```cpp
/*
 * Class name:  BSTree.h
 * Class description:  The apparatus for the collection of nodes that make up the binary search tree.
 *
 * Programmer:  Chad Philip Johnson
 * Date Created:  Friday, April 26th, 2013
 * Last Date Modified:  Thursday, May 09th, 2013
 *
 * Sources Used:
 *      BSTNode.h
 *          - For the individual nodes contained within the binary search tree.
 */

#include <iostream>

#include "BSTNode.h"

#ifndef BSTREE_H
#define BSTREE_H

template<typename T>
class BSTree
{
    public:
        /**
         * Default constructor for the BSTree class.  Sets the count to zero and the root node pointer to NULL.
         */
        BSTree()
        : uintCount( 0 ), pobjBSTNodeRootNode( NULL )
        {
            /* empty */
        }

        /**
         * Destructor for the BSTree class.  Frees the memory associated with the root node pointer.
         */
        ~BSTree()
        {
            this->clear( this->pobjBSTNodeRootNode );
        }

        /**
         * Return the current number of nodes in the tree.
         * @return The unsigned integer value representing the number of nodes in the tree.
         */
        unsigned int getCount() const
        {
            return this->uintCount;
        }

```

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

```
        void inOrder()
        {
            this->inOrder( this->pobjBSTNodeRootNode );
        }

        /**
         * Calls the private member function reverseOrder.
         */
        void reverseOrder()
        {
            this->reverseOrder( pobjBSTNodeRootNode );
        }


    private:
        /**
         * Free all memory within the passed tree.
         * @param pobjBSTNodeClearNode Root node of the tree to be deleted and have its memory freed.
         */
        void clear( BSTNode<T>*& pobjBSTNodeClearNode )
        {
            if( this->uintCount > 0 )
            {
                if( pobjBSTNodeClearNode->getLeftChild() != NULL )
                {
                    this->clear( pobjBSTNodeClearNode->getLeftChild() );
                }

                if( pobjBSTNodeClearNode->getRightChild() != NULL )
                {
                    this->clear( pobjBSTNodeClearNode->getRightChild() );
                }

                delete pobjBSTNodeClearNode;
                pobjBSTNodeClearNode    = NULL;
                this->uintCount--;
            }
        }

        /**
         * Insert a value into the tree.
         * @param tmpNewData The value to be inserted into the tree.
         * @param pobjBSTNodeInsertNode The root node of the tree where the value is to be inserted.
         */
        void insert( T tmpNewData, BSTNode<T>*& pobjBSTNodeInsertNode )
        {
            if( uintCount > 0 )
            {
                if( tmpNewData == pobjBSTNodeInsertNode->getData() )
                {
```

```
                    pobjBSTNodeInsertNode->getData().incrementCount();

                }
                else
                {
                    if( tmpNewData < pobjBSTNodeInsertNode->getData() )
                    {
                        if( pobjBSTNodeInsertNode->getLeftChild() == NULL )
                        {
                            pobjBSTNodeInsertNode->setLeftChild( new BSTNode<T>( tmpNewData ) );
                            this->uintCount++;
                        }
                        else
                        {
                            this->insert( tmpNewData, pobjBSTNodeInsertNode->getLeftChild() );
                        }
                    }

                    if( tmpNewData > pobjBSTNodeInsertNode->getData() )
                    {
                        if( pobjBSTNodeInsertNode->getRightChild() == NULL )
                        {
                            pobjBSTNodeInsertNode->setRightChild( new BSTNode<T>( tmpNewData ) );
                            this->uintCount++;
                        }
                        else
                        {
                            this->insert( tmpNewData, pobjBSTNodeInsertNode->getRightChild() );
                        }
                    }
                }
            }
            else
            {
                pobjBSTNodeInsertNode   = new BSTNode<T>( tmpNewData );
                this->uintCount++;
            }
        }

        /**
         * Find a value within the tree.
         * @param tmpTargetData The data value to be found in the tree.
         * @param pobjBSTNodeSelectedNode The root node of the tree to be searched.
         * @return True if the value exists in the tree; false if not.
         */
        bool find( T tmpTargetData, BSTNode<T>* pobjBSTNodeSelectedNode )
        {
            if( pobjBSTNodeSelectedNode == NULL )
            {
                return false;
```

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

    /**
     * Remove a value from the tree, if it exists, and free the memory for the associated node.
     * @param tmpTargetData The data to be removed from the tree.
     * @param pobjBSTNodeSelectedNode The root node of the tree from which the value will be removed.
     * @return True if the data was removed from the tree; false if not.
     */
    bool remove( T tmpTargetData, BSTNode<T>*& pobjBSTNodeSelectedNode )
    {
        if( pobjBSTNodeSelectedNode == NULL )
        {
            return false;
        }
        else
        {
            if( tmpTargetData == pobjBSTNodeSelectedNode->getData() )
            {
                if( pobjBSTNodeSelectedNode->getLeftChild() == NULL )
                {
                    BSTNode<T>* pobjBSTNodeMarkDelete = pobjBSTNodeSelectedNode;
                    // command automatically works backwards to assign the new value of leftchild to the parent's pobjBSTNodeLeftChild pointer variable
                    pobjBSTNodeSelectedNode     = pobjBSTNodeSelectedNode->getRightChild();
                    delete pobjBSTNodeMarkDelete;
                    pobjBSTNodeMarkDelete               = NULL;
                    this->uintCount--;
                }
                else
                {
                    this->removeMax( pobjBSTNodeSelectedNode->getData(), pobjBSTNodeSelectedNode->getLeftChild() );
```

```
                }

                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.
     */
```

```
        T* get( T tmpTargetData, BSTNode<T>* pobjBSTNodeSelectedNode )
        {
            if( pobjBSTNodeSelectedNode != NULL )
            {
                if( tmpTargetData == pobjBSTNodeSelectedNode->getData() )
                {
                    return &( pobjBSTNodeSelectedNode->getData() );
                }
                else
                {
                    if( tmpTargetData < pobjBSTNodeSelectedNode->getData() )
                    {
                        return this->get( tmpTargetData, pobjBSTNodeSelectedNode->getLeftChild() );
                    }
                    else
                    {
                        return this->get( tmpTargetData, pobjBSTNodeSelectedNode->getRightChild() );
                    }
                }
            }

            return NULL;
        }

        /**
         * Print a list of all values in the tree (ascending order).
         * @param pobjBSTNodePrintNode The root node of the tree or subtree to have values printed.
         */
        void inOrder( BSTNode<T>* pobjBSTNodePrintNode )
        {
            if( this->uintCount > 0 )
            {
                if( pobjBSTNodePrintNode->getLeftChild() != NULL )
                {
                    this->inOrder( pobjBSTNodePrintNode->getLeftChild() );
                }
                cout << pobjBSTNodePrintNode->getData().getValue() << " " << pobjBSTNodePrintNode->getData().getCount() << endl;

                if( pobjBSTNodePrintNode->getRightChild() != NULL )
                {
                    this->inOrder( pobjBSTNodePrintNode->getRightChild() );
                }
            }
            else
            {
                cout << "";
            }
        }

        /**
```

```
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
```