Module - 4

Trees - 1

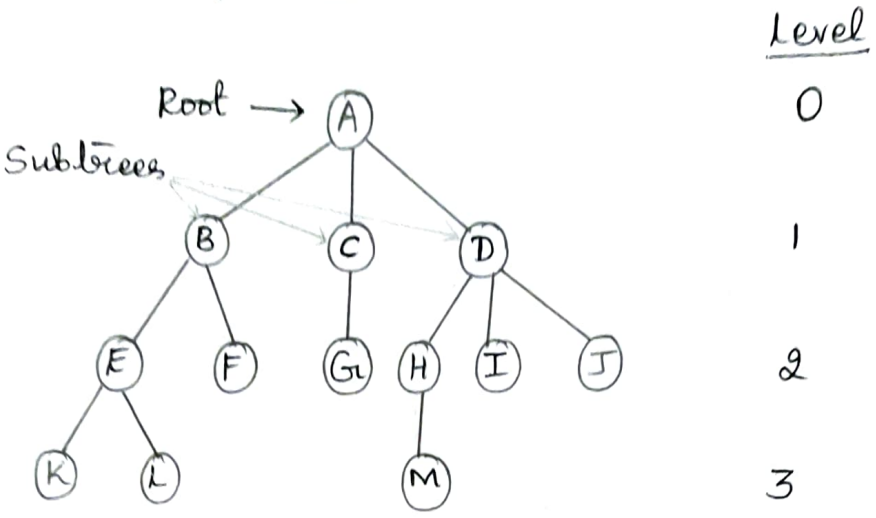
Syllabus :-

Terminologies, Binary Trees, Properties of Binary trees Array and linked representation of Binary trees, Binary Tree Traversals - Inorder, Postorder, Preorder Threaded binary trees, Binary Search Trees - Definition Insertion, Deletion, Traversal and Searching operation on Binary Search Tree. Application of Trees - Evaluation of Expression.

Definition :-

- Tree is a Data structure used to represent hierarchical relationship existing among several data items.
  Each data item is referred as node.
  Each node may be empty or may be connected to some other nodes.
- Stacks, queues and linked list are linear in nature
- In the field of CS, we come across many situations where the data are interrelated in hierarchical structure.
- In few situations, linear representation of data is not possible
- Solution for such problem is a data structure called trees which is a non-primitive and non-linear data structure.
- A tree is a finite set of one or more nodes such that there is a specially designated node called the root.
- Remaining nodes are partitioned into $n \geq 0$ disjoint set. These nodes are called subtrees of the root

Example of tree :-



- Tree consists of 'N' nodes with maximum 'N-1' edges and every individual element is called as 'Node'

Basic Terminologies of tree :-

1) Root :-
   - 1st node written at the top of the tree
   - Root node does not have parent
   - Every tree must have only 1 root node.

   Eg :- 'A' is the root node from the above example.

2) Edge :-
   - connecting link between any 2 nodes is called edge.
   - A tree has 'N' nodes & 'N-1' edges

   Eg :- In the above example, there 13 nodes & 12 edges.

3) Parent :-
   - A node which has branch from it to any other node (or)
   - A node having left subtree or right subtree or having both is said to be a parent node for left or right subtree.

Eg :- In the given example,
The parent of B, C, D is A
The parent of E & F is B
The parent of G is C and so on.

4) Child :-

- Node obtained from the parent node is called as child node.
- A parent node can have zero or more child

Eg :- In the given example,
B, C & D are the children of A
K & L are the children of E
M is a child of H

5) Siblings :-

- Two or more nodes having same parent is called as siblings.
- Nodes which belongs to same parent is also known as siblings.

Eg :- In the given example,
B, C & D are siblings, bcz they have same parent A
E & F are siblings.
H, I & J are siblings.

6) Leaf :-

- Node which does not have any child node is called as leaf node.
- Node that has a degree of zero is called as leaf node or Terminal node.

Eg :- K, L, F, G, M, I & J are leaf nodes.

7) Degree of a Node / Degree :-

- Total number of children of a node is called degree

- Number of subtrees of a node is called degree

Eg :- Degree of node A = 3
Degree of node B = 2
Degree of node C = 1
Degree of node G = 0

8) Level :-

- Distance of a node from the root is called level of the node
- Distance from root to itself is '0'. So level of root node is '0'
- Node B, C, & D is at distance of 1 from root node, so its level is 1
- Node E, F, G, H, I & J is at distance of 2 from root node, so its level is 2

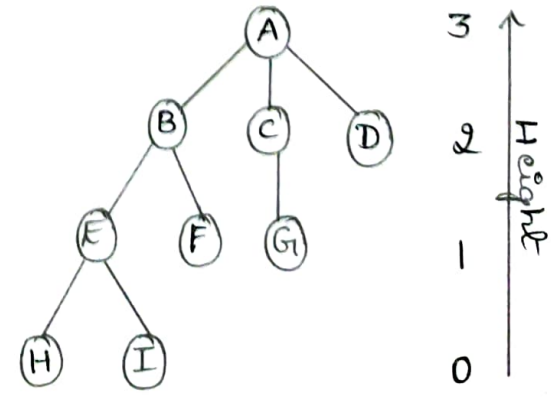
9) Height of a node :-

- Height of a tree is defined as maximum level of any leaf in the tree.

(OR)

- Total number of edges from leaf node to a particular node in the longest path is called as height of that node.
- Height will be calculated from leaf to root in bottom up manner.

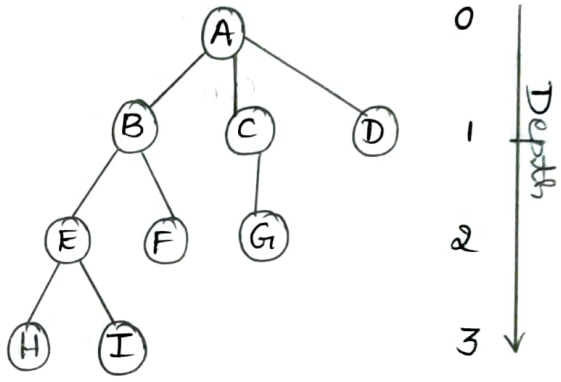
Eg :- In the given example,
Height of a tree = 3
Height of node C (from G1) = 1
Height of node B (from I) = 2
Height of node A from G1 = 2
Height of node A from H = 3



10) Depth of Node :-

- Total number of edges from root node to a particular node is called depth of that node.
- Depth will be calculated from root to leaf in top-down manner.

Eg :- Depth of node B = 1
Depth of node I = 3
Depth of node G = 2



- Height of all the leaf nodes will be 'zero'.

11) Path :-

- Sequence of nodes & edges from one node to another node is called path.

12) Internal Nodes :-

- Nodes except leaf nodes in a tree are called internal nodes

13) External Nodes :-

- NULL link of any node in a tree is an external node i.e, leaf nodes are the external nodes.

14) Left subtree :-

- All the nodes that are towards the left hand hand side of a node

Eg :- Left subtree of A are B, E, F, K & L
Left subtree of B are E, K, L

15) Right subtree :-

- All the nodes that are towards the right hand side of a node.

Eg :- Right subtree of A are D, H, I, J & M

Binary Trees:-

- A binary tree is a special type of tree which has finite set of nodes that is either empty or consist of a root and 2 subtrees called left subtree & right subtree i.e, every node can have a maximum of 2 child.
- A binary tree can be partitioned into 3 subgroups namely Root, left subtree and right sub tree.

Root:-

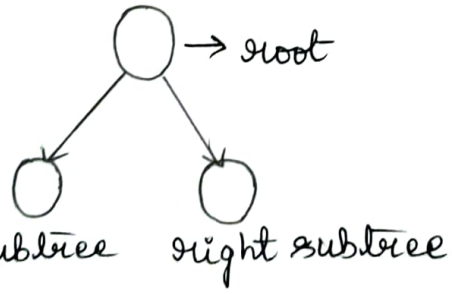
- If tree is not empty, the first node is called root node

Left subtree:-

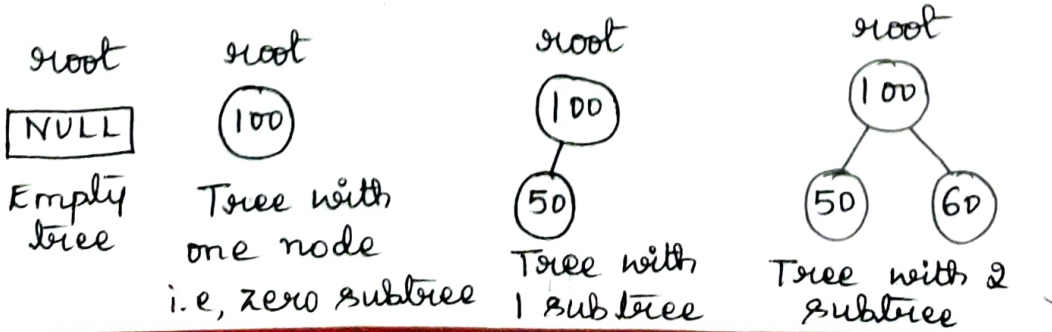
- It is a tree which is connected to the left of root. Since it comes towards left of root, it is called left subtree

Right subtree:-

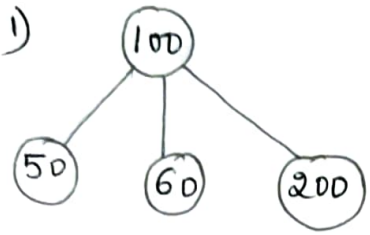
- It is a tree which is connected to the right of root. Since it comes towards right of root, it is called right subtree.

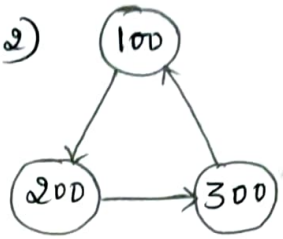


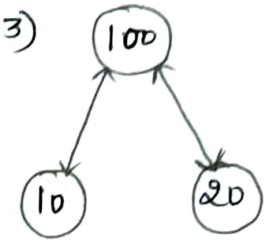
- An empty tree is also a binary tree.
- Binary means almost two i.e, zero, one or two subtrees are possible.
- But more than 2 subtrees are not permitted.



Whether the below following trees are binary tree or not?

1) 

2) 

3) 

- Above trees are not binary trees because,

* 1st tree has more than 2 subtrees
* 2nd tree has a cycle from 100 to 200 & so on. Binary trees should not have cycle.
* 3rd tree, node 100 has subtree 10 & 10 has subtree 100 i.e, bidirectional. If 10 is a subtree of 100 then 100 cannot be subtree of 10. So it is not a binary tree.

Different types of binary trees :-

1) Strictly binary tree (Full binary tree)
2) Complete binary tree
3) Skewed tree

1) Strictly binary tree :-

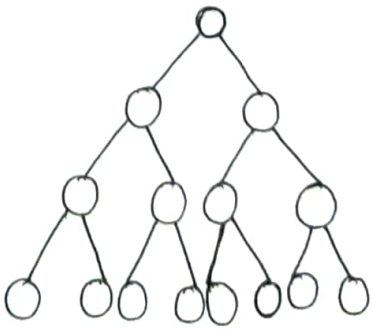
- A tree having 2^i nodes in any given level 'i' is called as strictly binary tree.
- Strict binary tree should have exactly 2 children or none.

Eg :- No of nodes at level 0 $= 2^0 = 1$

No of nodes at level 1 $= 2^1 = 2$

No of nodes at level 2 $= 2^2 = 4$

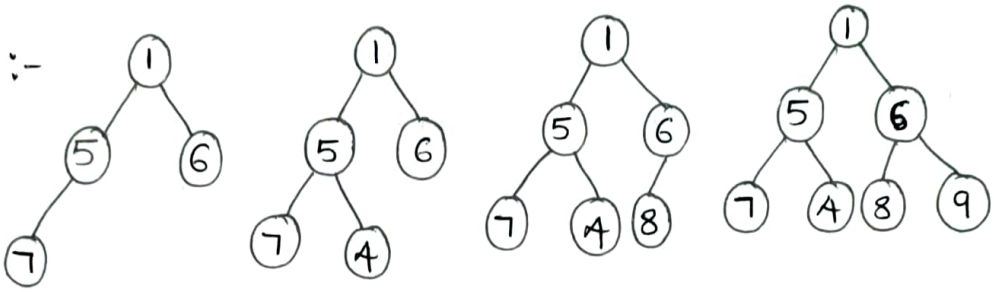
No of nodes at level 3 $= 2^3 = 8$



2) Complete binary tree :-

- Nodes must have exactly 2 children at every level.
- If the nodes in the last level are not completely filled then all the nodes in that level should be filled only from left to right.

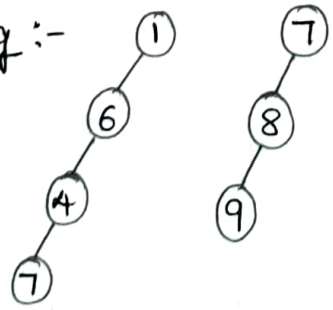
Eg :-



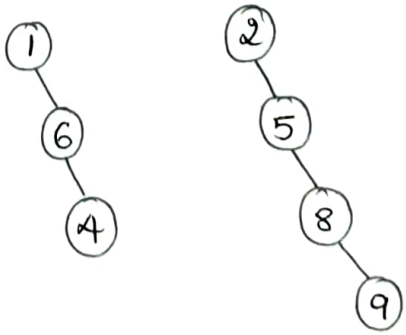
3) Skewed tree :-

- It is a tree consisting of only left subtree or only right subtree.
- A tree with only left subtree is called as left skewed binary tree.
- A tree with only right subtree is called as right skewed binary tree.

Eg :-



Left Skewed Tree



Right Skewed Tree

Properties of Binary tree :-

1) The maximum number of nodes on level 'i' of a binary tree $= 2^i$ for $i \geq 0$
2) The maximum number of nodes in a binary tree of depth $K = 2^K - 1$
3) The number of leaf nodes is equal to number of nodes of degree 2.

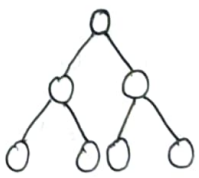
(or)

If n_0 is a leaf node & n_2 is no of nodes of degree 2 then $n_0 = n_2 + 1$

Proof :-

1) The maximum number of nodes on level 'i' of a binary tree $= 2^i$ for $i \geq 0$

- Root is the only node on level 'i'
- Consider a complete binary tree and observe:



No of nodes at level $0 = 2^0 = 1$
No of nodes at level $1 = 2^1 = 2$
No of nodes at level $2 = 2^2 = 4$
⋮
No of nodes at level $i = 2^i$

∴ Total no of nodes in the full binary tree of level i
$= 2^0 + 2^1 + 2^2 + \cdots + 2^i$

2) Total no of nodes in the full binary tree of level 'i'
$= 2^0 + 2^1 + 2^2 + \cdots + 2^i$

- Above series is a geometric progression whose sum is given by

$S = a(r^n - 1)/(r-1)$ where $a = 1$, $n = i+1$ & $r = 2$

- So, total no of nodes $n_t = a(r^n - 1)/(r-1)$

$$= 1(2^{i+1} - 1)/(2-1)$$

$$\boxed{n_t = 2^{i+1} - 1} \quad \text{—— (1)}$$

- Depth of the tree K = maximum level + 1

$$K = i + 1$$

Substituting this value in eqn (1)

$$n_t = 2^{i+1} - 1$$

$$n_t = 2^K - 1$$

$$\therefore \boxed{K = 2^K - 1}$$

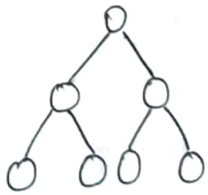
∴ Maximum number of nodes in a binary tree of depth

$$K = 2^K - 1$$

3) The number of leaf nodes is equal to number of nodes of degree 2.

(or)

If n_0 is a leaf node and n_2 is no of nodes of degree 2 then $n_0 = n_2 + 1$



- Consider a binary tree of degree -2 nodes
- Let 'N' be the no of nodes and
  'B' be the no of branches
- Since all nodes in a binary tree are of degree - 2 then

Total no of nodes in the tree $N = n_0 + n_1 + n_2$ —— (1)

$n_0 \rightarrow$ no of nodes in degree 0

$n_1 \rightarrow$ no of nodes in degree 1

$n_2 \rightarrow$ no of nodes in degree 2

observe,

In a tree, total no of nodes = no of branches + 1

i.e, $N = B + 1$ —— (2)

- If there is node with degree 1, no of branches = 1

So, for n_1, no of nodes of degree 1, no of branches = n_1 —— (3)

- If there is node with degree 2, no of branches = 2

So, for n_2, no of nodes of degree 2, no of branches = $2n_2$ —— (4)

Add eqn (3) & (4)

$$B = n_1 + 2n_2 \quad \text{——} (5)$$

Substitute eqn (5) in (2)

$$N = B + 1$$

$$N = n_1 + 2n_2 + 1 \quad \text{——} (6)$$

∴ From (1) & (6)

$$N = n_0 + n_1 + n_2 \quad \& \quad N = n_1 + 2n_2 + 1$$

$$n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$$

$$n_0 - n_2 - 1 = 0$$

$$\boxed{n_0 = n_2 + 1}$$

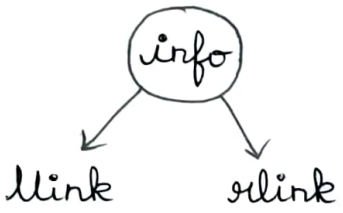
Binary Tree Representation :-

The storage representation of binary trees can be classified as follows:

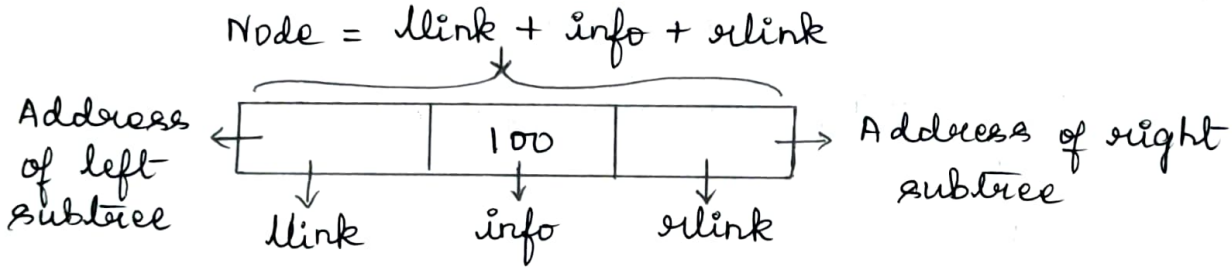
1) Array representation
   ↳ uses static allocation technique
2) Linked representation
   ↳ uses dynamic allocation technique.

1) Linked Representation :-

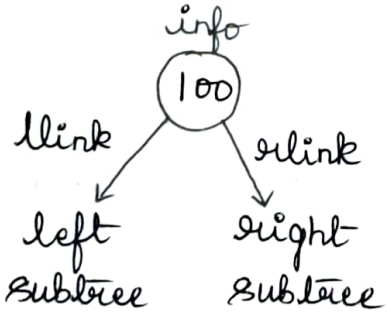
- In a linked representation, a node in a tree has 3 fields
  - i) Info → which contains the actual information
  - ii) llink → which contains the address of left subtree
  - iii) rlink → which contains the address of right subtree



- Linked representation is applicable only for binary trees
- The pictorial representation of a node in a binary tree is as shown:



- Above node can also be written as



- Using the above node structure it is difficult to determine the parent of a node.
- Node can be represented using self-referential structure.

```
struct node
{
   int info;
   struct node *llink;
   struct node *rlink;
};
```

- In the above example, llink and rlink is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'llink & rlink' as the referencing pointer.
- A pointer variable 'root' can be used to point to the root node always.
- If the tree is empty, pointer variable 'root' points to NULL indicating that the tree is empty. It can be declared and initialized as

```
NODE root = NULL (or)
struct node *root = NULL
```

Examples:-

1) Represent the given binary tree using linked representation

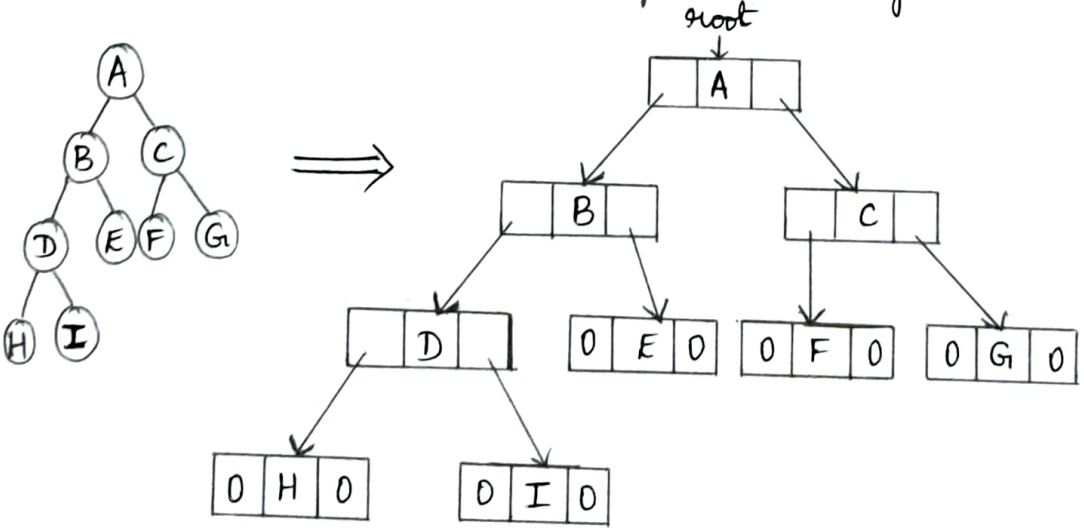


Fig:- Linked representation of a binary tree

2)

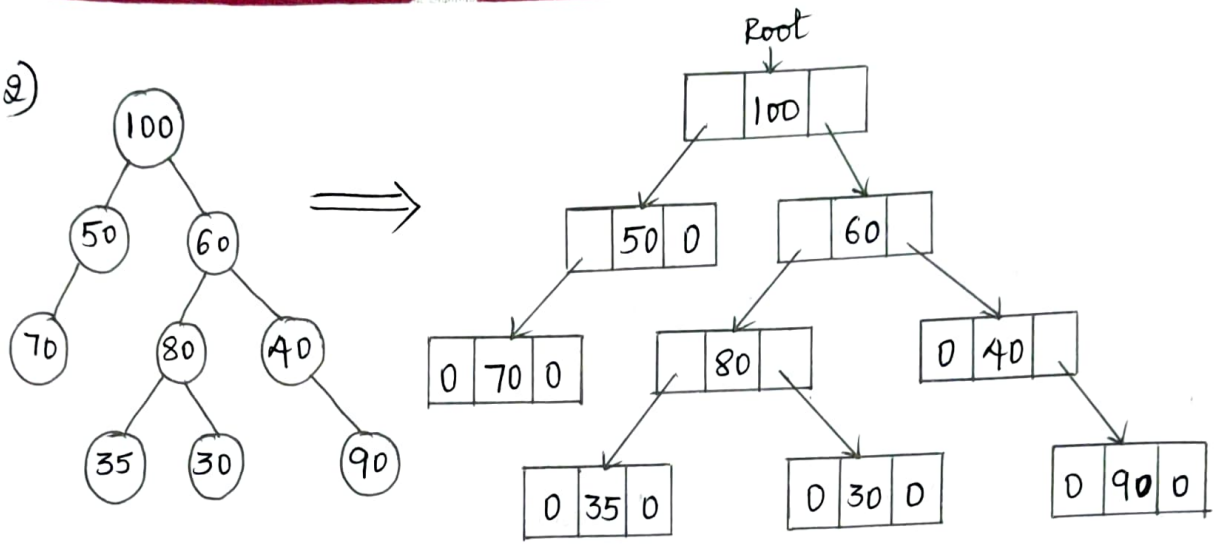


Fig: Linked representation of a binary tree.

2) Array Representation :-

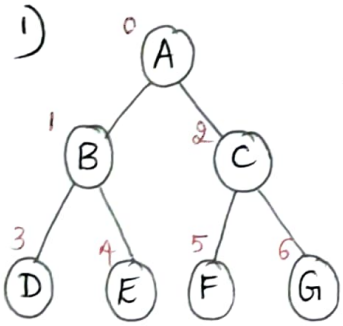
- A tree can be represented using an array, which is called as sequential representation.
- Nodes are numbered sequentially from 0.
- Node with position '0' is considered as root node.
- If an index 'i' is 0, it gives the position of root node.
- Given the position of any other node i, $2i+1$ gives the position of the left child and $2i+2$ gives the position of right child.

(or)

- If 'i' is the position of the left child, $i+1$ gives the position of right child.
- If 'i' is the position of the right child, $i-1$ gives the position of left child.
- Given the position of any node 'i', the parent position is given by $(i-1)/2$
- If the value of 'i' is odd, it points to the left child otherwise, it points to the right child.
- Binary tree is always completed from left to right.

Represent the following binary trees using array representation :-

1)

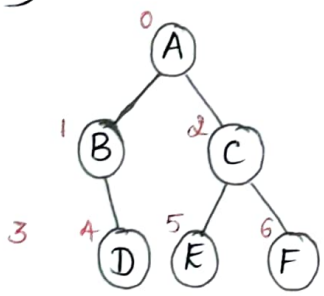


⟹

0	1	2	3	4	5	6	→ Index 'i'
A	B	C	D	E	F	G	→ Value of nodes

Fig :- Array representation of a binary tree.

2)

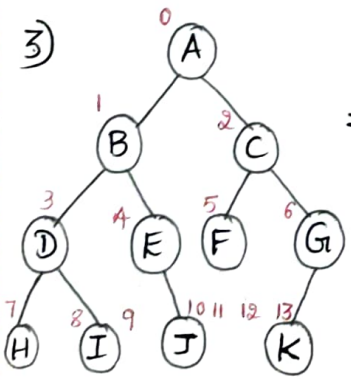


⟹

0	1	2	3	4	5	6
A	B	C		D	E	F

Fig :- Array representation of a binary tree

3)

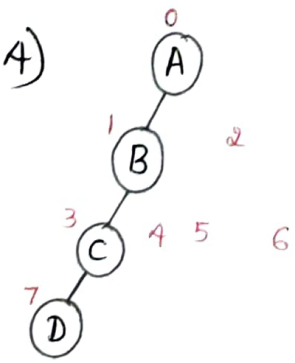


⟹

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

Fig :- Array representation of a binary tree

4)

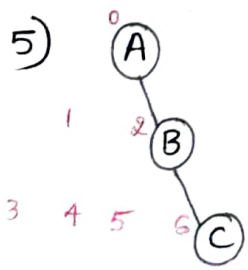


⟹

0	1	2	3	4	5	6	7
A	B		C				D

Fig :- Array representation of a binary tree

5)



⟹

0	1	2	3	4	5	6
A		B				C

Fig :- Array representation of a binary tree

From example – 1 :-

- $i = 1$ then

  $2i+1 \rightarrow 2(1)+1 = 3 \rightarrow$ left child

  $2i+2 \rightarrow 2(1)+2 = 4 \rightarrow$ right child

- $i = 2$ then

  $2i+1 \rightarrow 2(2)+1 = 5 \rightarrow$ left child

  $2i+2 \rightarrow 2(2)+2 = 6 \rightarrow$ right child

- If 'i' is the position of the left child, i+1 gives the position of right child

- $i = 1$ then → left child

  $i+1 \rightarrow 1+1 = 2 \rightarrow$ right child

From example – 2 :-

- $i = 2$ then → right child

  $i-1 \rightarrow 2-1 = 1 \rightarrow$ left child

- $i = 4$ then → right child

  $i-1 \rightarrow 4-1 = 3 \rightarrow$ left child

From example – 1 :-

- $i = 5$ then $(i-1)/2 = (5-1)/2 = 2$

  ∴ 2 is the parent node of index value 5

  i.e, C is parent of F

$i = 5$ is odd value, so it is a left child

- $i = 4$ then $(i-1)/2 = (4-1)/2 = 1.5 = 1$

  ∴ Index value 1 is the parent node of index value 4 i.e, B is parent of E

$i = 4$ is even value, so it is a right child.

Advantages of array representation :-

- Easy for implementation
- Fast access
- Good for complete binary trees

Disadvantage of array representation :-

- Wastes the memory for skewed tree, because in skewed tree one side of binary tree will be left empty, so the empty side of the binary tree will also be left empty inside the array memory.
- Implementation of operation requires re-arranging of elements.

Binary Tree Traversal :-

What is the meaning of traversing a tree?

- Traversing is a method of visiting each node of a tree exactly once in a systematic order.
- During traversal, we may print the info field of each node visited.

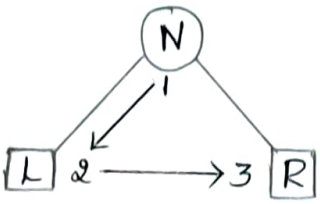
Different traversal techniques :-

1) Preorder traversal
2) Postorder traversal
3) Inorder traversal

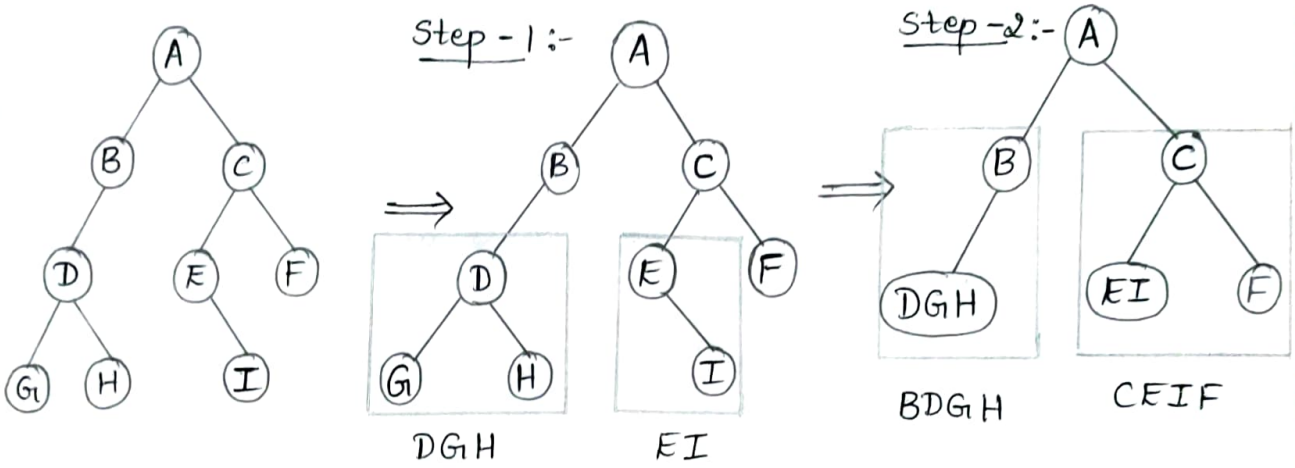
1) Preorder Traversal :-

It can be recursively defined as follows:

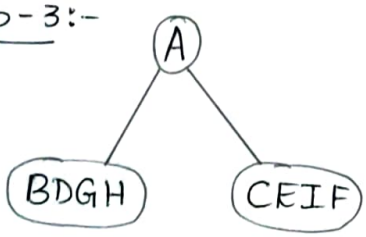
- Process the root Node [N]
- Traverse the left subtree in preorder [L]
- Traverse the Right subtree in preorder [R]



Example :-



Step - 3 :-



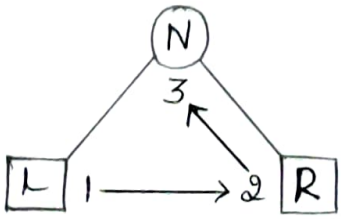
$\Rightarrow$ ABDGHCEIF

∴ Preorder traversal for the given binary tree is ABDGHCEIF

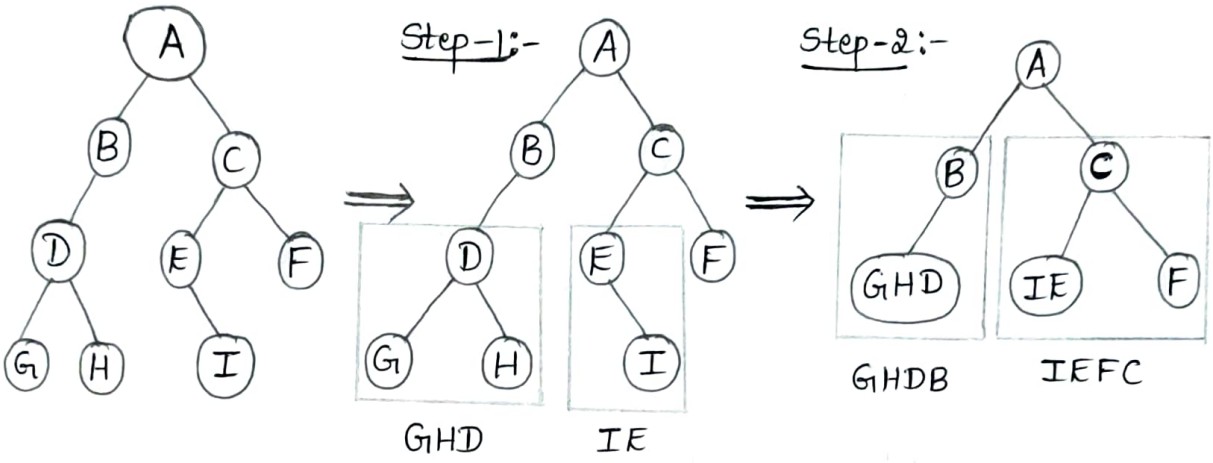
2) Postorder Traversal :-

It can be recursively defined as follows:

- Traverse the Left subtree in postorder [L]
- Traverse the Right subtree in postorder [R]
- Process the Root node [N]



Example :-



Step-3:-

∴ Postorder traversal for the given binary tree is GHDBIEFCA

C function to traverse the tree in preorder :-

```
Void preorder (NODE root)
{
  if (root == NULL) return;
  printf ("%d", root → info);
  preorder (root → llink);
  preorder (root → rlink);
}
```

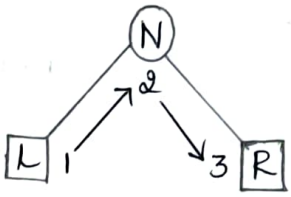
C function to traverse the tree in postorder :-

```
Void postorder (NODE root)
{
  if (root == NULL) return;
  postorder (root → llink);
  postorder (root → rlink);
  printf ("%d", root → info); }
```

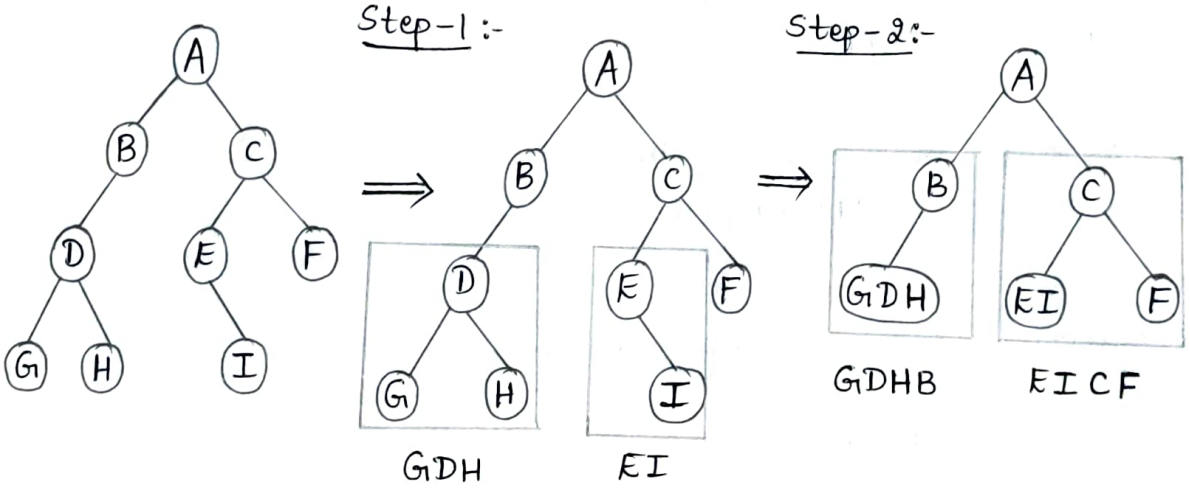
3) Inorder Traversal :-

It can be recursively defined as

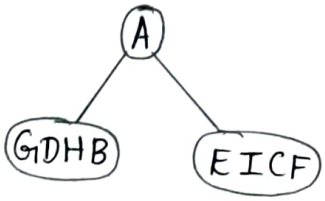
- Traverse the Left subtree in inorder [L]
- Process the root Node [N]
- Traverse the Right subtree in inorder [R]



Example :-



Step-3 :-



$\Rightarrow$ GDHBAEICF

∴ Inorder traversal for the given binary tree is GDHBAEICF

C function to traverse the tree in inorder:-

```
Void inOrder (NODE root)
{
  if (root == NULL) return;
  inOrder (root → llink);
  printf ("%d", root → info);
  inOrder (root → rlink);
}
```

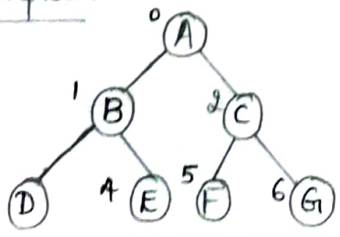
C function to create a node in a tree:-

```
typedef struct node * treePointer;
typedef struct {
        int data;
        treePointer leftChild, rightChild;
        } node;
```

4) Level order Traversal:-

- The nodes in a tree are numbered starting with the root on level 0, continuing with nodes on level 1, level 2 and so on.
- Nodes on any level are numbered from 'left' to 'right'
- Visiting the nodes using the ordering suggested by the node numbering is called level order traversing.
- For level order traversing, we will make use of queue (FIFO) concept.

Example:-



Level
0
1
2

output of level order traversing is

A B C D E F G

Algorithm for level order traversal:-

1) Insert the root node identified by variable 'root' into queue
It is done using statement

```
front = 0, rear = -1;
q[++rear] = root;
```

2) Delete a node from front end and visit that node by displaying the statement

```
cur = q[front++];
printf("%d\n", cur->info);
```

3) If node visited in step ② has left subtree, insert that node into queue.

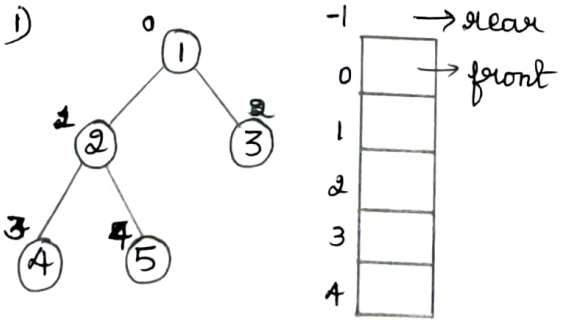
```
if (cur->llink != NULL)
q[++rear] = cur->llink;
```

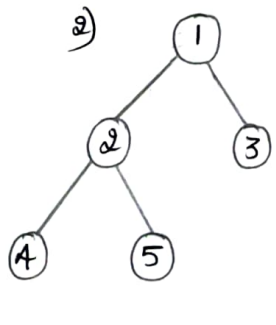
4) If node visited in step ② has right subtree, insert that node into queue.

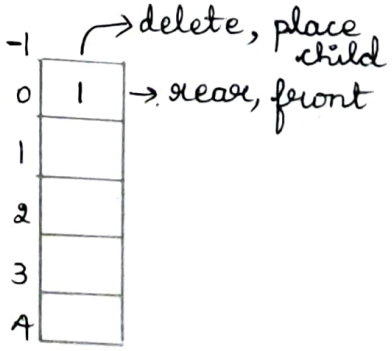
```
if (cur->rlink != NULL)
q[++rear] = cur->rlink;
```

5) Repeat through step ② as long as queue is not empty.

Consider the example:





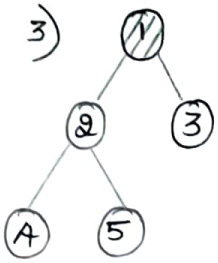


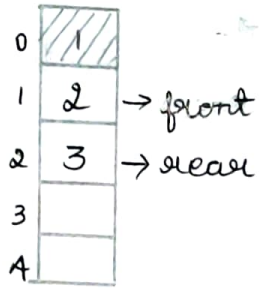
output:- 1

- In the above diagram, root node is inserted into queue and then rear is incremented to '0'.
- After inserting root node, then visit the next nodes and check whether it contains left or right sub tree.

- If either left or right subtree is present then insert it into the queue by deleting the previously visited node and display it as an output
- Shaded portion indicates that the data is deleted in the queue and displayed as an output.

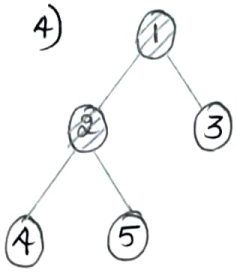
3)

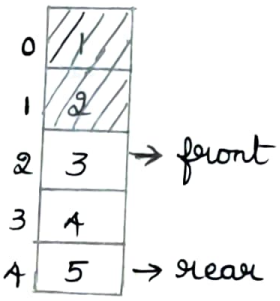




output :- 1

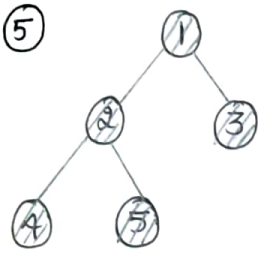
4)

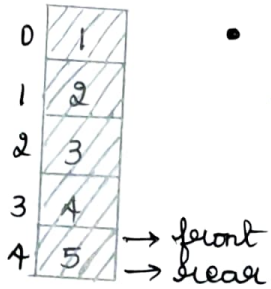




output :- 1 2

5)





output :- 1 2 3 4 5

- As, there is no child nodes for 3, 4 and 5, delete them one by one and display them as an output.

C function to implement level order traversal :-

```
Void level_order (NODE root)
{
  NODE q[MAX_QUEUE], cur;
  int front=0, rear=-1;
  q[++rear] = root;
  while (front <= rear)
  {
    cur = q[front++];
    printf("%d", cur->info);
    if (cur->llink != NULL)
    q[++rear] = cur->llink;
    if (cur->rlink != NULL)
     q[++rear] = cur->rlink;
  }
  printf("\n"); }
```

Threaded Binary Trees :-

Limitations of binary tree :-

- In a binary tree, more than 50% of link field have null values and more memory space is wasted by storing null values.
- Traversing a tree with binary tree is time consuming. These limitations can be overcome by threaded binary tree.

Definition :-

- In the linked representation of any binary tree, there are more null links than actual pointers.
- These null links are replaced by the address of some nodes (pointers) called threads which points to other nodes in the tree and the tree is termed as threaded binary tree.

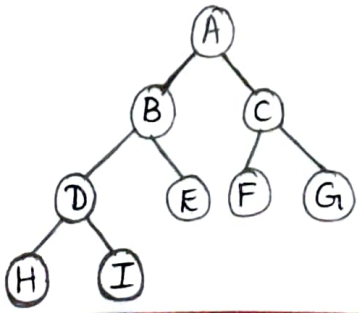
To construct the threads use the following rules :-

- Assume that ptr represents a node.

1) If ptr → left child is NULL i.e, left link of a node contains NULL then replace the NULL link with a pointer to the inorder predecessor of ptr.

2) If ptr → right child is NULL i.e, right link of a node contains NULL then replace the NULL link with a pointer to the inorder successor of ptr.

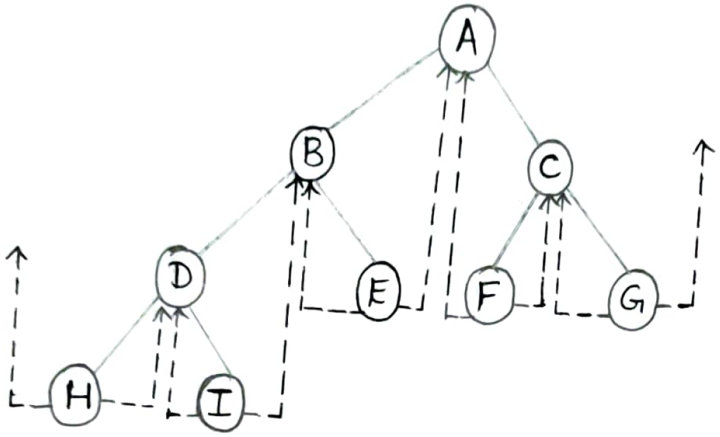
Consider a binary tree :-



Inorder Traversal

H D I B E A F C G

Threaded tree corresponding to binary tree :-



Dotted lines → Threads
Straight lines → Child Nodes

- In the above figure, all the leaf nodes have the pointer to left and right child as NULL, so we have to replace the NULL values to the
  inorder predecessor → for left child and
  inorder successor → for right child
- Inorder predecessor and successor will be indicated by dotted lines as threads.
- Predecessor and Successor are obtained from the inorder traversal result of the given binary tree.
- Above tree has 9 Nodes and 10 links which have been replaced by threads.
- Traverse the given binary tree in inorder then the result is

  H D I B E A F C G

- Eg :- Node E is a leaf node and has a pointer to left and right child as NULL.
  So, it can have predecessor thread pointing to node B and successor thread pointing to node A from the result of inorder traversal.
- There should be no loose threads in threaded binary tree.
- In the above figure of threaded tree, 2 threads have been left dangling

- Two dangling threads are → one child in the left of H and the other in the right child of G.
- In threaded binary tree, we cannot leave the threads loose or dangling, so we are going to assume a header node for all threaded binary tree.
- Header nodes will be used only for dangling threads.
- Header node points to the root node.
- Dangling threads → Threads that will not be having any predecessor or successor nodes in the inorder traversal.

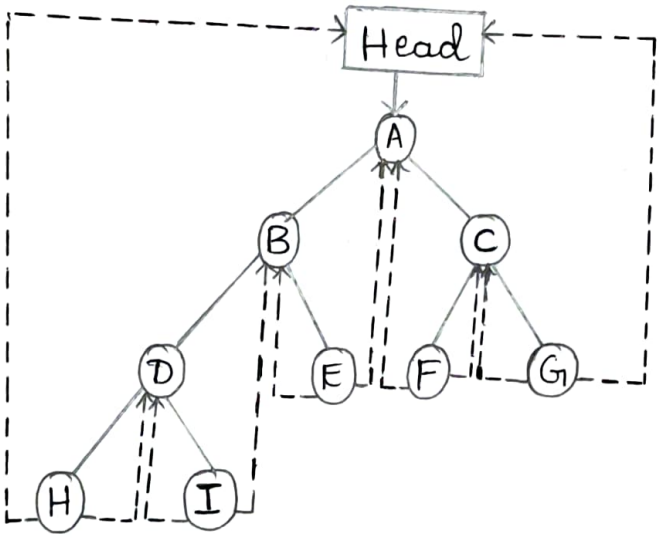
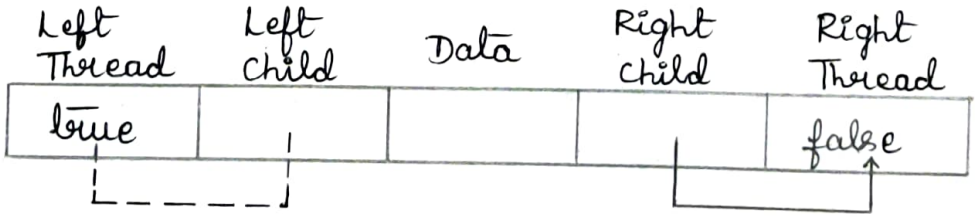


Fig :- Threaded Binary Tree with Head Node

Empty threaded binary tree :-



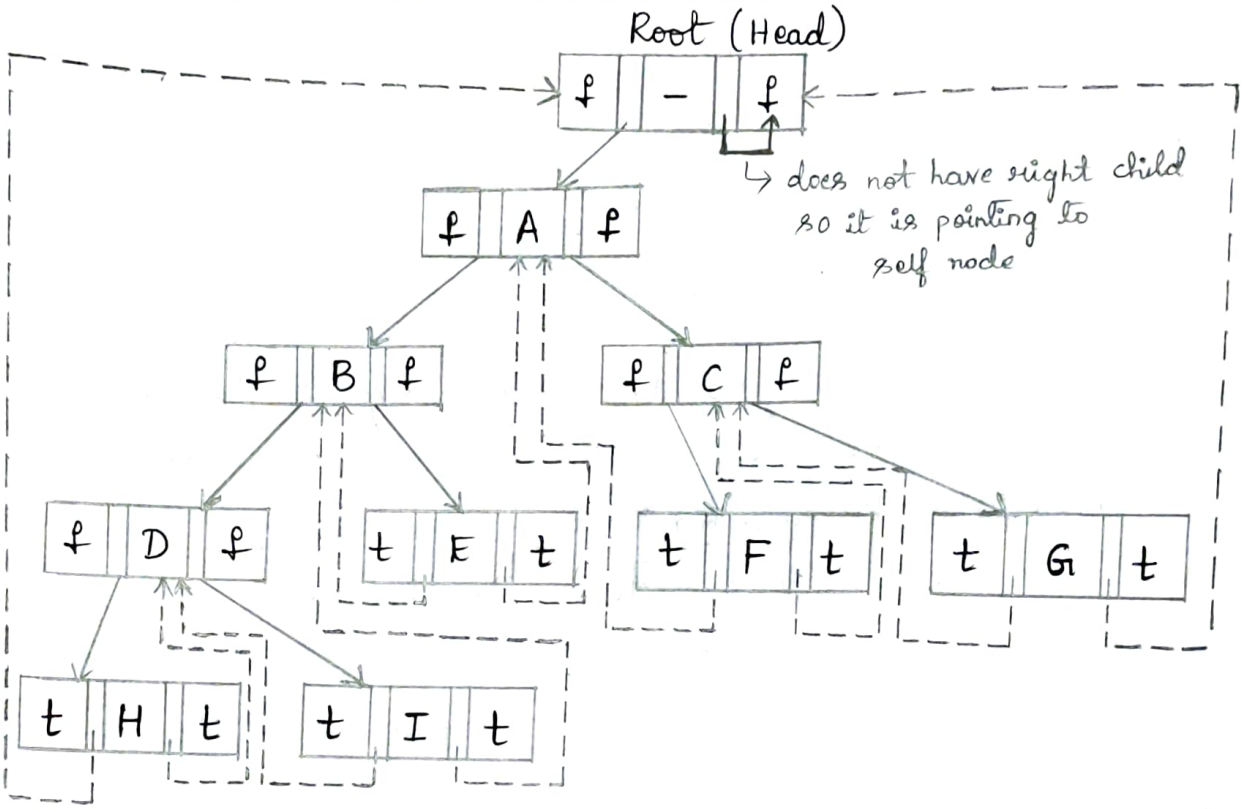
Node structure in C declaration :-

```
typedef struct threadedTree *threadPointer;
typedef struct {
    short int leftThread;
    threadPointer leftChild;
    char data;
```

```
    threadPointer rightChild;
    short int rightThread;
} threadedTree;
```

Memory representation of threaded tree :-

- When trees are represented in memory, it should be able to distinguish between threads and pointers.
- This can be done by adding 2 additional fields to node structure i.e, leftThread and rightThread.
- If ptr → leftThread = TRUE, then ptr → leftChild contains a thread, otherwise it contains a pointer to the left child i.e, a self pointer.
- If ptr → rightThread = TRUE, then ptr → rightChild contains a thread, otherwise it contains a pointer to the right child i.e, a self pointer.
- In memory representation, if the nodes have threads then it will be represented by 'true' value or else it is denoted by 'false' value
- In memory blocks, true value is denoted by 't' and false value is denoted by 'f'
- Variable 'root' points to the header node of the tree where left child of the 'root' points to the start of the first node of the actual tree. i.e, root node of the actual tree.
- This is true for all threaded trees.

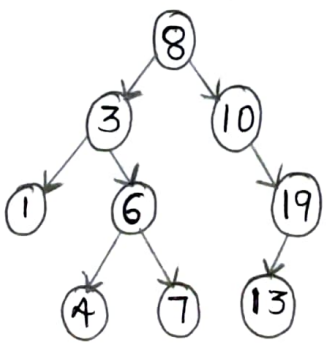


Binary Search Tree (BST) :-

- A binary search tree is a binary tree in which for each node say 'x' in the tree, elements in the left subtree are less than info(x) and elements in the right subtree are greater than info(x).
- Every node in the tree should satisfy this condition
- Binary Search Tree divides all its sub tree into 2 segments
  1) Left subtree
  2) Right subtree
- Binary search tree can be defined as :

  left_subtree (keys) ≤ Node (key) ≤ Right_subtree (keys)

Eg :-



Binary Search Tree Constructions :-

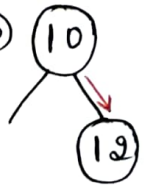
- Construction of BST consists of 3 steps

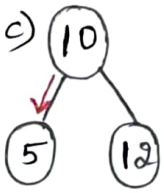
1) Place the first value in the given list as root Node
2) For next value in the given list of values, compare with root value.
   - If greater than root, place it towards "Right".
   - If lesser than root, place it towards "Left".
3) Repeat step -(2) for all values apart from first value i.e, root.

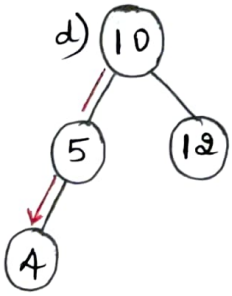
Construct a BST by inserting the following sequence of numbers :-

1) 10, 12, 5, 4, 20, 8, 7, 15, 13

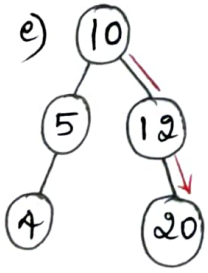
a) 

b) 

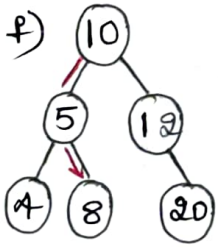
c) 

d) 

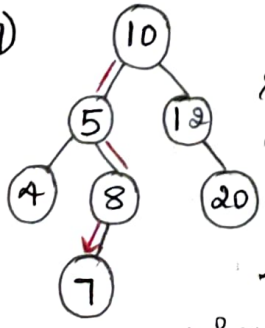
4 is less than 10 and when compared with 5, again its less than 5, so place it in 'left' of 5

e) 

20 is greater than 10, & when compared to 12, again it is greater than 12, so place it in 'right' of 12.

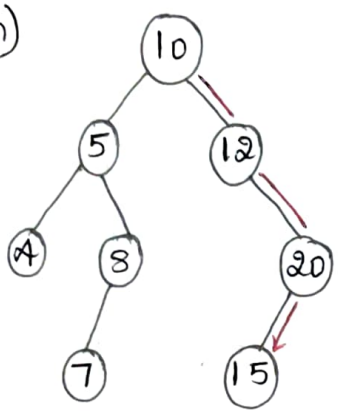
f) 

8 is lesser than 10, so take left sub tree, when compared to 5, it is greater than 5, so place it in 'right' of 5

g) 

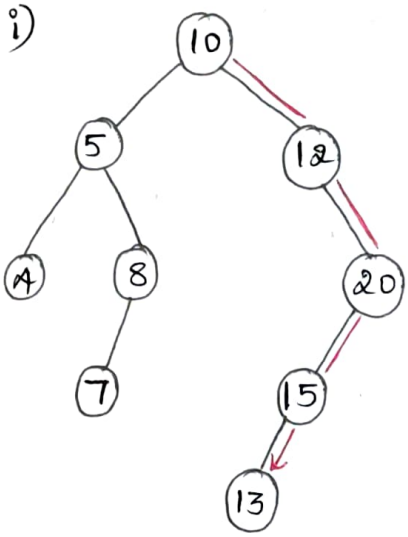
7 is lesser than 10, so take left and when compared to 5, it is greater than 5, so take right & again when compared to 8, it is lesser than 8, so place it in left of 8

h)



15 is greater than 10 so take right & when it is compared to 12, again it is greater than 12 so take right & when it is compared to 20, it is less than 20, so take left and place it in left of 20.

i)

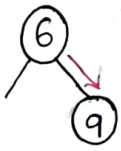


13 is ~~has~~ greater than 10, so take right & when it is compared to 12 again it is greater than 12, so take right & when it is compared to 20 it less than 20, so take left and when it is compared to 15 again it is less than 15 so take left and place it in left of 15.

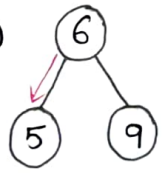
2) 6, 9, 5, 2, 8, 15, 24, 14, 7

a)

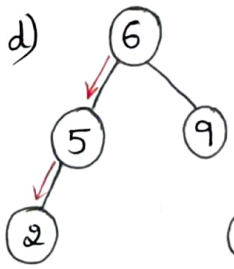
b)



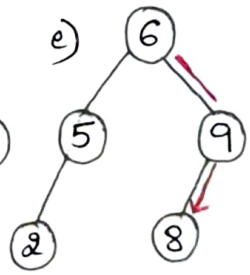
c)



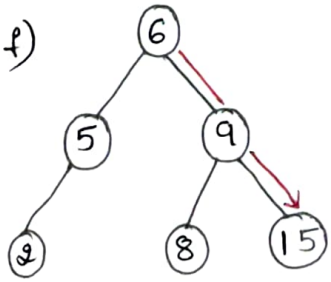
d)



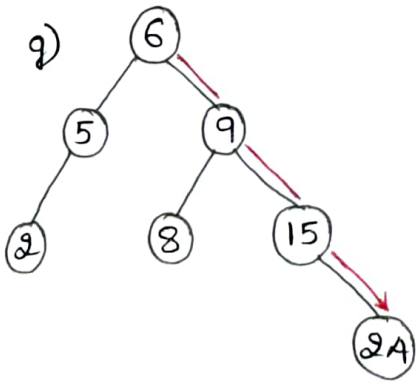
e)



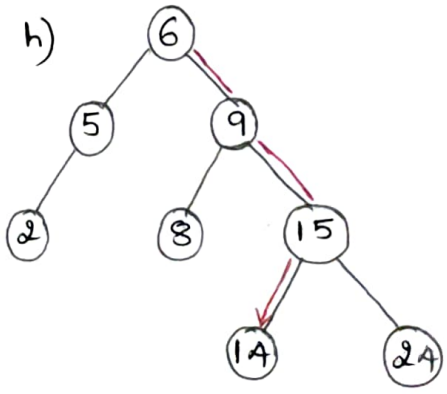
f)



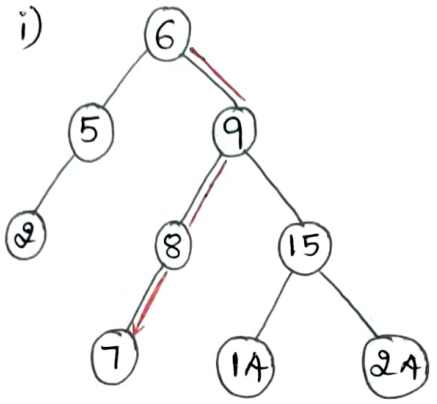
g)



h)



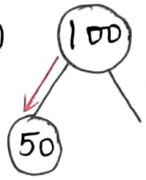
i)



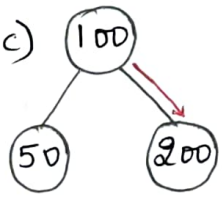
3) 100, 50, 200, 25, 90, 150, 300, 80, 180

a)

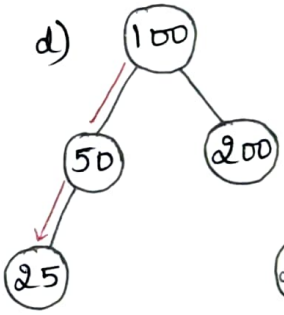
b)



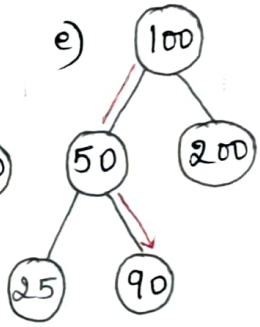
c)



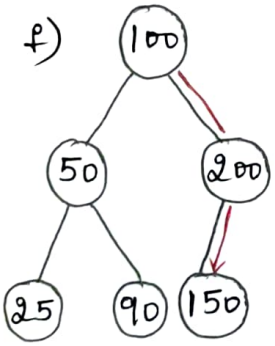
d)



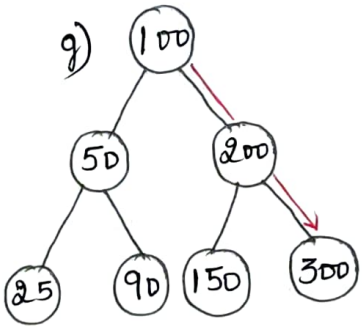
e)



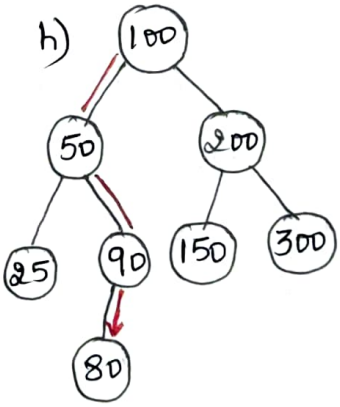
f)



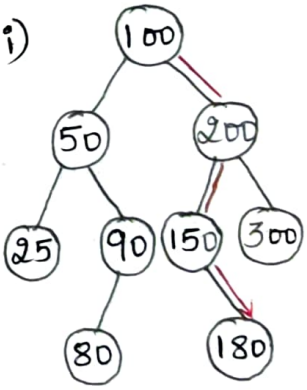
g)



h)



i)



4) 8, 3, 10, 1, 6, 19, 4, 7, 13 → construct BST

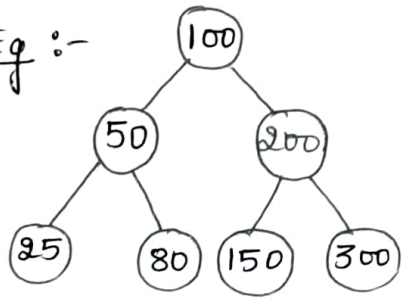
Binary Search Tree Operations :-

- Basic operations of BST (Binary Search Tree) are
  1) Insertion
  2) Searching
  3) Traversal
  4) Deletion

1) Insertion Operation :-

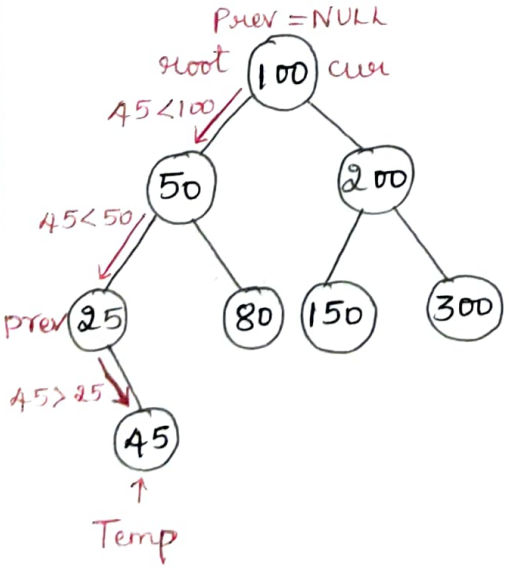
- Whenever an element is to be inserted, first locate its proper location, start searching from the root node then,
- If the data is less than the key value (root), search for empty location in left subtree and insert the data
- Otherwise, search for the empty location in right subtree and insert the data.

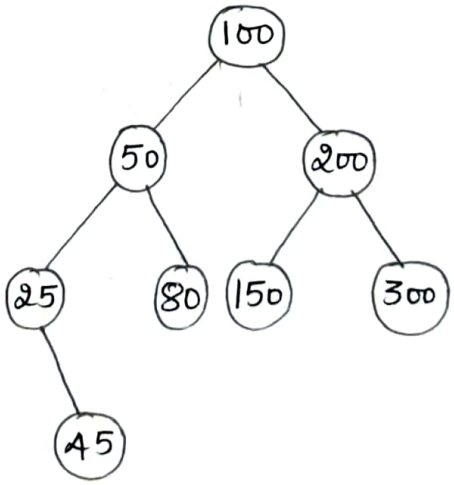
Eg :-



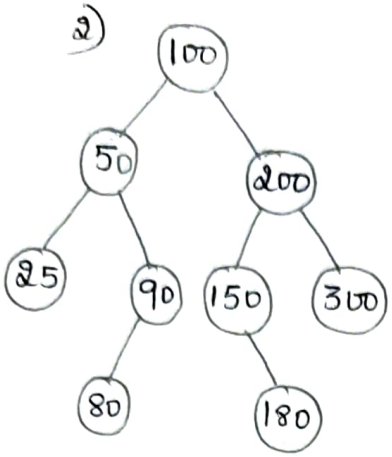
Item to be inserted = 45

- 'Prev' always points to the parent of 'cur' node.
- Initially, prev → NULL & cur = root.
- when cur = NULL, in such case we have found the appropriate position to insert a new element/item.
- Unlill cur = NULL, keep updating the value of 'cur'.

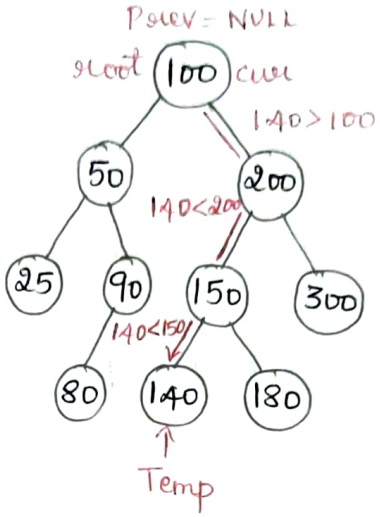


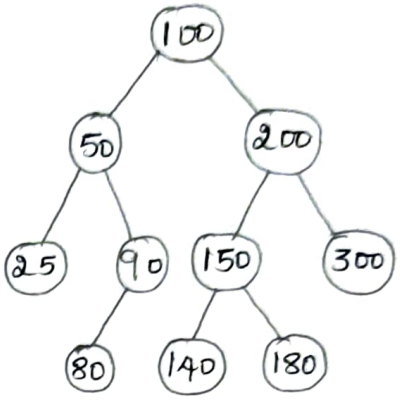


2)



Item to be inserted = 140





C function to insert the node in BST:-

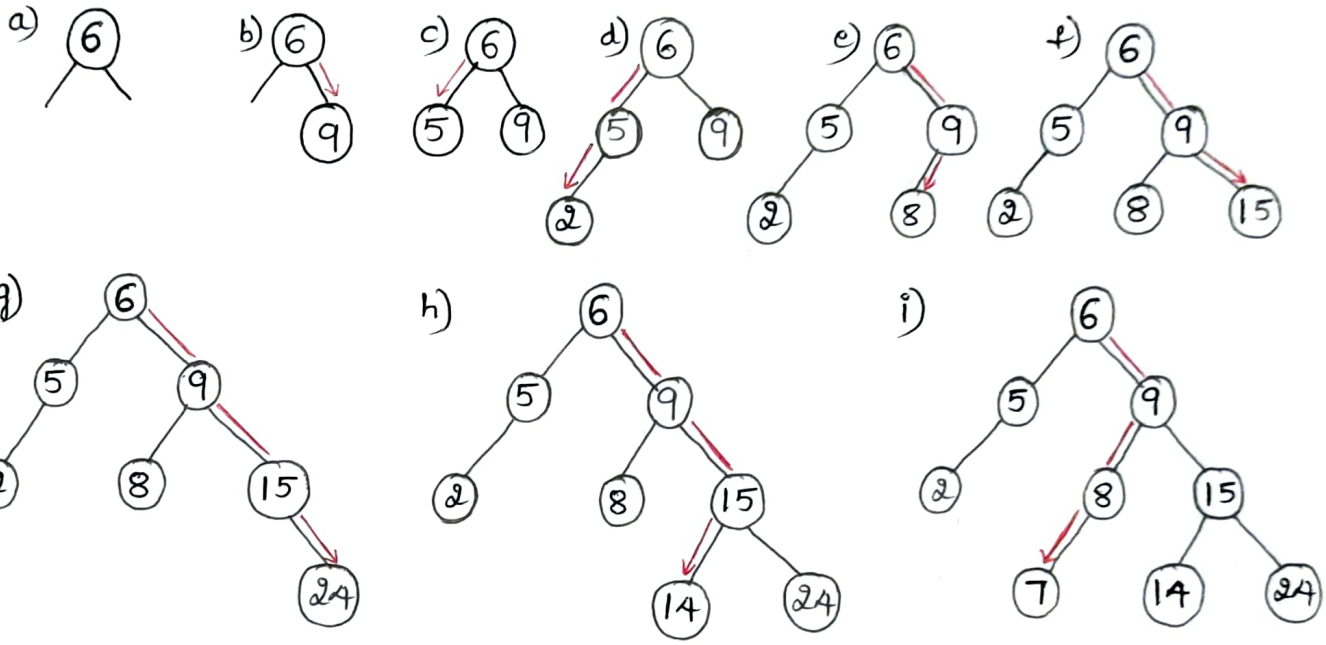
```
NODE insert (int item, NODE root)
{
  NODE temp, cur, prev;
  MALLOC (temp, 1, struct node);
  temp → info = item;
  temp → llink = NULL;
  temp → rlink = NULL;
  if (root == NULL) return temp;
  prev = NULL;
  cur = root;
  while (cur != NULL)
  {
    prev = cur;
    if (item < cur → info)
      cur = cur → llink;
    else
      cur = cur → rlink;
  }
  if (item < prev → info)
    prev → llink = temp;
  else
    prev → rlink = temp; return root; }
```

2) Traversal Operation :-

- Traversal operation on BST is similar to the traversal of binary tree i.e, preorder traversal
  postorder traversal
  inorder traversal.

Eg :- Construct a BST on a given value and show traversal using all traversal function

6, 9, 5, 2, 8, 15, 24, 14, 7



Preorder : 6 5 2 9 8 7 15 14 24

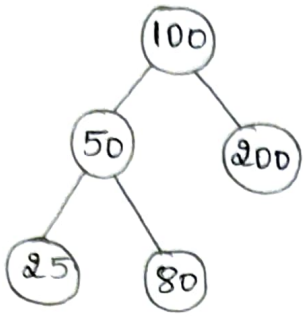
Postorder : 2 5 7 8 14 24 15 9 6

Inorder : 2 5 6 7 8 9 14 15 24

3) Search Operation :-

- Whenever an element is to be searched in BST, start searching from root node, then
- If the root = element, search is successful
- Else, if the data is less than the key value (root), search in left subtree
- Else, if the data is greater than the key value (root), search in right subtree.

Eg :-



Item to be searched is '80'

- Item = 80 is compared with root = 100, item is lesser than root.
- So traverse towards left subtree.
- Again item is compared with '50', and item is greater than root i.e, 50.
- So traverse towards right subtree.
- Again item is compared with 80, and it is a successful search, the item is found.

Function to search an item in BST using iteration :-

```
NODE search (int item, NODE root)
{
  NODE cur;
  if (root == NULL) return NULL;
  cur = root;
  while (cur != NULL)
  {
    if (item == cur → info)
    return cur;
    if (item < cur → info)
    cur = cur → llink;
    else
    cur = cur → rlink;
  }
  return NULL;
}
```

Function to search an item in BST using recursion :-

```
NODE search (int item, NODE root)
{
  if (root == NULL) return NULL;
```

```
if (item == root->info)
    return root;
if (item < root->info)
    return search(item, root->llink);
    return search(item, root->rlink);
}
```

4) Deletion Operation :-

- There are 3 different cases that needs to be considered for deleting a node from BST.

i) Node with no children (or) leaf node

ii) Node with one child (An empty left subtree & non empty right subtree & vice versa)

iii) Node with two children (Non empty left subtree & non empty right subtree)

Case -1 :- Delete a leaf node / node with no children

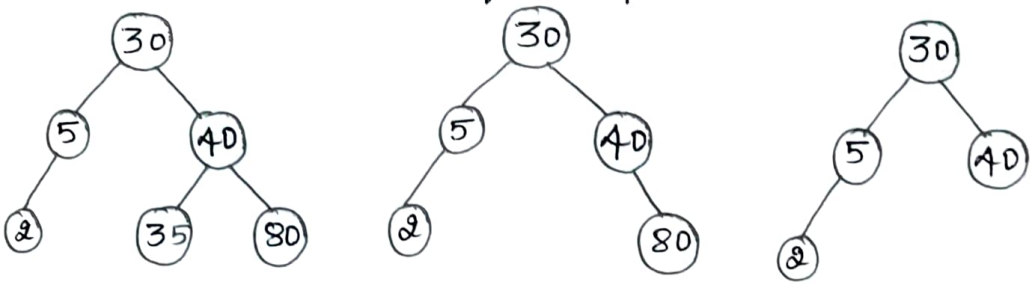


Fig :- (a) (b) (c)

- Deletion of a leaf node is quite easy.
- From the above fig (a), if we want to delete 'node 35' then left child field of its parent i.e, left child of 'node 40' is set to NULL and the node is freed.

```
parent->llink = NULL;
```

- This gives us the tree as in fig (b).
- Similarly, to delete 'node 80' from tree then right child field of its parent i.e, right child of 'node 40' is set to NULL and the node is freed.

parent→rlink = NULL;

- This gives us the tree as in fig (c).

Case – 2 :- Delete a node with one child.

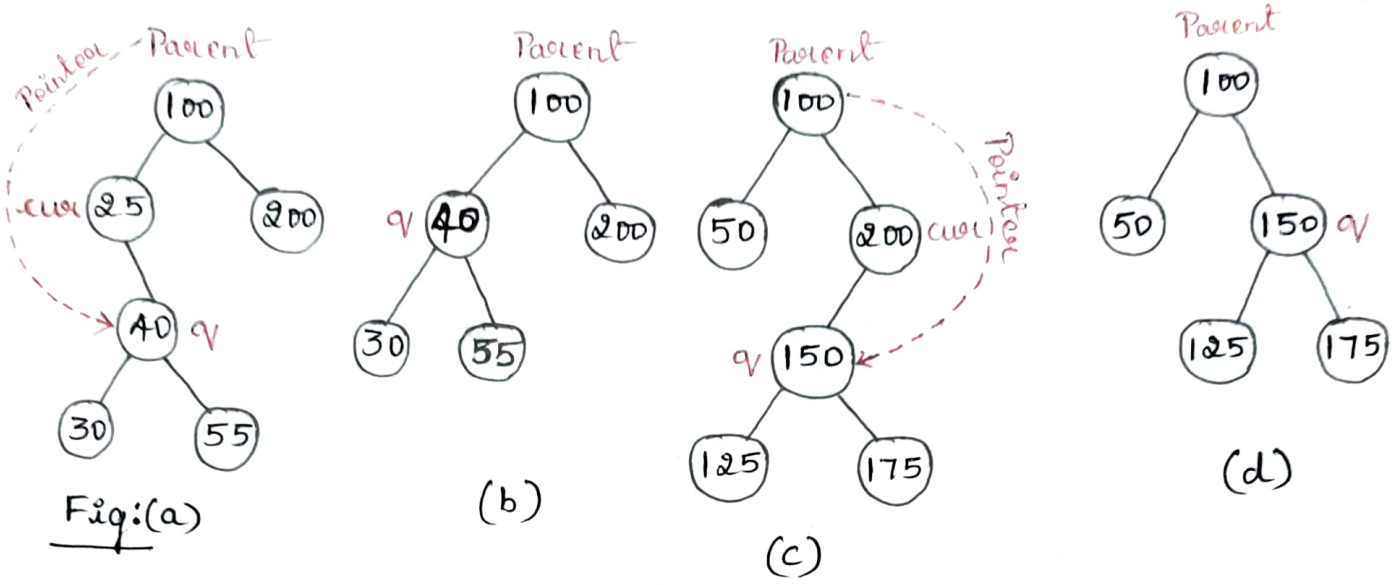


Fig:(a) (b) (c) (d)

- In both fig (a) and (c), contains a single child. i.e, Node with 'cur' variable.
- Node to be deleted is denoted by 'cur' and 'parent' denotes the parent of the node to be deleted.
- other than non – empty child of the node to be deleted is denoted by variable 'q'.
- In fig – (a), 40 is the 'q' because that tree has only one child. i.e Node 25 has only one child i.e, right child denoted by q and left child is empty.
- In fig – (c), 150 is the 'q' because that tree has only one child. i.e, Node 200 has only one child i.e, left child denoted by q and right child is empty.
- Node 'cur' is deleted using free() and then non – empty child node of the deleted node will be pointed to (or) replaced to the root node or parent node of the deleted node.

- If the deleted node was in left side then non-empty child of it will be placed towards the left side of the parent node or else it will be placed towards the right side.
- Node saved in the variable 'q' will be attached to the parent node of the deleted node based on its position of deleted node as shown in fig (b) & fig (d)

In general,

- 'cur' denotes the node to be deleted and in both cases one of the subtree is empty and the other is non-empty.
- The node identified by 'parent' is the parent of node 'cur'.
- The non-empty subtree can be obtained and is saved in a variable 'q'. The corresponding code is:

```
if (cur->llink == NULL)
    q = cur->rlink;
else if (cur->rlink == NULL)
    q = cur->llink;
```

Case-3:- Delete a node with two children
(or)
Non empty left subtree and non-empty right subtree.

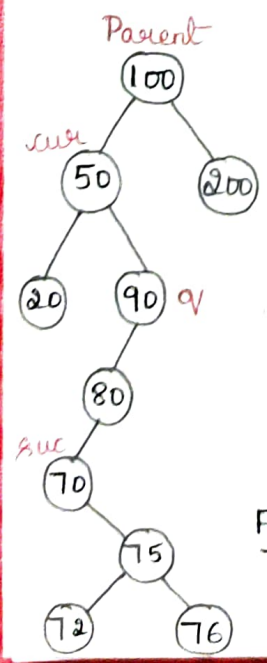


Fig:- (a)

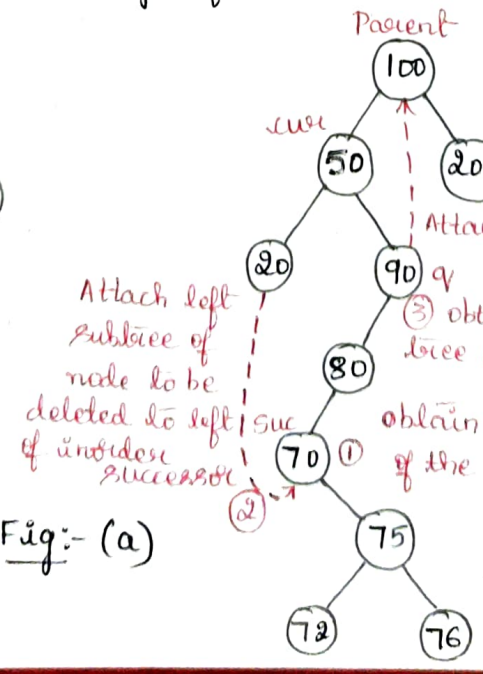


Fig:- (b)

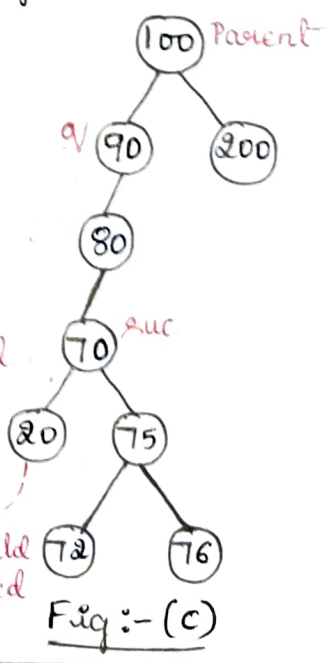


Fig:- (c)

Inorder traversal of fig(a):-

20, 50, 70, 72, 75, 76, 80, 90, 100, 200

suc

- In fig(a), 'cur' denotes the node to be deleted and 'parent' denotes the parent of the node 'cur'.

Step-1:-

- Find the inorder successor of the node to be deleted.

```
suc = cur → rlink;
while (suc → llink != NULL)
{
  suc = suc → llink;
}
```

Step-2:-

- Attach left subtree of the node to be deleted to the left of the successor of the node to be deleted.

```
suc → llink = cur → llink;
```

Step-3:-

- Obtain the right subtree of the node to be deleted and denoted by 'q'.

```
q = cur → rlink;
```

Attach the node q to parent:-

- After step-① and ② it is required to attach right subtree of the node to be deleted to the parent of the node to be deleted.
- If parent of the node to be deleted does not exist, then return 'q' itself as the root node.

```
if (parent == NULL) return q;
```

- If parent exists, then attach 'q' to parent node based on the direction of 'cur' node.
- If 'cur' is left child, attach 'q' to left of parent or else attach it to right of parent.

```
if (cur == parent -> llink)
    parent -> llink = q;
else
    parent -> rlink = q;
```

- once the node 'q' is attached to the parent, then the node pointed to by 'cur' can be deleted and then return the address of the root node.

```
free (cur);
return root;
```

Function to delete an item from the tree:-

```
NODE delete_item (int item, NODE root)
{
  NODE cur, parent, suc, q;
  if (root == NULL)
{
  printf ("Tree empty! Item not found");
  return root;
}
 parent = NULL, cur = root;
 while (cur != NULL)
{
  if (item == cur -> info) break;
  parent = cur;
  if (cur = item < cur -> info)
  cur = cur -> llink;
  else
  cur = cur -> rlink;
}
 if (cur == NULL)
{
  printf ("Item not found");
  return root;
}
```

Search

```
if (cur → llink == NULL)
  q = cur → rlink;
else if (cur → rlink = NULL)
  q = cur → llink;
else
{
  suc = cur → rlink;
 while (suc → llink != NULL)
   suc = suc → llink;
 suc → llink = cur → llink;
 q = cur → rlink;
}
 if (parent == NULL) return q;
 if (cur == parent → llink)
  parent → llink = q;
else
  parent → rlink = q;
free (cur);
return root;
}
```

case - 2

case - 3

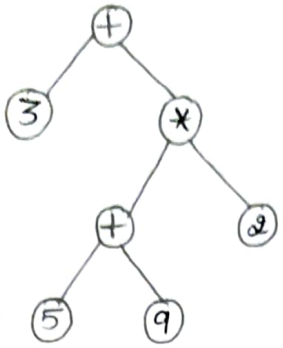
Applications of trees :-

- Evaluation of expressions

Expression trees :-

- A sequence of operators and operands that reduces to a single value is called as expression.
- Expression tree is a binary tree that satisfy the following properties
  - Any leaf is an operand
  - Root and internal nodes are operators
  - Subtrees represent sub expressions with root of the subtree as an operator.

Eg :- $3 + ((5 + 9) * 2)$



- Expression tree is constructed from bottom to top.
- Level of the tree indicates precedence.
- Operations at the root is the always the last operation performed.
- Operations at higher level of the tree are evaluated later than those below them.
- A node containing an operator is not a leaf node whereas a node containing an operand is a leaf node.
- If the expression tree is traversed in
  preorder – we get prefix expression
  postorder – we get postfix expression
  inorder – we get infix expression
- If we apply traversal procedure on binary expression trees the results are as follows.
  Inorder – Infix expression
  Preorder – Prefix expression
  Postorder – Postfix expression
- Binary expression tree can be represented by two types of expressions.
  - Algebraic expression
  - Boolean expression.

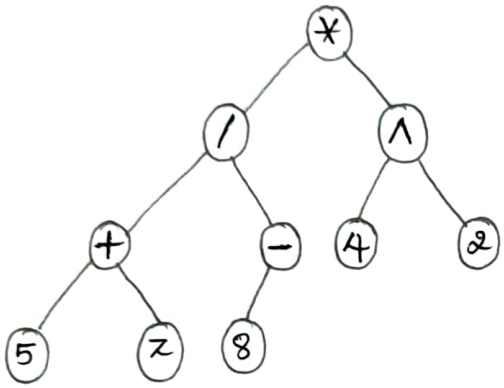
Algebraic expression trees :-

- Algebraic expression represents the expression that contains numbers, variables and unary and binary operators.
- Some of the common operators are *, +, ÷, −, ∧, ∼, /
  ∧ → Exponentiation
  ∼ → Negation

- Operators are present in the internal nodes.
- Numbers and Variables are present in the leaf node.

- Nodes of binary operators have 2 child nodes.
- Nodes of unary operators have 1 child node.

Eg :- $((5+z) / -8) * (A \wedge 2)$



- -8 → unary minus is used to change the sign of +ve value to a -ve value.
- Unary operations are the operation with only one operand
- Unary minus changes +ve number to a -ve number.

- In the above example, since unary minus is used, the direction of node '8' is also changed towards left even though it is present towards the right in expression.

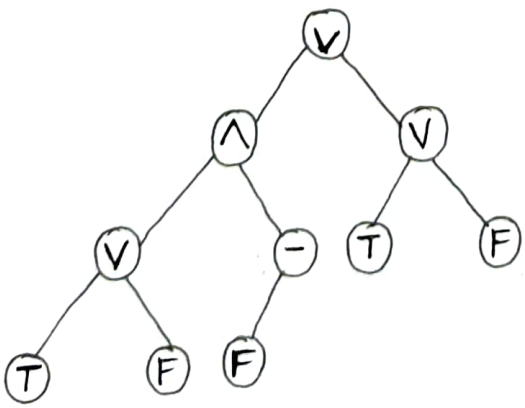
Boolean expression trees :-

- It is represented very similar to algebraic expression, the only difference being the specific values and operators used.
- Boolean expression uses true and false as constant values and the operators include AND and OR

  AND → ∧ OR → ∨

Eg :-

1) ((T OR F) AND (-F)) OR (T OR F)



Infix expression :-

- In this expression, highest precedence operator will be evaluated towards leaf node i.e, *, / and ^
- Lowest precedence operator will be evaluated towards root node i.e, + & -

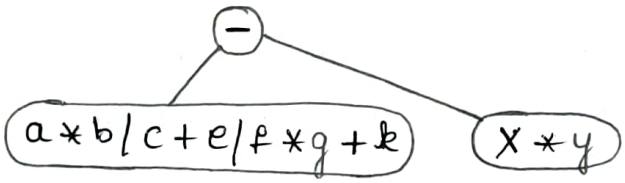
Eg :-

1) a * b / c + e / f * g + k - x * y

Associativity	
^	R → L
*, /	L → R
+, -	L → R

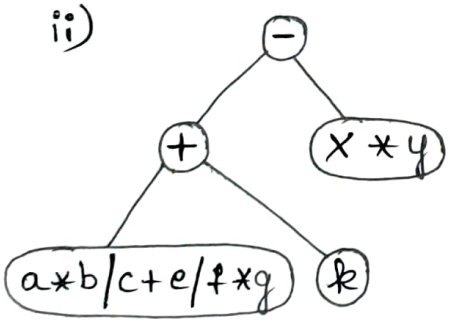
- In the above expression, multiple '+ and -' operators are present, so if we follow L → R associativity then '-' will be evaluated at last.
  ∴ '-' is the root node.

i)

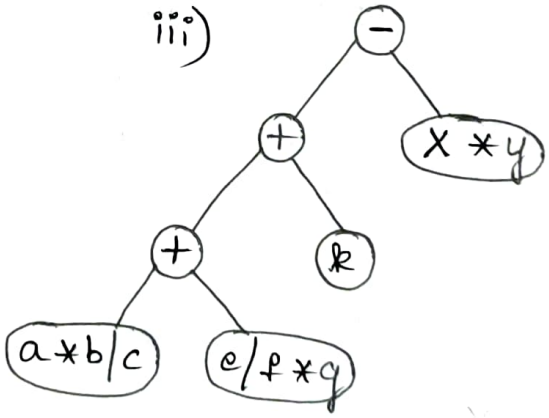


- Evaluate left subtree and then evaluate right subtree.
- Follow the same procedure based on the precedence and associativity of operators.

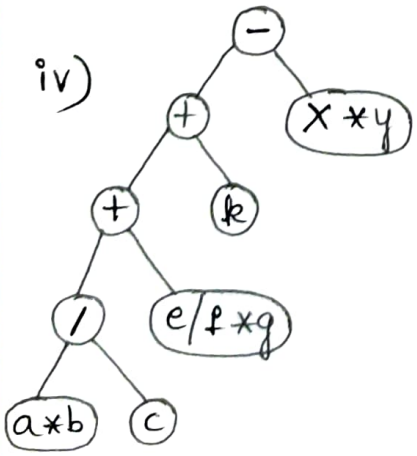
ii)



iii)

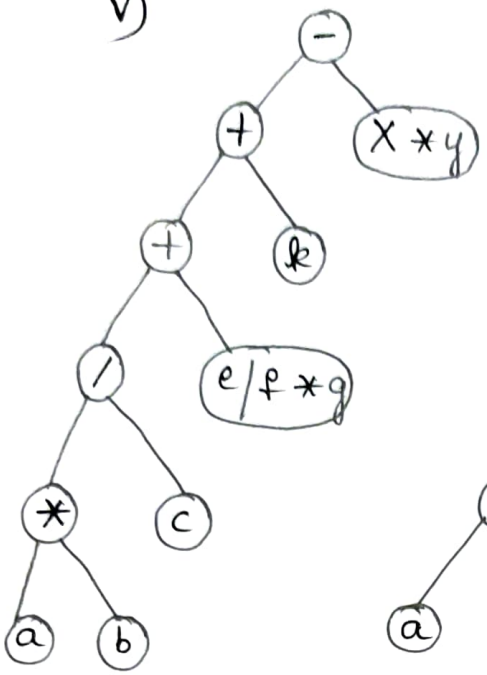


iv)

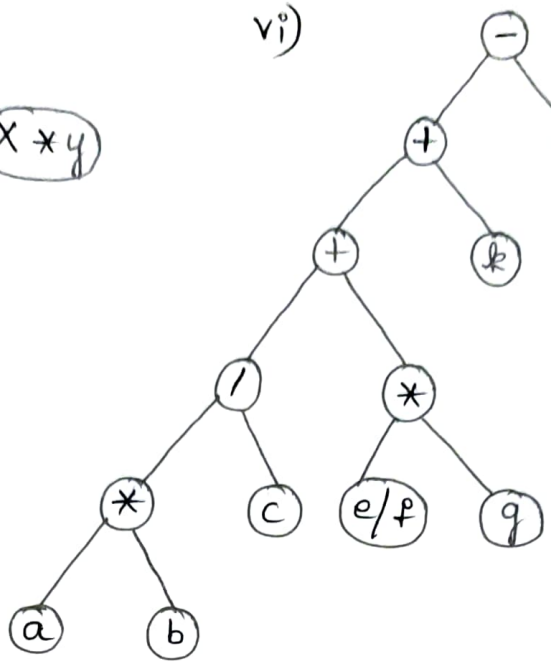


In a*b/c, * and / has same precedence so scan from L → R and '/' will be evaluated at last. Hence, it becomes the root node.

v)

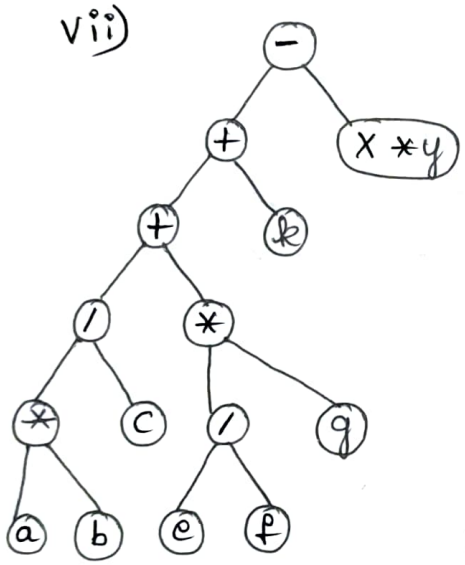


vi)

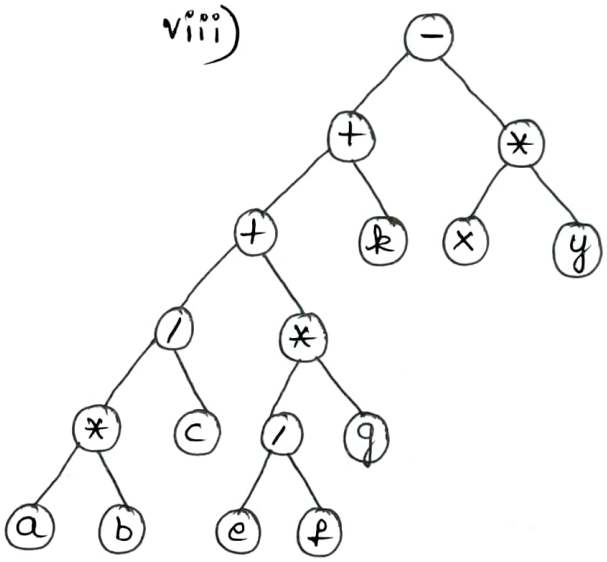


In e/f*g, */ has same precedence so scan from L→R and '*' will be evaluated at last. Hence, it becomes the root node.

vii)

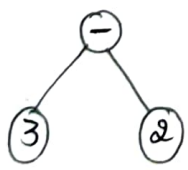


viii)

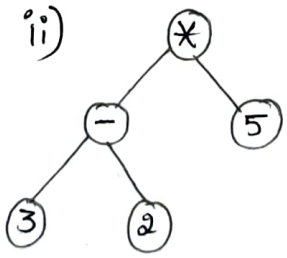


2) $((6+(3-2)*5) \wedge 2+3)$

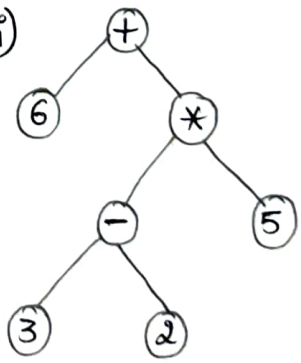
i)



ii)

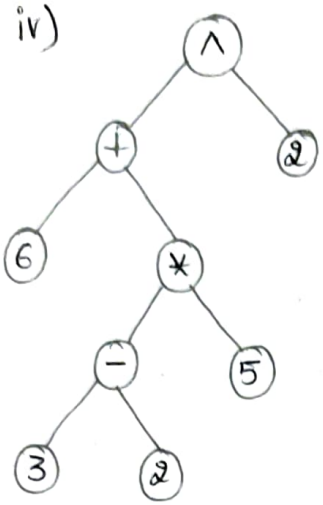


iii)



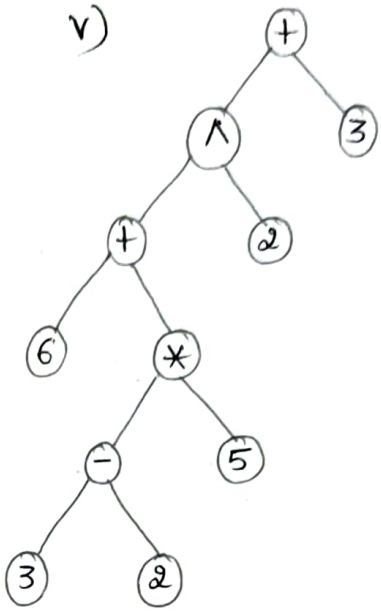
'*' has highest priority than '+' and highest priority operator should be evaluated towards leaf node.

iv)



'^' has highest priority than '+' and it cannot be a root node due to high priority. Hence, it will be having only '2' as a right child.

v)



Inorder / Infix expression

6+3-2*5^2+3

Preorder / Prefix expression

+^+6*-3252 3

Postorder / Postfix expression

632-5*+2^3+

Postfix Expression :-

- Using postfix expression, we can construct expression tree by the following steps

Step-1 :- Reading postfix expression. If the given expression is in infix, convert it into postfix.

Step-2 :- Read only one symbol at a time.

Step-3 :- If symbol is

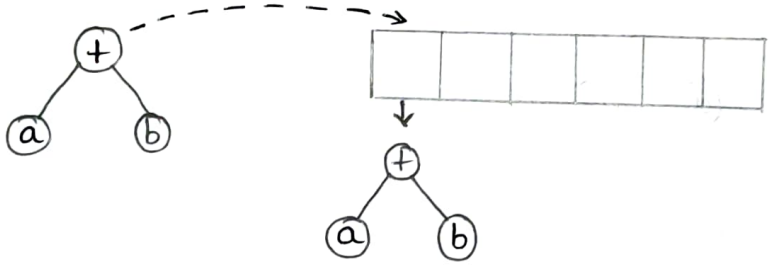
- operand → create a node and push pointer onto stack.
- operator → pop 2 pointers and form a new tree
  - First popped → Right child
  - Second popped → Left child
- Then push the pointer of this new tree onto stack.

Construct expression tree for the following expression :- (24)

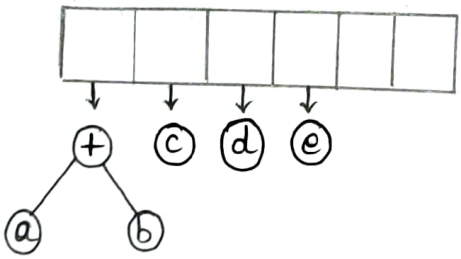
1) a b + c d e + * *

- check whether the given expression is in postfix form. If it is not in postfix form convert it.
- Above expression is in postfix form, so read first 2 operands and place it on stack.

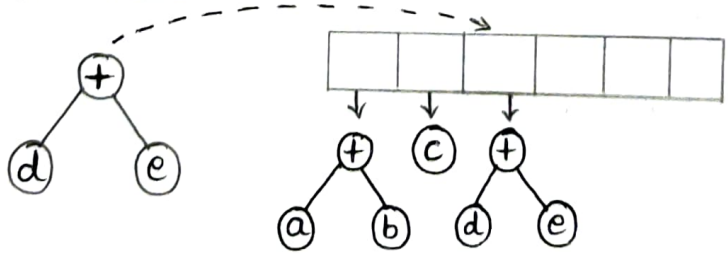
- Next character is '+', so pop 2 pointers from stack and place operator in between them for tree construction and push the newly constructed tree onto stack.



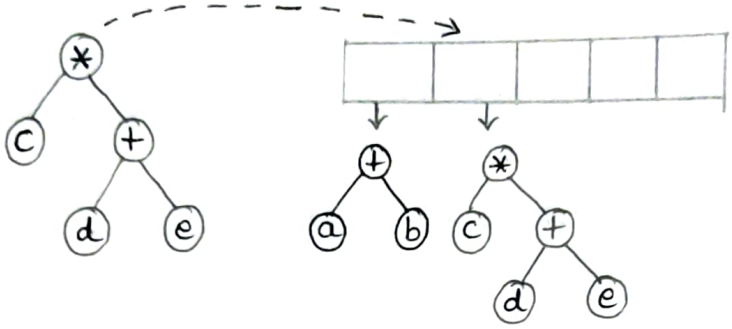
- Next characters are 'c' 'd' and 'e', so push all the operands onto stack one by one.



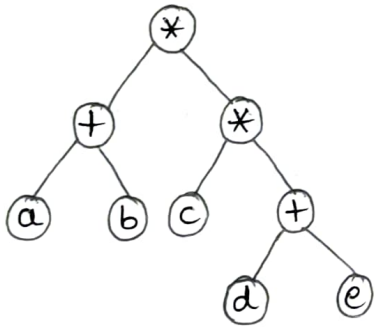
- Next, we found operator '+', pop 2 items from stack and place operator '+' in between them, push newly constructed tree onto stack.



- Next, we found operator '*' pop 2 items and construct new tree and push it onto stack.

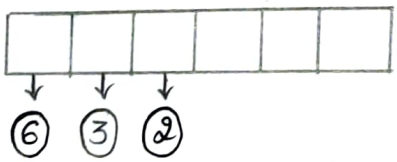


- Next, the last symbol is '*', pop 2 items and construct new tree.

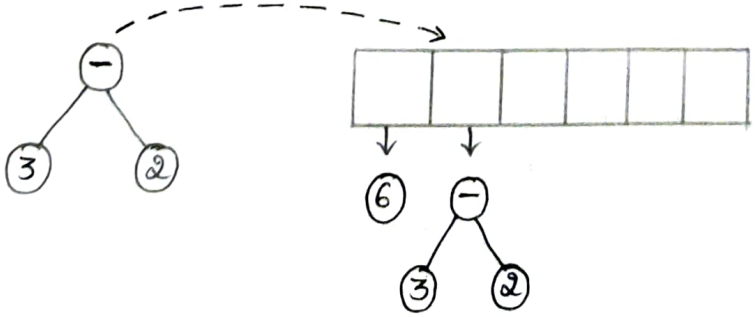


② 6 3 2 - 5 * + 1 $ 7 +

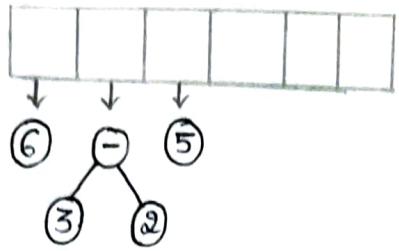
- Given expression is in postfix expression and first three characters are operand, so we can push them onto stack one by one.



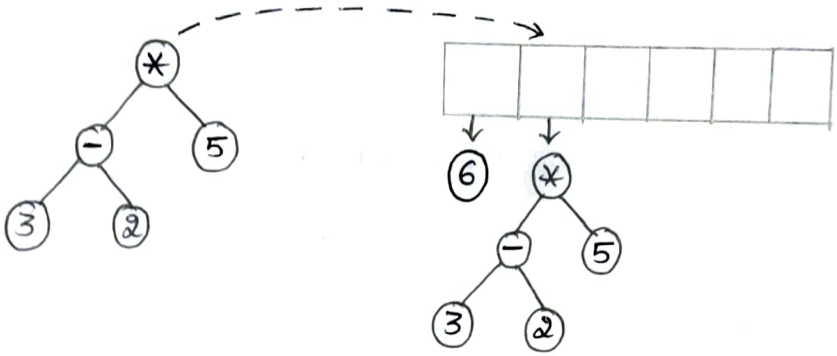
- Next operator '-', so pop 2 items & construct a new tree and push it onto stack.



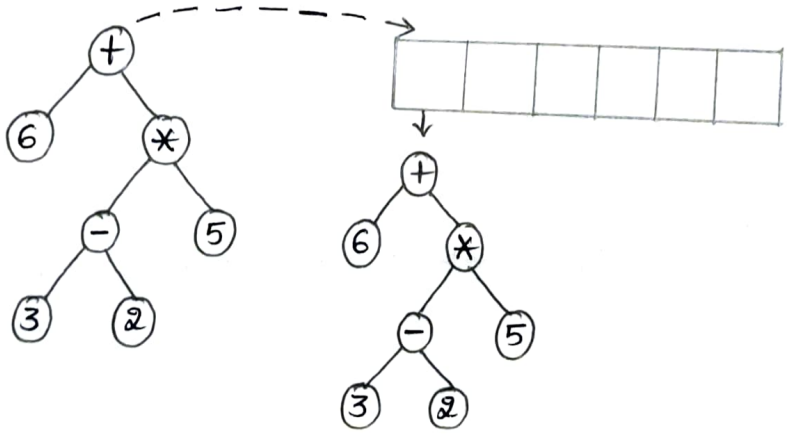
- Next symbol is '5', push it onto stack



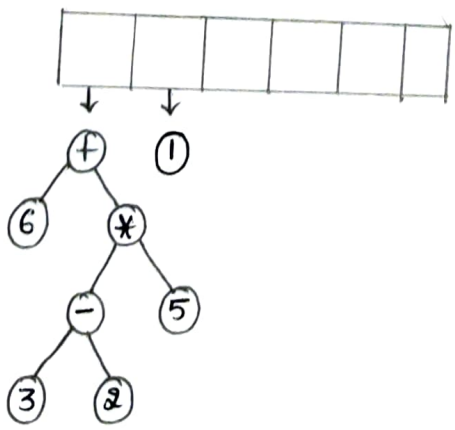
- Next symbol is '*', pop 2 items & construct a new tree and push it onto stack.



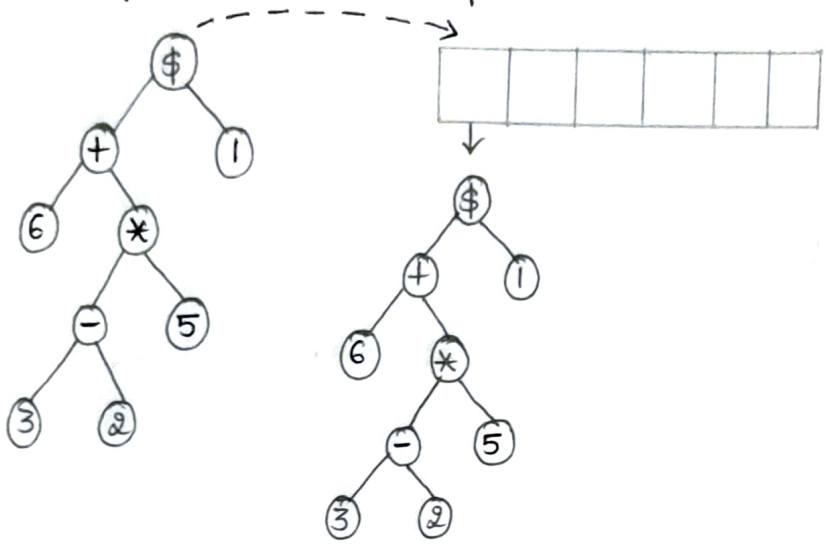
- Next symbol is '+', pop 2 items & construct a new tree and push it onto stack.



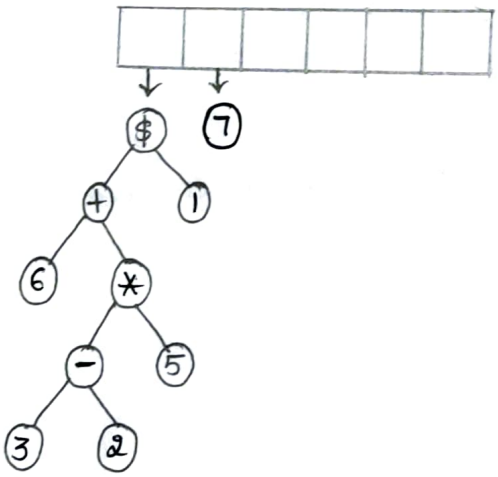
- Next symbol is '1', push it onto stack.



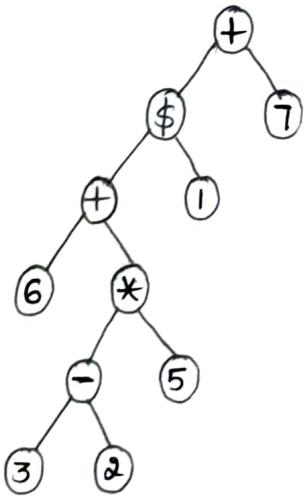
- Next symbol is '$', pop 2 items & construct a new tree.



- Next symbol is '7', push it onto stack.



- Last symbol is '+', pop 2 items and construct a new tree



3) $3+A*(7-6)/A+3$

- Given expression is in infix, so convert it into postfix expression.

$3+A*(7-6)/A+3$

$3+A*T_1/A+3$

$3+T_2/A+3$

$3+T_3+3$

T_4+3

T_5

Postfix Conversion

$T_1 = 76-$

$T_2 = AT_1*$

$T_3 = T_2A/$

$T_4 = 3T_3+$

$T_5 = T_43+$

- Then perform the back substitution

$T_5 = T_43+$

$= 3T_3+3+$

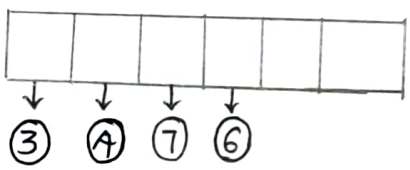
$= 3T_2A/+3+$

$= 3AT_1*A/+3+$

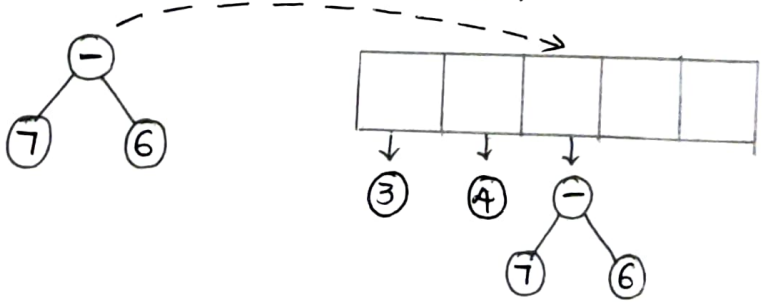
$= 3A76-*A/+3+$

∴ Postfix expression → $3476-*A/+3+$

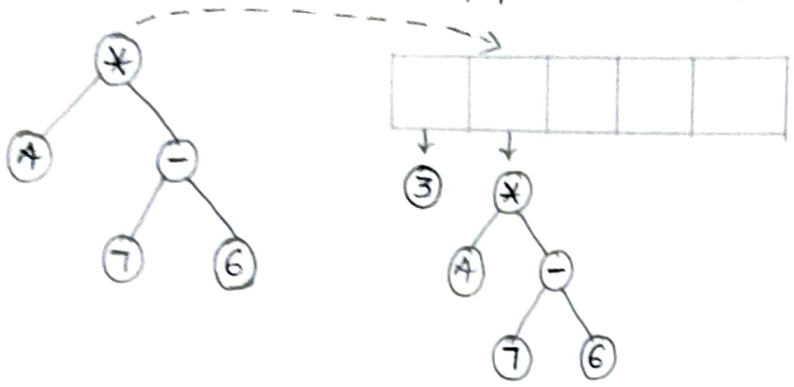
- Follow the same procedure to construct a expression tree using postfix expression.
- First 4 symbols are operands '3, 4, 7, 6', so push them onto stack one by one.



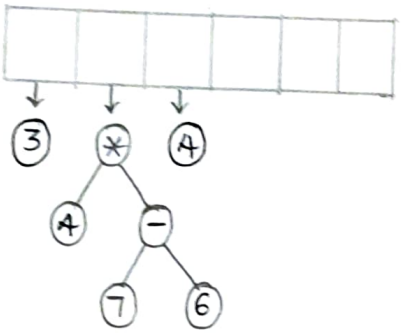
- Next '−' operator, so pop 2 items and construct a tree.



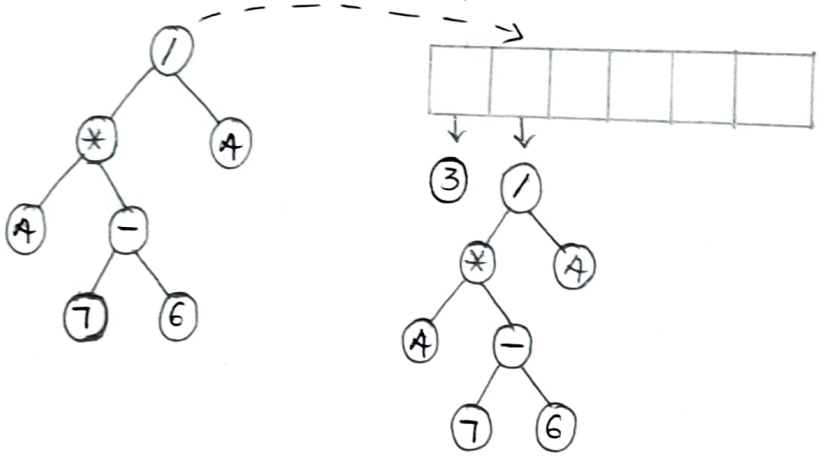
- Next '*' operator, so pop 2 items & construct a new tree.



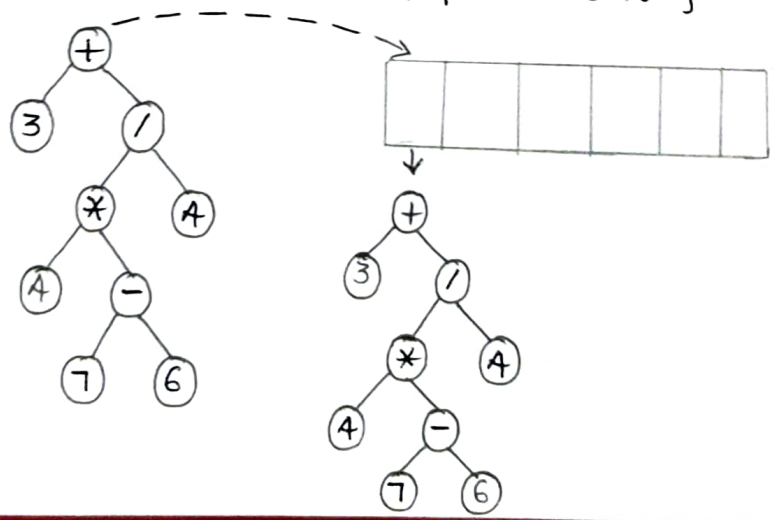
- Next 'A' operand, so push it onto stack.



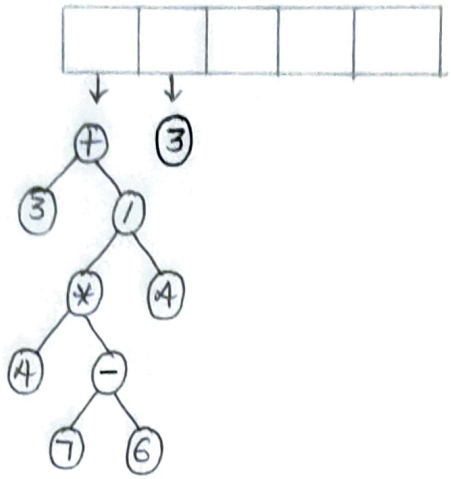
- Next '/' operator, so pop 2 items & construct a new tree.



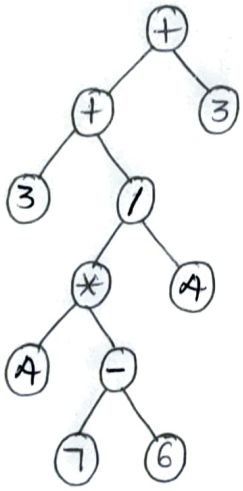
- Next '+' operator, so pop 2 items & construct a new tree.



- Next '3' operand, so push it onto stack.



- Next '+' operator, so pop 2 items and construct a new tree.



Inorder → 3+4*7-6/4+3

Preorder → ++3/*4-7643

Postorder → 3476-*4/+3+