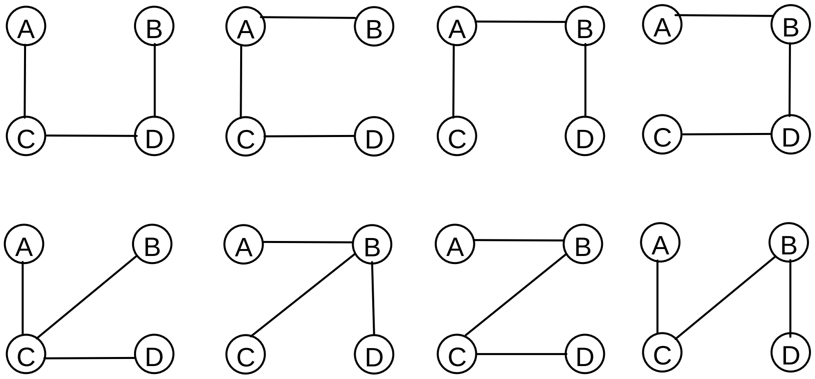
Fig.34

Consider the undirected graph G given in Fig.34 in which:

V = {A,B,C,D}

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

Its spanning trees are:



Minimum Spanning Tree (MST) of Weighted Graph

A spanning tree T of a connected, undirected, weighted graph G is said to be minimum spanning tree of G, if the sum of weights of all the edges in T is minimum.

Obviously there can be many spanning trees of a graph consisting of different sets of n-1 edges with different lengths. Among these that spanning tree, in which the sum of weights of all the edges is minimum, is called minimum spanning tree (MST) of weighted graph G.

Example:

- Consider the connected, undirected, weighted graph given in Fig.35.
- Its minimum spanning tree (MST) is given in Fig.36:

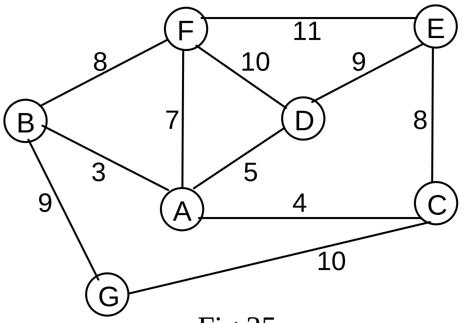


Fig.35

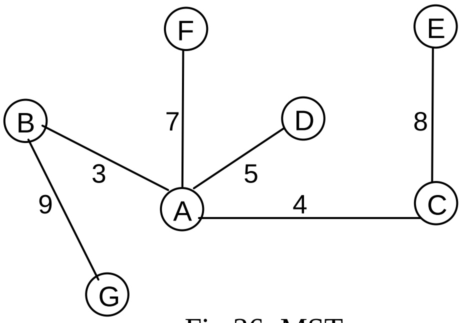


Fig.36: MST

Cost of MST = 9+3+7+5+4+8 = **36**

Applications of MST

1. To build a least cost electric lines/water supply lines/sewage lines/telephone lines etc. between cities (or localities) by connecting them using cables/wires/pipes etc.
2. To connect n pins/terminals by n-1 wires using least length of wire.
3. To build bridges to link a group of islands by spending least building cost of the Government (the bridges will enable the public to travel from one island to another).

Algorithms for Building Minimum Spanning Tree of Weighted Graph

Joseph Kruskal in 1956, and Robert C. Prim in 1957 gave algorithms for building a MST for a connected, undirected, weighted graph. These are known as:

1. **Kruskal's algorithm**
2. **Prim's algorithm**

Kruskal’s Algorithm for Building Minimum Spanning Tree of Weighted Graph

This algorithm finds the Minimum Spanning Tree T of a weighted graph G=(V,E) with n nodes using the concept of forest of trees.

Algorithm kruskal(V,E)

```
define a forest T=($V_T$,$E_T$) such that $V_T$ =V and $E_T$ ={ }.
sort the edges E in ascending order of their weights.
while (n-1 edges are not added to T) do
    select a least cost unprocessed edge (u,v) of G and add it to T (if addition of this edge
    forms a cycle in T then discard it).
▷subgraph T is the MST of the given graph
return T
```

Time complexity of Kruskal’s algorithm is **O(E log V))**, where V and E denote the number of vertices and number of edges in the graph respectively.

Prob: Find Minimum Spanning Tree (MST) of the weighted graph G=(V,E) given in Fig.37 using Kruskal's algorithm.

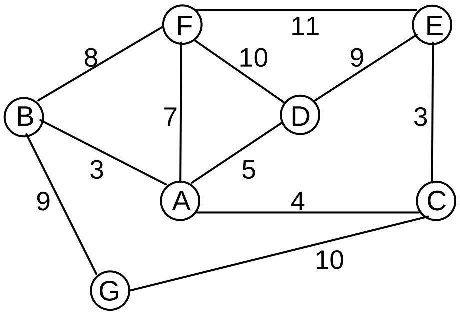
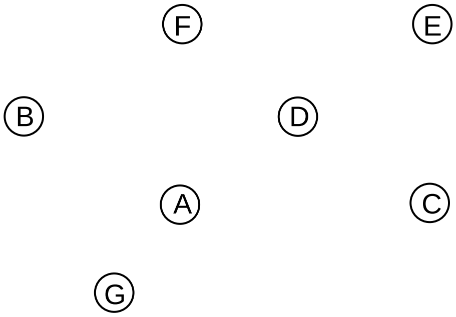


Fig.37

Solu: Above graph contains 7 vertices and 11 edges. So, its minimum spanning tree formed will be having (7 – 1) = 6 edges.

1. Define a forest T=(V_T,E_T) that contains all the nodes but no edges.

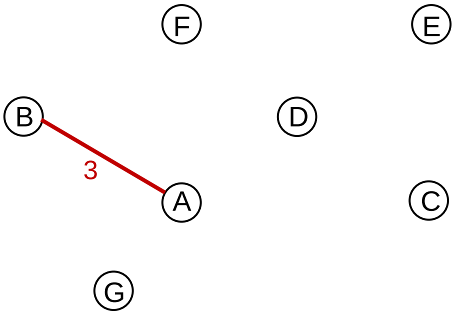


2. Sort the edges E in ascending order of their weights:

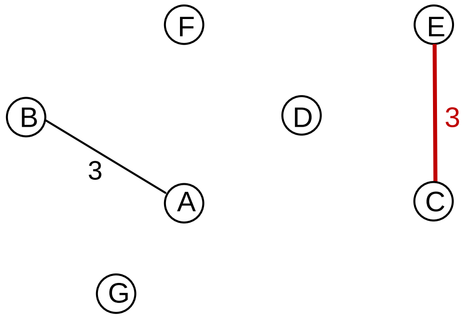
u	A	C	A	A	A	B	D	B	C	D	E
v	B	E	C	D	F	F	E	G	G	F	F
weight	3	3	4	5	7	8	9	9	10	10	11

3. Repeatedly select a least cost unprocessed edge and add it to T (reject it if its addition creates a cycle) until n-1 i.e. 6 edges are added to T.

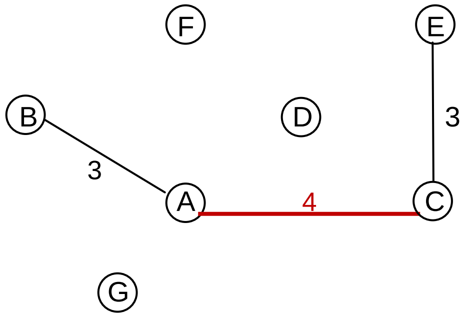
The least cost unprocessed edge is AB. Hence edge AB is added to T.



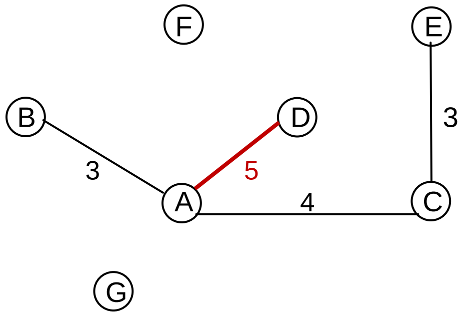
Next least cost unprocessed edge is CE. Hence edge CE is added to T.



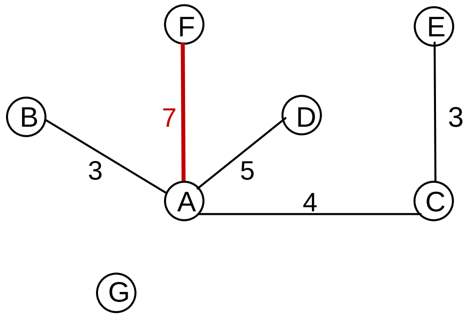
Next least cost unprocessed edge is AC. Hence edge AC is added to T.



Next least cost unprocessed edge is AD. Hence edge AD is added to T.



Next least cost unprocessed edge is AF. Hence edge AF is added to T.



Next least cost unprocessed edge is BF. But addition of BF creates a cycle, hence it is discarded.

Next least cost unprocessed edge is DE. But addition of DE creates a cycle, hence it is discarded.

Next least cost unprocessed edge is BG. Hence edge BG is added to T.

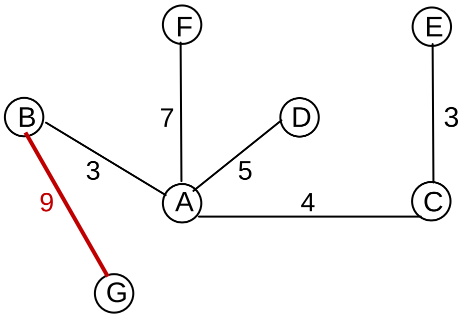


Fig.38: Minimum Spanning Tree

Now since n-1 edges have been added to T, hence we stop.

Thus, resultant subgraph T given in Fig.38 is the desired MST of the given graph G.

Cost of the minimum spanning tree (MST) = (9+3+7+5+4+8) = **31**

Exercise: **Find MST of the weighted graph G=(V,E) given in Fig.39 using Kruskal's algorithm.**

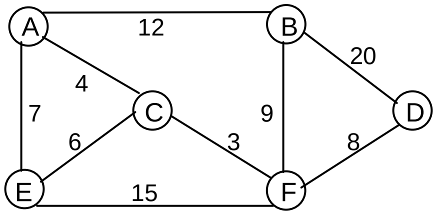


Fig.39

Ans:

The subgraph T given in Fig.40 is the desired minimum spanning tree of the given graph G.

Cost of the MST = (6+4+3+9+8) = **30**

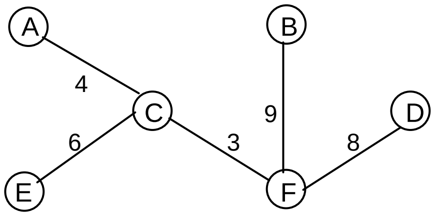


Fig.40: MST

****