```
// TREVER WAGENHALS
// PROGRAM 10

#ifndef BST_T_H
#define BST_T_H
#include <cassert>
#include <ostream>
#include <climits>

using namespace std;

#include "CursorCntl.h"

template <typename NodeData>
class BST
{
private:
    // Tree node class definition
    struct Node
    {
        // Constructors
        Node() : left(0), right(0) {}
        Node(const NodeData &d) : data(d), left(0), right(0) { }

        // Data Members
        NodeData    data;    // The "contents" of the node
        Node        *left;   // Link to the left successor node
        Node        *right;  // Link to the right successor node
    };

public:
   // Constructor
   BST() : root(0), current(0) { }

   // True if the tree is empty
   bool Empty() const { return root == 0;}

   // Search for an entry in the tree. If the entry is found,
   // make it the "current" entry. If not, make the current entry
   // NULL. Return true if the entry is found; otherwise return false.
   bool Search(NodeData &d);

   // Add a new node to the tree.
   void Insert(NodeData &d);

   // Delete the current node.
   void Delete();
   NodeData RemoveLeaf() { return RRemoveLeaf(root); }
   // Output the tree to the "os" in the indicated sequence.
   void OutputInOrder(ostream &os) const;    // Output inorder
   void OutputPreOrder(ostream &os) const;   // Output preorder
   void OutputPostOrder(ostream &os) const;  // Output postorder
   void OutputByLevel(ostream &os) const;          // Output by level

   // Retrieve the data part of the current node.
   NodeData Current() { return current->data; }

   // Show the binary tree on the screen.
   void ShowTree() const;
```

```
private:
   Node *root;      // Points to the root node
   Node *current;   // Points to the current node
   Node *parent;    // Points to current node's parent

   // Recursive Search
   bool RSearch(Node *subTree, NodeData &d);

   NodeData RRemoveLeaf(Node *&r);
   // Recursive Insert
   void RInsert(Node *&subTree, NodeData &d);

   // Recursive Traversal Functions
   void ROutputInOrder(Node *subTree, ostream &os) const;
   void ROutputPreOrder(Node *subTree, ostream &os) const;
   void ROutputPostOrder(Node *subTree, ostream &os) const;
   // Find the parent of leftmost right successor of the current node.
   Node *ParentOfLeftMostRightSucc(Node *node, Node *parent) const;

   // Show the binary tree on the screen.
   void RShowTree(Node *subTree, int x, int y) const;
};

template <typename NodeData>
NodeData BST<NodeData>::RRemoveLeaf(Node *&r)
{
    // Make sure that the tree is not empty.
    assert(r != 0);
    // Does this node have any successors?
    if (r->left != 0)
        // There is a left successor, traverse left subtree
        return RRemoveLeaf(r->left);
    else if (r->right != 0)
        // There is a right successor, traverse right subtree
        return RRemoveLeaf(r->right);
    else
    {
        // There are no successors; it's a leaf node, capture node data
        NodeData result = r->data;
        // Delete the leaf node
        delete r;
        // Mark the subtree empty
        r = 0;
        // Return the data from the removed node.
        return result;
    }
}

// Public insert function to call RInsert
template <typename NodeData>
void BST<NodeData>::Insert(NodeData &d)
{
    RInsert(root, d);
}

// Public OutputInOrder function to call ROutputInOrder
template <typename NodeData>
void BST<NodeData>::OutputInOrder(ostream &os) const
{
    ROutputInOrder(root, os);
}
```

```
// Public OutputPreOrder function to call ROutputPreOrder
template <typename NodeData>
void BST<NodeData>::OutputPreOrder(ostream &os) const
{
    ROutputPreOrder(root, os);
}

// Public OutputPostOrder function to call ROutputPostOrder
template <typename NodeData>
void BST<NodeData>::OutputPostOrder(ostream &os) const
{
    ROutputPostOrder(root, os);
}

// Public search function call resursive search
template <typename NodeData>
bool BST<NodeData>::Search(NodeData &d)
{
    parent = 0;
    return RSearch(root, d);
}

template <typename NodeData>
void BST<NodeData>::ROutputPostOrder(Node *subTree, ostream &os) const
{
    if (subTree != NULL)
    {
        ROutputPostOrder(subTree->left, os);
        ROutputPostOrder(subTree->right, os);
        subTree->data.Show(os);
        cout << endl;
    }
}

const unsigned XRoot = 40;        // Column number for root node

template <typename NodeData>
void BST<NodeData>::RShowTree(Node *subTree, int x, int y) const
{
  const unsigned VertSpacing = 7;   // Vertical spacing constant
  const unsigned HorizSpacing = 10; // Horizontal spacing of tree nodes
  const unsigned MaxLevels = 4;     // The number of levels that fit on the screen

  // If the tree is not empty display it.
  if (subTree != 0 && x < MaxLevels)
    {
    // Show the left sub-tree.
    RShowTree(subTree->left, x+1, y+VertSpacing/(1<<x));

    // Show the root.
    gotoxy(XRoot+HorizSpacing*x, y);
    subTree->data.Show(cout);
    cout << endl;

    // Show the right subtree.
    RShowTree(subTree->right, x+1, y-VertSpacing/(1<<x));
    }
}
```

```
// Delete a node in the tree by name
template <typename NodeData>
void BST<NodeData>::Delete()
{
    // Temp node so that we can free memory of removed node
    Node *temp = current;
    // 2 successors
    if (current->left != NULL && current->right != NULL)
    {
        // set parent to left-most right succ
        parent = ParentOfLeftMostRightSucc(current->right, current);
        // if current is parent, there are no left nodes
        if (current == parent)
        {
            temp = parent->right;
            current->right = temp->right;
        }
        // Grab node to update current's data and null out its parents
        // left pointer
        else
        {
            temp = parent->left;
            parent->left = NULL;
        }
        current->data = temp->data;
    }
    // 1 or 0 successors
    else
    {
        Node *succ = NULL;
        // left successor
        if (current->left != NULL)
            succ = current->left;
        // right succesor
        else if (current->right != NULL)
            succ = current->right;

        // if temp is first node, update first node
        if (temp == root)
            root = succ;
        // if current is left node of parrent, make
        // the parent of left the successor
        else if (current == parent->left)
            parent->left = succ;
        else
            parent->right = succ;
    }
    delete temp; // free memory
}
```

```
// Find the parent of the left most right succ for delete cases with 2 succ
template <typename NodeData>
typename BST<NodeData>::Node* BST<NodeData>::ParentOfLeftMostRightSucc(Node *node, Node *parent)
const
{
    // run until proper parent found
    for (;;)
    {
        // if left node, make current node the parent
        // and make current node it's left succ
        if (node->left != NULL)
        {
            parent = node;
            node = node->left;
        }
        // no more succ, return parent
        else
            break;
    }
    return parent;
}

template <typename NodeData>
void BST<NodeData>::ShowTree() const
{
  const unsigned YRoot = 11;      // Line number of root node
  const unsigned ScrollsAt = 24;  // Screen scrolls after line 24

  int xOld;                       // Old cursor x coordinate
  int yOld;                       // Old cursor y coordinate

  // Save cursor position
  getxy(xOld, yOld);

  // Has the screen scrolled yet?
  int deltaY = 0;

  if (yOld > ScrollsAt)
    deltaY = yOld - ScrollsAt+1;

  // Clear the right half of the screen.
  for (int y=0; y<ScrollsAt+1; y++)
    {
    gotoxy(XRoot,y+deltaY);
    clreol();
    }

  // Show the tree and offset if scrolled.
  RShowTree(root, 0, YRoot+deltaY);

  // Restore old cursor position.
  gotoxy(xOld,yOld);
}
```

```
325
326 // Recursive search, returning true if data is in tree else false
327 template <typename NodeData>
328 bool BST<NodeData>::RSearch(Node *subTree, NodeData &d)
329 {
330     //If node is empty, return false
331     if (subTree == NULL)
332         return false;
333     // if node data = data searching for, return true
334     else if (subTree->data == d)
335     {
336         current = subTree;
337         return true;
338     }
339     //If d is less than nodeData, look left
340     else if (d < subTree->data)
341     {
342         parent = subTree;
343         return RSearch(subTree->left, d);
344     }
345     //If d is greater than nodeData, look right
346     else
347     {
348         parent = subTree;
349         return RSearch(subTree->right, d);
350     }
351 }
352
353 template <typename NodeData>
354 void BST<NodeData>::RInsert(Node *&subTree, NodeData &d)
355 {
356     // If node is empty, inserts here
357     if (subTree == NULL)
358     {
359         Node *newNode = new(nothrow) Node(d);
360         assert(newNode != NULL);
361         subTree = newNode;
362     }
363     // If d is less than data, go left and repeat
364     else if (d < subTree->data)
365         RInsert(subTree->left, d);
366     // If d is more than data, go right and repeat
367     else if (d > subTree->data)
368         RInsert(subTree->right, d);
369     // Node data found in tree, update node's count
370     else if (d == subTree->data)
371         subTree->data.Update();
372 }
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
```

```
390
391 template <typename NodeData>
392 void BST<NodeData>::ROutputInOrder(Node *subTree, ostream &os) const
393 {
394     // If node is not empty
395     if (subTree != NULL)
396     {
397         // Recursively go all the way down to left most node
398         ROutputInOrder(subTree->left, os);
399         // Output value
400         subTree->data.Show(os);
401         cout << endl;
402         // Now go down a right node to recursively grab next left most node
403         ROutputInOrder(subTree->right, os);
404     }
405 }
406
407 template <typename NodeData>
408 void BST<NodeData>::ROutputPreOrder(Node *subTree, ostream &os) const
409 {
410     if (subTree != NULL)
411     {
412         // Output data
413         subTree->data.Show(os);
414         cout << endl;
415         // Go left
416         ROutputPreOrder(subTree->left, os);
417         // Go right
418         ROutputPreOrder(subTree->right, os);
419     }
420 }
421
422 #endif
```