1.Write a C program to find the depth or height of a tree.

Here is some C code to get the height of the tree

```
tree_height(mynode *p)
{
	if(p==NULL)
		return(0);
	return(max(tree_height(p->left), tree_height(p->right))+1);
}
```

The degree of the leaf is zero. The degree of a tree is the max of its element degrees. A binary tree of height n, h > 0, has at least h and at most (2^h -1) elements in it. The height of a binary tree that contains n, n>0, elements is at most n and atleast log(n+1) to the base 2.
Log(n+1) to the base 2 = h
n = (2^h - 1)

2.Write a C program to determine the number of elements (or size) in a tree.

```
int tree_size(struct node* node)
{
	if (node==NULL)
	{
		return(0);
	}
	else
	{
		return(tree_size(node->left) + tree_size(node->right) + 1);
	}
}
```

3.Write a C program to delete a tree (i.e, free up its nodes)

```
delete(node *ptr)
{
	if(!ptr)
		return;
	delete(ptr->left);
	delete(ptr->right);
	free(ptr);
}
```

4.Write C code to determine if two trees are identical

Here is a C program using recursion

```
int identical(struct node* a, struct node* b)
{
	if (a==NULL && b==NULL)
	{
		return(true);
	}
	else if (a!=NULL && b!=NULL)
	{
		return(a->data == b->data &&
		identical(a->left, b->left) &&
		identical(a->right, b->right));
	}
	else
		return(false);
}
```

5.Write a C program to find the mininum value in a binary search tree

Here is some sample C code. The idea is to keep on moving till you hit the left most node in the tree

```
int minValue(struct node* node)
{
	struct node* current = node;
	while (current->left != NULL)
	{
		current = current->left;
	}
	return(current->data);
}
```

On similar lines, to find the maximum value, keep on moving till you hit the right most node of the tree.

6.Write a C program to create a mirror copy of a tree (left nodes become right and right nodes become left)!

This C code will create a new mirror copy tree.

```
mynode *copy(mynode *root)
{
	mynode *temp;
	if(root==NULL)
		return(NULL);
```

```
        temp = (mynode *) malloc(sizeof(mynode));
        temp->value = root->value;
        temp->left = copy(root->right);
        temp->right = copy(root->left);
        return(temp);
}
```

This code will will only print the mirror of the tree

```
void tree_mirror(struct node* node)
{
        struct node *temp;
        if (node==NULL)
        {
                return;
        }
        else
        {
                tree_mirror(node->left);
                tree_mirror(node->right);
                // Swap the pointers in this node
                temp = node->left;
                node->left = node->right;
                node->right = temp;
        }
}
```

7.Write C code to return a pointer to the nth node of an inorder traversal of a BST.

```
// Get the pointer to the nth inorder node in "nthnode"

void nthinorder(mynode *root, int n, mynode **nthnode)
{
        static whichnode;
        static found;
        if(!found)
        {
                if(root)
                {
                        nthinorder(root->left, n , nthnode);
                        if(++whichnode == n)
                        {
                                printf("\nFound %dth node\n", n);
                                found = 1;
                                *nthnode = root; // Store the pointer to the nth node.
                        }
```

```
                nthinorder(root->right, n , nthnode);
            }
        }
}
```

There seems to be an easier way to do this, or so they say. Suppose each node also has a weight associated with it. This weight is the number of nodes below it and including itself. So, the root will have the highest weight (weight of its left subtree + weight of its right subtree + 1). Using this data, we can easily find the nth inorder node.

Note that for any node, the (weight of the leftsubtree of a node + 1) is its inorder rankin the tree!. Thats simply because of how the inorder traversal works (left->root->right). So calculate the rank of each node and you can get to the nth inorder node easily. But frankly speaking, I really dont know how this method is any simpler than the one I have presented above. I see more work to be done here (calculate thw weights, then calculate the ranks and then get to the nth node!).

Also, if (n > weight(root)), we can error out saying that this tree does not have the nth node you are looking for.

8.Write C code to implement the preorder(), inorder() and postorder() traversals. Whats their time complexities?

Here are the C program snippets to implement these traversals...

Preorder

```
preorder(mynode *root)
{
    if(root)
    {
        printf("Value : [%d]", root->value);
        preorder(root->left);
        preorder(root->right);
    }
}
```

Postorder

```
postorder(mynode *root)
{
    if(root)
    {
        postorder(root->left);
        postorder(root->right);
        printf("Value : [%d]", root->value);
    }
}
```

Inorder

```
inorder(mynode *root)
{
	if(root)
	{
		inorder(root->left);
		printf("Value : [%d]", root->value);
		inorder(root->right);
	}
}
```

Time complexity of traversals is O(n).

9.Write a C program to create a copy of a tree

Here is a C program which does that...

```
mynode *copy(mynode *root)
{
	mynode *temp;
	if(root==NULL)
		return(NULL);
	temp = (mynode *) malloc(sizeof(mynode));
	temp->value = root->value;
	temp->left = copy(root->left);
	temp->right = copy(root->right);
	return(temp);
}
```

10.Write C code to check if a given binary tree is a binary search tree or not?

Here is a C program which checks if a given tree is a Binary Search Tree or not...

```
int isThisABST(struct node* mynode)
{
	if (mynode==NULL)
		 return(true);
	if (node->left!=NULL && maxValue(mynode->left) > mynode->data)
		return(false);
	if (node->right!=NULL && minValue(mynode->right) <= mynode->data)
		return(false);
	if (!isThisABST(node->left) || !isThisABST(node->right))
		return(false);
```

```
	return(true);
}
```

11.Write a C program to delete a node from a Binary Search Tree?

The node to be deleted might be in the following states

- The node does not exist in the tree - In this case you have nothing to delete.
- The node to be deleted has no children - The memory occupied by this node must be freed and either the left link or the right link of the parent of this node must be set to NULL.
- The node to be deleted has exactly one child - We have to adjust the pointer of the parent of the node to be deleted such that after deletion it points to the child of the node being deleted.
- The node to be deleted has two children - We need to find the inorder successor of the node to be deleted. The data of the inorder successor must be copied into the node to be deleted and a pointer should be setup to the inorder successor. This inorder successor would have one or zero children. This node should be deleted using the same procedure as for deleting a one child or a zero child node. Thus the whole logic of deleting a node with two children is to locate the inorder successor, copy its data and reduce the problem to a simple deletion of a node with one or zero children.

Implementation:

```
tree *del(tree *root)
{
	if(!root)
	{
		cout<<"Tree is empty\n";
		return root;
	}
	else
	{
		int num;
		cout<<"Enter the element to be deleted:";
		cin>>num;
		tree *parent,*temp,*q;
		temp = root;
		parent = NULL;
		while(temp != NULL && temp->data != num)
		{
			parent = temp;
			temp = (num>temp->data)?temp->right:temp->left;
		}
		if(temp == NULL)
		{
```

```
				cout<<"Element not found\n";
				return root;
			}
			else
			{
				if(temp->left == NULL)
					q = temp->right;
				else
				{
					if(temp->right == NULL)
						q = temp->left;
					else
					{
						int value;
						q = temp;
						parent = NULL;
						temp = temp->right;
						while(temp->left != NULL)
						{
							parent = temp;
							temp = temp->left;
						}
						value =  temp->data;
						if(parent == NULL)
							q->right = temp->right;
						else
							parent->left = temp->left;
						delete temp;
						q->data = value;
						return root;
					}
				}
			}
			if(parent == NULL)
			{
				delete temp;
				return q;
			}
			else
				if(parent->left == temp)
					parent->left = q;
				else
					if(parent->right == temp)
						parent->right = q;
			delete temp;
			return root;
		}
	}
```

12.Write C code to search for a value in a binary search tree (BST).

```
mynode *search(int value, mynode *root)
{
        while(root!=NULL && value!=root->value)
        {
        root = (value < root->value)?root->left:root->right;
        }
        return(root);
}
```

13.Write C code to count the number of leaves in a tree

```
void count_leaf(mynode *root)
{
        if(root!=NULL)
        {
                count_leaf(root->left);
                if(root->left == NULL && root->right==NULL)
                {
                        // This is a leaf!
                        count++;
                }
                count_leaf(root->right);
        }
}
```

14.Construct a tree given its inorder and preorder traversal strings. Similarly construct a tree given its inorder and post order traversal strings.

For Inorder And Preorder traversals
inorder = g d h b e i a f j c
preorder = a b d g h e i c f j

Scan the preorder left to right using the inorder sequence to separate left and right subtrees. For example, "a" is the root of the tree;"gdhbei" are in the left subtree; "fjc" are in the right subtree. "b" is the next root; "gdh" are in the left subtree; "ei" are in the right subtree. "d" is the next root; "g" is in the left subtree; "h" is in the right subtree.

For Inorder and Postorder traversals

Scan postorder from right to left using inorder to separate left and right subtrees.
inorder = g d h b e i a f j c
postorder = g h d i e b j f c a

Tree root is "a"; "gdhbei" are in left subtree; "fjc" are in right subtree.

For Inorder and Levelorder traversals

Scan level order from left to right using inorder to separate left and right subtrees.
inorder = g d h b e i a f j c
level order = a b c d e f g h i j
Tree root is "a"; "gdhbei" are in left subtree; "fjc" are in right subtree.

Here is some working code which creates a tree out of the Inorder and Postorder traversals. Note that here the tree has been represented as an array. This really simplifies the whole implementation.
Converting a tree to an array is very easy
Suppose we have a tree like this

```
        A
      B     C
    D  E  F    G
```

The array representation would be

```
a[1] a[2] a[3] a[4] a[5] a[6] a[7]
A     B     C     D     E     F     G
```

That is, for every node at position j in the array, its left child will be stored at position (2*j) and right child at (2*j + 1). The root starts at position 1.

```
// CONSTRUCTING A TREE GIVEN THE INORDER AND PREORDER SEQUENCE
#include<stdio.h>
#include<string.h>
#include<ctype.h>
/*-------------------------------------------------------------
* Algorithm
*
* Inorder And Preorder
* inorder = g d h b e i a f j c
* preorder = a b d g h e i c f j
* Scan the preorder left to right using the inorder to separate left
* and right subtrees. a is the root of the tree; gdhbei are in the
* left subtree; fjc are in the right subtree.
*-----------------------------------------------------------*/
static char io[]="gdhbeiafjc";
static char po[]="abdgheicfj";
static char t[100][100]={'\0'}; //This is where the final tree will be stored
static int hpos=0;
void copy_str(char dest[], char src[], int pos, int start, int end);
void print_t();
int main(int argc, char* argv[])
{
	int i,j,k;
	char *pos;
	int posn;
```

```
	// Start the tree with the root and its
	// left and right elements to start off
	for(i=0;i<strlen(io);i++)
	{
		if(io[i]==po[0])
		{
			copy_str(t[1],io,1,i,i); // We have the root here
			copy_str(t[2],io,2,0,i-1); // Its left subtree
			copy_str(t[3],io,3,i+1,strlen(io)); // Its right subtree
			print_t();
		}
	}
	// Now construct the remaining tree
	for(i=1;i<strlen(po);i++)
	{
		for(j=1;j<=hpos;j++)
		{
			if((pos=strchr((const char *)t[j],po[i]))!=(char *)0 && strlen(t[j])!=1)
			{
				for(k=0;k<strlen(t[j]);k++)
				{
					if(t[j][k]==po[i]){posn=k;break;}
				}
				printf("\nSplitting [%s] for po[%d]=[%c] at %d..\n", t[j],i,po[i],posn);
				copy_str(t[2*j],t[j],2*j,0,posn-1);
				copy_str(t[2*j+1],t[j],2*j+1,posn+1,strlen(t[j]));
				copy_str(t[j],t[j],j,posn,posn);
				print_t();
			}
		}
	}
}
// This function is used to split a string into three seperate strings
// This is used to create a root, its left subtree and its right subtree
void copy_str(char dest[], char src[], int pos, int start, int end)
{
	char mysrc[100];
	strcpy(mysrc,src);
	dest[0]='\0';
	strncat(dest,mysrc+start,end-start+1);
	if(pos>hpos)
		hpos=pos;
}
void print_t()
{
	int i;
	for(i=1;i<=hpos;i++)
	{
```

```
		printf("\nt[%d] = [%s]", i, t[i]);
	}
	printf("\n");
}
```

15.Given an expression tree, evaluate the expression and obtain a paranthesized form of the expression.

The code below prints the paranthesized form of a tree.

```
infix_exp(p)
{
	if(p)
	{
		printf("(");
		infix_exp(p->left);
		printf(p->data);
		infix_exp(p->right);
		printf(")");
	}
}
```

Creating a binary tree for a postfix expression

```
mynode *create_tree(char postfix[])
{
	mynode *temp, *st[100];
	int i,k;
	char symbol;
	for(i=k=0; (symbol = postfix[i])!='\0'; i++)
	{
		temp = (mynode *) malloc(sizeof(struct node));
		temp->value = symbol;
		temp->left = temp->right = NULL;
		if(isalnum(symbol))
			st[k++] = temp;
		else
		{
			temp->right = st[--k];
			temp->left = st[--k];
			st[k++] = temp;
		}
	}
	return(st[--k]);
}
```

Evaluate a tree

```
float eval(mynode *root)
{
	float num;
	switch(root->value)
	{
		case '+' : return(eval(root->left) + eval(root->right)); break;
		case '-' : return(eval(root->left) - eval(root->right)); break;
		case '/' : return(eval(root->left) / eval(root->right)); break;
		case '*' : return(eval(root->left) * eval(root->right)); break;
		case '$' : return(eval(root->left) $ eval(root->right)); break;
		default : if(isalpha(root->value))
			 {
				printf("%c = ", root->value);
				scanf("%f", &num);
				return(num);
			 }
			else
			{
				return(root->value - '0');
			}
	}
}
```

16.How do you convert a tree into an array?

The conversion is based on these rules
If i > 1, i/2 is the parent
If 2*i > n, then there is no left child, else 2*i is the left child.
If (2*i + 1) > n, then there is no right child, else (2*i + 1) is the right child.
Converting a tree to an array is very easy

Suppose we have a tree like this

```
     A
   B   C
  D E F G
```

The array representation would be

```
a[1] a[2] a[3] a[4] a[5] a[6] a[7]
  A    B    C    D    E    F    G
```

That is, for every node at position i in the array, its left child will be stored at position (2*i) and right child at (2*i + 1). The root starts at position 1.