

```

1  #ifndef BST_T_H
2  #define BST_T_H
3  #include <cassert>
4  #include <ostream>
5  #include <climits>
6
7  using namespace std;
8
9  #include "CursorCntl.h"
10 #include "Queue_T.h"
11
12 template <typename NodeData>
13 class BST
14 {
15 private:
16     // Tree node class definition
17     struct Node
18     {
19         // Constructors
20         Node() : left(0), right(0) {}
21         Node(const NodeData &d) : data(d), left(0), right(0) { }
22
23         // Data Members
24         NodeData data;    // The "contents" of the node
25         Node *left;      // Link to the left successor node
26         Node *right;     // Link to the right successor node
27     };
28
29 public:
30     // Constructor
31     BST() : root(0), current(0) { }
32
33     // True if the tree is empty
34     bool Empty() const { return root == 0;}
35
36     // Search for an entry in the tree. If the entry is found,
37     // make it the "current" entry. If not, make the current entry
38     // NULL. Return true if the entry is found; otherwise return false.
39     bool Search(NodeData &d);
40
41     // Add a new node to the tree.
42     void Insert(NodeData &d);
43
44     // Delete the current node.
45     void Delete();
46
47     // Output the tree to the "os" in the indicated sequence.
48     void OutputInOrder(ostream &os) const;    // Output inorder
49     void OutputPreOrder(ostream &os) const;   // Output preorder
50     void OutputPostOrder(ostream &os) const;  // Output postorder
51     void OutputByLevel(ostream &os) const;    // Output by level
52
53     // Retrieve the data part of the current node.
54     NodeData Current() { return current->data; }
55
56     // Show the binary tree on the screen.
57     void ShowTree() const;
58
59
60
61
62
63
64
65

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```