```
/*Author: Eric Gustin
  Assignment: CPSC223-01 HW09 bst_collection.h
  Description: Binary Search Tree implementation of the
  collection class. Contains all collection methods including remove.
*/
#ifndef BST_COLLECTION_H
#define BST_COLLECTION_H

#include <vector>
#include "collection.h"


template <typename K, typename V>
class BSTCollection : public Collection<K,V>
{
public:

 // create an empty linked list
 BSTCollection();

 // copy a linked list
 BSTCollection(const BSTCollection<K,V>& rhs);

 // assign a linked list
 BSTCollection<K,V>& operator=(const BSTCollection<K,V>& rhs);

 // delete a linked list
 ~BSTCollection();

 // insert a key - value pair into the collection
 void insert(const K& key, const V& val);

 // remove a key - value pair from the collection
 void remove(const K& key);

 // find the value associated with the key
 bool find(const K& key, V& val) const;

 // find the keys associated with the range
 void find(const K& k1, const K& k2, std::vector<K>& keys) const;

 // return all keys in the collection
 void keys(std::vector<K>& keys) const;

 // return collection keys in sorted order
 void sort(std::vector<K>& keys) const;

 // return the number of keys in collection
 int size() const;

 // return the height of the tree
 int height() const;

private:

 // binary search tree node structure
 struct Node {
  K key;
  V value;
  Node* left;
  Node* right;
 };

 // root node of the search tree
 Node* root;

 // number of k-v pairs in the collection
 int collection_size;

 // helper to recursively empty search tree
 void make_empty(Node* subtree_root);

 // helper to recursively build sorted list of keys
 void inorder(const Node* subtree, std::vector<K>& keys) const;

 // helper to recursively build sorted list of keys
 void preorder(const Node* subtree, std::vector<K>& keys) const;

 // helper to recursively find range of keys
 void range_search(const Node* subtree, const K& k1, const K& k2,
 std::vector<K>& keys) const;
```

```
 // return the height of the tree rooted at subtree_root
 int height(const Node* subtree_root) const;

 // helper to recursively remove key in BST
 Node* remove_helper(Node* subtree_root, const K& key);
};


template <typename K, typename V>
BSTCollection<K,V>::BSTCollection() :
collection_size(0), root(nullptr)
{
}


template <typename K, typename V>
void BSTCollection<K,V>::make_empty(Node* subtree_root)
{
 // this is a recursive helper function
 // base case
 if (subtree_root == nullptr)
  return;
 // postorder delete
 make_empty(subtree_root->left);
 make_empty(subtree_root->right);
  delete subtree_root;
  subtree_root = nullptr;
  --collection_size;
}


template <typename K, typename V>
BSTCollection<K,V>::~BSTCollection()
{
 make_empty(root);
}


template <typename K, typename V>
BSTCollection<K,V>::BSTCollection(const BSTCollection<K,V>& rhs)
: collection_size(0), root(nullptr)
{
 *this = rhs;
}


template <typename K, typename V>
BSTCollection <K,V>& BSTCollection<K,V>::operator=(const BSTCollection<K,V>& rhs)
{
 if (this == &rhs)
  return *this;

 // delete current
 make_empty(root);
 root = nullptr;
 // build tree
 std::vector<K> ks;
 preorder(rhs.root, ks);

 V val;
 // find value that corresponds to current key, and insert into tree
 for (int i = 0; i < ks.size(); ++i) {
  find(ks[i], val);
  insert(ks[i], val);
 }

 return *this;
}

template <typename K, typename V>
void BSTCollection<K,V>::insert(const K& key, const V& val)
{
 // make a new leaf node
 Node* curr = new Node;
 curr->key = key;
 curr->value = val;
 curr->left = nullptr;
 curr->right = nullptr;

 // case: tree is empty
 if (root == nullptr)
  root = curr;
```

```
 else {
  Node* temp = root;
  while (temp != nullptr) {
   // case: the key that I'm inserting is less than the internal node "temp"
   if (curr->key < temp->key) {
    // leaf node found. Insert curr node into the tree as a new leaf.
    if (temp->left == nullptr) {
     temp->left = curr;
     temp = nullptr;
    }
    // continue to traverse tree since "temp" is not a leaf
    else
     temp = temp->left;
   }
   // case: the key that I'm inserting is greater than the internal node "temp"
   else {
    // leaf node found. Insert curr node into the tree as a new leaf node
    if (temp->right == nullptr) {
     temp->right = curr;
     temp = nullptr;
    }
    // countinue to traverse tree since "temp" is not a leaf
    else
     temp = temp->right;
   }
  }
 }
 ++collection_size;
}


template <typename K, typename V>
void BSTCollection<K,V>::remove(const K& key)
{
 // remove desired node
 if (collection_size > 0) {
  if (root->key == key)
   root = remove_helper(root, key);
  else
   remove_helper(root, key);
 }
}

template <typename K, typename V>
typename BSTCollection<K,V>::Node*
BSTCollection<K,V>::remove_helper(Node* subtree_root, const K& key)
{
 // go left
 if (subtree_root && key < subtree_root->key)
  subtree_root->left = remove_helper(subtree_root->left, key);

 // go right
 else if (subtree_root && key > subtree_root->key)
  subtree_root->right = remove_helper(subtree_root->right, key);

 else if (subtree_root && key == subtree_root->key) {
  //case1: leaf.
  if (subtree_root->left == nullptr && subtree_root->right == nullptr) {
   delete subtree_root;
   subtree_root = nullptr;
   --collection_size;
   return nullptr;
  }
  // case2: one child
  else if (subtree_root->left == nullptr ^ subtree_root->right == nullptr) {
   Node* replace = nullptr;
   // ternanry conditional
   (subtree_root->left != nullptr) ? (replace = subtree_root->left) : (replace = subtree_root->right);
   delete subtree_root;
   subtree_root = nullptr;
   --collection_size;
   return replace;

  }
  //case3: two children.
  else if (subtree_root->left != nullptr && subtree_root->right != nullptr) {
   Node* successor = subtree_root->right;
   // right child doesn't have a left child, so inorder successor = right child
   if (successor->left == nullptr) {
    // copy contents of sucessor into subtree_root
    subtree_root->key = successor->key;
```

```
    subtree_root->value = successor->value;
    subtree_root->right = successor->right;
    delete successor;
    successor = nullptr;
    --collection_size;
   }
   else {
    // iteratively find inorder successor
    while (successor->left != nullptr)
     successor = successor->left;
    K k = successor->key;
    V v = successor->value;
    // remove leaf node
    remove_helper(root, successor->key);
    // copy contents or inorder successor into subtree_root
    subtree_root->key = k;
    subtree_root->value = v;
   }
  }
 }
 return subtree_root;
}

template <typename K, typename V>
bool BSTCollection<K,V>::find(const K& key, V& val) const
{
 Node* temp = root;
 while (temp != nullptr) {
  // key found in tree
  if (temp->key == key) {
   val = temp->value;
   return true;
  }
  // temp node is greater than key. Traverse left
  if (temp->key > key)
   temp = temp->left;
  // temp node is less than key. Traverse right
  else
   temp = temp->right;
 }
 return false;
}


template <typename K, typename V>
void BSTCollection<K,V>::
range_search(const Node* subtree, const K& k1, const K& k2, std::vector<K>& ks) const
{
 // this is a recursive helper function
 // base statement
 if (subtree == nullptr)
  return;

 // modified inorder traversal. Only traverses nodes between k1 and k2 (inclusive)
 // traverse left until k1 >= the subtree's key
 if (k1 < subtree->key)
  range_search(subtree->left, k1, k2, ks);
 // insert qualifying nodes into vector
 if (k1 <= subtree->key && k2 >= subtree->key)
  ks.push_back(subtree->key);
 // traverse right until k2 <= subtree's key
 if (k2 > subtree->key)
  range_search(subtree->right, k1, k2, ks);
}


template <typename K, typename V>
void BSTCollection<K,V>::find(const K& k1, const K& k2, std::vector<K>& ks) const
{
 // defer to the range search ( recursive ) helper function
 range_search(root, k1, k2, ks);
}


template <typename K, typename V>
void BSTCollection<K,V>::inorder(const Node* subtree, std::vector<K>& ks) const
{
 // this is a recursive helper function
 // base statement
 if (subtree == nullptr)
  return;
 // left, "print", right
```

```
 inorder(subtree->left, ks);
 ks.push_back(subtree->key);
 inorder(subtree->right, ks);
}


template <typename K, typename V>
void BSTCollection<K,V>::preorder(const Node* subtree, std::vector<K>& ks) const
{
 // this is a recursive helper function
 // base statement
 if (subtree == nullptr)
  return;
 // "print" --> left --> right
 ks.push_back(subtree->key);
 preorder(subtree->left, ks);
 preorder(subtree->right, ks);
}

template <typename K, typename V>
void BSTCollection<K,V>::keys(std::vector<K>& ks) const
{
 // defer to the inorder ( recursive ) helper function
 inorder(root, ks);
}


template <typename K, typename V>
void BSTCollection<K,V>::sort(std::vector<K>& ks) const
{
 // defer to the inorder ( recursive ) helper function
 inorder(root, ks);
}


template <typename K, typename V>
int BSTCollection<K,V>::size() const
{
 return collection_size;
}


template <typename K, typename V>
int BSTCollection<K,V>::height(const Node* subtree_root) const
{
 // this is a recursive helper function
 // base case
 if (subtree_root == nullptr)
  return 0;
 //if (subtree_root->left == nullptr && subtree_root->right == nullptr)
 // return 1;

 return 1 + std::max(height(subtree_root->left), height(subtree_root->right));
}


template <typename K, typename V>
int BSTCollection<K,V>::height() const
{
 // defer to the height ( recursive ) helper function
 return height(root);
}

#endif
```