IX. BINARY SEARCH TREES

Uptil now all data structures that we have covered (Stack,Queue,Linked List) are linear in nature ie. they have a definate order of placement. Now we shall study Binary Trees which requires a different thought process as it is a non linear data structure.

A Binary Tree consists of a main node known as the Root. The Root then has two sub-sections, ie. the left and the right half. The data subsequently stored after the root is created depends on it's value compared to the root.
Suppose the root value is 10 and the Value to be added is 15, then the data is added to the right section of the root.
The Basic idea is that every node can be thought of a binary tree itself. Each node has two pointers, one to the left and the other to the right. Depending on the value to be stored, it is placed after a node's right pointer if the value of the node is lesser than the one to be added or the node's left pointer if viceversa.

Let's take an Example. To add the Following List of Numbers, we end up with a Binary Tree like this:

32 16 34 1 87 13 7 18 14 19 23 24 41 5 53

http://www.dreamincode.net/forums/index.php?act=Attach&type=post&id=7756

Here's How:
**: KEEP ADDING DATA IN THE TREE ON PAPER AFTER EACH STEP BELOW TO UNDERSTAND
HOW THE TREE IS FORMED.

- Since **32** is the First Number to be added, 32 becomes the root of the tree.
- Next Number is **16** which is lesser than 32 Hence 16 becomes left node of 32.
- **34**. Since 34 > 32 , 34 becomes the right node of the ROOT.
- **1**. Since 1 < 32 we jump to the left node of the ROOT. But since the left node has already been taken we test 1 once again. Since 1 < 16, 1 becomes the left node of 16.
- **87**. Since 87 > 32 we jump to the right node of the root. Once again this space is occupied by 34. Now since 87 > 34, 87 becomes the right node of 34.
- **13**. Since 13 < 32 we jump to left node of the root. There, 13 < 16 so we continue towards the left node of 16. There 13 > 1, so 13 becomes the right node of 1.
- Similarly work out addition till the end ie. before Number **53**.
- **53**. Since 53 > 32 we jump to the right node of the root. There 53 > 34 so we continue to the right node of 34. There 53 < 87 so we continue towards the left node of 87. There 53 > 41 so we jump to the right node of 41. Since the Right node of 41 is empty 53 becomes the right node of 41.

This should give you an idea of how a Binary Tree works. You must know that:

- The linking of nodes to nodes in a Binary Tree is one to one in nature ie. a node cannot be pointed by more than 1 node.
- A Node can point to two different sub-nodes at the most.

Here in the binary tree above there are a few nodes whose left and right pointers are empty ie. they have no sub-node attached to them. So Nodes 5,14,18, 19,23,24,41 have their left nodes empty.

There are three popular ways to display a Binary Tree. Displaying the trees contents is known as transversal. There are three ways of transversing a tree iw. in inorder,preorder and postorder transversal methods. Description of each is shown below:

PREORDER:

- Visit the root.
- Transverse the left leaf in preorder.
- Transverse the right leaf in preorder.

INORDER:

- Transverse the left leaf in inorder.
- Visit the root.
- Transverse the right leaf in inorder.

POSTORDER:

- Transverse the left leaf in postorder.
- Transverse the right leaf in postorder.
- Visit the root.

Writing code for these three methods are simple if we understand the recursive nature of a binary tree. Binary tree is recursive, as in each node can be thought of a binary tree itself. It's just the order of displaying data that makes a difference for transversal.

Deletion from a Binary Tree is a bit more difficult to understand. For now just remember that for deleting a node, it is replaced with it's next inorder successor. I'll explain everything after the Binary Tree code.
Now that you've got all your Binary Tree Fundas clear, let's move on with the Source code.

```
#include <iostream>

using namespace std;

#define YES 1
#define NO 0

class tree
{
    private:
        struct leaf
        {
            int data;
            leaf *l;
            leaf *r;
        };
        struct leaf *p;

    public:
        tree();
        ~tree();
        void destruct(leaf *q);
        tree(tree& a);
        void findparent(int n,int &found,leaf* &parent);
        void findfordel(int n,int &found,leaf *&parent,leaf* &x);
        void add(int n);
        void transverse();
        void in(leaf *q);
        void pre(leaf *q);
        void post(leaf *q);
        void del(int n);
};

tree::tree()
{
    p=NULL;
}

tree::~tree()
{
```

```
    destruct(p);
}

void tree::destruct(leaf *q)
{
    if(q!=NULL)
    {
        destruct(q->l);
        del(q->data);
        destruct(q->r);
    }
}
void tree::findparent(int n,int &found,leaf *&parent)
{
    leaf *q;
    found=NO;
    parent=NULL;

    if(p==NULL)
        return;

    q=p;
    while(q!=NULL)
    {
        if(q->data==n)
        {
            found=YES;
            return;
        }
        if(q->data>n)
        {
            parent=q;
            q=q->l;
        }
        else
        {
            parent=q;
            q=q->r;
        }
    }
```

```
}

void tree::add(int n)
{
    int found;
    leaf *t,*parent;
    findparent(n,found,parent);
    if(found==YES)
        cout<<"\nSuch a Node Exists";
    else
    {
        t=new leaf;
        t->data=n;
        t->l=NULL;
        t->r=NULL;

        if(parent==NULL)
            p=t;
        else
            parent->data > n ? parent->l=t : parent->r=t;
    }
}

void tree::transverse()
{
    int c;
    cout<<"\n1.InOrder\n2.Preorder\n3.Postorder\nChoice: ";
    cin>>c;
    switch©
    {
        case 1:
            in(p);
            break;

        case 2:
            pre(p);
            break;

        case 3:
            post(p);
```

```
            break;
    }
}

void tree::in(leaf *q)
{
    if(q!=NULL)
    {
        in(q->l);
        cout<<"\t"<<q->data<<endl;
        in(q->r);
    }

}

void tree::pre(leaf *q)
{
    if(q!=NULL)
    {
        cout<<"\t"<<q->data<<endl;
        pre(q->l);
        pre(q->r);
    }

}

void tree::post(leaf *q)
{
    if(q!=NULL)
    {
        post(q->l);
        post(q->r);
        cout<<"\t"<<q->data<<endl;
    }

}

void tree::findfordel(int n,int &found,leaf *&parent,leaf *&x)
{
    leaf *q;
```

```
    found=0;
    parent=NULL;
    if(p==NULL)
        return;

    q=p;
    while(q!=NULL)
    {
        if(q->data==n)
        {
            found=1;
            x=q;
            return;
        }
        if(q->data>n)
        {
            parent=q;
            q=q->l;
        }
        else
        {
            parent=q;
            q=q->r;
        }
    }
}

void tree::del(int num)
{
    leaf *parent,*x,*xsucc;
    int found;

    // If EMPTY TREE
    if(p==NULL)
    {
        cout<<"\nTree is Empty";
        return;
    }
    parent=x=NULL;
    findfordel(num,found,parent,x);
```

```
    if(found==0)
    {
        cout<<"\nNode to be deleted NOT FOUND";
        return;
    }

    // If the node to be deleted has 2 leaves
    if(x->l != NULL && x->r != NULL)
    {
        parent=x;
        xsucc=x->r;

        while(xsucc->l != NULL)
        {
            parent=xsucc;
            xsucc=xsucc->l;
        }
        x->data=xsucc->data;
        x=xsucc;
    }

    // if the node to be deleted has no child
    if(x->l == NULL && x->r == NULL)
    {
        if(parent->r == x)
            parent->r=NULL;
        else
            parent->l=NULL;

        delete x;
        return;
    }

    // if node has only right leaf
    if(x->l == NULL && x->r != NULL )
    {
        if(parent->l == x)
            parent->l=x->r;
        else
            parent->r=x->r;
```

```

        delete x;
        return;
    }

    // if node to be deleted has only left child
    if(x->l != NULL && x->r == NULL)
    {
        if(parent->l == x)
            parent->l=x->l;
        else
            parent->r=x->l;

        delete x;
        return;
    }
}

int main()
{
    tree t;
    int data[]={32,16,34,1,87,13,7,18,14,19,23,24,41,5,53};
    for(int iter=0 ; iter < 15 ; i++)
        t.add(data[iter]);

    t.transverse();
    t.del(16);
    t.transverse();
    t.del(41);
    t.transverse();
    return 0;
}
```

OUTPUT:
1.InOrder
2.Preorder
3.Postorder
Choice: 1
1
5
7
13
14
16
18
19
23
24
32
34
41
53
87

1.InOrder
2.Preorder
3.Postorder
Choice: 2
32
18
1
13
7
5
14
19
23
24
34
87
41
53

1.InOrder
2.Preorder
3.Postorder
Choice: 3
5
7

14
13
1
24
23
19
18
53
87
34
32
Press any key to continue

NOTE: Visual C++ may give Runtime Errors with this code. Compile with Turbo C++.

Just by looking at the output you might realise that we can print out the whole tree in ascending order by using inorder transversal. Infact Binary Trees are used for Searching [Binary Search Trees {BST}] as well as in Sorting.
The Addition of data part seems fine. Only the deletion bit needs to be explained.

For deletion of data there are a few cases to be considered:

- If the leaf to be deleted is not found.
- If the leaf to be deleted has no sub-leafs.
- If the leaf to be deleted has 1 sub-leaf.
- If the leaf to be deleted has 2 sub-leafs.

CASE 1:
Dealing with this case is simple, we simply display an error message.

CASE 2:
Since the node has no sub-nodes, the memory occupied by this should be freed and either the left link or the right link of the parent of this node should be set to NULL. Which of these should be set to NULL depends upon whether the node being deleted is a left child or a right child of its parent.

CASE 3:
In the third case we just adjust the pointer of the parent of the leaf to be deleted such that after deletion it points to the child of the node being deleted.

CASE 4:
The last case in which the leaf to be deleted has to sub-leaves of its own is rather complicated.The whole logic is to locate the inorder successor, copy it's data and reduce the problem to simple deletion of a node with one or zero leaves. Consider in the above program...(Refer to the previous tree as well) when we are deleting 16 we search for the next inorder successor. So we simply set the data value to 5 and delete the node with value 5 as shown for cases 2 and 3.

That's It! *phew*
Binary Trees are used for various other things which even include Compression algorithms,binary searching,sorting etc. A lot of Huffman, Shannon-Fano and other Compression algorithms use Binary Trees. If you want source code of these Compression codes you can freely contact me at my email.