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

// newNode() allocates a new node with the given data and NULL left and
//right pointers.
struct node* newNode(int data)
{
	// Allocate memory for new node
	struct node* node
		= (struct node*)malloc(sizeof(struct node));

	// Assign data to this node
	node->data = data;

	// Initialize left and right children as NULL
	node->left = NULL;
	node->right = NULL;
	return (node);
}

struct node* root=NULL;
struct node* insert(int item, struct node* r)
{
struct node* temp=newNode(item);
	if(r==NULL)
	{
		r=temp;
		return r;
	}
	else
	{
		char direction[20];
		printf("enter direction in uppercase: ");
		scanf("%s",direction);
		struct node* current;
		struct node* prev;
		prev=NULL;
		current=r;
		int i;
		for(i=0;i<strlen(direction)&&current!=NULL;i++)
		{
			prev=current;
			if(direction[i]=='L')
				current=current->left;
```

```
				else
					current=current->right;
			}
			if(current!=NULL||i!=strlen(direction))
			{
				printf("insertion not possible");
				free(temp);
				return r;
			}
			if(direction[i-1]=='L')
				prev->left=temp;
			else
				prev->right=temp;
		}
		return r;
}

struct node* parent(struct node *curr,int ele, struct node *prev)
{
		if(curr!=NULL)
		{
			parent(curr->left, ele, curr);
			if(ele==curr->data)
			{
				printf("\n parent : %d ",prev->data);
				return prev;
			}
			parent(curr->right, ele , curr);
		}
}

//If target is present in tree, then prints the ancestors and returns true, otherwise returns false.
int printAncestors(struct node *root, int target)
{
  /* base cases */
  if (root == NULL)     return(0);
  if (root->data == target)      return(1);
  /* If target is present in either left or right subtree of this node,
     then print this node */
  if ( printAncestors(root->left, target) ||
       printAncestors(root->right, target) )
  {
    printf(" %d ",root->data);
    return 1;
  }
  /* Else return false */
  return 0;
}

int depth(struct node *ptr)
{
```

```
int ldepth,rdepth;
if(ptr==NULL)    return 0;
else
   {   ldepth=depth(ptr->left);
          rdepth=depth(ptr->right);
          if(ldepth>rdepth)          return ldepth+1;
          else                  return rdepth+1;
   }
}
int max1(int a,int b)
{
          int m=(a>b)?a:b;
          return(m);
}
int FindHeight(struct node *r)
{
          if(root==NULL)  return(0);
          return(max1(FindHeight(r->left),FindHeight(r->right))  +1);
}
int compare(struct node *ptr, struct node *temp)
{
   if(ptr)
   {
          if(ptr->left)
                 return 0;
          if(ptr->right) if(ptr->right!=temp->right) return 0;
      return 1;
   }

}
/*
Post order iterative
1. Push root to first stack.
2. Loop while first stack is not empty
2.1 Pop a node from first stack and push it to second stack
2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack
*/
// An iterative function to do post order
// traversal of a given binary tree
void postOrderIterative(struct node* root)
{
if (root == NULL) return;
// Create two stacks
struct node* s1[50]; int top1=-1;
struct node* s2[50]; int top2=-1;
// push root to first stack
s1[++top1]=root;
struct node* node;
// Run while first stack is not empty
while (top1>=0) {
```

```
// Pop an item from s1 and push it to s2
node=s1[top1--];
s2[++top2]=node;

// Push left and right children
// of removed item to s1
if (node->left) s1[++top1]=node->left;
if (node->right) s1[++top1]=node->right;
}//while

// Print all elements of second stack
while (top2>=0) {
node = s2[top2--];
printf(" %d ", node->data);
}
}
/*
2. Iterative Preorder traversal algorithm
1. Create an empty stack nodeStack and push root node to stack
2. Do following while nodeStack is not empty
        a. Pop an item from stack and print it.
        b. Push right child of popped item to stack
        c. Push left child of popped item to stack
*/
// An iterative process to print preorder traversal of Binary tree
void iterativePreorder(struct node *root)
{
   if (root == NULL)       return;      // Base Case
   struct node *curr=root;     // 1. Create an empty stack and push root to it
   struct node* S[50];  int top1=-1;  //single stack
   S[++top1]=curr;
while(top1>=0) {     //2. Do following while nodeStack is not empty
curr=S[top1--];                          //a. Pop an item from stack and print it.
printf(" %d ",curr->data);    //a. printing
if (curr->right) S[++top1]=curr->right;
//b. Push right child of popped item to stack
if (curr->left) S[++top1]=curr->left;
//c. Push left child of popped item to stack
  } //while
} //iterative preorder
/*
Iterative inorder traversal
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S if not NULL and set current = current->left until current is NULL(repeat)
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item,
                              set current =    popped_item->right
    c) Go to step 3.
5) stack is empty then we are done.
```

```
*/
void iterativeinorder(struct node* root)
{
   // Base Case
   if (root == NULL)  return;    //empty tree
struct node *S[50]; int top=-1;                              //1) Create an empty stack S.
struct node *curr=root;              //2) Initialize current node as root
  for(;;){
            for(;curr;curr=curr->left) S[++top]=curr; //3) Push the current node to S //if not NULL and set
current = current->left until current becomes //NULL(repeat)
                        curr=S[top--];
                        if (curr==NULL) break; //5) stack is empty then we are done.
                        printf(" %d ",curr->data);  curr=curr->right;
}
}

struct node* copy(struct node *ptr)
{
   struct node *temp;
   if(ptr)
   {
temp= (struct node*)malloc(sizeof(struct node));
            if(ptr->left) temp->left=copy(ptr->left);
            if(ptr->right) temp->right=copy(ptr->right);
            temp->data=ptr->data;
            return(temp);
   }
   return(NULL);
}

void level_order(struct node *ptr)
{
   int front=-1;
   int rear=-1;
   struct node *Q[10];
   if(!ptr)   return;
   Q[++rear]=ptr;
   do
   {
            ptr=Q[++front];
            if(ptr)
            {
              printf(" %d ",ptr->data);
               if(ptr->left)      Q[++rear]=ptr->left;
               if(ptr->right)     Q[++rear]=ptr->right;
            }
   }while(front!=rear);
}

void rinorder(struct node* node) {
   if (node == NULL) return;
```

```
    rinorder(node->left);
    printf(" %d ", node->data);
    rinorder(node->right);
}
void rpreorder(struct node* node) {
    if (node == NULL) return;
    printf(" %d ", node->data);
    rpreorder(node->left);
    rpreorder(node->right);
}
void rpostorder(struct node* node) {
    if (node == NULL) return;
    rpostorder(node->left);
    rpostorder(node->right);
    printf(" %d ", node->data);
}
void main()
{

        int ch,f,ele;
        int a;
        struct node *p;
do
{
printf("\n1:create,2:pre.3:in,4:post,5:parent \n6. level order 7. depth 8.copy 9. ancestors \n 10.
rpreorder 11. rinorder 12. rpostorder 13.exit\n");
scanf("%d",&ch);
switch(ch)
{
        case 1:  printf("enter element: ");scanf("%d",&a);
                 root=insert(a,root);
                 break;
        case 2: iterativePreorder(root); break;
        case 3:  iterativeinorder(root); break;
        case 4:  postOrderIterative(root); break;
        case 5:
                 printf("enter element: ");scanf("%d",&ele);
                 parent(root,ele,root);
                 break;
        case 6:  level_order(root); break;
        case 7:  printf("\nDepth of the tree: %d \n",depth(root));
                 break;
        case 8:  struct node* ne=copy(root);
                 printf("New tree created with preorder sequence: ");
                 iterativePreorder(ne);
                 break;
        case 9:
                 printf("enter element: ");scanf("%d",&ele);
                 printAncestors(root,ele);
                 break;
        case 10: rpreorder(root); break;
```

```
            case 11: rinorder(root); break;
            case 12: rpostorder(root); break;
            case 13: exit(0);
    }
    }while(1);
    }
```