```
// TREVER WAGENHALS
// PROGRAM 8

#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;
```

```
66
67
68
69  private:
70     Node *root;      // Points to the root node
71     Node *current;   // Points to the current node
72     Node *parent;    // Points to current node's parent
73
74     // Recursive Search
75     bool RSearch(Node *subTree, NodeData &d);
76
77     // Recursive Insert
78     void RInsert(Node *&subTree, NodeData &d);
79
80     // Recursive Traversal Functions
81     void ROutputInOrder(Node *subTree, ostream &os) const;
82     void ROutputPreOrder(Node *subTree, ostream &os) const;
83     void ROutputPostOrder(Node *subTree, ostream &os) const;
84     // Find the parent of leftmost right successor of the current node.
85     Node *ParentOfLeftMostRightSucc(Node *node, Node *parent) const;
86
87     // Show the binary tree on the screen.
88     void RShowTree(Node *subTree, int x, int y) const;
89  };
90
91  // Public insert function to call RInsert
92  template <typename NodeData>
93  void BST<NodeData>::Insert(NodeData &d)
94  {
95      RInsert(root, d);
96  }
97
98  // Public OutputInOrder function to call ROutputInOrder
99  template <typename NodeData>
100 void BST<NodeData>::OutputInOrder(ostream &os) const
101 {
102     ROutputInOrder(root, os);
103 }
104
105 // Public OutputPreOrder function to call ROutputPreOrder
106 template <typename NodeData>
107 void BST<NodeData>::OutputPreOrder(ostream &os) const
108 {
109     ROutputPreOrder(root, os);
110 }
111
112 // Public OutputPostOrder function to call ROutputPostOrder
113 template <typename NodeData>
114 void BST<NodeData>::OutputPostOrder(ostream &os) const
115 {
116     ROutputPostOrder(root, os);
117 }
118
119 // Public search function call resursive search
120 template <typename NodeData>
121 bool BST<NodeData>::Search(NodeData &d)
122 {
123     parent = 0;
124     return RSearch(root, d);
125 }
126
127
128
129
130
```

```
131
132
133
134 // Delete a node in the tree by name
135 template <typename NodeData>
136 void BST<NodeData>::Delete()
137 {
138     // Temp node so that we can free memory of removed node
139     Node *temp = current;
140     // 2 successors
141     if (current->left != NULL && current->right != NULL)
142     {
143         // set parent to left-most right succ
144         parent = ParentOfLeftMostRightSucc(current->right, current);
145         // if current is parent, there are no left nodes
146         if (current == parent)
147         {
148             temp = parent->right;
149             current->right = temp->right;
150         }
151         // Grab node to update current's data and null out its parents
152         // left pointer
153         else
154         {
155             temp = parent->left;
156             parent->left = NULL;
157         }
158         current->data = temp->data;
159     }
160     // 1 or 0 successors
161     else
162     {
163         Node *succ = NULL;
164         // left successor
165         if (current->left != NULL)
166             succ = current->left;
167         // right succesor
168         else if (current->right != NULL)
169             succ = current->right;
170
171         // if temp is first node, update first node
172         if (temp == root)
173             root = succ;
174         // if current is left node of parrent, make
175         // the parent of left the successor
176         else if (current == parent->left)
177             parent->left = succ;
178         else
179             parent->right = succ;
180     }
181     delete temp; // free memory
182 }
183
184 template <typename NodeData>
185 void BST<NodeData>::ROutputPostOrder(Node *subTree, ostream &os) const
186 {
187     if (subTree != NULL)
188     {
189         ROutputPostOrder(subTree->left, os);
190         ROutputPostOrder(subTree->right, os);
191         subTree->data.Show(os);
192         cout << endl;
193     }
194 }
195
```

```
196
197
198
199 // Find the parent of the left most right succ for delete cases with 2 succ
200 template <typename NodeData>
201 typename BST<NodeData>::Node* BST<NodeData>::ParentOfLeftMostRightSucc(Node *node, Node *parent)
    const
202 {
203     // run until proper parent found
204     for (;;)
205     {
206         // if left node, make current node the parent
207         // and make current node it's left succ
208         if (node->left != NULL)
209         {
210             parent = node;
211             node = node->left;
212         }
213         // no more succ, return parent
214         else
215             break;
216     }
217     return parent;
218 }
219
220 const unsigned XRoot = 40;        // Column number for root node
221
222 template <typename NodeData>
223 void BST<NodeData>::RShowTree(Node *subTree, int x, int y) const
224 {
225   const unsigned VertSpacing = 7;   // Vertical spacing constant
226   const unsigned HorizSpacing = 10; // Horizontal spacing of tree nodes
227   const unsigned MaxLevels = 4;     // The number of levels that fit on the screen
228
229   // If the tree is not empty display it.
230   if (subTree != 0 && x < MaxLevels)
231     {
232     // Show the left sub-tree.
233     RShowTree(subTree->left, x+1, y+VertSpacing/(1<<x));
234
235     // Show the root.
236     gotoxy(XRoot+HorizSpacing*x, y);
237     subTree->data.Show(cout);
238     cout << endl;
239
240     // Show the right subtree.
241     RShowTree(subTree->right, x+1, y-VertSpacing/(1<<x));
242     }
243 }
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
```

```
260
261 template <typename NodeData>
262 void BST<NodeData>::ShowTree() const
263 {
264   const unsigned YRoot = 11;      // Line number of root node
265   const unsigned ScrollsAt = 24;  // Screen scrolls after line 24
266
267   int xOld;                       // Old cursor x coordinate
268   int yOld;                       // Old cursor y coordinate
269
270   // Save cursor position
271   getxy(xOld, yOld);
272
273   // Has the screen scrolled yet?
274   int deltaY = 0;
275
276   if (yOld > ScrollsAt)
277     deltaY = yOld - ScrollsAt+1;
278
279   // Clear the right half of the screen.
280   for (int y=0; y<ScrollsAt+1; y++)
281     {
282     gotoxy(XRoot,y+deltaY);
283     clreol();
284     }
285
286   // Show the tree and offset if scrolled.
287   RShowTree(root, 0, YRoot+deltaY);
288
289   // Restore old cursor position.
290   gotoxy(xOld,yOld);
291 }
292
293 // Output tree nodes level by level
294 template <typename NodeData>
295 void BST<NodeData>::OutputByLevel(ostream &os) const
296 {
297     // Queue a queue of nodes
298     Queue<Node *> queue;
299     // Add root to start of queue
300     queue.Enqueue(root);
301     while (!queue.Empty())
302     {
303         // Add left and right successors of head node
304         if (queue.Head()->left != 0)
305             queue.Enqueue(queue.Head()->left);
306         if(queue.Head()->right != 0)
307             queue.Enqueue(queue.Head()->right);
308         // Show data for head node
309         queue.Head()->data.Show(os);
310         cout << endl;
311         // Remove head node
312         queue.Dequeue();
313     }
314 }
315
316
317
318
319
320
321
322
323
324
```

```
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
```