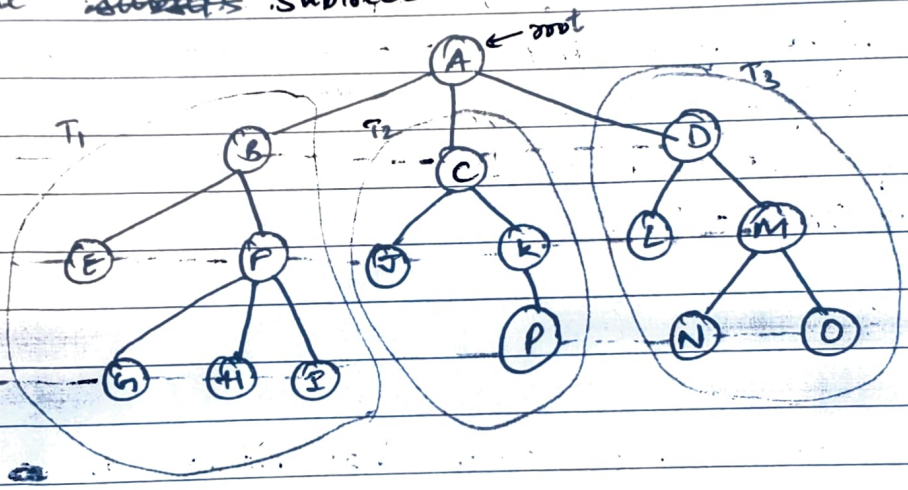
Trees

A tree is a non-linear, non-primitive data structure where in data is stored in a hierarchal fashion.

A tree is a finite set of one or more nodes that exhibits the parent-child relationship such that

1. There is a special node called the root node.
2. The remaining nodes are partitioned into disjoint subsets of nodes like T_1, T_2 upto n, termed as the subtrees.



(1) Siblings

Two or more nodes having a common parent.

eg: J & K are siblings of C.

(2) Ancestors

The nodes obtained in the path from a specific node say x while moving upwards towards the root node are termed as ancestors of x.

eg: Ancestors of node H are F and B.

(3) Degree of a node

The number of sub trees of a given node is termed as the degree of a node.

eg: degree of node N is 0. F is 3.

↳ have degree of 0

④ **Leaf Node**

The nodes in a tree which has degree of 0 is termed as leaf nodes.

eg: E, G, H, I, P, N, O are leaf nodes

⑤ **Level of a tree (node).**

The distance of a node from the root node is termed as the level of a tree.

eg: Level of root node is 0

⑥ **Height of a tree.**

The number of nodes processed to reach the leaf node present in last level from the root node is termed as the height of the tree.

eg: Here height is 4.

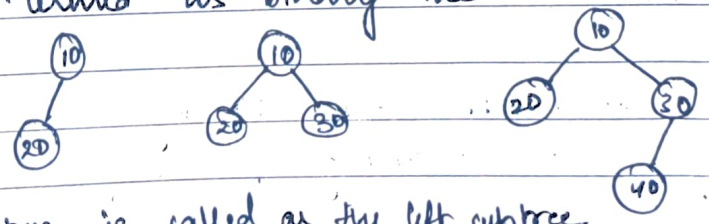
Height of root node is 1.

Ht = max level + 1.

• **Binary tree**

A tree in which each node has either 0, 1 or 2 subtrees is termed as binary tree.

eg:



The first subtree is called as the left subtree.

The second " " right subtree.

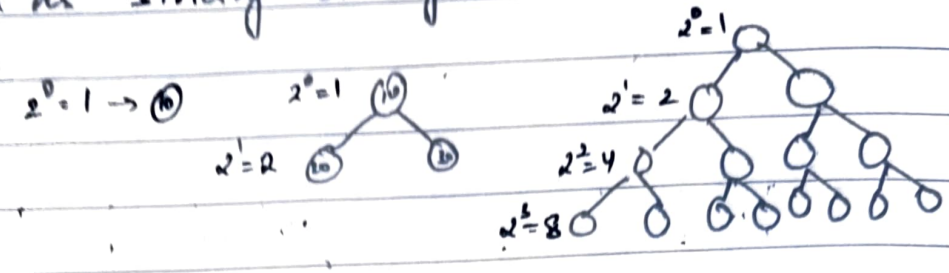
Types of Binary trees

① Strictly binary tree
② Complete binary tree
③ Skewed binary tree
④ Expression tree
⑤ Binary search tree

Strictly binary tree (Full bin tree)

A bin tree having 2^i nodes at any given level i is referred as strictly binary tree.

eg:

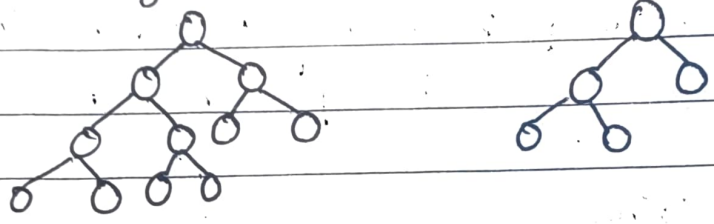


(2) Complete bin tree

A bin tree in which at every level apart from the last level if it is completely filled we term it as complete bin tree.

In the last level the nodes should be filled from left to right

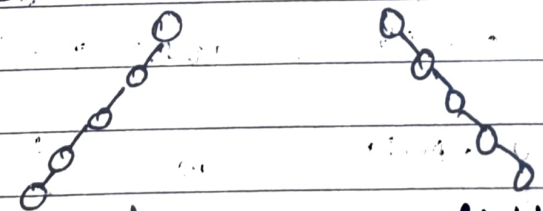
eg



(3) Skewed binary tree

If a bin tree is built only on one side its called as skewed bin tree.

eg:



Left skewed

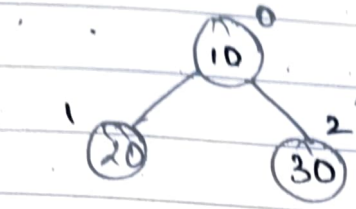
Right skewed.

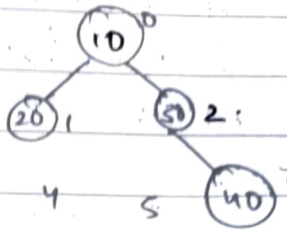
Representation of a binary tree

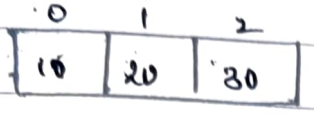
A bin tree can be represented in 2 ways:

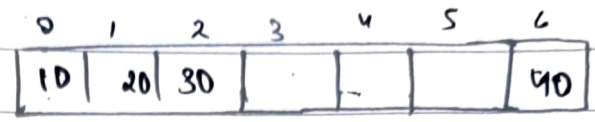
1. Array rep^n
2. List rep^n

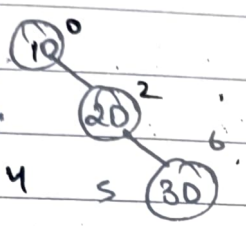
① Array representation

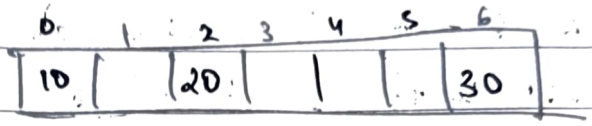












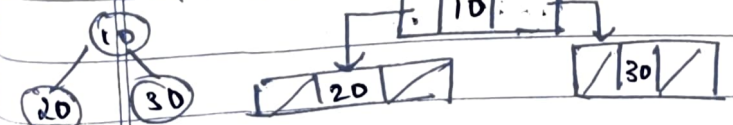
not useful cause wasting too much space.

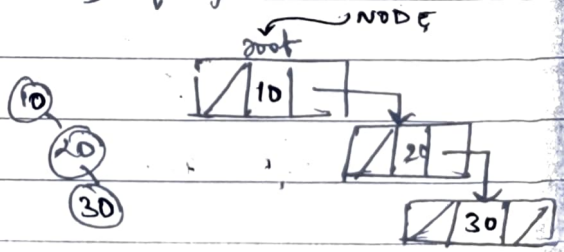
* If a tree is a complete binary tree then it is efficient to use arrays to represent a tree.

② List representation

We represent each node with the following structure def^n

```
struct node
{ int info;
  struct node *left;   //address of left subtree/child
  struct node *right;  //address of right child
};
typedef struct node *NODE;
```





Operations on Binary tree.

① Traversal operation

It is a method of visiting all the nodes of a tree exactly once.

Types of traversal:

1. Pre order traversal
2. Inorder "
3. Post order "

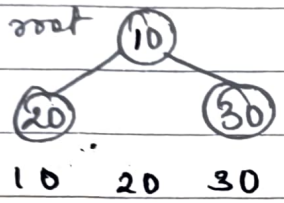
} can be used only if tree is binary.
If not binary, convert it into bin tree & then perform traversal.

1. Preorder traversal (NLR)

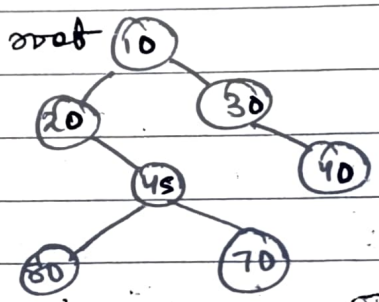
root left right

The recursive def^n of the preorder traversal is

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



10 20 30



10 20 45 80 70 30 40

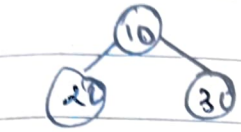
A recursive func^n for preorder:

```
void preorder (NODE root)
{
    if(root != NULL)
    { printf("%d \t", root->info);
      preorder(root->left);
      preorder(root->right);
    }
}
```

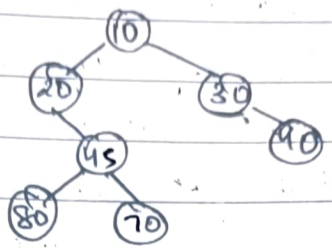
(2) Inorder traversal (LNR)

The recursive def^n of inorder traversal is:

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



20 10 30



20 80 45 70 10 30 40

Recursive def^n:

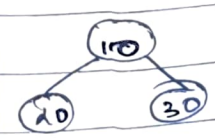
```
Void inorder (NODE root)
{  if (root != NULL)
     { inorder (root->left);
       printf ("%d \t", root->info);
       inorder (root->right);
     }
}
```

I should be at 4th position.

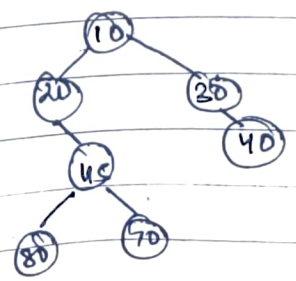
(3) Postorder traversal (LRN)

The recursive def^n to perform postorder traversal is:

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



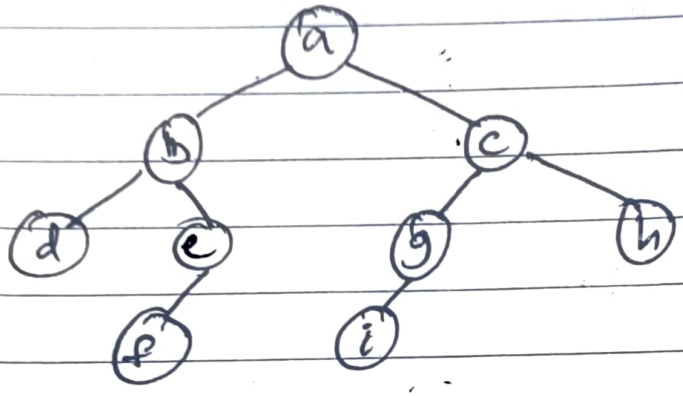
20 30 10



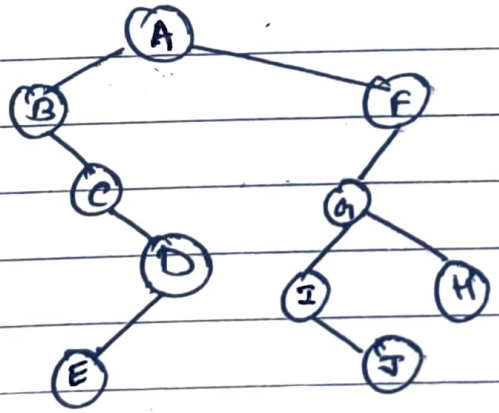
80 70 45 20 40 30 10

```
void postorder (NODE root)
{   if (root != NULL)
    {   postorder (root->left);
        postorder (root->right);
        printf ("%d\t", root->info);
    }
}
```

Q Perform the traversal for given graph.



preorder → a b d e f c g i h
inorder → d b f e a i g c h
postorder → d f e b i g h c a

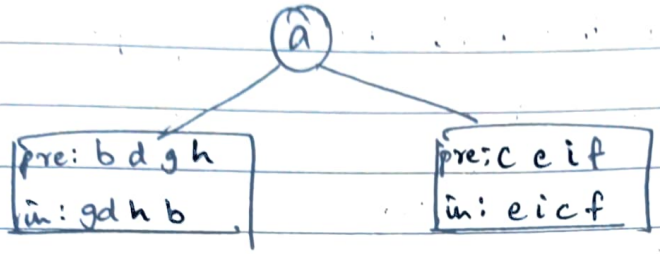


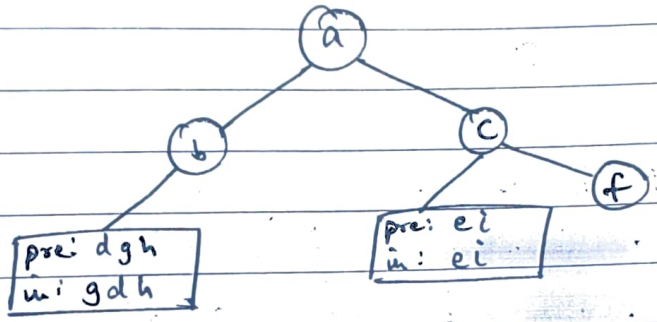
NLR pre → A B C D E F G I J H
LNR in → B C E D A I J G H F
LRN post → E D C B J I H G F A

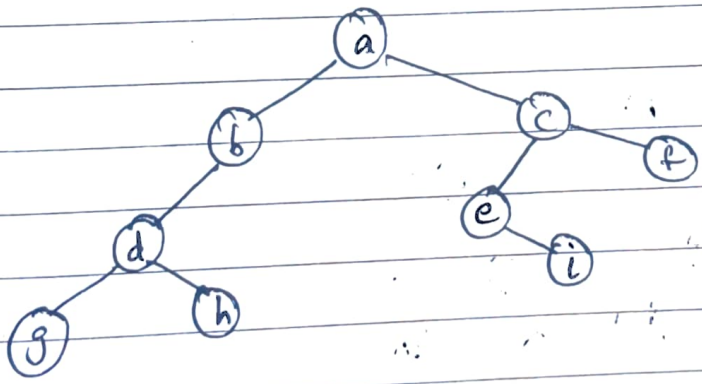
comb^n of pre & in
↑ or post & in

Q Construct a bin tree from a given traversal.

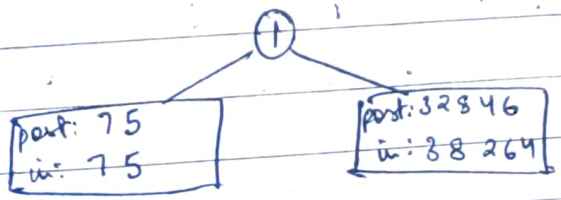
root
↑
preorder → a b d g h c e i f
inorder → g d h b a e i c f



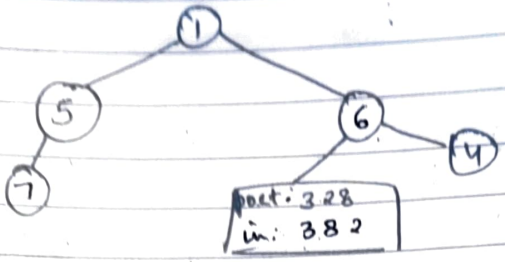


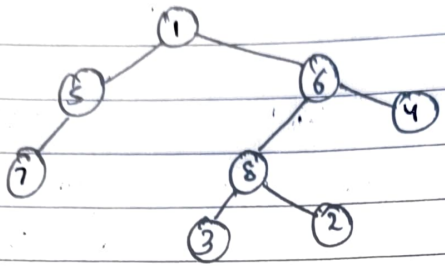


Q post: 7 5 3 2 8 4 6 1 LRN
in: 7 5 1 3 8 2 6 4 LNR



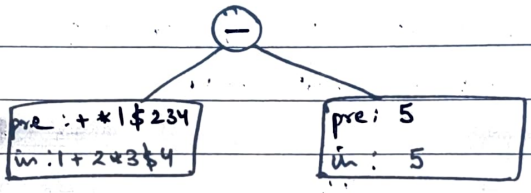
others are operators

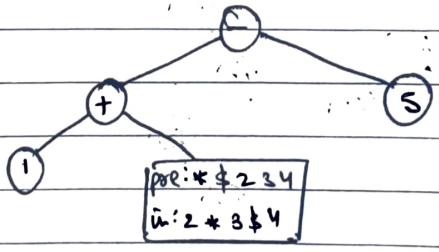


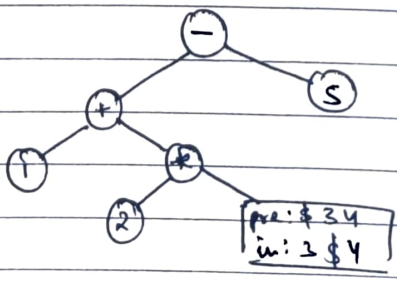


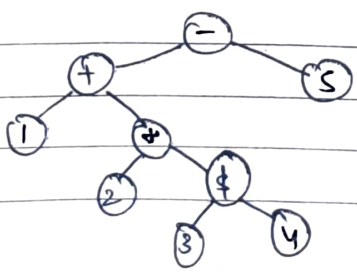
8 preorder: − + * 1 $ 2 3 4 5 NLR

inorder: 1 + 2 * 3 $ 4 − 5 LNR







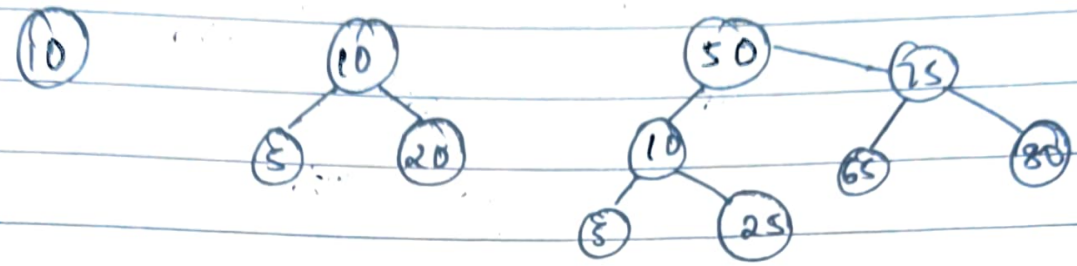


(1 + (2 * (3 $ 4)) − 5

Binary search tree

It is a binary tree such that for each node x in the binary tree, the elements in the left subtree are less than data of x and the elements in the right subtree are greater than / equal to data of x.

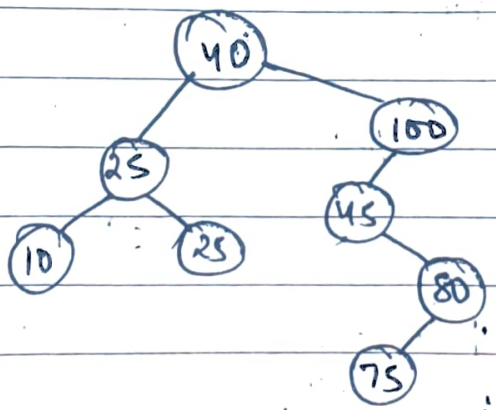
eg:



Q Construct a BST for a given set of elements:

40 100 25 45 25 80 10 75

→ 1st element always root node



Compare 100 with 40

25 < 40

45 with 40

45 < 100

25 < 40

25 < 25

80 > 40

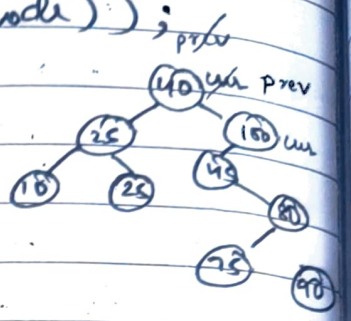
80 < 100

80 > 45

10 < 40

Function to create a bin search tree.

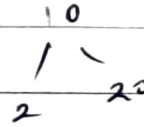
```
NODE createBST (NODE root, int item)
{  NODE temp, cur, prev;
   temp = (NODE) malloc (sizeof (struct node));
   temp -> data = item;
   temp -> left = NULL;
   temp -> right = NULL;
   if (root == NULL)
      return temp;
   cur = root;
   prev = NULL;
   while (cur != NULL)
    {  prev = cur;
       if ( item < cur -> data )
              cur = cur -> left;
       else
              cur = cur -> right;
    }
   if (prev -> data > item)
       prev -> left = temp;
   else
       prev -> right = temp;
   return root;
}
```



LAB 7 Create BST, traverse & delete a node.

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

```
typedef struct node *NODE;
// create BST
// write preorder, inorder, postorder
NODE delete(NODE root, int key)
{ NODE temp;
  if(root == NULL)
     return root;
  if(key < root->data)
     root->lft = delete(root->lft, key);
  else if(key > root->data)
     root->right = delete(root->right, key);
  else
  { if(root->lft == NULL)
    { temp = root->right;
      free(root);
      return temp;
    }
    if(root->right == NULL)
    { temp = root->lft;
      free(root);
      return temp;
    }
    temp = inordersuccessor(root->right)
    root->data = temp->data;
    root->right = delete(root->right, temp->data);
  }
  return root;
}
```

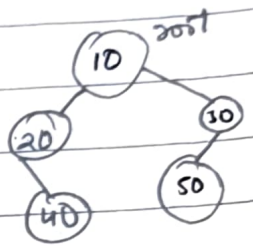


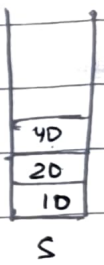
```
int main()
{
    NODE root = NULL;
    int item, key, ch;
    for(;;)
    { printf("1. Insert \n 2. Delete \n 3. Preorder \n
              4. Inorder \n 5. Postorder \n 6. Exit \n");
      printf("Read choice \n");
      scanf("%d", &ch);
      switch(ch)
      { case 1: printf("\n Enter element to be inserted: ");
                scanf("%d", &item);
                root = creatBST(root, item);
                break;

        case 2: printf("\n Enter the node to be deleted \n");
                scanf("%d", &key);
                root = delete(root, key);
                break;

        case 3: printf("\n Preorder traversal is \n");
                preorder(root);
                break;
        case 4: printf("\n Inorder traversal is \n");
                inorder(root);
                break;
        case 5: printf("\n Postorder traversal is \n");
                postorder(root);
                break;
        default: exit(0);
      }
    }
}
```

```
NODE inordersuccessor (NODE root)
{ NODE cur = root;
  while (cur->left != NULL)
      cur = cur->left;
  return cur;
}
```

A C function to perform iterative preorder traversal

```
void preorder_iter (NODE root)
{ NODE cur, s[20]; int top = -1;
  if (root == NULL)
    { printf ("\n Empty tree");
      return;
    }
  cur = root;
  while (1)
  { while (cur != NULL)
      { printf ("%d\t", cur->data);
        s[++top] = cur;
        cur = cur->left;
      }
    if (top != -1)
    {
      cur = s[top--];
      cur = cur->right;
    }
    else
      return;
  }
}
```





Write a iterative func to perform the iterative inorder traversal

```
void inorder_iter(NODE root)
{   NODE cur, s[20];
    int top = -1;
    if (root == NULL)
    {  printf("\n Empty tree");
       return;
    }
    cur = root;
    while(1)
    {  while(cur != NULL)
       {    s[++top] = cur;
            cur = cur -> left;
       }
       if (top != -1)
       {  cur = s[top--];
          printf("%d\t", cur -> data);
          cur = cur -> right;
       }
       else
          return;
    }
}
```

Q Write an iterative func to perform postorder traversal

```
void postorder_iter(NODE root)
{   NODE cur, s[20];
    int top = -1;
    if (root == NULL)
    {  pf("\n Empty tree");
       return;
    }
```

```
cur = root;
while(1)
{   while (cur != NULL)
    {  s[++top] = cur;
       cur = cur -> left;
    }
    if (top != -1)
    {  cur = s[top--];
       cur = cur -> right;
    }
    else
        printf ("%d\t", cur -> data);
}
}
```

2) To write a C func to check or to find 2 binary trees are same or not.

```
int issameBT (NODE root1, NODE root2)
{

  if (root1 == NULL && root2 == NULL)
      return 1;
  if ((root1 != NULL && root2 != NULL)
      return root1->data == root2->data
    && issameBT (root1->left, root2->left)
    && issameBT (root1->right, root2->right))
}
```

3) To check whether 2 BST are same or not.

```
int issameBST (NODE root1, NODE root2)
{  if (root1 == NULL && root2 == NULL)
      return 1;
   if (root1 != NULL && root2 != NULL)
```

8 To count the no. of nodes in a binary tree.

```
int count = 0          // is a global variable
count_nodes (NODE root)
{
    if (root == NULL)
        return 0;
    count++;
    count_nodes (root->left);
    countnodes (root->right);
}
```

8 To count the no. of leaf nodes in B.T

```
int count = 0;
void countleaf (NODE root)
{
    if (root == NULL)
        return;
    countleaf (root->left);
    if (root->left == NULL && root->right == NULL)
        count++;
    countleaf (root->right);
}
```

9 To seach a key in BST

```
NODE search (NODE root, int key)
{
    if (root == NULL)
        return NULL;
    if (key == root->data)
        return root;
```

```
    if (key < root->data)
        return search (root->left, key);
    return search (root->right, key);
}
```

Q To find the height of a given BT

```
int max (int a, int b)
{
    if (a > b)
        return a;
    return b;
}

int height (NODE root)
{
    if (root == NULL)
        return -1;
    if (root != NULL)
        return 1 + max (height(root->left), height(root->right));
}
```

Q To find the max element in BST

```
NODE max ( NODE root )
{
    NODE cur = root;
    while (cur->right != NULL)
        cur = cur->right;
    return cur;
}
```

Expression Tree

Expression tree is a binary tree wherein a give infix expression can be represented as a binary tree with operands as leaf nodes and operators as internal nodes.

Algorithm to construct an exp tree.

1. Scan the given expression from left to right.
2. Initialize 2 stacks namely a tree stack and operator stack.
3. If the scanned symbol is
   (a) an operand - construct a node for the operand & push the node onto the tree stack.
   (b) an operator - if operator stack is empty or the precedence of the operator on the top of the operator stack is less than the precedence of the scanned operator, construct a node for the operator & push it onto the operator stack.
   else pop 2 nodes from the tree stack & attach them as the right & the left child & push the operator node onto the tree stack and push the scanned operator onto the operator stack.
   (pop an operator node from the operator stack).
4. Until the operator stack becomes empty, pop an operator node from the operator stack and 2 nodes from the tree stack, attaching them as the right and the left child and push the operator node onto the tree stack.
5. Return tree stack to get your exp. tree.