

```
1  /*
2   * Class name: BSTree.h
3   * Class description: The apparatus for the collection of nodes that make up the binary search tree.
4   *
5   * Programmer: Chad Philip Johnson
6   * Date Created: Friday, April 26th, 2013
7   * Last Date Modified: Thursday, May 09th, 2013
8   *
9   * Sources Used:
10  *     BSTNode.h
11  *         - For the individual nodes contained within the binary search tree.
12  */
13
14 #include <iostream>
15
16 #include "BSTNode.h"
17
18 #ifndef BSTREE_H
19 #define BSTREE_H
20
21 template<typename T>
22 class BSTree
23 {
24     public:
25     /**
26      * Default constructor for the BSTree class. Sets the count to zero and the root node pointer to NULL.
27      */
28     BSTree()
29     : uintCount( 0 ), pobjBSTNodeRootNode( NULL )
30     {
31         /* empty */
32     }
33
34     /**
35      * Destructor for the BSTree class. Frees the memory associated with the root node pointer.
36      */
37     ~BSTree()
38     {
39         this->clear( this->pobjBSTNodeRootNode );
40     }
41
42     /**
43      * Return the current number of nodes in the tree.
44      * @return The unsigned integer value representing the number of nodes in the tree.
45      */
46     unsigned int getCount() const
47     {
48         return this->uintCount;
49     }
50 }
```

```
51
52     /**
53      * Frees the memory associated with the root node pointer.
54      */
55     void clear()
56     {
57         this->clear( this->pobjBSTNodeRootNode );
58     }
59
60     /**
61      * Calls the private member function insert.
62      * @param tmpNewData The data to be inserted into the tree.
63      */
64     void insert( T tmpNewData )
65     {
66         this->insert( tmpNewData, this->pobjBSTNodeRootNode );
67     }
68
69     /**
70      * Calls the private member function find.
71      * @param tmpTargetData The data to be found within the tree.
72      * @return True if a node containing the value already exists in the tree; false if not.
73      */
74     bool find( T tmpTargetData )
75     {
76         return this->find( tmpTargetData, this->pobjBSTNodeRootNode );
77     }
78
79     /**
80      * Calls the private member function remove.
81      * @param tmpTargetData The data to be removed from the tree.
82      * @return True if a node containing the value has been removed from the tree; false if not.
83      */
84     bool remove( T tmpTargetData )
85     {
86         return this->remove( tmpTargetData, this->pobjBSTNodeRootNode );
87     }
88
89     /**
90      * Calls the private member function get.
91      * @param tmpTargetData The data to be found within the tree (and the associated node)
92      * @return A pointer to the node containing the specified value.
93      */
94     T* get( T tmpTargetData )
95     {
96         return this->get( tmpTargetData, this->pobjBSTNodeRootNode );
97     }
98
99     /**
100      * Calls the private member function inOrder.
101      */
```

```
101 void inOrder()
102 {
103     this->inOrder( this->pobjBSTNodeRootNode );
104 }
105
106 /**
107 * Calls the private member function reverseOrder.
108 */
109 void reverseOrder()
110 {
111     this->reverseOrder( pobjBSTNodeRootNode );
112 }
113
114
115 private:
116 /**
117 * Free all memory within the passed tree.
118 * @param pobjBSTNodeClearNode Root node of the tree to be deleted and have its memory freed.
119 */
120 void clear( BSTNode<T>*& pobjBSTNodeClearNode )
121 {
122     if( this->uintCount > 0 )
123     {
124         if( pobjBSTNodeClearNode->getLeftChild() != NULL )
125         {
126             this->clear( pobjBSTNodeClearNode->getLeftChild() );
127         }
128
129         if( pobjBSTNodeClearNode->getRightChild() != NULL )
130         {
131             this->clear( pobjBSTNodeClearNode->getRightChild() );
132         }
133
134         delete pobjBSTNodeClearNode;
135         pobjBSTNodeClearNode = NULL;
136         this->uintCount--;
137     }
138 }
139
140 /**
141 * Insert a value into the tree.
142 * @param tmpNewData The value to be inserted into the tree.
143 * @param pobjBSTNodeInsertNode The root node of the tree where the value is to be inserted.
144 */
145 void insert( T tmpNewData, BSTNode<T>*& pobjBSTNodeInsertNode )
146 {
147     if( uintCount > 0 )
148     {
149         if( tmpNewData == pobjBSTNodeInsertNode->getData() )
150         {
```

```

151         pobjBSTNodeInsertNode->getData().incrementCount();
152     }
153     else
154     {
155         if( tmpNewData < pobjBSTNodeInsertNode->getData() )
156         {
157             if( pobjBSTNodeInsertNode->getLeftChild() == NULL )
158             {
159                 pobjBSTNodeInsertNode->setLeftChild( new BSTNode<T>( tmpNewData ) );
160                 this->uintCount++;
161             }
162             else
163             {
164                 this->insert( tmpNewData, pobjBSTNodeInsertNode->getLeftChild() );
165             }
166         }
167     }
168
169     if( tmpNewData > pobjBSTNodeInsertNode->getData() )
170     {
171         if( pobjBSTNodeInsertNode->getRightChild() == NULL )
172         {
173             pobjBSTNodeInsertNode->setRightChild( new BSTNode<T>( tmpNewData ) );
174             this->uintCount++;
175         }
176         else
177         {
178             this->insert( tmpNewData, pobjBSTNodeInsertNode->getRightChild() );
179         }
180     }
181 }
182
183 else
184 {
185     pobjBSTNodeInsertNode = new BSTNode<T>( tmpNewData );
186     this->uintCount++;
187 }
188
189 */
190
191 /**
192 * Find a value within the tree.
193 * @param tmpTargetData The data value to be found in the tree.
194 * @param pobjBSTNodeSelectedNode The root node of the tree to be searched.
195 * @return True if the value exists in the tree; false if not.
196 */
197 bool find( T tmpTargetData, BSTNode<T>* pobjBSTNodeSelectedNode )
198 {
199     if( pobjBSTNodeSelectedNode == NULL )
200     {
201         return false;

```

```

201 }
202 else
203 {
204     if( tmpTargetData == pobjBSTNodeSelectedNode->getData() )
205     {
206         return true;
207     }
208     else
209     {
210         if( tmpTargetData < pobjBSTNodeSelectedNode->getData() )
211         {
212             return this->find( tmpTargetData, pobjBSTNodeSelectedNode->getLeftChild() );
213         }
214         else
215         {
216             return this->find( tmpTargetData, pobjBSTNodeSelectedNode->getRightChild() );
217         }
218     }
219 }
220 }
221 /**
222 * Remove a value from the tree, if it exists, and free the memory for the associated node.
223 * @param tmpTargetData The data to be removed from the tree.
224 * @param pobjBSTNodeSelectedNode The root node of the tree from which the value will be removed.
225 * @return True if the data was removed from the tree; false if not.
226 */
227 bool remove( T tmpTargetData, BSTNode<T>*& pobjBSTNodeSelectedNode )
228 {
229     if( pobjBSTNodeSelectedNode == NULL )
230     {
231         return false;
232     }
233     else
234     {
235         if( tmpTargetData == pobjBSTNodeSelectedNode->getData() )
236         {
237             if( pobjBSTNodeSelectedNode->getLeftChild() == NULL )
238             {
239                 BSTNode<T>* pobjBSTNodeMarkDelete = pobjBSTNodeSelectedNode;
240                 // command automatically works backwards to assign the new value of leftchild to the parent's
241                 // pobjBSTNodeLeftChild pointer variable
242                 pobjBSTNodeSelectedNode      = pobjBSTNodeSelectedNode->getRightChild();
243                 delete pobjBSTNodeMarkDelete;
244                 pobjBSTNodeMarkDelete        = NULL;
245                 this->uintCount--;
246             }
247             else
248             {
249                 this->removeMax( pobjBSTNodeSelectedNode->getData(), pobjBSTNodeSelectedNode->getLeftChild() );

```

```

250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
        }

        return true;
    }
    else
    {
        if( tmpTargetData < pobjBSTNodeSelectedNode->getData() )
        {
            return this->remove( tmpTargetData, pobjBSTNodeSelectedNode->getLeftChild() );
        }
        else
        {
            return this->remove( tmpTargetData, pobjBSTNodeSelectedNode->getRightChild() );
        }
    }
}

/**
 * Helper function for the remove function: finds the largest value within a tree or subtree and swaps the
 * value for this node with the value from the original calling node. The discovered node is then
 * deleted so that its memory is freed.
 * @param tmpReplaceData The data value from the calling node that will be replaced by the value of the
 * discovered maximum node value.
 * @param pobjBSTNodeSelectedNode The root node of the tree or subtree to be searched.
 */
void removeMax( T& tmpReplaceData, BSTNode<T*>*& pobjBSTNodeSelectedNode )
{
    if( pobjBSTNodeSelectedNode->getRightChild() == NULL )
    {
        tmpReplaceData = pobjBSTNodeSelectedNode->getData();

        BSTNode<T>* pobjBSTNodeMarkDelete = pobjBSTNodeSelectedNode;
        pobjBSTNodeSelectedNode = pobjBSTNodeSelectedNode->getRightChild();
        delete pobjBSTNodeMarkDelete;
        pobjBSTNodeMarkDelete = NULL;
        this->uintCount--;
    }
    else
    {
        removeMax( tmpReplaceData, pobjBSTNodeSelectedNode->getRightChild() );
    }
}

/**
 * Find a data value and return the associated node.
 * @param tmpTargetData The data value to be found.
 * @param pobjBSTNodeSelectedNode The root node of the tree or subtree to be searched.
 * @return Pointer to the node contained the searched value.
 */

```

```

300 T* get( T tmpTargetData, BSTNode<T>* pobjBSTNodeSelectedNode )
301 {
302     if( pobjBSTNodeSelectedNode != NULL )
303     {
304         if( tmpTargetData == pobjBSTNodeSelectedNode->getData() )
305         {
306             return &( pobjBSTNodeSelectedNode->getData() );
307         }
308         else
309         {
310             if( tmpTargetData < pobjBSTNodeSelectedNode->getData() )
311             {
312                 return this->get( tmpTargetData, pobjBSTNodeSelectedNode->getLeftChild() );
313             }
314             else
315             {
316                 return this->get( tmpTargetData, pobjBSTNodeSelectedNode->getRightChild() );
317             }
318         }
319     }
320
321     return NULL;
322 }
323
324 /**
325 * Print a list of all values in the tree (ascending order).
326 * @param pobjBSTNodePrintNode The root node of the tree or subtree to have values printed.
327 */
328 void inOrder( BSTNode<T>* pobjBSTNodePrintNode )
329 {
330     if( this->uintCount > 0 )
331     {
332         if( pobjBSTNodePrintNode->getLeftChild() != NULL )
333         {
334             this->inOrder( pobjBSTNodePrintNode->getLeftChild() );
335         }
336         cout << pobjBSTNodePrintNode->getData().getValue() << " " << pobjBSTNodePrintNode->getData().getCount() << endl;
337
338         if( pobjBSTNodePrintNode->getRightChild() != NULL )
339         {
340             this->inOrder( pobjBSTNodePrintNode->getRightChild() );
341         }
342     }
343     else
344     {
345         cout << "";
346     }
347 }
348
349 /**

```

```
350 * Print a list of all values in the tree (descending order).
351 * @param pobjBSTNodePrintNode The root node of the tree or subtree to have values printed.
352 */
353 void reverseOrder( BSTNode<T>* pobjBSTNodePrintNode )
354 {
355     if( this->uintCount > 0 )
356     {
357         if( pobjBSTNodePrintNode->getRightChild() != NULL )
358         {
359             this->reverseOrder( pobjBSTNodePrintNode->getRightChild() );
360         }
361
362         cout << pobjBSTNodePrintNode->getData().getValue() << " " << pobjBSTNodePrintNode->getData().getCount() << endl;
363
364         if( pobjBSTNodePrintNode->getLeftChild() != NULL )
365         {
366             this->reverseOrder( pobjBSTNodePrintNode->getLeftChild() );
367         }
368     }
369     else
370     {
371         cout << "";
372     }
373 }
374
375 unsigned int uintCount;
376 BSTNode<T>* pobjBSTNodeRootNode;
377
378 };
379
380 #endif
381
```