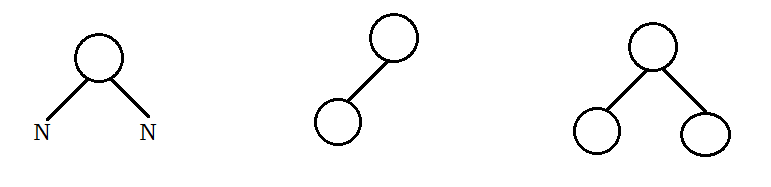
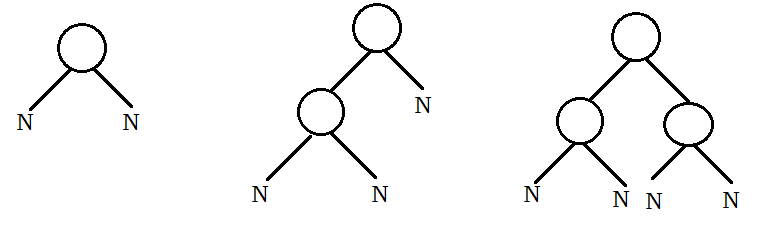
**Binary Search Tree**

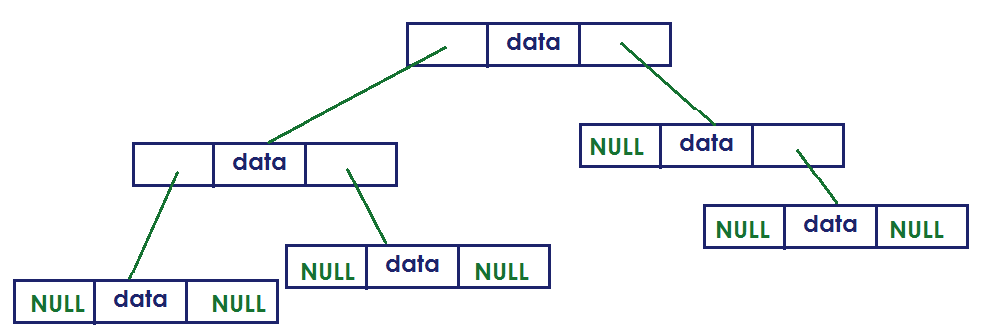
* BST is a non-linear data structure.
* One data element is connected to multiple elements in this structure.
* We represent the data in nodes.
* Every node has 3 fields
  1. Data filed
  2. Left child
  3. Right child
* In BST, every node has at most 2 children.



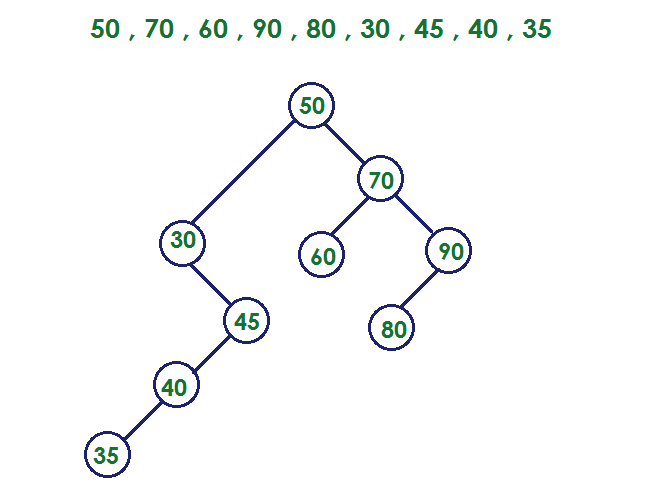
**The tree with N nodes having N+1 null nodes.**



**Tree memory representation with NULL nodes:**



* We store information into BST by comparing with Parent node.
* Least value is connected to left side of Parent node.
* Highest value is connected to right side of Parent node.
* BST not allow duplicates.
* Keys(elements) must be unique to store the data.



**Construct BST with these node values:**

50, 20, 80, 10, 40, 30, 35, 60, 70

**Node structure:** We represent the node using user data type called structure.

struct Node

{

int data;

struct Node \*left;

struct Node \*right;

};

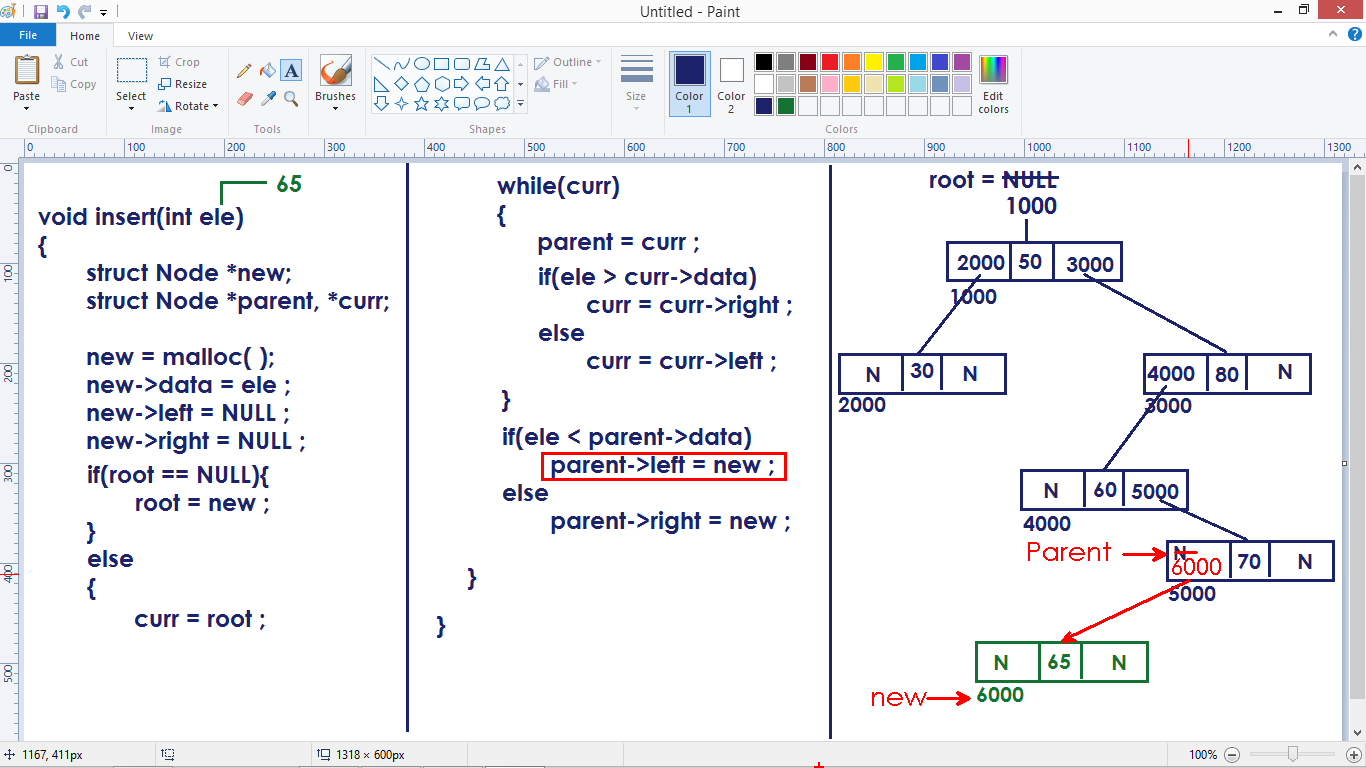
struct Node \*root=NULL;

**Operations:**

1. Insert
2. Delete
3. Traverse

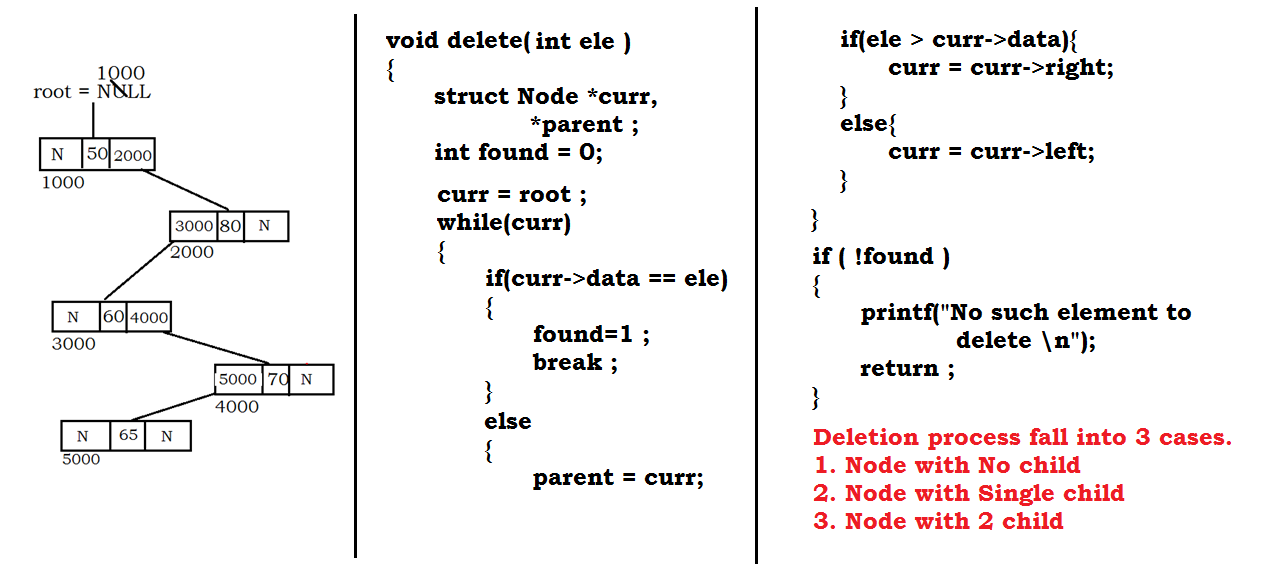
**Insertion:**

* Construct the Node.
* Place the Node data.
* Find the Parent node in the tree.
* Connect to Left or Right depends on the value of new node.

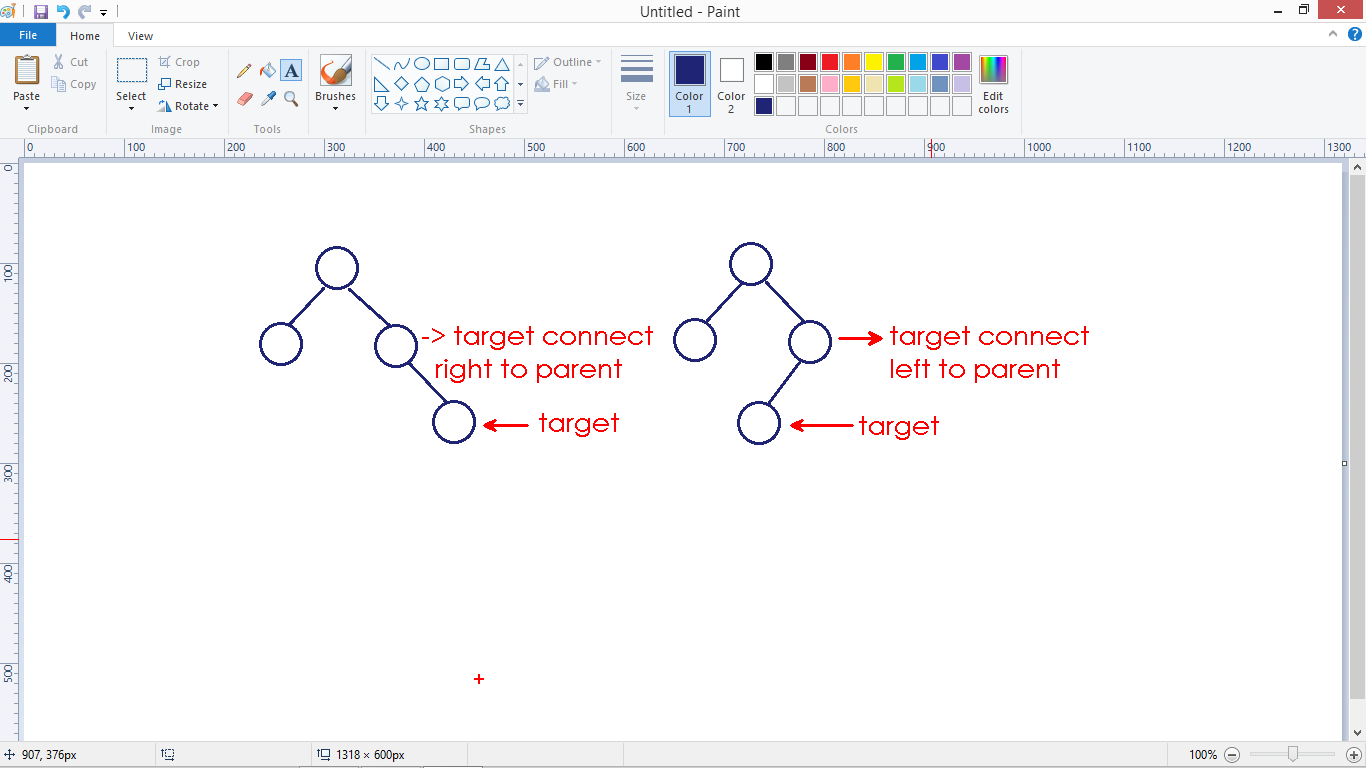


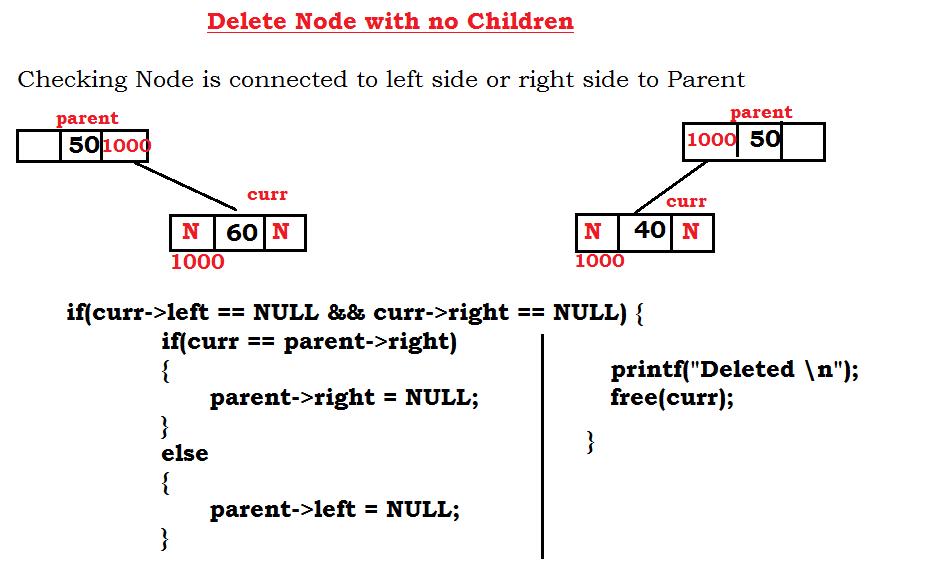
**Node deletion:**

* We need to find whether the element is present or not.
* If the element is present, we need to delete the node and re-arrange other nodes.
* If not present, display “Element not Found”.



**Case 1:** Deleting node with no child





if(curr->left==NULL && curr->right==NULL)

{

if(curr==parent->right){

parent->right=NULL;

}

else{

parent->left=NULL;

}

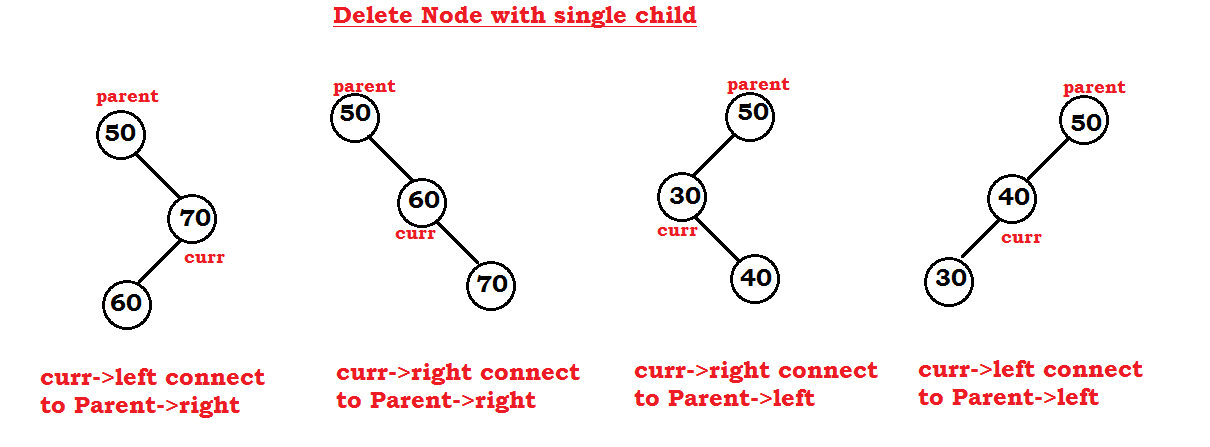
printf(“Deleted \n”);

free(curr);

}

**Case 2:** Deleting element that has single child

* We need to consider the following 4 situations to delete the node with single child.
* After deletion, we need to connect the node which is connected to the deleted node to parent.



if((curr->left != NULL && curr->right==NULL) || (curr->right != NULL && curr->left==NULL))

{

if(curr->left != NULL && curr->right==NULL)

{

if(curr == parent->right)

{

parent->right = curr->left;

}

else

{

parent->left = curr->left;

}

curr->left=NULL;

free(curr);

}

else

{

if(curr == parent->right)

{

parent->right = curr->right;

}

else

{

parent->left = curr->right;

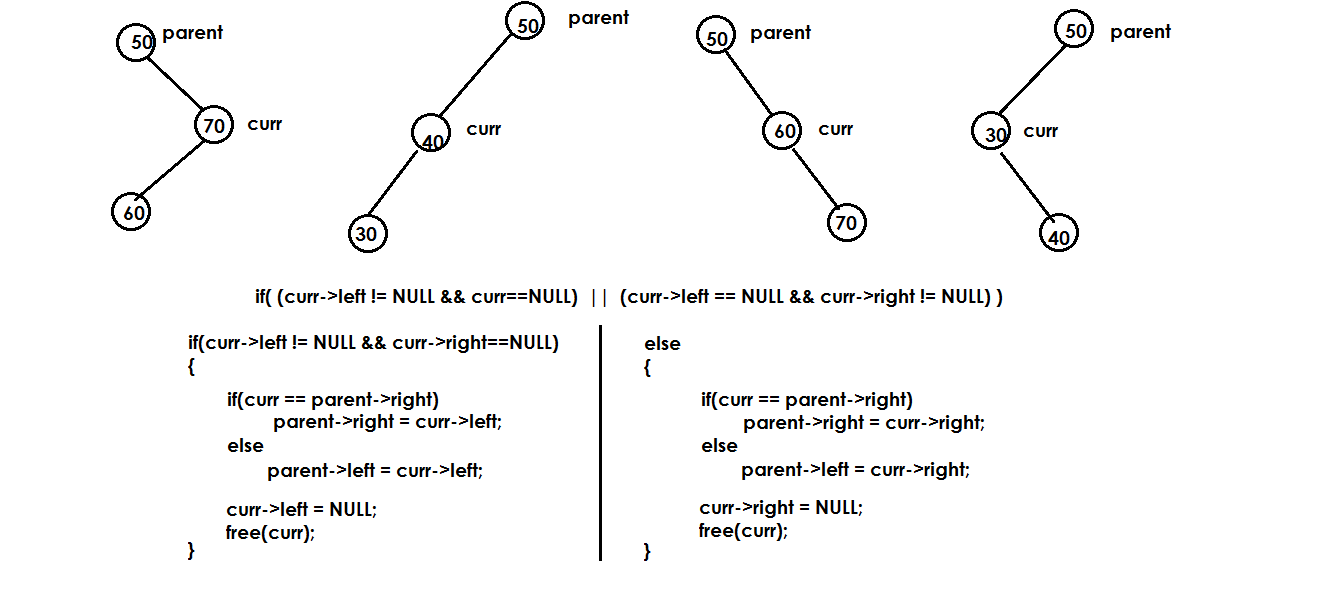
}

curr->right=NULL;

free(curr);

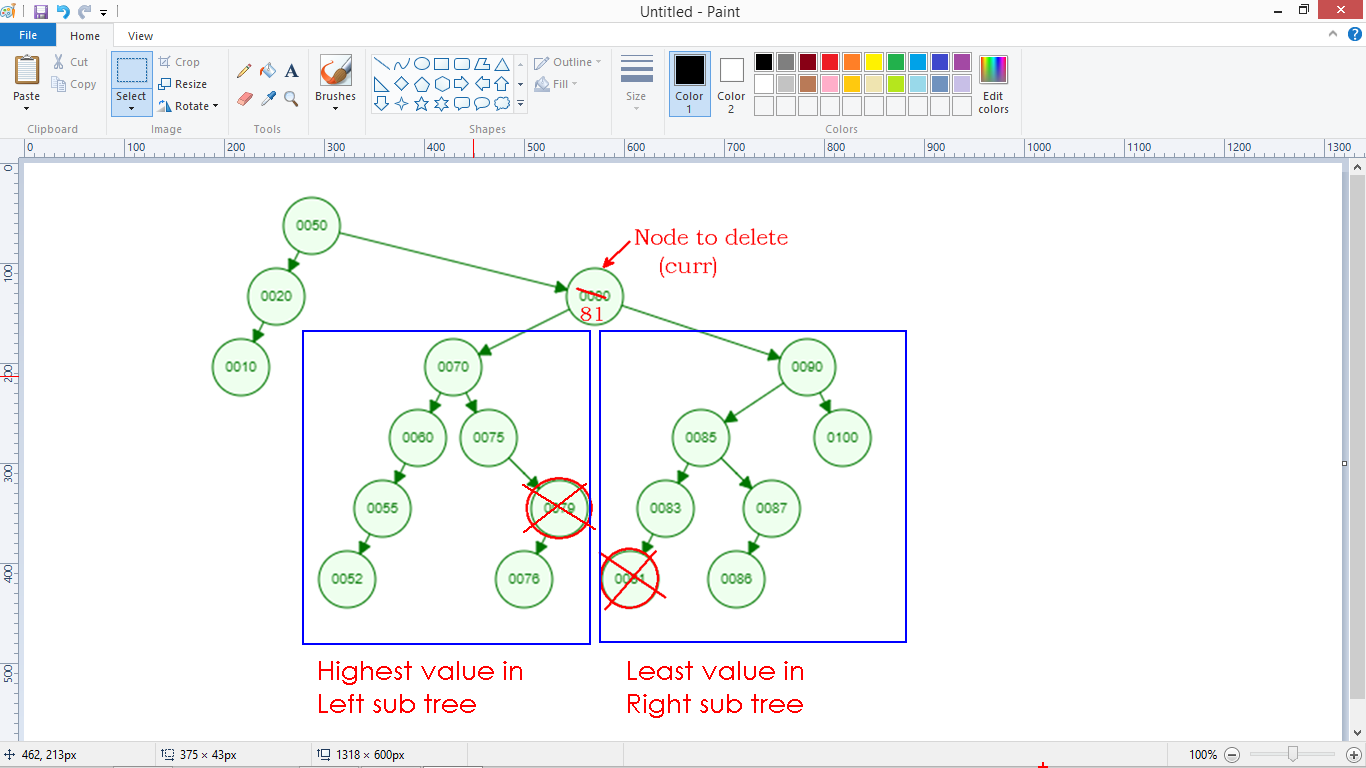
}

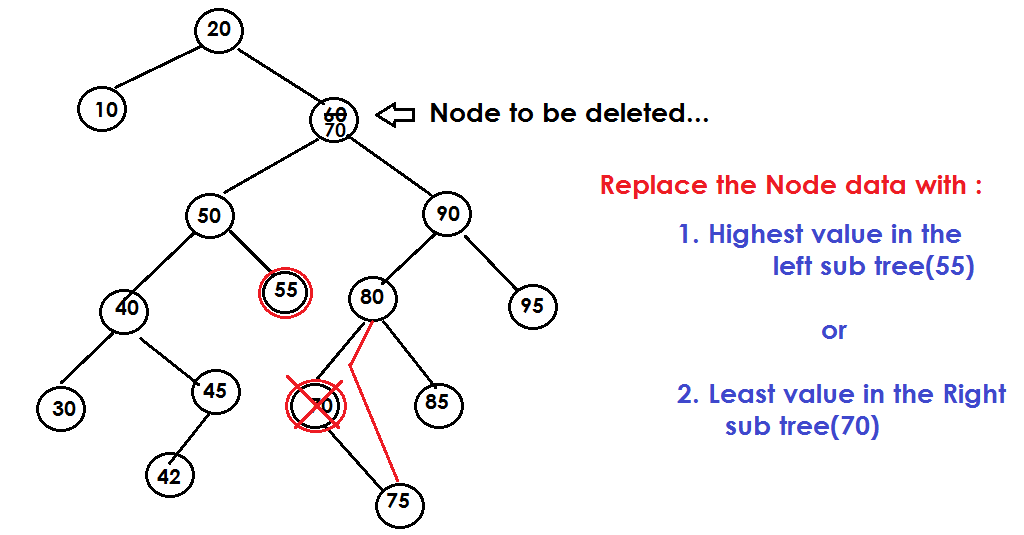
}



**Case 3:** Deleting the node has 2 children.

* Replace the current node data with
  + Least element in the right sub tree or
  + Highest element in the left sub tree.
* Removes the data swapped node.





**Here we are replacing the least value in Right sub tree:**

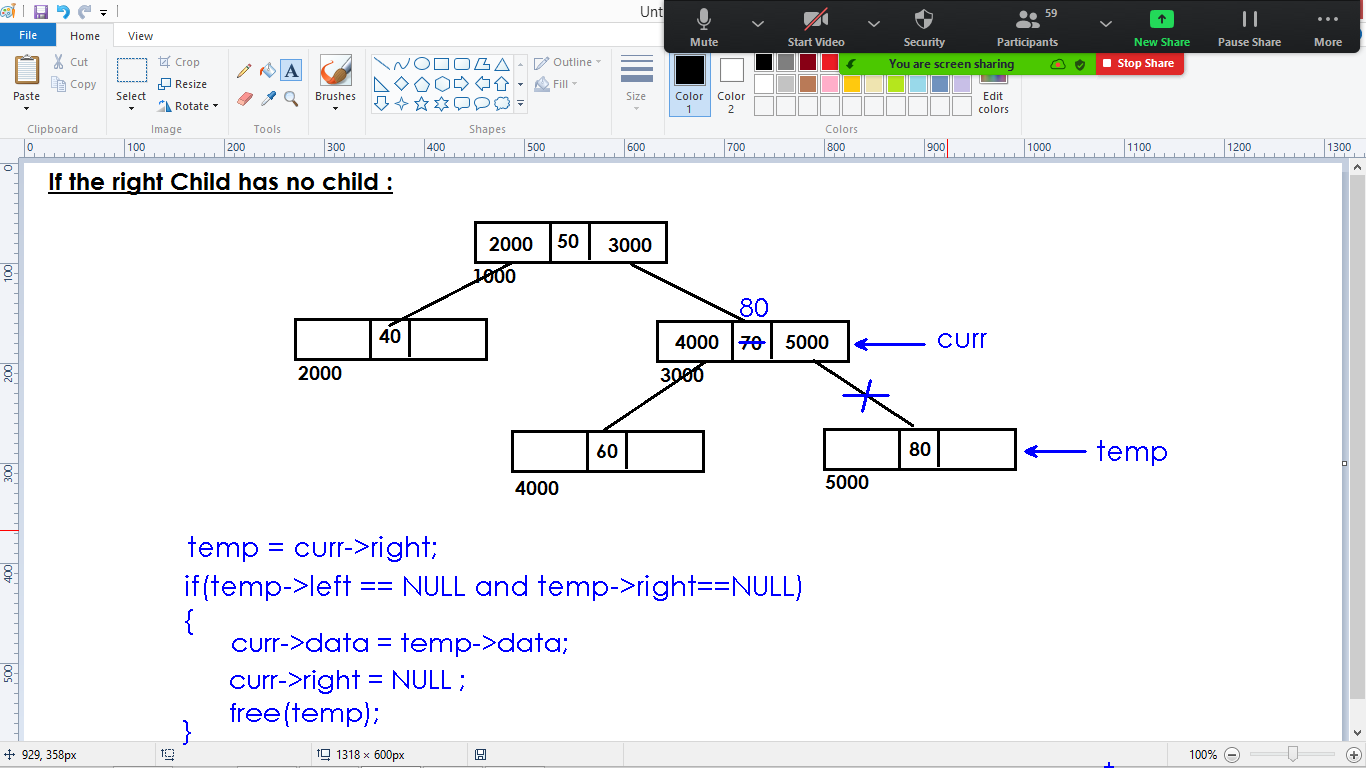
**We confirm that Node has 2 Children with following condition:**

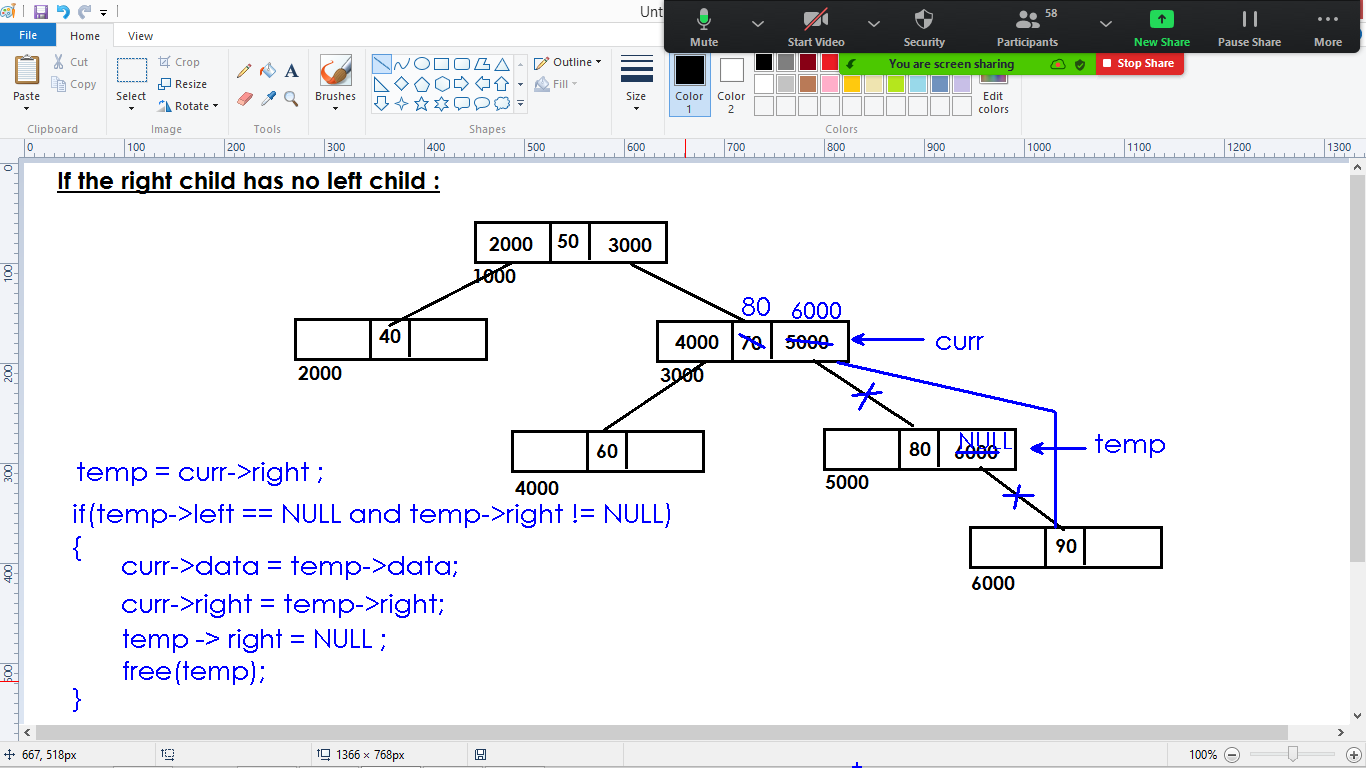
if(curr->left!=NULL && curr->right != NULL)

{

Logic...

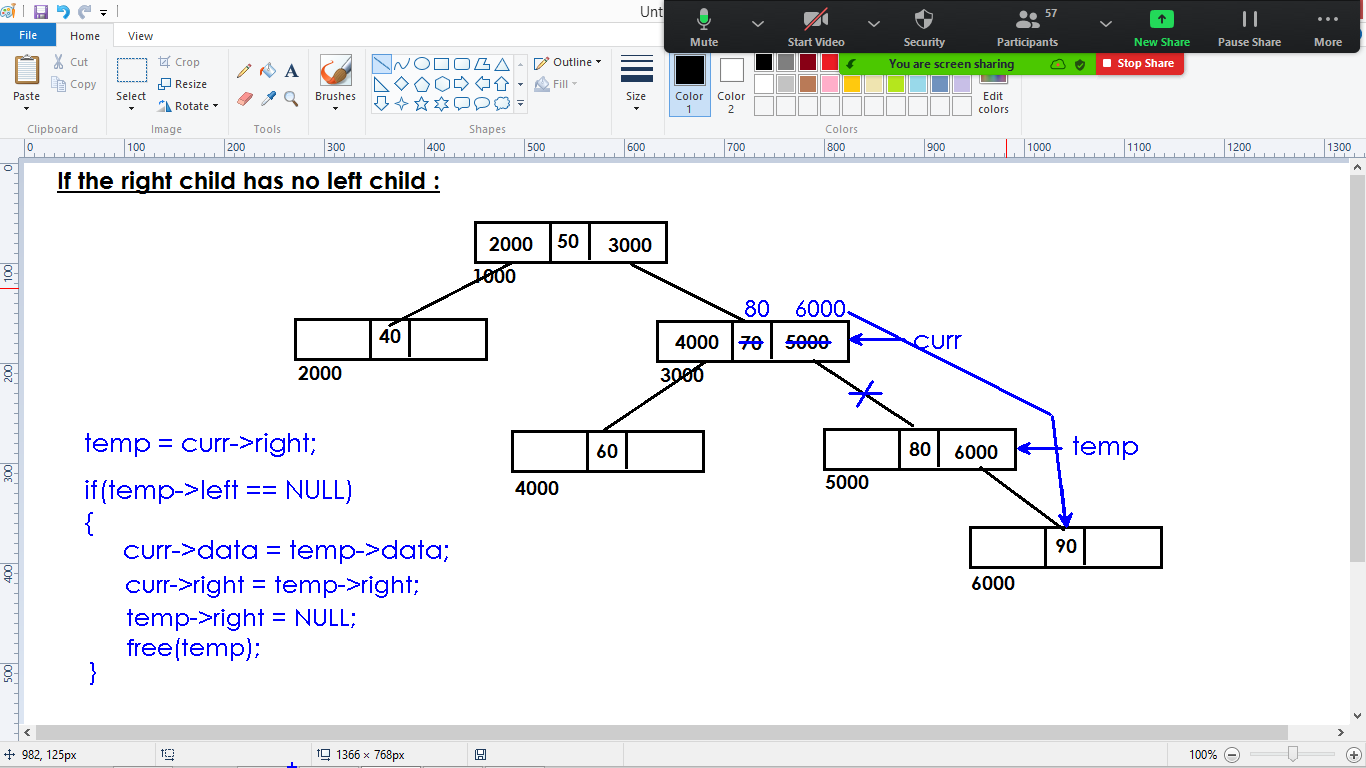
}

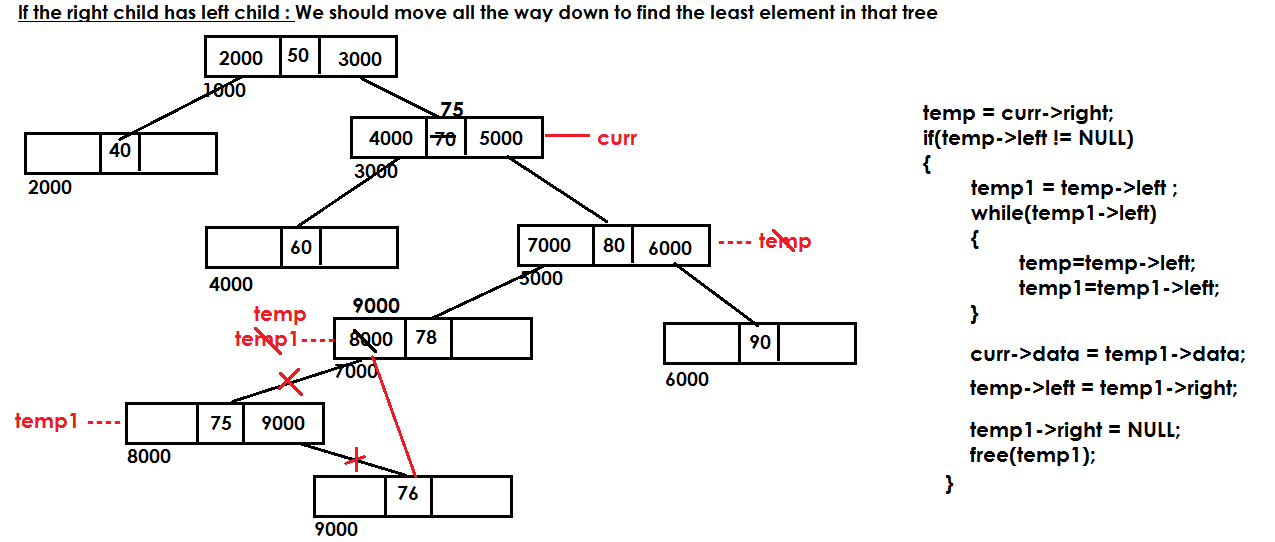
****

****

**In the above 2 cases, we check the condition separately and implemented the logic.**

**We can write same logic for the above 2 cases as follows:**

****

****

**If curr-> right has no left child and right child**:

t1 = curr->right ;

if(t1->left == NULL && t1->right == NULL)

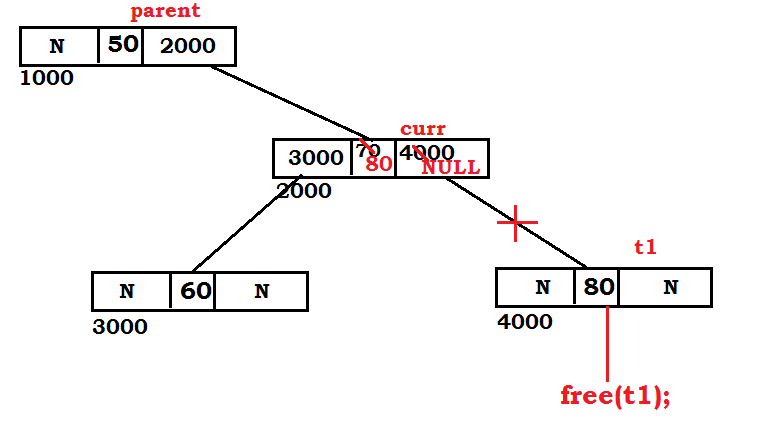
{

curr->data = t1->data;

curr->right = t1->right;

free(t1);

}



**If curr-> right has no left child but right child is present:**

if(t1->right!=NULL && t1->left==NULL)

{

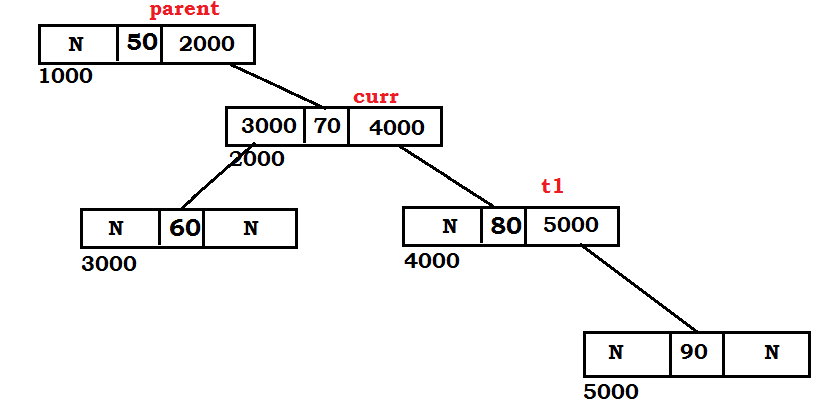
curr->data=t1->data;

curr->right=t1->right;

t1->right=NULL;

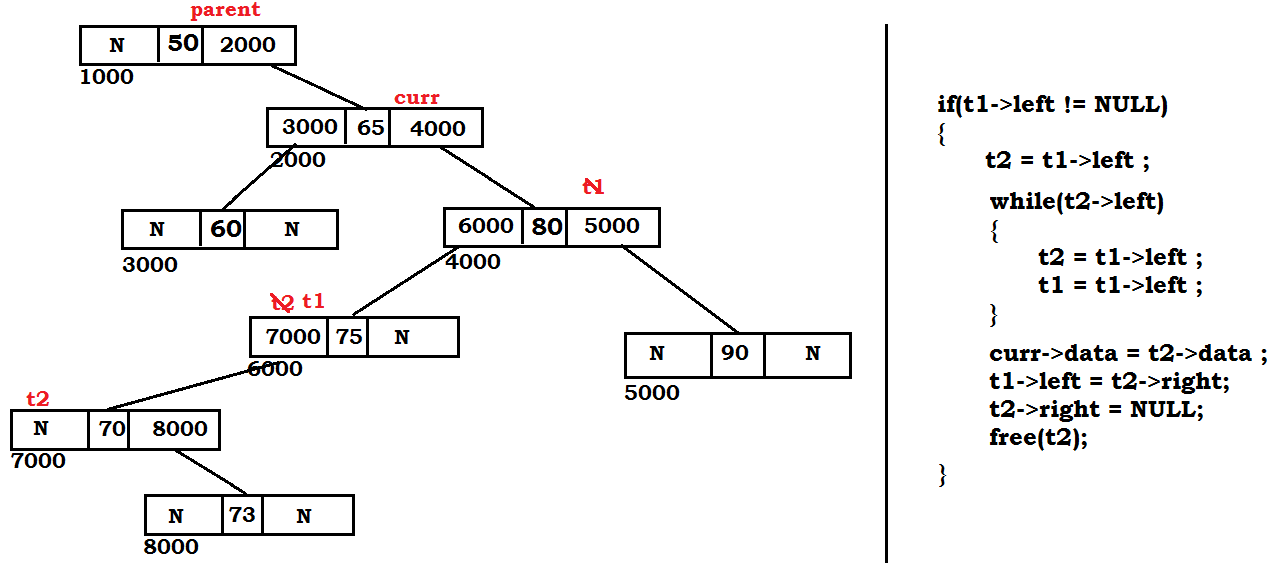
free(t1);

}

****

**If curr->right has left child:**

* We can find the least values on the left side of tree.
* We need to move all the way down to find the least values in right sub tree.
* Once we found, we replace the current node data with least node data and delete that node.

****

t1 = curr->right;

if(t1->left != NULL)

{

t2 = t1->left;

}

while(t2->left)

{

t1 = t1->left;

t2 = t2->left;

}

curr->data = t2->data;

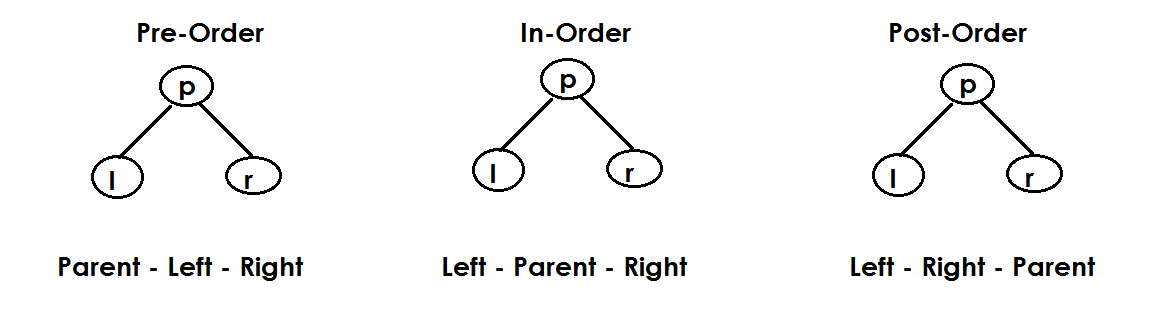
t1->left = t2->right;

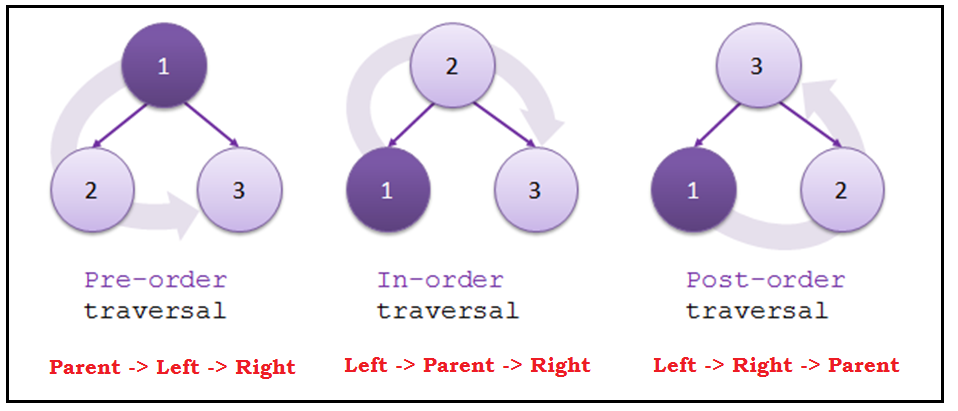
t2->right = NULL;

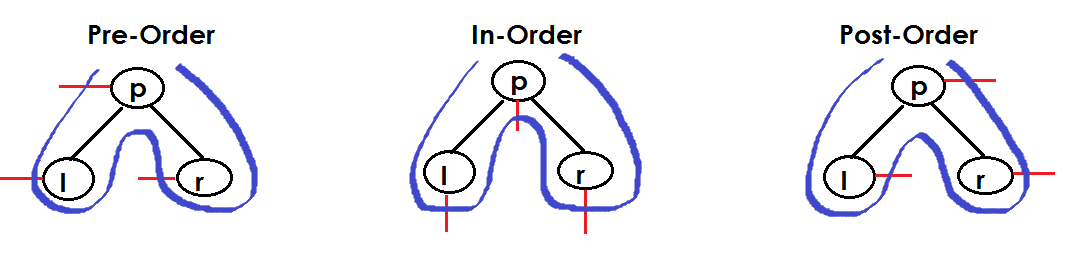
free(t2);

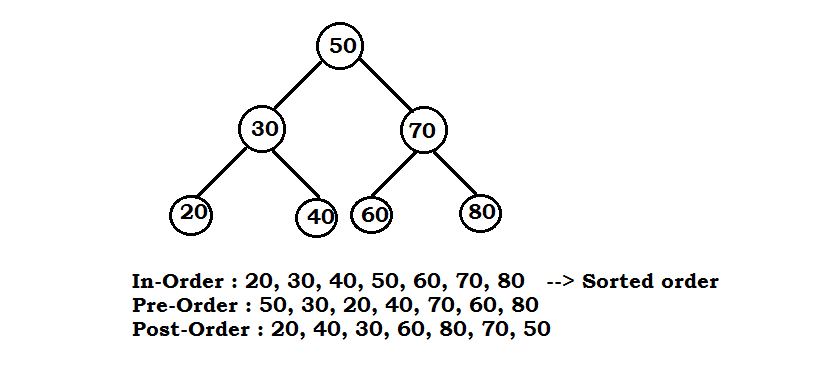
**Traversal:**

* We can traverse the tree in 3 ways
  + In order traversal
  + Pre order traversal
  + Post order traversal

****

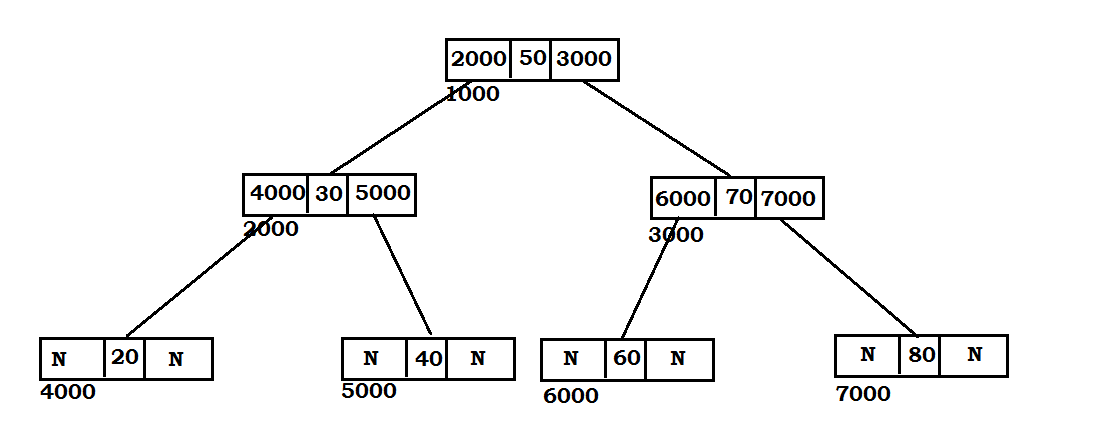
****

****

****

**BST Traversal:**

* We traverse (display) the elements of BST using recursion.
* We have 3 options to traverse the elements
  + Pre-order
  + In-order
  + Post-order



**Inorder traversal:**

void print\_inorder()

{

if(root){

inorder(root);

}

else{

print(“Empty BST”);

}

}

void inorder(struct Node \*p)

{

if(p->left)

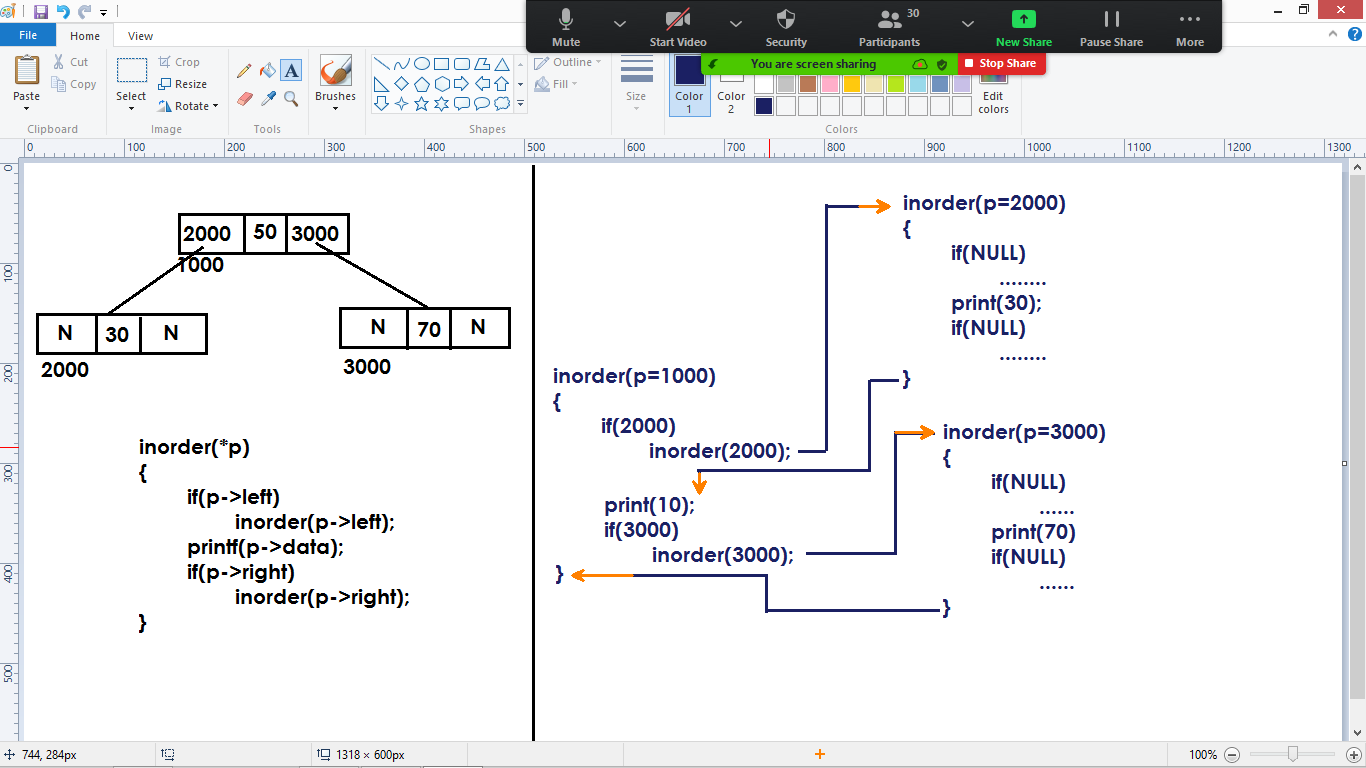
inorder(p->left);

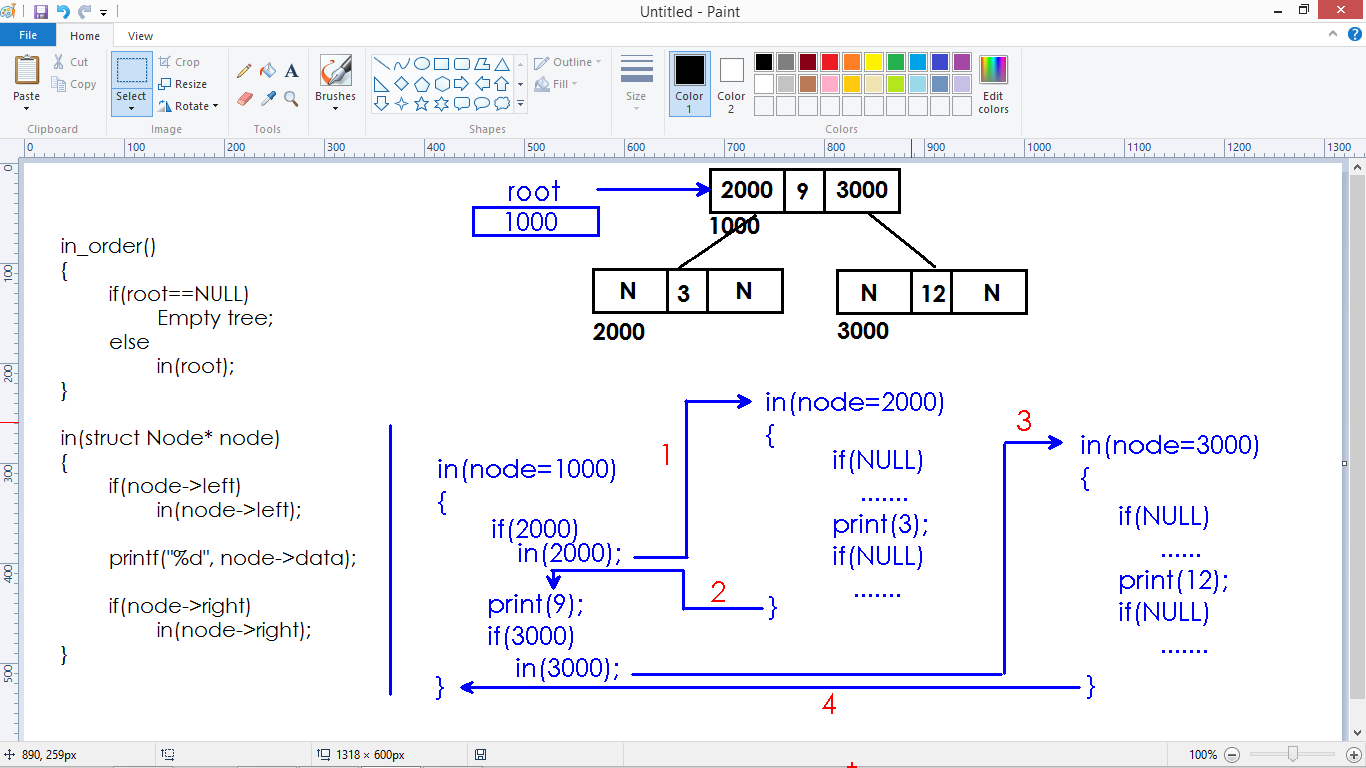
printf("%d ", p->data);

if(p->right)

inorder(p->right);

}





**Pre-order traversal:**

void print\_preorder()

{

if(root){

preorder(root);

}

else{

print("Empty BST");

}

}

void preorder(struct Node \*p)

{

printf("%d ", p->data);

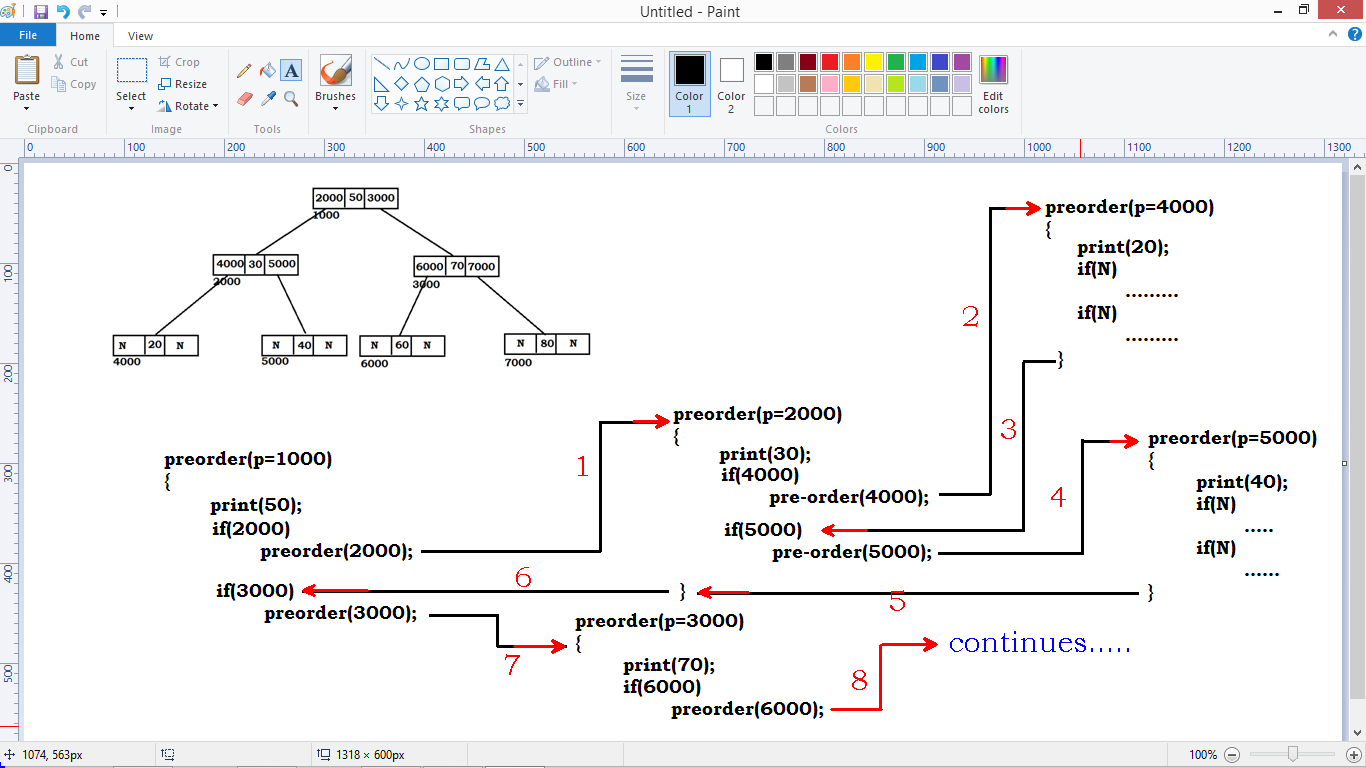
if(p->left)

preorder(p->left);

if(p->right)

preorder(p->right);

}



**Postorder traversal:**

void print\_postorder()

{

if(root){

postorder(root);

}

else{

print("Empty BST");

}

}

void postorder(struct Node \*p)

{

if(p->left)

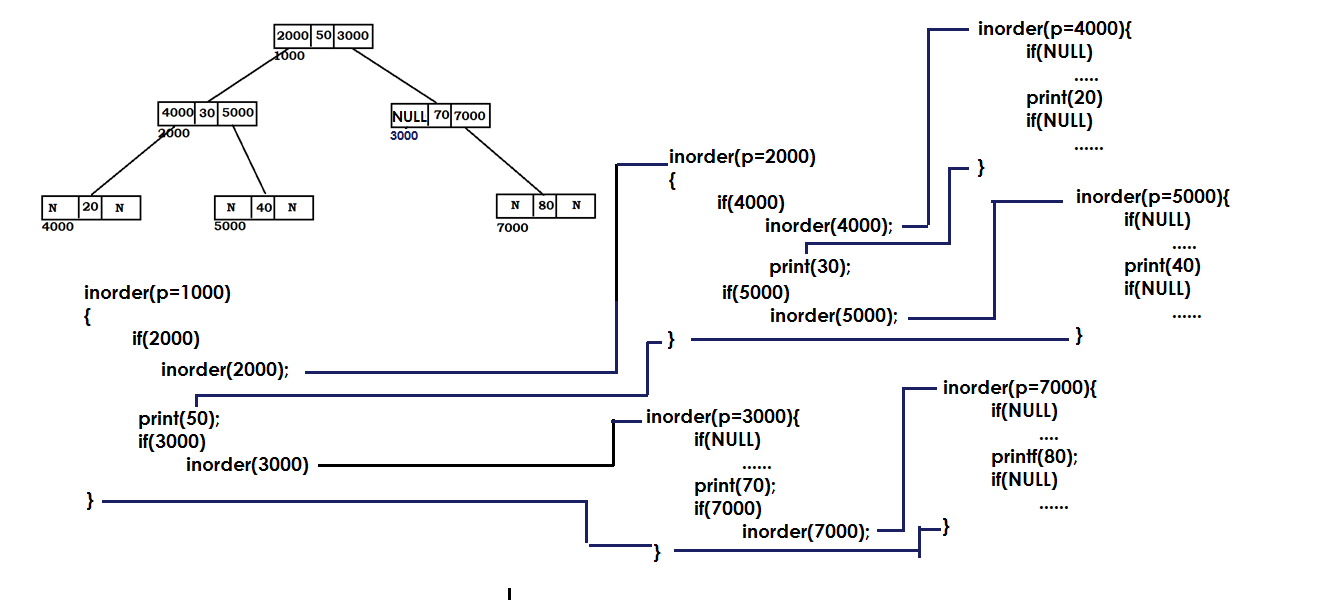
postorder(p->left);

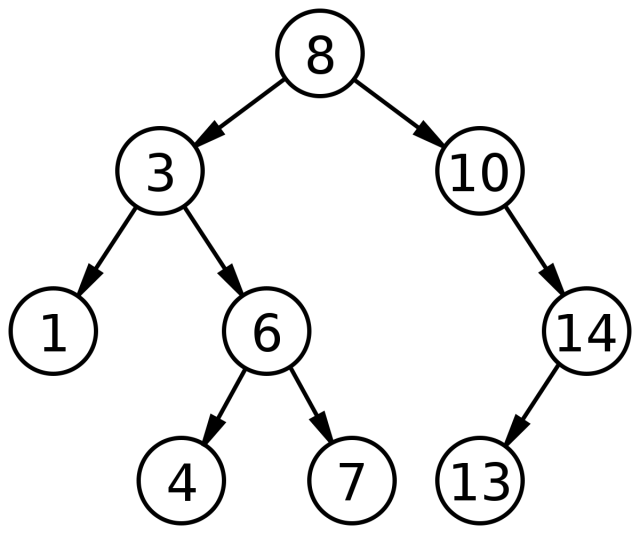
if(p->right)

postorder(p->right);

printf("%d ", p->data);

}





InOrder:

PreOrder:

PostOrder:

Elements : 6,9,2,8,4,0,7,1,6,3

Construct BST and Traverse Post order.

**BST insertion using recursion:**

/\* BST insertion using recursion \*/

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*left;

struct Node \*right;

};

struct Node \*root = NULL;

struct Node\* new\_node(int);

void print\_inorder();

void inorder(struct Node\*);

struct Node\* insert(struct Node\*, int);

int main()

{

int ch, ele;

while(1)

{

printf("1. Insert \n");

printf("2. Inorder \n");

printf("3. Quit \n");

printf("Enter choice : ");

scanf("%d", &ch);

switch(ch)

{

case 1 : printf("Enter element : ");

scanf("%d" , &ele);

root = insert(root , ele);

break;

case 2 : print\_inorder();

break;

case 3 : exit(1);

default : printf("Invalid Choice \n\n");

}

}

return 0;

}

struct Node\* insert(struct Node\* node, int ele)

{

if(node==NULL)

{

return new\_node(ele);

}

if(ele < node->data)

{

node->left = insert(node->left , ele);

}

if(ele > node->data)

{

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

}

return node;

}

struct Node\* new\_node(int data)

{

struct Node\* node;

node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

void print\_inorder()

{

if(root==NULL)

{

printf("BST is empty \n\n");

}

else

{

inorder(root);

printf("\n\n");

}

}

void inorder(struct Node\* node)

{

if(node->left)

{

inorder(node->left);

}

printf("%d \t", node->data);

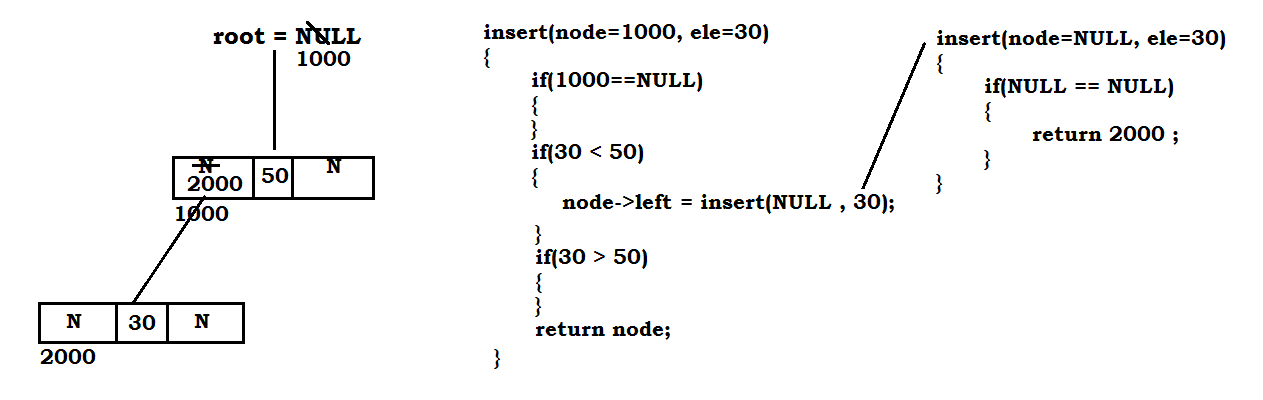
if(node->right)

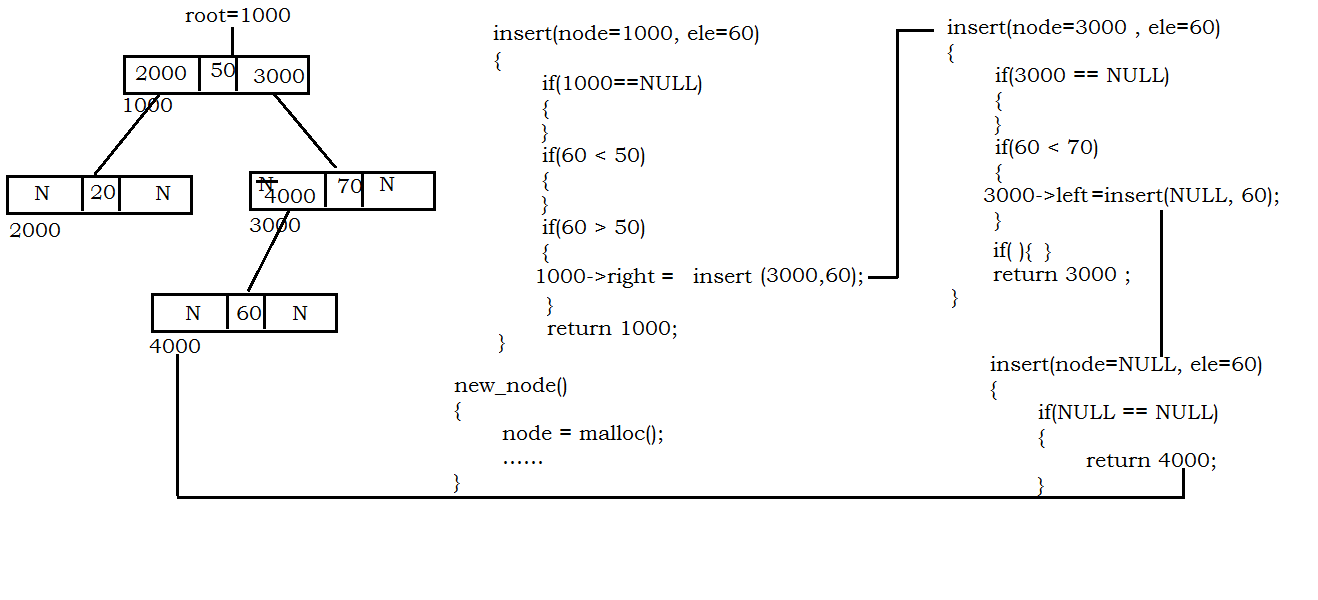
{

inorder(node->right);

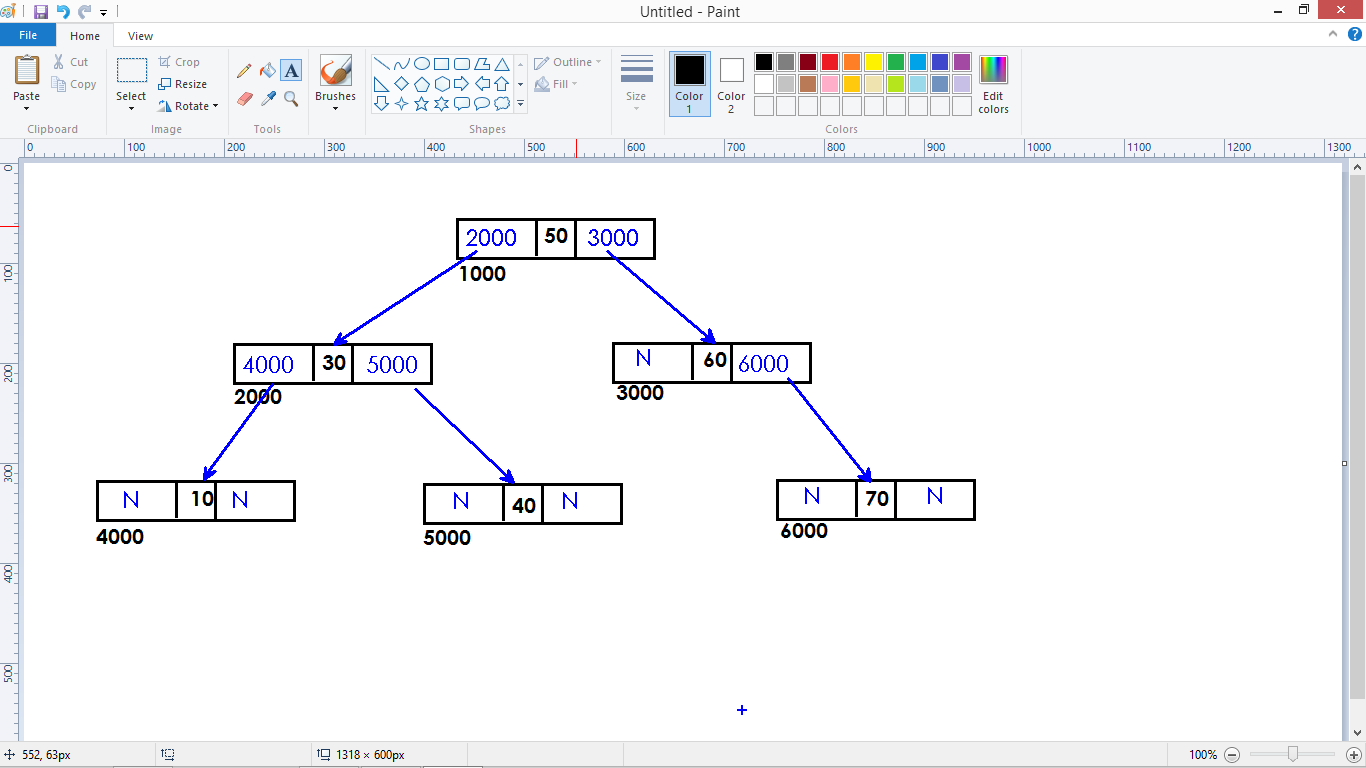
}

}





**Sample BST:**



**Pre-order traversal with recursion:**

Parent -> Left Child -> Right Child

**Code:**

preorder(struct Node\* node)

{

print(node->data);

if(node->left)

preorder(node->left);

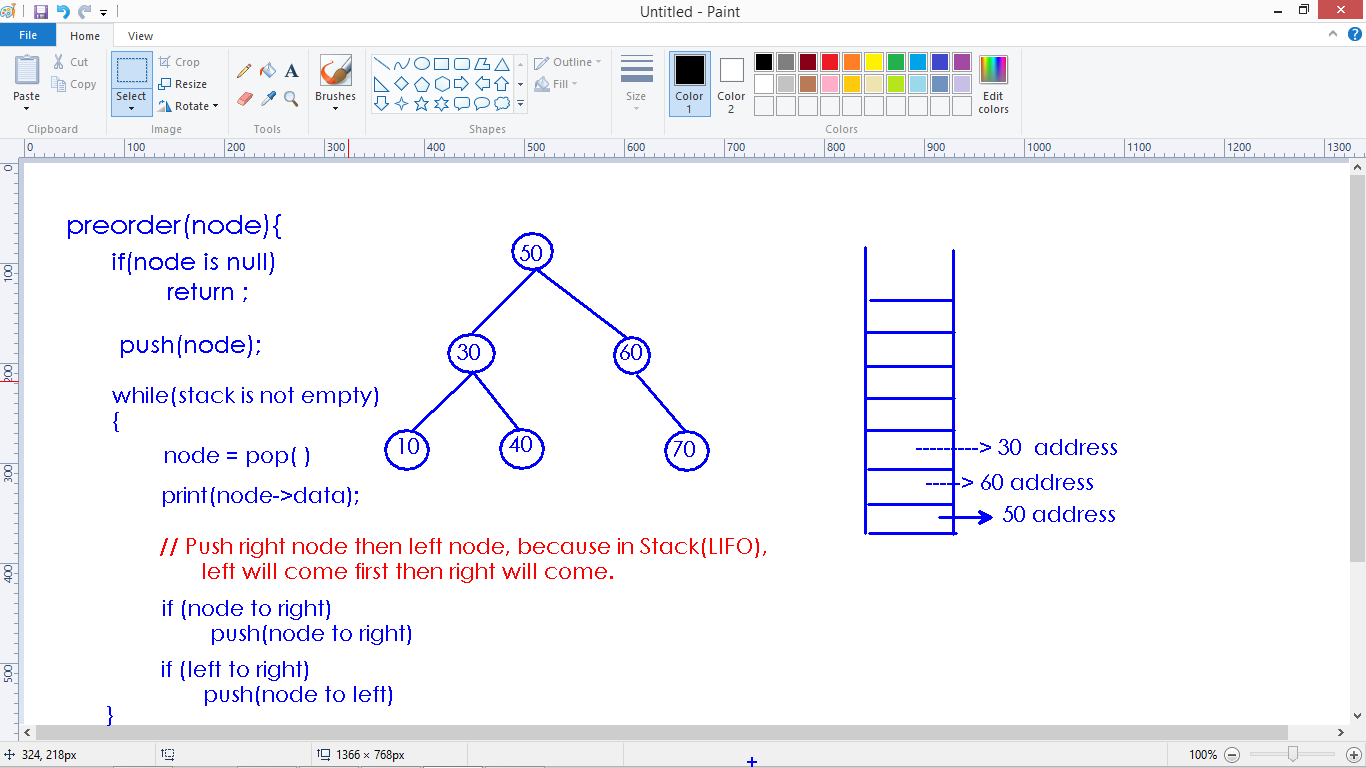
if(node->right)

preorder(node->right);

}

**Pre-order without recursion:**

* We use Stack data structure to iterate the BST.



**Code:**

struct Node \*stack[20];

int top=-1;

preorder(struct Node\* node)

{

if(node==NULL)

return;

push(node);

while(top != -1)

{

node = pop();

print("%d \n", node->data);

if(node->right)

push(node->right);

if(node->left)

push(node->left);

}

}

void push(struct Node\* node)

{

stack[++top] = node;

}

struct Node\* pop()

{

return stack[top--];

}

**Inorder traversal:**

Left child -> Parent -> Right Child

**Code With recursion:**

inorder(struct Node\* node)

{

if(node->left)

inorder(node->left);

print(node->data);

if(node->right)

inorder(node->right);

}

**Code without recursion:**

