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

using namespace std;

#include "CursorCntl.h"
#include "Queue_T.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();

   // 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);

   // 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;
};

// 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);
}
```

```cpp
// 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)
    {
        // if current is parent, there are no left node succ
        if (current == ParentOfLeftMostRightSucc(current->right, current))
        {
            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
}

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;
    }
}
```

```
// 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
{
    Node *temp = node;

    // run until proper parent found
    for (;;)
    {
        // if left node, make current node the parent
        // and make current node it's left succ
        if (temp->left != NULL)
        {
            parent = temp;
            temp = temp->left;
        }
        // no more succ, return parent
        else
            break;
    }
    return parent;
}

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));
    }
}
```

```
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);
}

// Output tree nodes level by level
template <typename NodeData>
void BST<NodeData>::OutputByLevel(ostream &os) const
{
    // Queue a queue of nodes
    Queue<Node *> queue;
    // Add root to start of queue
    queue.Enqueue(root);
    while (!queue.Empty())
    {
        // Add left and right successors of head node
        if (queue.Head()->left != 0)
            queue.Enqueue(queue.Head()->left);
        if(queue.Head()->right != 0)
            queue.Enqueue(queue.Head()->right);
        // Show data for head node
        queue.Head()->data.Show(os);
        cout << endl;
        // Remove head node
        queue.Dequeue();
    }
}
```

```cpp
// Recursive search, returning true if data is in tree else false
template <typename NodeData>
bool BST<NodeData>::RSearch(Node *subTree, NodeData &d)
{
    //If node is empty, return false
    if (subTree == NULL)
        return false;
    // if node data = data searching for, return true
    else if (subTree->data == d)
    {
        current = subTree;
        return true;
    }
    //If d is less than nodeData, look left
    else if (d < subTree->data)
    {
        parent = subTree;
        return RSearch(subTree->left, d);
    }
    //If d is greater than nodeData, look right
    else
    {
        parent = subTree;
        return RSearch(subTree->right, d);
    }
}

template <typename NodeData>
void BST<NodeData>::RInsert(Node *&subTree, NodeData &d)
{
    // If node is empty, inserts here
    if (subTree == NULL)
    {
        Node *newNode = new(nothrow) Node(d);
        assert(newNode != NULL);
        subTree = newNode;
    }
    // If d is less than data, go left and repeat
    else if (d < subTree->data)
        RInsert(subTree->left, d);
    // If d is more than data, go right and repeat
    else if (d > subTree->data)
        RInsert(subTree->right, d);
    // Node data found in tree, update node's count
    else if (d == subTree->data)
        subTree->data.Update();
}
```

```
template <typename NodeData>
void BST<NodeData>::ROutputInOrder(Node *subTree, ostream &os) const
{
    // If node is not empty
    if (subTree != NULL)
    {
        // Recursively go all the way down to left most node
        ROutputInOrder(subTree->left, os);
        // Output value
        subTree->data.Show(os);
        cout << endl;
        // Now go down a right node to recursively grab next left most node
        ROutputInOrder(subTree->right, os);
    }
}

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

#endif
```