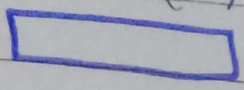
preorder Traversal

(go to main function

struct node *root= new node(1)



∵ go to the newNode() function

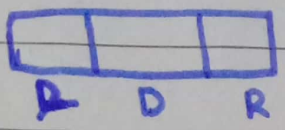
Node *temp = new Node;

∵ go to struct Node {

int data;

struct Node *left, *right;

}

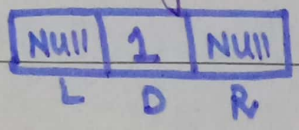


∵ go back to To newNode(int data)

{ temp→data = data;

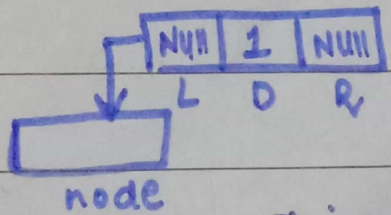
temp→left = temp→right = NULL;

return temp;



∵ go back to the main function

root→left = newNode(2)



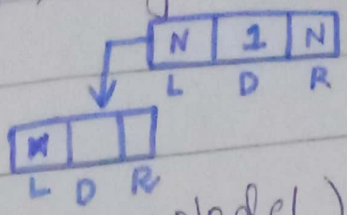
∵ call new node() function

Node *temp = new Node;

∵ call struct node {

int data;

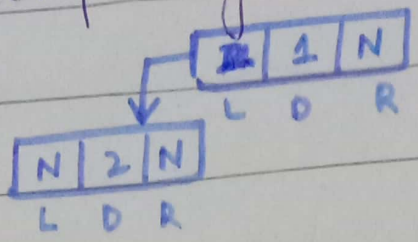
struct Node * left, * right;



∴ go back to the newNode() func

temp→data = data

temp→left = temp→right = NULL;

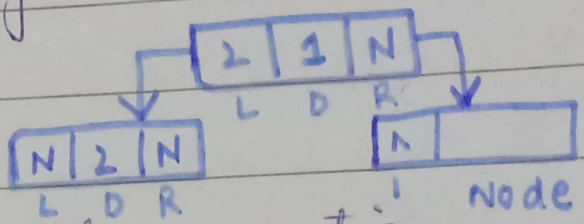


return temp;

}

∴ go again to main function

temp→right = newNode(3).



∴ call newnode function

Node * temp = new Node;

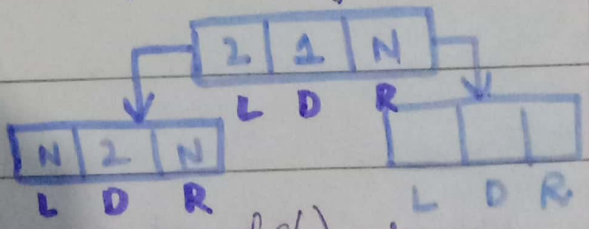
∴ call struct node function

struct node {

int data;

struct node * left, * right;

}

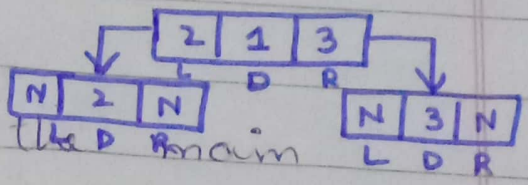


∴ go back to newnode() func

temp→data = data

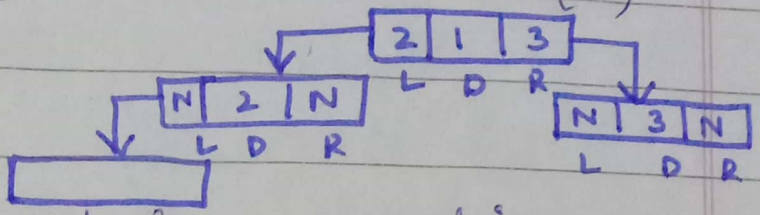
temp→left = temp→right = NULL

return temp;



∴ go back to main function

root → left → left = new Node(4)



∴ call newNode function

Node * temp = new Node;

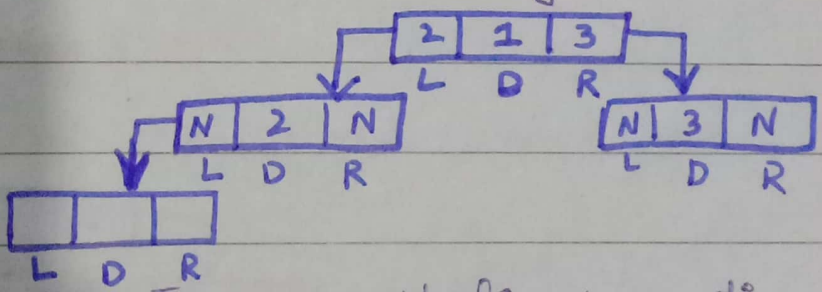
∴ call struct Node function

struct node {

int data;

struct node * left, * right;

}



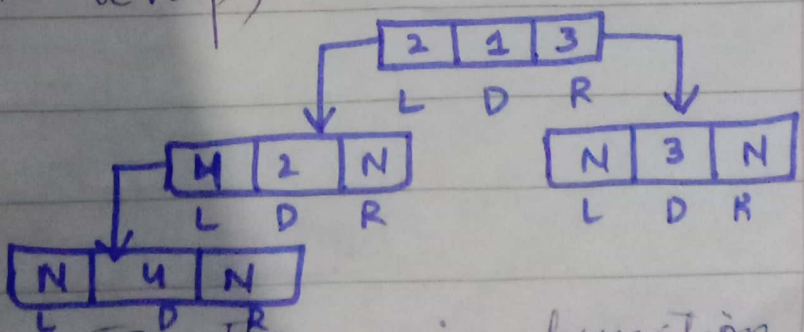
∴ go back to new Node function

temp → data = data.

temp → left = temp → right = Null

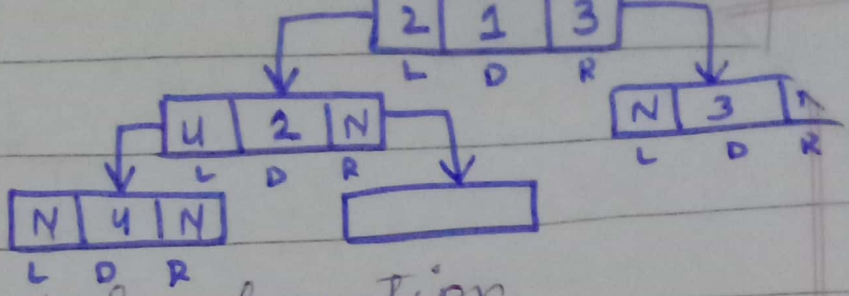
return temp;

}



∴ go back to the main function

root → left → right = new Node(5)



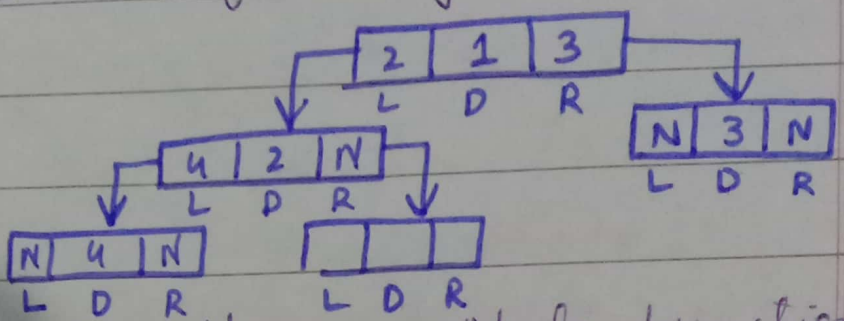
∴ call new Node function

Node * temp = new Node{};

∴ go to struct node function

int data;

struct node * left, * right;

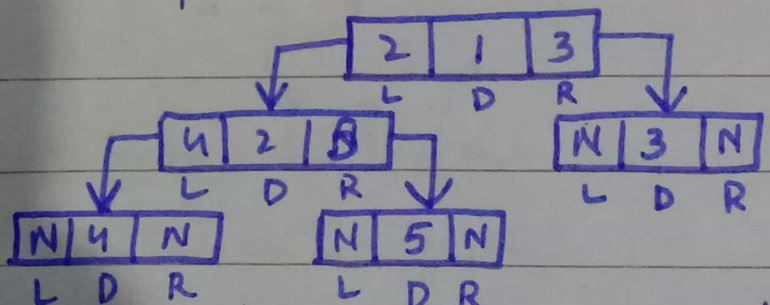


∴ go back to the newNode function

temp → data = data;

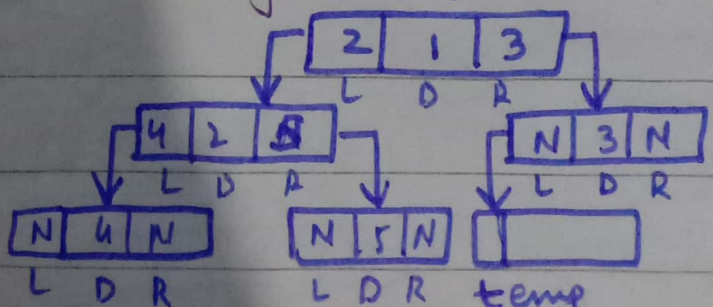
temp → left = temp → right = Null;

return Temp;



∴ go back to the main function

~~ro temp~~ root → right → left = newNode(6)



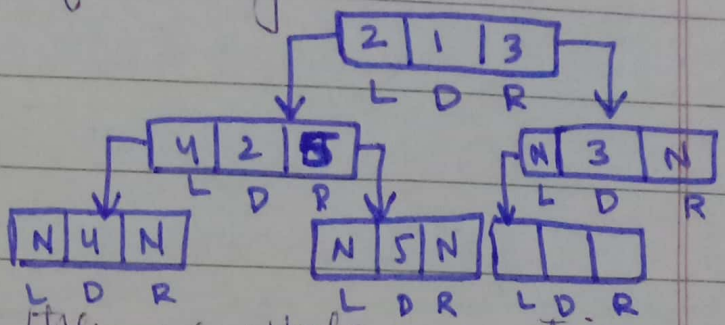
∴ call new Node function

Node *temp = newNode;

∴ go to struct node function

int data;

struct node *left, *right;

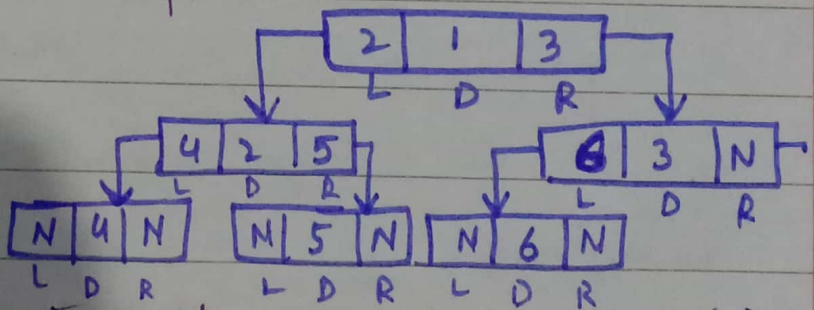


∴ go back to the newNode function

temp → data = data;

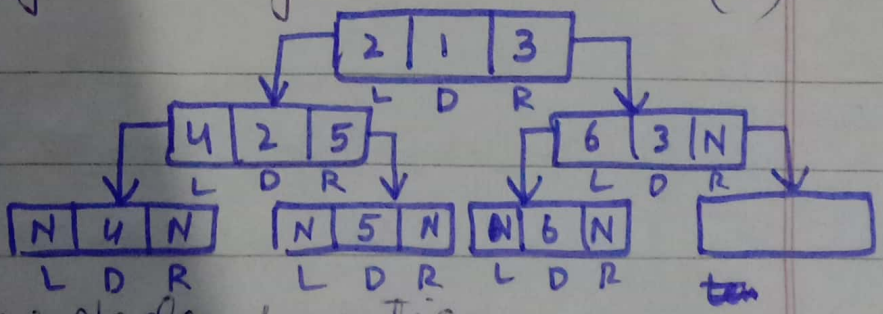
temp → left = temp → right = Null;

return temp;



∴ go back to the main function

root → right → right = new Node(7)



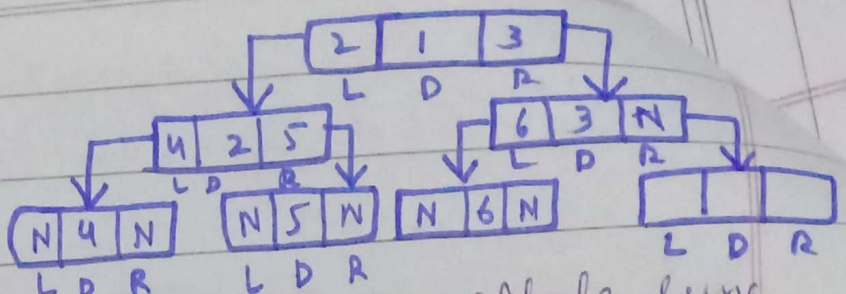
∴ call new Node function

Node *temp = new Node;

∴ go to struct node function

int data;

struct node *left, *right;



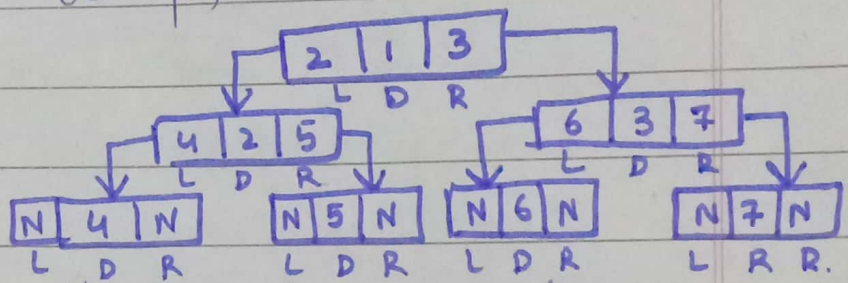
∵ go back to the newNode func

temp → data = data;

temp → left = temp → right = Null.

return temp;

}



∵ go to the main function

print preorder (root),

∵ go to the printpreorder function

1

if (root == NULL) → false

return ;

}

cout<< node → data;

print preorder(root → left);

print preorder (root → right);

print 2.

print pre-order(root → left) 2

∵ go to if condition

2

if (root == NULL) → false

return ;

}

cout << node → data;

print 2

print preorder (root → left) → 4

cout << node → data

print 4

∵ Go back to the root 2 and go to the right

~~cout<<~~

print preorder (root → right) → 5

cout<< node → data.

print 5.

∵ Go back to the root 1 and go to the right

print preorder (root → right) → 3

cout<< node → data

print 3.

∵ Go back to the left side of the root 3.

print preorder (root → left) → 6

print 6

∵ Go to the right side of root 3

print preorder (root → right) → 7

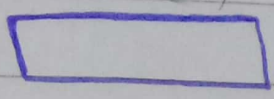
print 7

1 2 4 5 3 6 7

Post Traversal:.

Go to main function

struct node * root = new node(1)



∵ go to the newNode() function

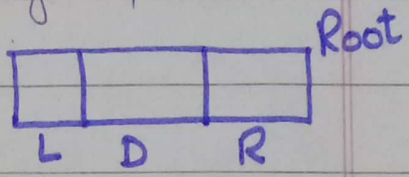
Node * temp = new Node;

∵ go to struct Node {

int data;

struct Node * left, * right;

}

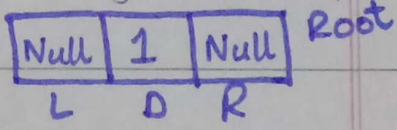


∵ go back to to newNode(int data)

temp → data = data;

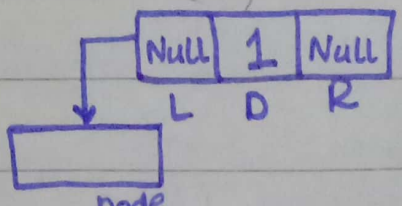
temp → left = temp → right = Null;

return temp;



∵ go back to the main function

root → left = newNode(2)



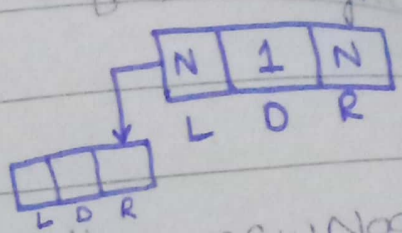
∵ Call new node() function

Node * temp = new Node;

∵ Call struct node {

int data;

struct Node *left, *right;

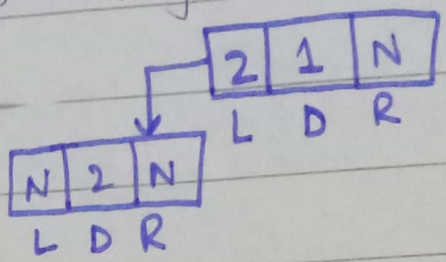


∵ go back to the newNode() func.

temp → data = data

Temp → left = temp → right = Null;

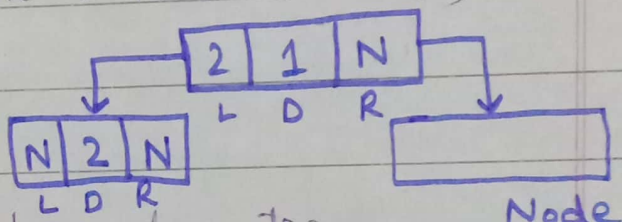
return temp;



}

∵ go again to main function

temp → right = newNode(3)



∵ Call new node function

Node * temp = new Node;

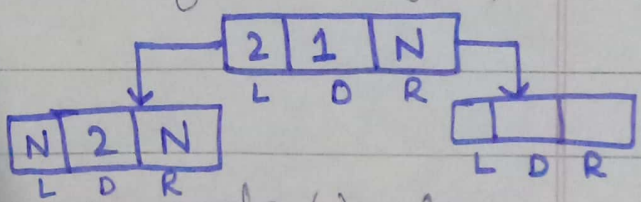
∵ call struct node function

struct node {

int data;

struct node *left, *right;

}

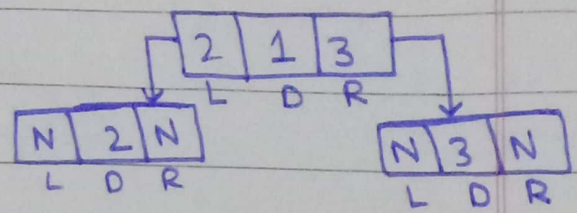


∵ go back to new node() func

temp → data = data

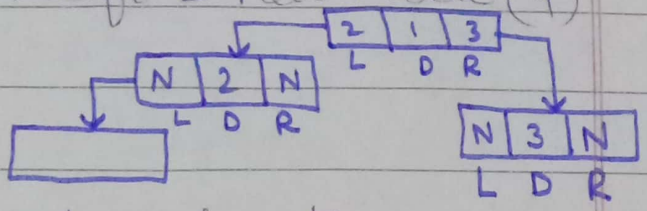
Temp → left = temp → right = Null

return temp;



∵ go back to the main function

root → left → left = new Node(4)



∵ Call new Node function

Node * temp = new Node;

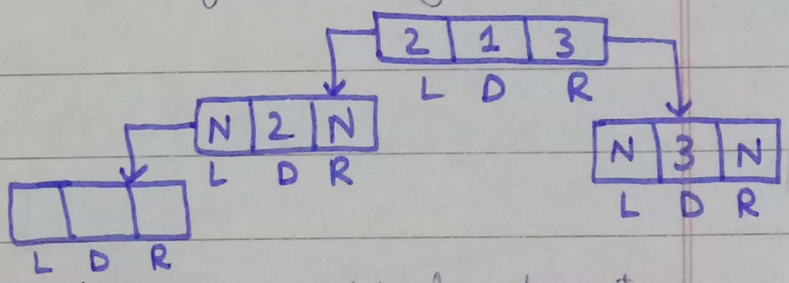
∵ Call struct Node function

struct node {

int data;

struct node *left, *right;

}



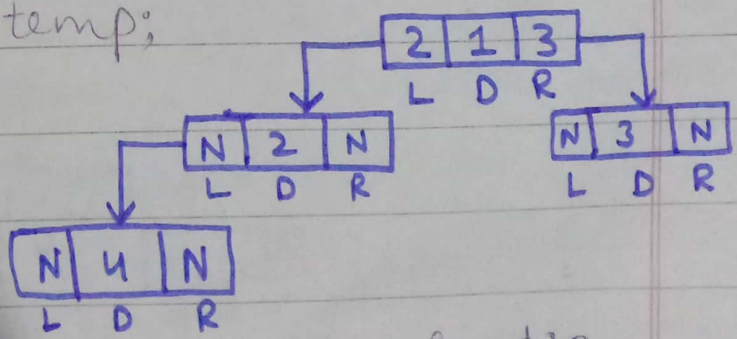
∵ go back to new Node function

temp → data = data

temp → left = temp → right = Null

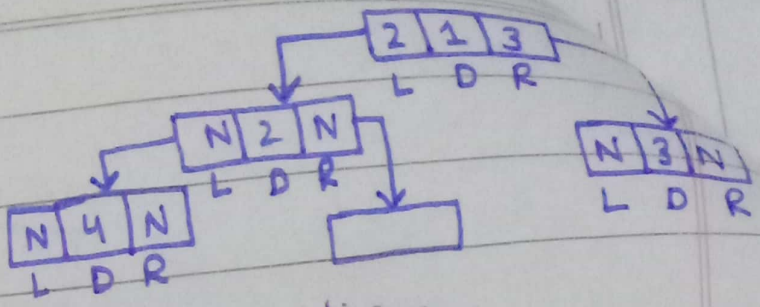
return temp;

}



∵ go back to the main function

root → left → right = new Node(5)



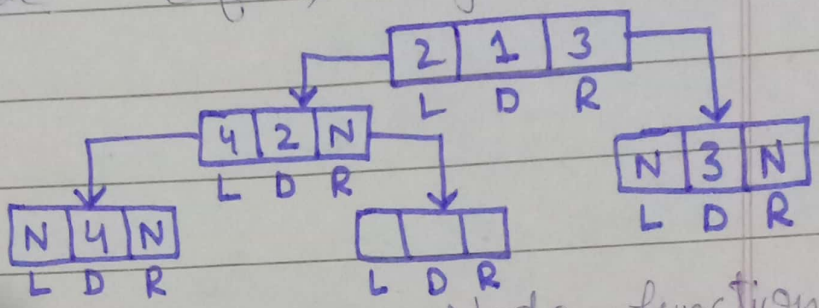
∵ Call new Node function

Node * temp = new Node;

∵ go to struct node function

int data;

struct node * left, * right;

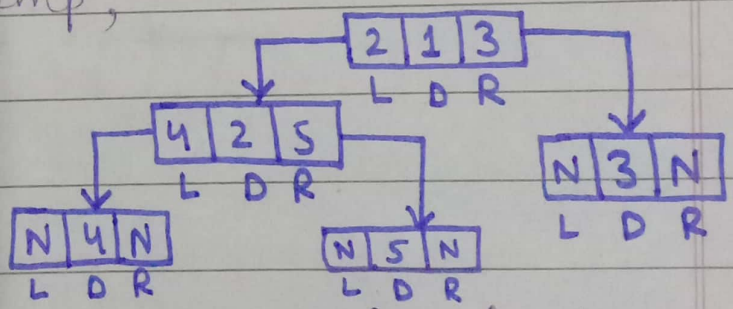


∵ go back to the new Node function

temp → data = data;

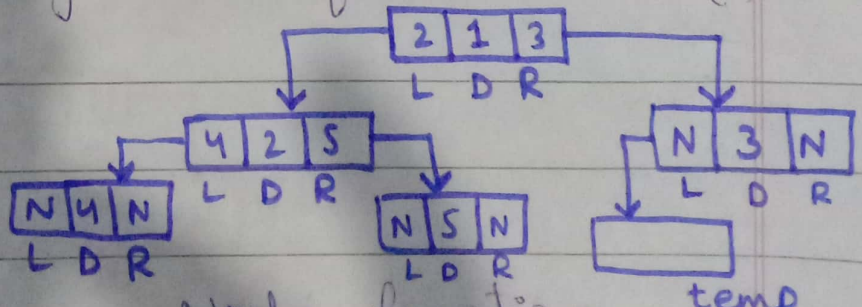
temp → left = temp → right = Null;

return Temp;

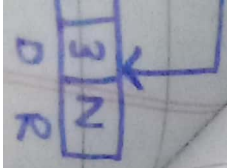


∵ go back to the main function

root → right → left = new Node(8)



∵ call new Node function.

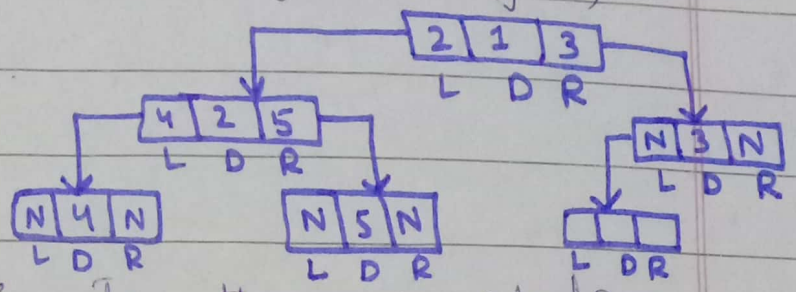


Node * temp = newNode;

∵ go to struct node function

int data;

struct node *left, *right;

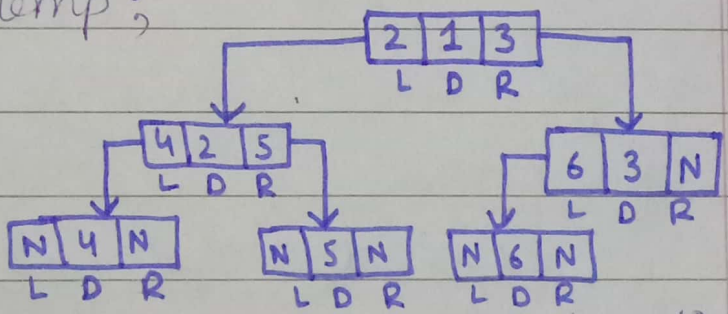


∵ go back to the newNode function.

temp → data = data;

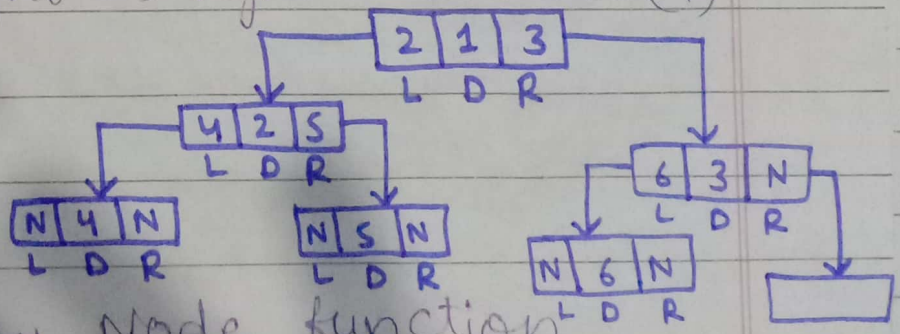
temp → left = temp → right = Null;

return temp;



∵ go back to the main function

root → right → right = newNode(7)



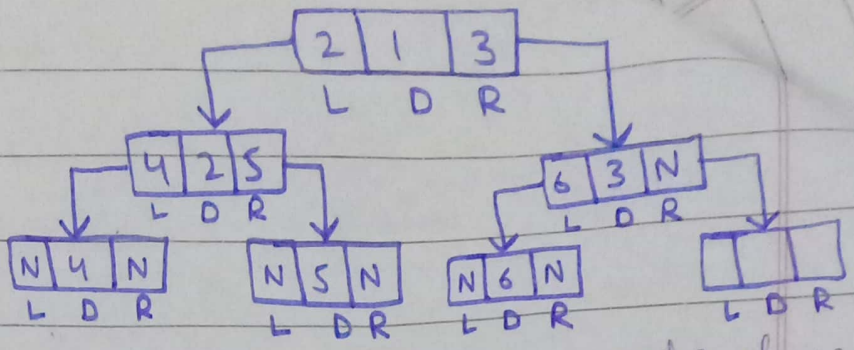
∵ Call newNode function

Node * temp = newNode;

∵ go to struct node function

int data;

struct node *left, *right;



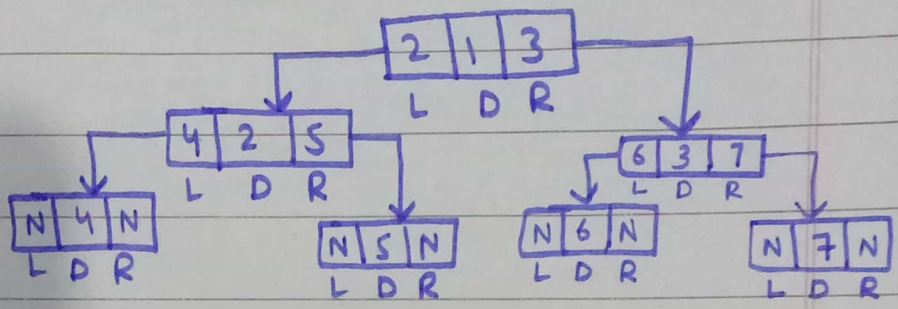
∵ go to back to the newnode func.

temp → data = data;

temp → left = temp → right = Null

return temp;

}



∵ go back to the main function

print postorder (root);

∵ go to the print preorder function

if (root == NULL) → false

return;

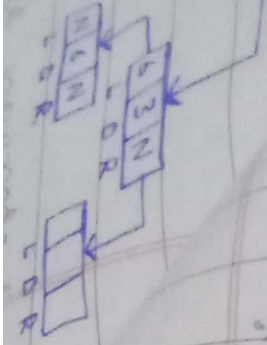
}

print postorder (root → left) → 2

∵ again go to if condition this condition is false

if (root == Null) → false

{ return; }



So the

printpostorder(root→left) → 4

∴ So there is no child of 4 so

print 4

∴ Back to the previous call

printpostorder(root→right) → 5

∴ So there is no child of 5 so

print 5

∴ Back to the main root 1

∴ print the data of the current node

print 2

∴ Now call printpostorder

printpostorder(root→right) → 3

∴ go to the printpostorder

printpostorder(root→left) → 6

So there is no child of 6.

print 6

∴ Back to the previous call

printpostorder(root→right) → 7

So there is no child of the node 7

∴ ~~print~~ ~~the~~ Back to the original root.

print 1

The final output of the postorder Traversal is

~~4 5~~ 4 5 2 6 7 3 1

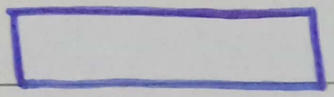
Topic:- Traversal BT (inorder)

- Go to main function

```
struct node* root = createdNode(1);
```

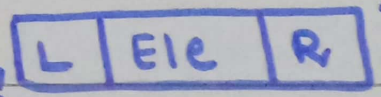
- go to create function

```
struct node* createNode(int value)
{
struct node * Node = new struct node;
```



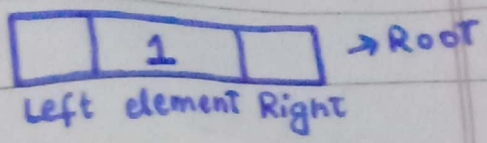
- go to struct part

```
struct node
{
    int element;
struct Node * left;
struct Node * right;
};
```



- go back to createnode function

Node → element = value (1)

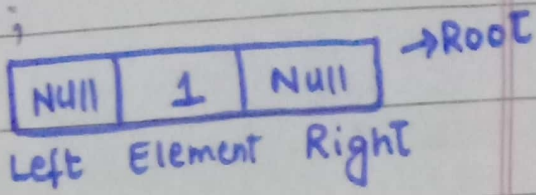


Node → left = NULL;

Node → right = NULL;

return node;

}



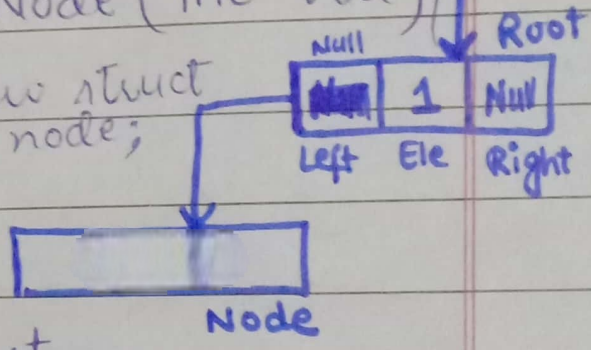
∵ again go to main function

root → left = createNode(2);

∵ go back to createNode function

struct node * createNode(int val) (2)

struct node * Node = new struct node;

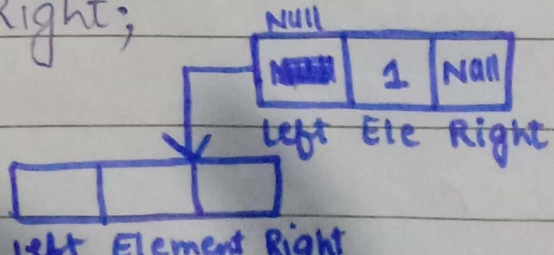


∵ go to struct part

int element;

struct Node * left;

struct Node * Right;



∵ go back to the createnode function.

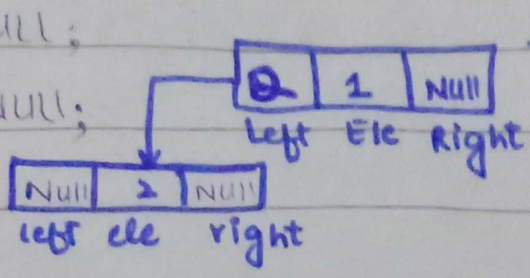
Node → element = val; (2)

Node → left = NULL;

Node → right = NULL;

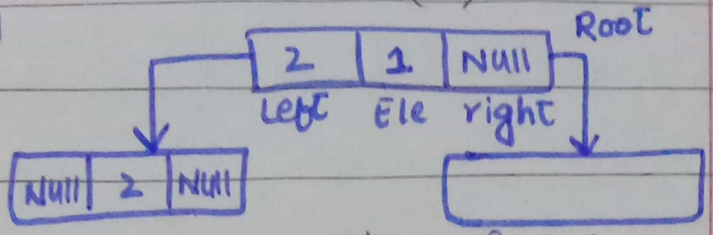
return node;

}



∵ again go back to the main function

root → right = creatNode (3)



∵ go ~~to~~ to the createNode( ) function

struct node * createNode (int val){

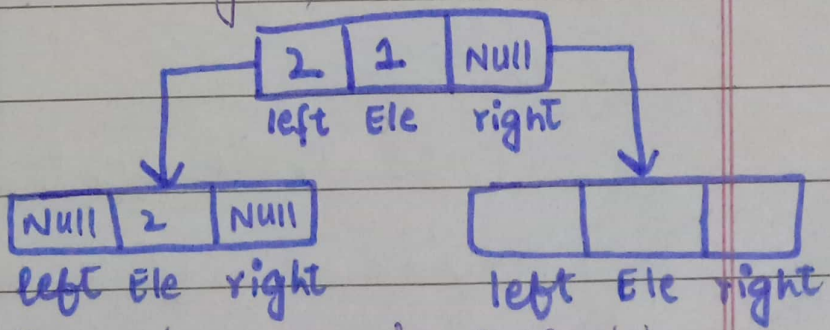
struct node * Node = new ~~Node~~ struct node;

∵ go to struct part

int element;

struct Node *left;

struct Node *right;



∵ go back to the createNode () function

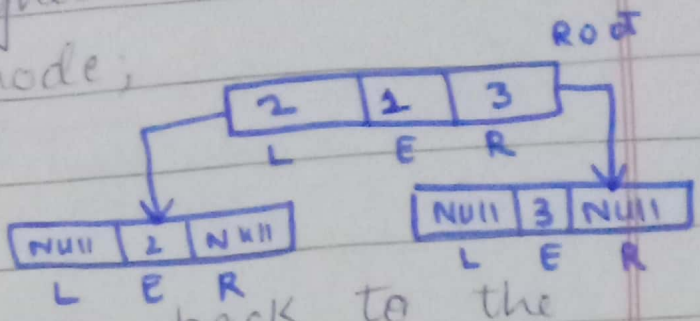
Node → element = 3 val;

Node → left = Null;

Node → right = NULL

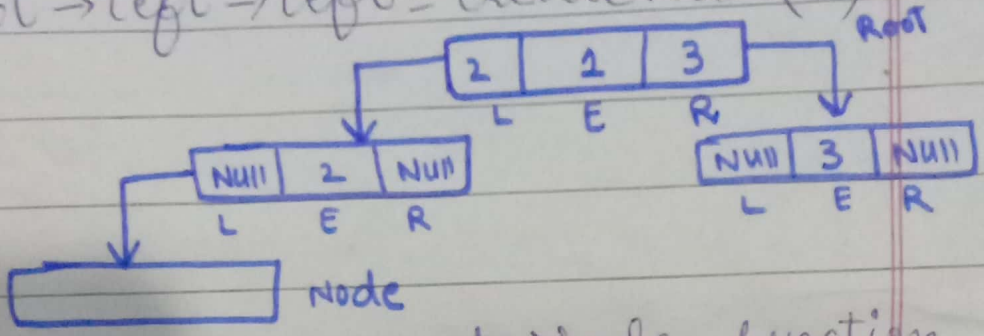
return node;

}



∴ again move back to the main function

root → left → left = createNode(4)



∴ move to createNode function 4

struct node * createNode(int val){

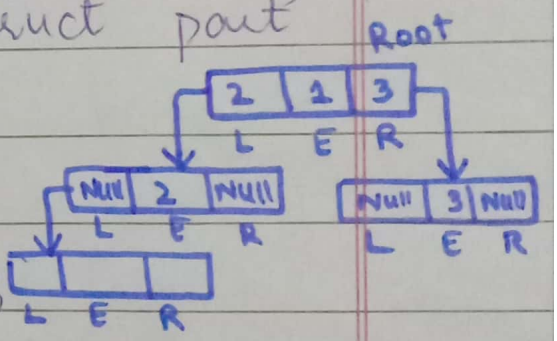
struct node * Node = new struct node();

∴ move to the struct part

int element;

struct node * left;

struct node * right;



∴ go back to createNode function

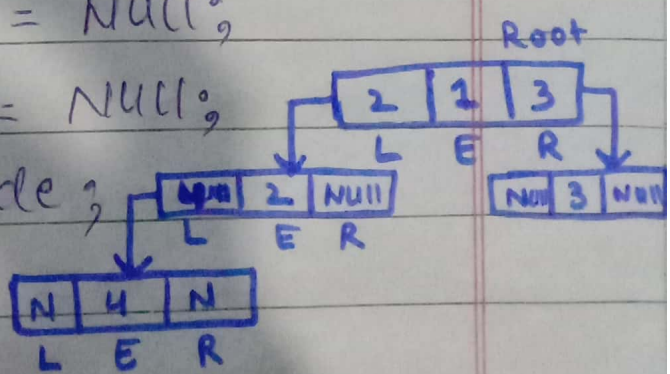
Node → element = val;

Node → left = NULL;

Node → right = NULL;

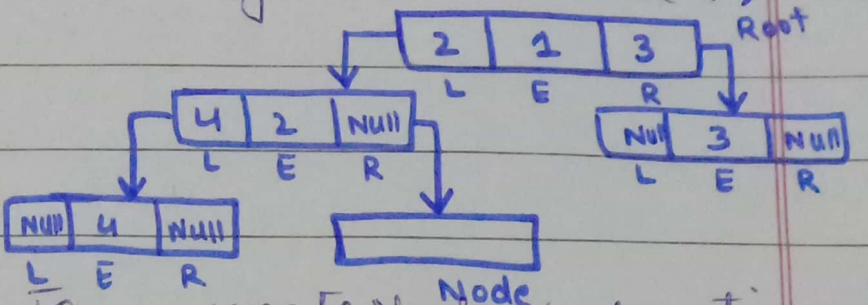
return Node;

}



∴ again move back to the main function

root → left → right = createNode(5);



∴ move to createNode() function

5

struct node * createNode(int val)

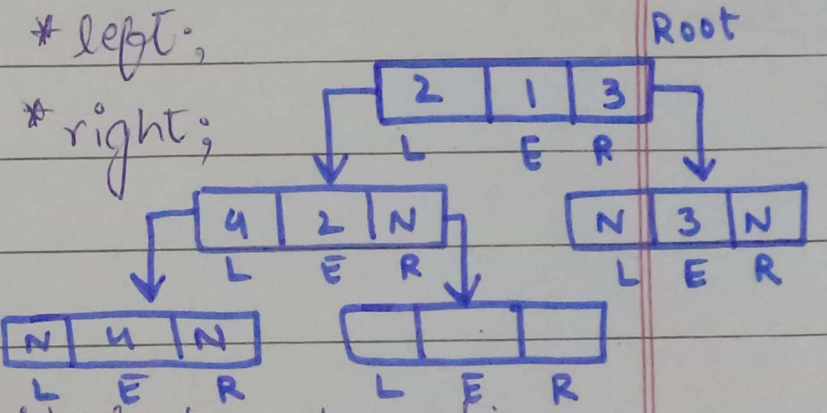
struct node * Node = new struct node();

∴ move to struct part.

int element;

struct node *left;

struct node *right;



∴ go createNode() function

5

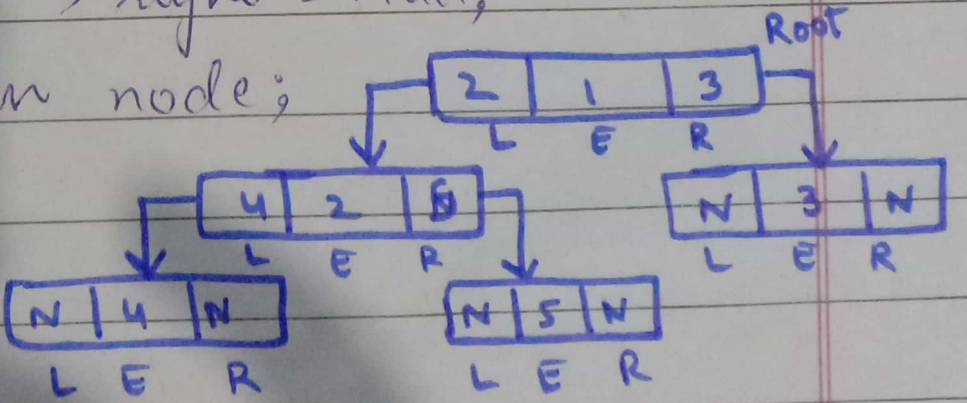
Node → Element = val;

Node → left = NULL;

Node → right = NULL;

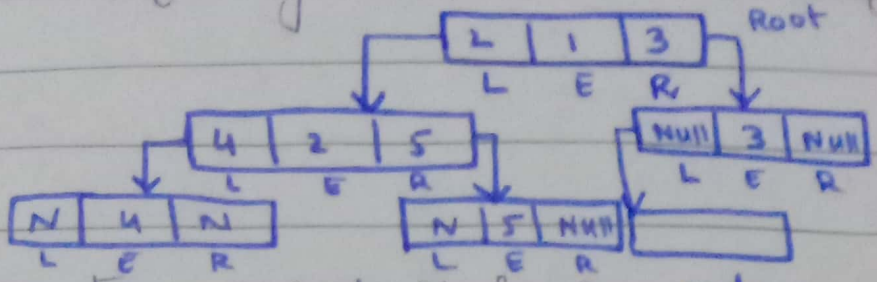
return node;

}



∴ again go back to the main function

root → ~~left~~ right → left = createNode(6)



∴ move to createNode function

struct node * createNode(int val)

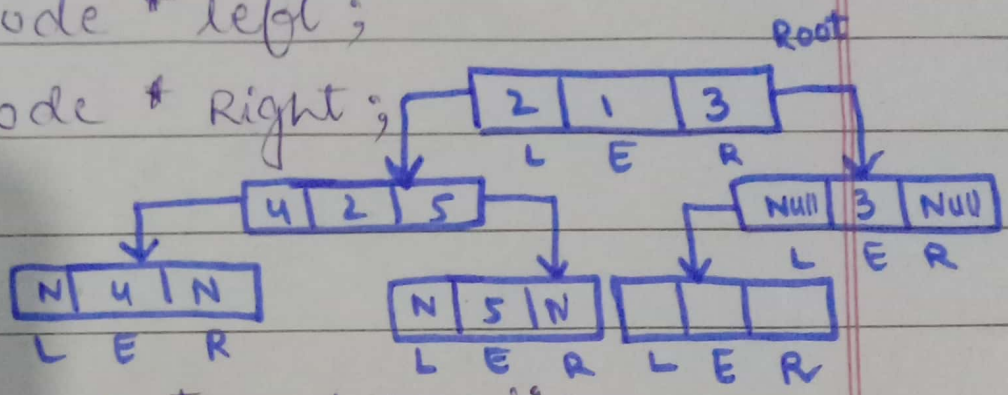
struct node * Node = new struct node();

∴ move to struct part

int element;

struct node * left;

struct node * Right;



∴ go to create function

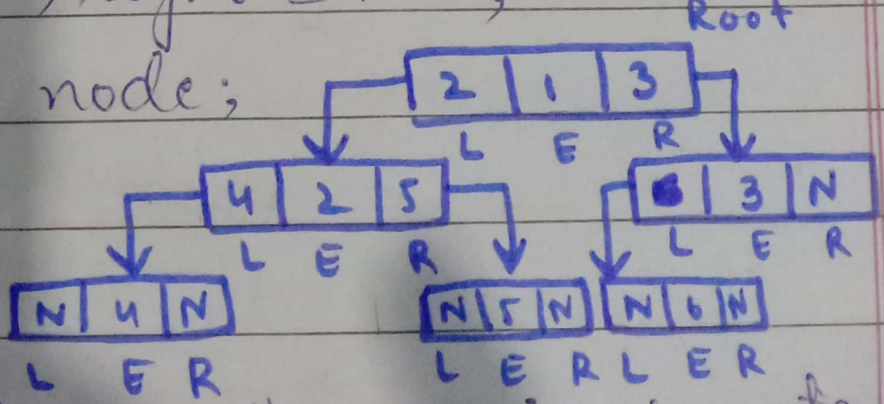
Node → element = val;

Node → left = NULL;

Node → right = NULL;

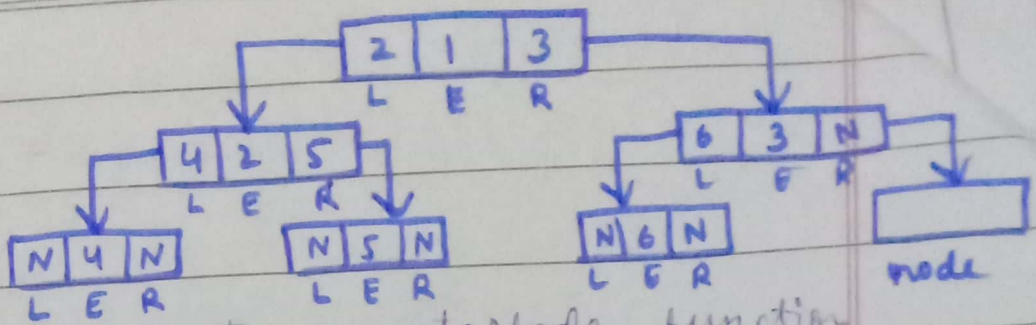
return node;

}



∴ go back to the main function

root → right → right = createNode(7)

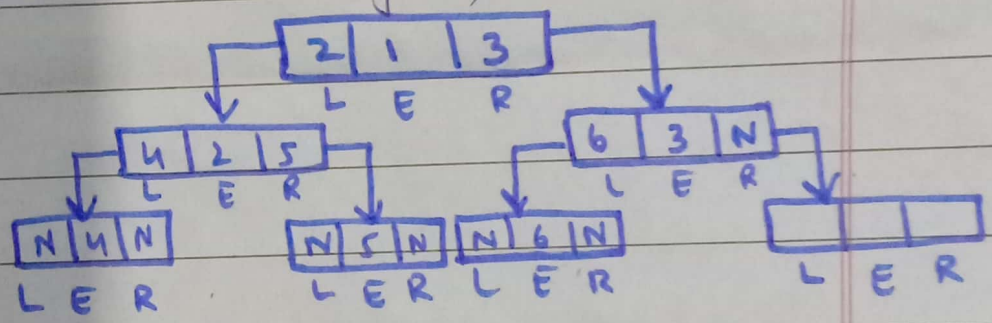


∴ move back to createNode function

```
struct node * createNode(int val){
struct node * Node = new struct node();
```

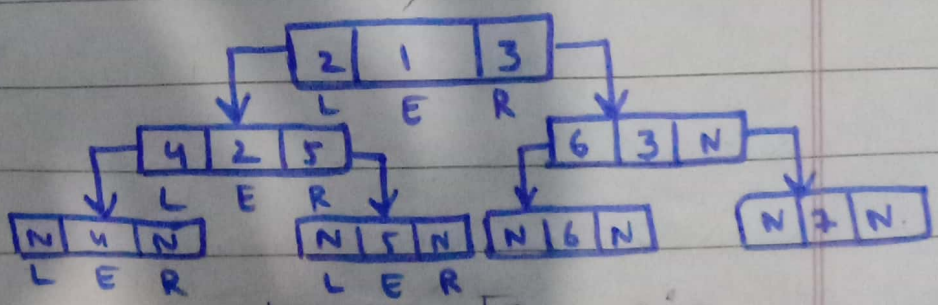
∵ move to struct part

```
int element;
struct node * left;
struct node * right;
```



∴ go back to createNode function.

```
Node -> element = val;
Node -> left = Null;
Node -> right = Null;
    return Node;
}
```



∴ again go back to main function

Traverse Inorder (root)

∴ go To the Traverse Inorder function

if (root == NULL) → false

return;

∴ call Traverse Inorder (root → left) → 2

∴ Go To the left of the root 2

Traverse Inorder (root → left) → 4

print 4.

∴ Go back To the previous node 2 and go the the right side.

Traverse Inorder (root → right) → 5

cout << root - element

print 5

∴ Back to the root 1

cout << root element → 1

∴ print the element of the root

print 1

∴ Go the the right child and then move to the left child

who's element is 6

print 6 → 6

∵ Go to the right child

who's element is 7

but first we print

the root 3

cout << root → element → 3

print 3.

∵ Go to right child

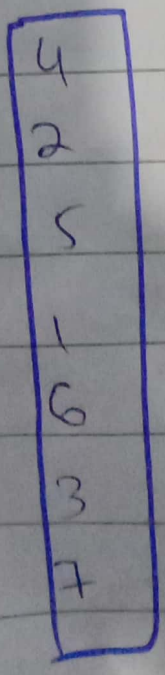
and print 7

print 7 → 7

So, the final output

of the inorder Tree

is



Insertion & deletion of Binary search Tree.

∴ Go to the main function

```
Node * root = null;
root = insert(root, 50);
```

Null root

∴ go to the insert function

```
Node * insert( Node * root, int key)
{
    if ( root == NULL) → True
    {
        return createNode(key)
    }
```

50

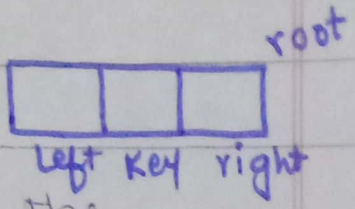
∴ go to the createNode function

```
Node * createNode( int key) {
Node * newNode = new Node;
```

∴ go to the struct Node

int key;

Node *left, *right.



∴ go back again to the

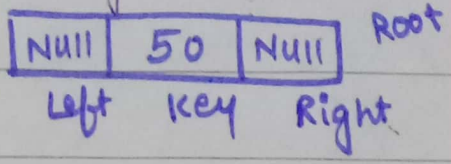
create Node function.

NewNode → key = key;

NewNode → ~~data~~ left = newNode →

right = NULL;

return NewNode;



}

∴ go back again to the main function.

insert(root, 30)

∴ go to the insert function.

Node * insert(Node * root, int key) 30

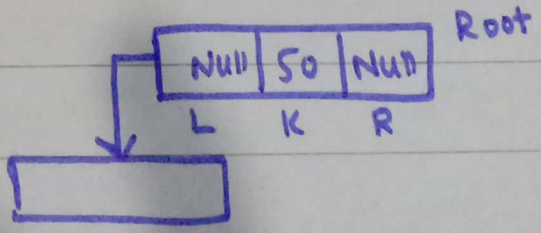
{

30 50

if (key < root → key) {

root → left = insert(root → left, key);

}



if (root == NULL)

{ return createNode(key);

}

∴ The left subtree is empty

so create a new node

if (node == NULL) → True.

{ { return createNode(key);

}

∴ go to the create node

Node * create Node (int key)

{

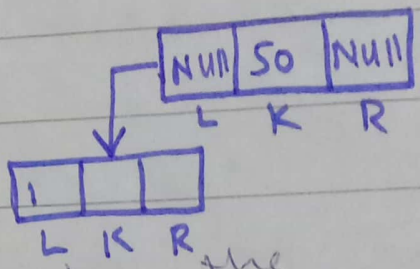
Node * NewNode = new Node;

∴ go to the struct node

int key;

Node * left

Node * Right;



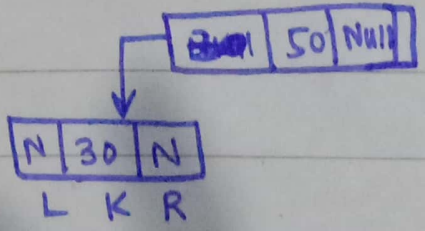
∴ go back again to the create Node function

30

NewNode → key = key

NewNode → left = NewNode → right = NULL,

return newNode



∴ again go back to the main function

insert (root, 20)

∴ go back again to the insert

Function.

Node *insert (Node *root, int key)

↳ if (node == Null) → false

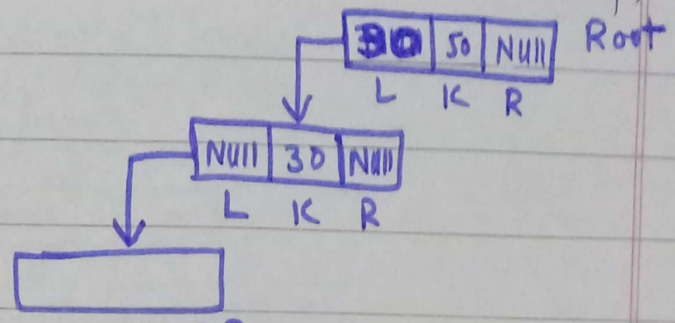
↳ return createNode (key);

}.

20 50

if (key < root → key)

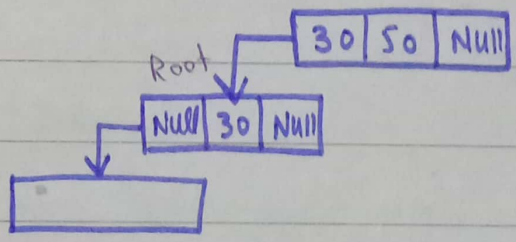
root → left = insert (root → left, key)



20 30

if (key < root → key)

root → left = insert (root → left, key)



if (node == Null) → True

↳ return createNode (key)

∴ go to the createNode function.

Node *createNode (int key)

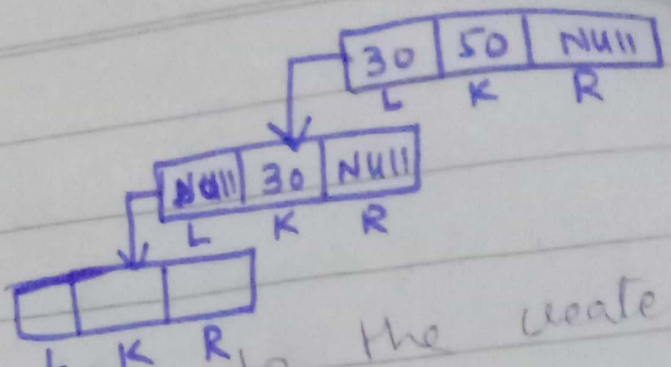
↳

Node * NewNode = newNode;

∴ go to the struct node.

int key

Node * left, * right

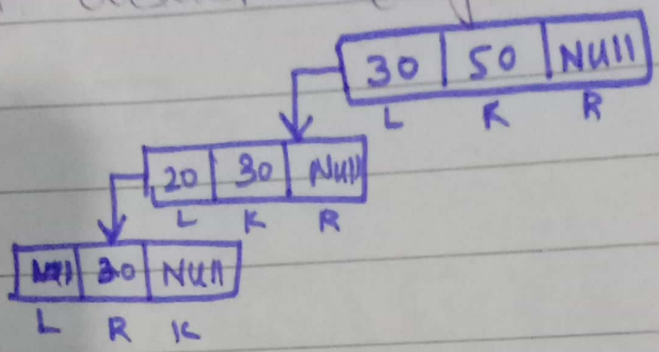


∵ again back to the create Node function

newNode → key = key

newNode → left = newNode → right = Null;

return createNode(key)



∵ again go back to the main function

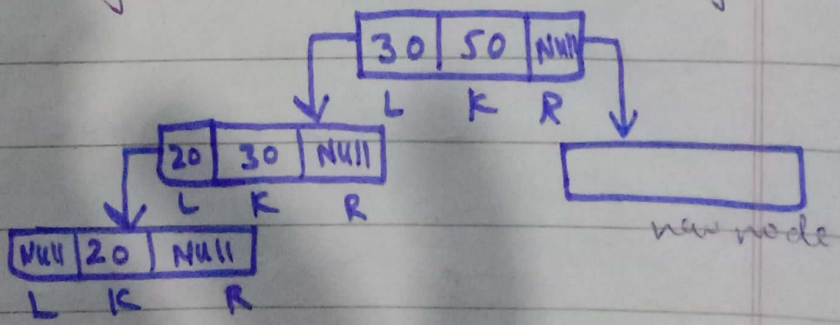
insert ~~node~~ (root, 70)

∵ go to the insert function

70 50

if (key > root → key)

root → right = insert (root → right, key)



if (node == NULL) → True

{

return createNode(key);

∴ go To the createNode function

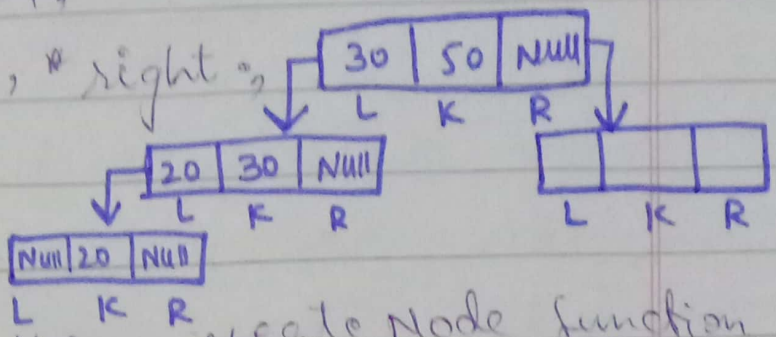
Node *newNode = NewNode

∴ go to the struct node

struct Node {

int key;

struct *left, *right;



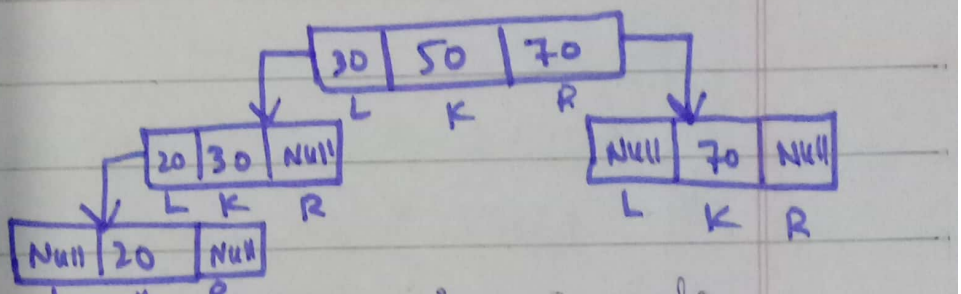
∴ go to the createNode function

70

NewNode → key = key;

NewNode → left = NewNode → right = Null

return NewNode,



∴ Go back to the main function

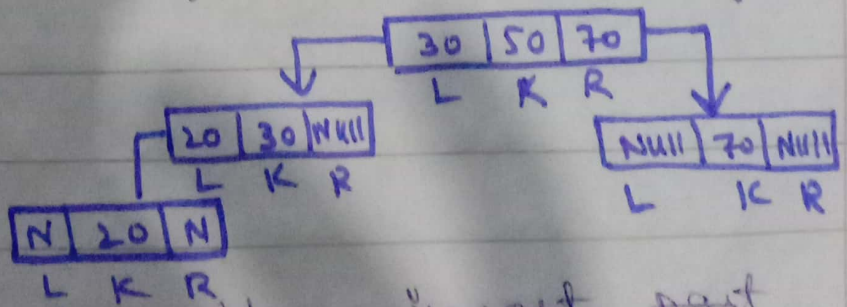
insert(root, 40)

∴ Go to the insert function

40 50

if (key < root → key)

root → left = insert (root → right, key)

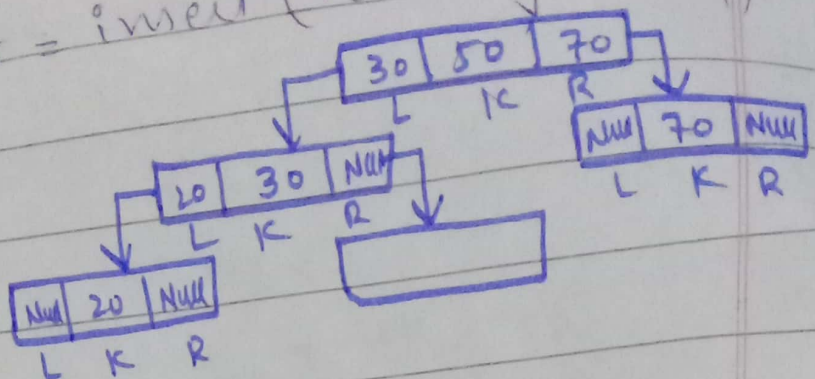


∴ go to the insert part function

40 30

if (key > root->key)

root->right = insert(root->right, key)



∴ go back to insert function

now node is Null.

if (node == NULL) → True

{ return createNewNode(key)

}

∴ go to the createNewNode function.

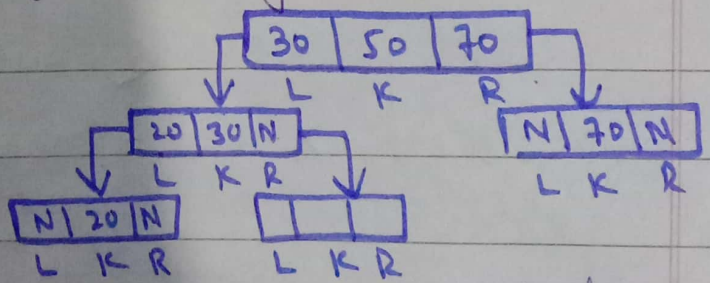
Node * createNode(int key)

{ Node * NewNode = NewNode,

∴ go to the struct node

int key;

Node * left, * right

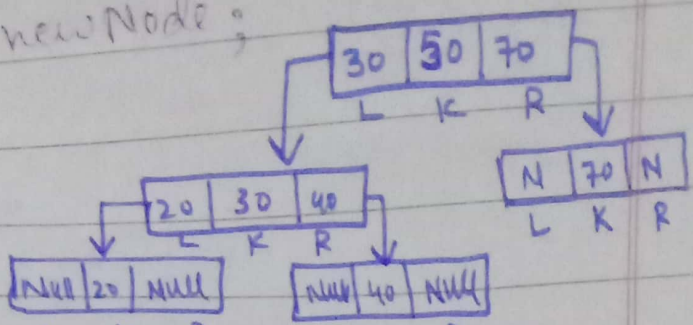


∴ again go to the createNode function

NewNode->key = key

NewNode->left = NewNode->right = NULL;

return newNode;



∴ call Traversal function after insertion

∴ call inorder Traversal (root → left)

inordertraversal(root → left) → 30 because it has 2 child more

∴ again call inorder Traversal (root → left)

inordertraversal (root → left) → 20

∴ 20 node have no child so

print 20

∴ Back to the previous root 30

print 30

∴ again call

inorderTravesal (root → right) → 40

∴ there is no child of node 40 so

print 40

∴ go back to the root 50

print 50

∴ call again

inorder traversal (root → right) → 70

print 70

20, 30, 40, 50, 70