A binary tree implemented in C++

Header file for a binary tree

```
//---------------------------------------------------------------
// File: Code202_Tree.h
// Purpose: Header file for a demonstration of a binary tree
// Programming Language: C++
// Author: Dr. Rick Coleman
//---------------------------------------------------------------
#ifndef CODE202_TREE_H
#define CODE202_TREE_H

#include <iostream>
using namespace std;

// Define a structure to be used as the tree node
struct TreeNode
{
    int      Key;
    float    fValue;
    int      iValue;
    char     cArray[7];
    TreeNode *left;
    TreeNode *right;
};

class Code202_Tree
{
    private:
        TreeNode *root;

    public:
        Code202_Tree();
        ~Code202_Tree();
        bool isEmpty();
        TreeNode *SearchTree(int Key);
        int Insert(TreeNode *newNode);
        int Insert(int Key, float f, int i, char *cA);
        int Delete(int Key);
        void PrintOne(TreeNode *T);
        void PrintTree();
    private:
        void ClearTree(TreeNode *T);
        TreeNode *DupNode(TreeNode * T);
        void PrintAll(TreeNode *T);
};

#endif
```

Implementation (.cpp) file for a binary tree

```
//---------------------------------------------------------------
// File: Code202_Tree.c
```

```
// Purpose: Implementation file for a demonstration of a binary tree
// Programming Language: C++
// Author: Dr. Rick Coleman
// Date: February, 2002
//---------------------------------------------------------------
#include <iostream>
#include <string.h>
#include "Code202_Tree.h"

using namespace std;

//-----------------------------------------
// Function: Code202_Tree()
// Purpose: Class constructor.
//-----------------------------------------
Code202_Tree::Code202_Tree()
{
    root = NULL;
    return;
}

//-----------------------------------------
// Function: Code202_Tree()
// Purpose: Class destructor.
//-----------------------------------------
Code202_Tree::~Code202_Tree()
{
    ClearTree(root);
    return;
}

//-----------------------------------------
// Function: ClearTree()
// Purpose: Perform a recursive traversal of
//        a tree destroying all nodes.
//-----------------------------------------
void Code202_Tree::ClearTree(TreeNode *T)
{
    if(T==NULL) return;  // Nothing to clear
    if(T->left != NULL) ClearTree(T->left); // Clear left sub-tree
    if(T->right != NULL) ClearTree(T->right); // Clear right sub-tree
    delete T;    // Destroy this node
    return;
}

//-----------------------------------------
// Function: isEmpty()
// Purpose: Return TRUE if tree is empty.
//-----------------------------------------
bool Code202_Tree::isEmpty()
{
    return(root==NULL);
}

//-----------------------------------------
// Function: DupNode()
// Purpose: Duplicate a node in the tree.  This
//        is used to allow returning a complete
//        structure from the tree without giving
//        access into the tree through the pointers.
// Preconditions: None
```

```
// Returns: Pointer to a duplicate of the node arg
//-----------------------------------------
TreeNode *Code202_Tree::DupNode(TreeNode * T)
{
    TreeNode *dupNode;

    dupNode = new TreeNode();
    *dupNode = *T;    // Copy the data structure
    dupNode->left = NULL;    // Set the pointers to NULL
    dupNode->right = NULL;
    return dupNode;
}

//-----------------------------------------
// Function: SearchTree()
// Purpose: Perform an iterative search of the tree and
//        return a pointer to a treenode containing the
//        search key or NULL if not found.
// Preconditions: None
// Returns: Pointer to a duplicate of the node found
//-----------------------------------------
TreeNode *Code202_Tree::SearchTree(int Key)
{
    int      ValueInTree = false;
    TreeNode *temp;

    temp = root;
    while((temp != NULL) && (temp->Key != Key))
    {
        if(Key < temp->Key)
            temp = temp->left;  // Search key comes before this node.
        else
            temp = temp->right; // Search key comes after this node
    }
    if(temp == NULL) return temp;    // Search key not found
    else
        return(DupNode(temp));    // Found it so return a duplicate
}

//-----------------------------------------
// Function: Insert()
// Insert a new node into the tree.
// Preconditions: None
// Returns: int (TRUE if successful, FALSE otherwise)
//-----------------------------------------
int Code202_Tree::Insert(TreeNode *newNode)
{
    TreeNode *temp;
    TreeNode *back;

    temp = root;
    back = NULL;

    while(temp != NULL) // Loop till temp falls out of the tree
    {
        back = temp;
        if(newNode->Key < temp->Key)
            temp = temp->left;
        else
            temp = temp->right;
    }
```

```
    // Now attach the new node to the node that back points to
    if(back == NULL) // Attach as root node in a new tree
        root = newNode;
    else
    {
        if(newNode->Key < back->Key)
            back->left = newNode;
        else
            back->right = newNode;
    }
   return(true);
}

//-----------------------------------------
// Function: Insert()
// Insert a new node into the tree.
// Preconditions: None
// Returns: int (TRUE if successful, FALSE otherwise)
//-----------------------------------------
int Code202_Tree::Insert(int Key, float f, int i, char *cA)
{
    TreeNode *newNode;

    // Create the new node and copy data into it
    newNode = new TreeNode();
    newNode->Key = Key;
    newNode->fValue = f;
    newNode->iValue = i;
    strcpy(newNode->cArray, cA);
    newNode->left = newNode->right = NULL;

    // Call other Insert() to do the actual insertion
    return(Insert(newNode));
}

//-----------------------------------------
// Function: Delete()
// Purpose: Delete a node from the tree.
// Preconditions: Tree contains the node to delete
// Returns: int (TRUE if successful, FALSE otherwise)
//-----------------------------------------
int Code202_Tree::Delete(int Key)
{
    TreeNode *back;
    TreeNode *temp;
    TreeNode *delParent;    // Parent of node to delete
    TreeNode *delNode;      // Node to delete

    temp = root;
    back = NULL;

    // Find the node to delete
    while((temp != NULL) && (Key != temp->Key))
    {
        back = temp;
        if(Key < temp->Key)
            temp = temp->left;
        else
            temp = temp->right;
    }
```

```
    if(temp == NULL) // Didn't find the one to delete
    {
        cout << "Key not found. Nothing deleted.\n";
        return false;
    }
    else
    {
        if(temp == root) // Deleting the root
        {
            delNode = root;
            delParent = NULL;
        }
        else
        {
            delNode = temp;
            delParent = back;
        }
    }

    // Case 1: Deleting node with no children or one child
    if(delNode->right == NULL)
    {
        if(delParent == NULL)    // If deleting the root
        {
            root = delNode->left;
            delete delNode;
            return true;
        }
        else
        {
            if(delParent->left == delNode)
                delParent->left = delNode->left;
            else
                delParent->right = delNode->left;
                delete delNode;
            return true;
        }
    }
    else // There is at least one child
    {
        if(delNode->left == NULL)    // Only 1 child and it is on the right
        {
            if(delParent == NULL)    // If deleting the root
            {
                root = delNode->right;
                delete delNode;
                return true;
            }
            else
            {
                if(delParent->left == delNode)
                    delParent->left = delNode->right;
                else
                    delParent->right = delNode->right;
                delete delNode;
                return true;
            }
        }
        else // Case 2: Deleting node with two children
        {
```

```
            // Find the replacement value.  Locate the node
            // containing the largest value smaller than the
            // key of the node being deleted.
            temp = delNode->left;
            back = delNode;
            while(temp->right != NULL)
            {
                back = temp;
                temp = temp->right;
            }
            // Copy the replacement values into the node to be deleted
            delNode->Key = temp->Key;
            delNode->fValue = temp->fValue;
            delNode->iValue = temp->iValue;
            strcpy(delNode->cArray, temp->cArray);

            // Remove the replacement node from the tree
            if(back == delNode)
                back->left = temp->left;
            else
                back->right = temp->left;
            delete temp;
            return true;
        }
    }
}

//-----------------------------------------
// Function: PrintOne()
// Purpose: Print data in one node of a tree.
// Preconditions: None
// Returns: void
//-----------------------------------------
void Code202_Tree::PrintOne(TreeNode *T)
{
    cout << T->Key << "\t\t" << T->fValue << "\t\t" << T->iValue << "\t\t"
        << T->cArray << "\n";
}

//-----------------------------------------
// Function: PrintAll()
// Purpose: Print the tree using a recursive
//        traversal
// Preconditions: None
// Returns: void
//-----------------------------------------
void Code202_Tree::PrintAll(TreeNode *T)
{
    if(T != NULL)
    {
        PrintAll(T->left);
        PrintOne(T);
        PrintAll(T->right);
    }
}

//-----------------------------------------
// Function: PrintTree()
// Purpose: Print the tree using a recursive
//        traversal.  This gives the user access
//        to PrintAll() without giving access to
```

```
//        the root of the tree.
// Preconditions: None
// Returns: void
//-----------------------------------------
void Code202_Tree::PrintTree()
{
    PrintAll(root);
}
```

Main file used to test the tree

```
//-----------------------------------------------------------------
// File: TreeMain.cpp
// Purpose: Main file for a demonstration of a binary tree
// Programming Language: C
// Author: Dr. Rick Coleman
// Date: February, 2002
//-----------------------------------------------------------------
#include <iostream>
#include <string.h>
#include "Code202_Tree.h"

using namespace std;

int main(void)
{
    Code202_Tree    *theTree;
    TreeNode        *newNode;

    // Do initialization stuff
    theTree = new Code202_Tree();

    cout <<"Building tree...\n";
    // Do simple insert of 15 nodes into tree.
    // Insert with keys in the order.
    //   8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15
    // First 5 nodes are inserted using Insert1(). Remainder using Insert2()

    // Node 1
    newNode = new TreeNode();
    newNode->Key = 8;
    newNode->iValue = 2;
    newNode->fValue = 2.3f;
    strcpy(newNode->cArray, "Node1");
    newNode->left = newNode->right = NULL;
    theTree->Insert(newNode);

    // Node 2
    // Note: Each time a new node is allocated we reuse the same pointer
    // Access to the previous node is not lost because it is not in the tree.
    newNode = new TreeNode();
    newNode->Key = 4;
    newNode->iValue = 4;
    newNode->fValue = 3.4f;
    strcpy(newNode->cArray, "Node2");
    newNode->left = newNode->right = NULL;
    theTree->Insert(newNode);

    // Node 3
    newNode = new TreeNode();
```

```
    newNode->Key = 12;
    newNode->iValue = 8;
    newNode->fValue = 4.5f;
    strcpy(newNode->cArray, "Node3");
    newNode->left = newNode->right = NULL;
    theTree->Insert(newNode);

    // Node 4
    newNode = new TreeNode();
    newNode->Key = 2;
    newNode->iValue = 16;
    newNode->fValue = 5.6f;
    strcpy(newNode->cArray, "Node4");
    newNode->left = newNode->right = NULL;
    theTree->Insert(newNode);

    // Node 5
    newNode = new TreeNode();
    newNode->Key = 6;
    newNode->iValue = 32;
    newNode->fValue = 6.7f;
    strcpy(newNode->cArray, "Node5");
    newNode->left = newNode->right = NULL;
    theTree->Insert(newNode);

    // Node 6
    // Remainder of the nodes are inserted using Insert2()
    theTree->Insert(10, 7.8f, 64, "Node6");

    // Node 7
    theTree->Insert(14, 8.9f, 128, "Node7");

    // Node 8
    theTree->Insert(1, 9.0f, 256, "Node8");

    // Node 9
    theTree->Insert(3, 0.9f, 512, "Node9");

    // Node 10
    theTree->Insert(5, 9.8f, 1024, "Node10");

    // Node 11
    theTree->Insert(7, 8.7f, 2048, "Node11");

    // Node 12
    theTree->Insert(9, 7.6f, 4096, "Node12");

    // Node 13
    theTree->Insert(11, 6.5f, 8192, "Node13");

    // Node 14
    theTree->Insert(13, 5.4f, 16384, "Node14");

    // Node 15
    theTree->Insert(15, 4.3f, 32768, "Node15");

    cout <<"All nodes inserted\n";

    // Print the tree
    cout <<"-----------------------------------------------------\n";
    theTree->PrintTree();
```

```
    cout <<"Press Enter to continue...";
    cin.get();
    cout <<"-----------------------------------------------------\n";

    // Find some nodes and print them
    cout <<"-----------------------------------------------------\n";
    cout <<"Testing the search function\n";
    newNode = theTree->SearchTree(13);
    if(newNode != NULL)
    {
        theTree->PrintOne(newNode);
        delete newNode; // Remember this is a duplicate node of the one in
                        // in the tree and main() is responsible for deleting it.
    }
    else
        cout <<"Search key not found.\n";

    newNode = theTree->SearchTree(6);
    if(newNode != NULL)
    {
        theTree->PrintOne(newNode);
        delete newNode;
    }
    else
        cout <<"Search key not found.\n";

    newNode = theTree->SearchTree(1);
    if(newNode != NULL)
    {
        theTree->PrintOne(newNode);
        delete newNode;
    }
    else
        cout <<"Search key not found.\n";

    newNode = theTree->SearchTree(25); // Note: there is no Key=25 in the tree
    if(newNode != NULL)
    {
        theTree->PrintOne(newNode);
        delete newNode;
    }
    else
        cout <<"Search key not found.\n";

    // Delete some nodes
    cout <<"-----------------------------------------------------\n";
    cout <<"Testing delete function\n";
    cout <<"-----------------------------------------------------\n";
    cout <<"Testing deleting a leaf...\n";
    theTree->Delete(7);    // Delete a known leaf
    theTree->PrintTree();
     cout <<"Press Enter to continue...";
    cin.get();
    cout <<"-----------------------------------------------------\n";

    cout <<"-----------------------------------------------------\n";
    cout <<"Testing deleting a node with 2 children...\n";
    theTree->Delete(12);    // Delete a node known to have 2 children
    theTree->PrintTree();
    cout <<"Press Enter to continue...";
    cin.get();
```

```
    cout <<"-------------------------------------------------------\n";

    cout <<"-------------------------------------------------------\n";
    cout <<"Testing deleting a node with 1 child...\n";
    theTree->Delete(6);    // Delete a node known to have only 1 child
    theTree->PrintTree();
    cout <<"Press Enter to continue...";
    cin.get();
    cout <<"-------------------------------------------------------\n";

    cout <<"-------------------------------------------------------\n";
    cout <<"Testing trying to delete a node that is not in the tree...\n";
    theTree->Delete(7);    // Delete a node that is not there
    theTree->PrintTree();
    cout <<"Press Enter to continue...";
    cin.get();
    cout <<"-------------------------------------------------------\n";

    cout <<"-------------------------------------------------------\n";
    cout <<"Testing deleting the root...\n";
    theTree->Delete(8);    // Delete the root
    theTree->PrintTree();
    cout <<"Done.\nPress Enter to continue...";
    cin.get();

    cout <<"-------------------------------------------------------\n";
    return 0;
}
```