```cpp
#ifndef BST_H
#define BST_H

#include <iostream>
#include <exception>
#include <cstdlib>
#include <utility>

/**
 * A templated class for a Node in a search tree.
 * The getters for parent/left/right are virtual so
 * that they can be overridden for future kinds of
 * search trees, such as Red Black trees, Splay trees,
 * and AVL trees.
 */
template <typename Key, typename Value>
class Node
{
public:
    Node(const Key &key, const Value &value, Node<Key, Value> *parent);
    virtual ~Node();

    const std::pair<const Key, Value> &getItem() const;
    std::pair<const Key, Value> &getItem();
    const Key &getKey() const;
    const Value &getValue() const;
    Value &getValue();

    virtual Node<Key, Value> *getParent() const;
    virtual Node<Key, Value> *getLeft() const;
    virtual Node<Key, Value> *getRight() const;

    void setParent(Node<Key, Value> *parent);
    void setLeft(Node<Key, Value> *left);
    void setRight(Node<Key, Value> *right);
    void setValue(const Value &value);

protected:
    std::pair<const Key, Value> item_;
    Node<Key, Value> *parent_;
    Node<Key, Value> *left_;
    Node<Key, Value> *right_;
};

/*
  -----------------------------------------
  Begin implementations for the Node class.
  -----------------------------------------
*/

/**
 * Explicit constructor for a node.
 */
template <typename Key, typename Value>
Node<Key, Value>::Node(const Key &key, const Value &value, Node<Key, Value> *parent) : item_(key, value),
                                                                                     parent_(parent),
                                                                                     left_(NULL),
                                                                                     right_(NULL)
{
}

/**
 * Destructor, which does not need to do anything since the pointers inside of a node
 * are only used as references to existing nodes. The nodes pointed to by parent/left/right
 * are freed by the BinarySearchTree.
 */
template <typename Key, typename Value>
Node<Key, Value>::~Node()
{
}

/**
 * A const getter for the item.
 */
template <typename Key, typename Value>
const std::pair<const Key, Value> &Node<Key, Value>::getItem() const
{
    return item_;
}

/**
 * A non-const getter for the item.
```

```
 */
template <typename Key, typename Value>
std::pair<const Key, Value> &Node<Key, Value>::getItem()
{
    return item_;
}

/**
 * A const getter for the key.
 */
template <typename Key, typename Value>
const Key &Node<Key, Value>::getKey() const
{
    return item_.first;
}

/**
 * A const getter for the value.
 */
template <typename Key, typename Value>
const Value &Node<Key, Value>::getValue() const
{
    return item_.second;
}

/**
 * A non-const getter for the value.
 */
template <typename Key, typename Value>
Value &Node<Key, Value>::getValue()
{
    return item_.second;
}

/**
 * An implementation of the virtual function for retreiving the parent.
 */
template <typename Key, typename Value>
Node<Key, Value> *Node<Key, Value>::getParent() const
{
    return parent_;
}

/**
 * An implementation of the virtual function for retreiving the left child.
 */
template <typename Key, typename Value>
Node<Key, Value> *Node<Key, Value>::getLeft() const
{
    return left_;
}

/**
 * An implementation of the virtual function for retreiving the right child.
 */
template <typename Key, typename Value>
Node<Key, Value> *Node<Key, Value>::getRight() const
{
    return right_;
}

/**
 * A setter for setting the parent of a node.
 */
template <typename Key, typename Value>
void Node<Key, Value>::setParent(Node<Key, Value> *parent)
{
    parent_ = parent;
}

/**
 * A setter for setting the left child of a node.
 */
template <typename Key, typename Value>
void Node<Key, Value>::setLeft(Node<Key, Value> *left)
{
    left_ = left;
}

/**
 * A setter for setting the right child of a node.
 */
```

```cpp
template <typename Key, typename Value>
void Node<Key, Value>::setRight(Node<Key, Value> *right)
{
    right_ = right;
}

/**
 * A setter for the value of a node.
 */
template <typename Key, typename Value>
void Node<Key, Value>::setValue(const Value &value)
{
    item_.second = value;
}

/*
  ---------------------------------------
  End implementations for the Node class.
  ---------------------------------------
*/

/**
 * A templated unbalanced binary search tree.
 */
template <typename Key, typename Value>
class BinarySearchTree
{
public:
    BinarySearchTree();                                                    // TODO
    virtual ~BinarySearchTree();                                           // TODO
    virtual void insert(const std::pair<const Key, Value> &keyValuePair); // TODO
    virtual void remove(const Key &key);                                   // TODO
    void clear();                                                          // TODO
    bool isBalanced() const;                                               // TODO
    void print() const;
    bool empty() const;

    template <typename PPKey, typename PPValue>
    friend void prettyPrintBST(BinarySearchTree<PPKey, PPValue> &tree);

public:
    /**
     * An internal iterator class for traversing the contents of the BST.
     */
    class iterator // TODO
    {
    public:
        iterator();

        std::pair<const Key, Value> &operator*() const;
        std::pair<const Key, Value> *operator->() const;

        bool operator==(const iterator &rhs) const;
        bool operator!=(const iterator &rhs) const;

        iterator &operator++();

    protected:
        friend class BinarySearchTree<Key, Value>;
        iterator(Node<Key, Value> *ptr);
        Node<Key, Value> *current_;
    };

public:
    iterator begin() const;
    iterator end() const;
    iterator find(const Key &key) const;
    Value &operator[](const Key &key);
    Value const &operator[](const Key &key) const;

protected:
    // Mandatory helper functions
    Node<Key, Value> *internalFind(const Key &k) const;              // TODO
    Node<Key, Value> *getSmallestNode() const;                       // TODO
    static Node<Key, Value> *predecessor(Node<Key, Value> *current); // TODO
    // Note:  static means these functions don't have a "this" pointer
    //        and instead just use the input argument.
    static Node<Key, Value> *successor(Node<Key, Value> *current);
    // Provided helper functions
    virtual void printRoot(Node<Key, Value> *r) const;
    virtual void nodeSwap(Node<Key, Value> *n1, Node<Key, Value> *n2);
```

```cpp
    // Add helper functions here
    int calculateHeight(Node<Key, Value> *r) const;
    void deleteNode(Node<Key, Value> *c);

protected:
    Node<Key, Value> *root_;
    // You should not need other data members
};

/*
-------------------------------------------------------------
Begin implementations for the BinarySearchTree::iterator class.
--------------------------------------------------------------
*/

/**
tializes an iterator with a given node pointer.
 */
template <class Key, class Value>
BinarySearchTree<Key, Value>::iterator::iterator(Node<Key, Value> *ptr)
{
    current_ = ptr;
}

/**
 * A default constructor that initializes the iterator to NULL.
 */
template <class Key, class Value>
BinarySearchTree<Key, Value>::iterator::iterator()
{
    current_ = NULL;
}

/**
 * Provides access to the item.
 */
template <class Key, class Value>
std::pair<const Key, Value> &
BinarySearchTree<Key, Value>::iterator::operator*() const
{
    return current_->getItem();
}

/**
 * Provides access to the address of the item.
 */
template <class Key, class Value>
std::pair<const Key, Value> *
BinarySearchTree<Key, Value>::iterator::operator->() const
{
    return &(current_->getItem());
}

/**
 * Checks if 'this' iterator's internals have the same value
 * as 'rhs'
 */
template<class Key, class Value>
bool
BinarySearchTree<Key, Value>::iterator::operator==(
    const BinarySearchTree<Key, Value>::iterator& rhs) const
{
    // TODO
    return(this->current_ == rhs.current_);
}


/**
 * Checks if 'this' iterator's internals have a different value
 * as 'rhs'
 */
template <class Key, class Value>
bool BinarySearchTree<Key, Value>::iterator::operator!=(
    const BinarySearchTree<Key, Value>::iterator &rhs) const
{
    return (this->current_ != rhs.current_);
}

/**
 * Advances the iterator's location using an in-order sequencing
 */
template <class Key, class Value>
```

```
typename BinarySearchTree<Key, Value>::iterator &
BinarySearchTree<Key, Value>::iterator::operator++()
{
    current_ = successor(current_);
    return *this;
}

/*
-------------------------------------------------------------
End implementations for the BinarySearchTree::iterator class.
-------------------------------------------------------------
*/

/*
-----------------------------------------------------
Begin implementations for the BinarySearchTree class.
-----------------------------------------------------
*/

/**
 * Default constructor for a BinarySearchTree, which sets the root to NULL.
 */
template <class Key, class Value>
BinarySearchTree<Key, Value>::BinarySearchTree()
{
    root_ = NULL;
}

template <typename Key, typename Value>
BinarySearchTree<Key, Value>::~BinarySearchTree()
{
    clear();
}

/**
 * Returns true if tree is empty
 */
template <class Key, class Value>
bool BinarySearchTree<Key, Value>::empty() const
{
    return root_ == NULL;
}

template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::print() const
{
    printRoot(root_);
    std::cout << "\n";
}

/**
 * Returns an iterator to the "smallest" item in the tree
 */
template <class Key, class Value>
typename BinarySearchTree<Key, Value>::iterator
BinarySearchTree<Key, Value>::begin() const
{
    BinarySearchTree<Key, Value>::iterator begin(getSmallestNode());
    return begin;
}

/**
 * Returns an iterator whose value means INVALID
 */
template <class Key, class Value>
typename BinarySearchTree<Key, Value>::iterator
BinarySearchTree<Key, Value>::end() const
{
    BinarySearchTree<Key, Value>::iterator end(NULL);
    return end;
}

/**
 * Returns an iterator to the item with the given key, k
 * or the end iterator if k does not exist in the tree
 */
template <class Key, class Value>
typename BinarySearchTree<Key, Value>::iterator
BinarySearchTree<Key, Value>::find(const Key &k) const
{
    Node<Key, Value> *curr = internalFind(k);
    BinarySearchTree<Key, Value>::iterator it(curr);
```

```
    return it;
}

/**
 * @precondition The key exists in the map
 * Returns the value associated with the key
 */
template <class Key, class Value>
Value &BinarySearchTree<Key, Value>::operator[](const Key &key)
{
    Node<Key, Value> *curr = internalFind(key);
    if (curr == NULL)
        throw std::out_of_range("Invalid key");
    return curr->getValue();
}

template <class Key, class Value>
Value const &BinarySearchTree<Key, Value>::operator[](const Key &key) const
{
    Node<Key, Value> *curr = internalFind(key);
    if (curr == NULL)
        throw std::out_of_range("Invalid key");
    return curr->getValue();
}

/**
 * An insert method to insert into a Binary Search Tree.
 * The tree will not remain balanced when inserting.
 * Recall: If key is already in the tree, you should
 * overwrite the current value with the updated value.
 */
template <class Key, class Value>
void BinarySearchTree<Key, Value>::insert(const std::pair<const Key, Value> &keyValuePair)
{
    // checking if insert is null
    if (root_ == nullptr)
    {
        // if it is, then we add new node
        root_ = new Node<Key, Value>(keyValuePair.first, keyValuePair.second, NULL);
        return;
    }
    else
    {
        // creating temp node
        Node<Key, Value> *c = root_;

        // iterating through while true
        while (true)
        {
            // checking if c key is greater
            if (keyValuePair.first < c->getKey())
            {
                // if c getLeft is null, then we add a new node
                if (c->getLeft() == NULL)
                {
                    Node<Key, Value> *ins2 = new Node<Key, Value>(keyValuePair.first, keyValuePair.second, c);
                    c->setLeft(ins2);
                    break;
                }
                // if not then we iterate to next left
                else
                {
                    c = c->getLeft();
                }
            }
            // checking if c key is less
            else if (keyValuePair.first > c->getKey())
            {
                // if c getRight is null, then we add a new node
                if (c->getRight() == NULL)
                {
                    Node<Key, Value> *ins = new Node<Key, Value>(keyValuePair.first, keyValuePair.second, c);
                    c->setRight(ins);
                    break;
                }
                // if not then we iterate to next left
                else
                {
                    c = c->getRight();
                }
            }
            else // both of them are the same
```

```
            {
                c->setValue(keyValuePair.second);
                break;
            }
        }
    }
}
/**
 * A remove method to remove a specific key from a Binary Search Tree.
 * Recall: The writeup specifies that if a node has 2 children you
 * should swap with the predecessor and then remove.
 */
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::remove(const Key &key)
{
    // root check
    if (root_ == NULL) {
        return;
    }

    // finding the key
    Node<Key, Value> *c = this->internalFind(key);

    // if c is null then we return
    if (c == NULL) {
        return;
    }

    // case for when there are 2 children
    // having this first bc after swap we will either be in a 0-child or 1-child case
    if(c->getLeft() != NULL && c->getRight() != NULL) {
        nodeSwap(c,predecessor(c));
    }

    // case for when there is one child and 0 children
    // creating temp node to store it
    Node <Key, Value> * one;
    // 0 children
    if (c->getRight() == NULL && c->getLeft() == NULL) {
        one = NULL;
    }

    // one child
    if (c->getRight()==NULL && c->getLeft()!=NULL) {
        one = c->getLeft();
    } else if(c->getRight()!=NULL && c->getLeft()==NULL){
        one = c->getRight();
    }

    // updating the parent node to complete the promote functionality
    Node <Key, Value> * two = c -> getParent();
    if (two == NULL) {
        root_ = one;
    } else {
        if (c->getParent()->getLeft() == c) {
            two->setLeft(one);
        } else if (c->getParent()->getRight() == c) {
            two->setRight(one);
        }
    }

    // check to set the parent of one to two
    if (one != NULL) {
        one->setParent(two);
    }
    // deleting c
    delete c;
}

// predecessor
template <class Key, class Value>
Node<Key, Value> *
BinarySearchTree<Key, Value>::predecessor(Node<Key, Value> *current)
{
    // checking if left exists
    if (current->getLeft() != NULL)
    {
        // if it does then we go all the way to the right
        Node<Key, Value> *cur = current->getLeft();
        while (cur->getRight() != NULL)
        {
            cur = cur->getRight();
```

```
        }
        return cur;
    }

    // if not, then we walk up the chain and find the first node
    // who is a right child of his parent
    Node<Key, Value> *cur = current;
    while (cur->getParent() != nullptr){
      if (cur->getParent()->getRight() == cur){
        return cur->getParent();
      }
      else{
        cur = cur->getParent();
      }
    }
    return nullptr;

}


// successor
template <class Key, class Value>
Node<Key, Value> *
BinarySearchTree<Key, Value>::successor(Node<Key, Value> *current)
{
    // check if right child exists
    if (current->getRight() != NULL)
    {
        // if it does, we go all the way to the left
        Node<Key, Value> *cur = current->getRight();
        while (cur->getLeft() != NULL)
        {
            cur = cur->getLeft();
        }
        return cur;
    }

    // if not, then we walk up the chain and find the first node
    // who is a left child of his parent
  Node<Key, Value> *cur = current;
    while (cur->getParent() != nullptr){
      if (cur->getParent()->getLeft() == cur){
        return cur->getParent();
      }
      else{
        cur = cur->getParent();
      }
    }
    return nullptr;
}

/**
 * A method to remove all contents of the tree and
 * reset the values in the tree for use again.
 */
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::clear()
{
    deleteNode(root_);
}

// helper function for delete
template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::deleteNode(Node<Key, Value> *c)
{
    // deleting everything by using post order recursion
    if (c == NULL)
    {
        return;
    }
    deleteNode(c->getLeft());
    deleteNode(c->getRight());
    c = NULL;
    delete c;
}

/**
 * A helper function to find the smallest node in the tree.
 */
template <typename Key, typename Value>
Node<Key, Value> *
BinarySearchTree<Key, Value>::getSmallestNode() const
```

```
{
    // iterating all the way to the left
    Node<Key, Value> *c = root_;
    while (c ->getLeft() != NULL)
    {
   c = c->getLeft();
    }
    return c;
}

/**
 * Helper function to find a node with given key, k and
 * return a pointer to it or NULL if no item with that key
 * exists
 */
template <typename Key, typename Value>
Node<Key, Value> *BinarySearchTree<Key, Value>::internalFind(const Key &key) const
{
    // if root is null or if the key is equal
    if (root_ == NULL)
    {
        return root_;
    }
    else if (root_->getKey() == key) {
        return root_;
    } else {
        // iterating through the tree to find the node
        Node<Key, Value> *c = root_;
        while (c != NULL) {
        if (c->getKey() == key)
        {
            return c;
        }
        if (key < c->getKey())
        {
            c = c->getLeft();
        }
        else if (key > c->getKey())
        {
            c = c->getRight();
        }
    }
    return NULL;
}


}

template <typename Key, typename Value>
int BinarySearchTree<Key, Value>::calculateHeight(Node<Key, Value> *parameter) const
{

    // checking if parameter is NULL
    if (parameter == NULL)
    {
        return 0 ;
    }

    // calculating height
    int l = calculateHeight(parameter->getLeft());
    int r = calculateHeight(parameter->getRight());

    int one = 1;
    // returning valid outputs for height values
    if (l == (-1))
    {
        return -1;
    }
    if (r == (-1))
    {
        return (-1);
    }
    if (std::abs(l - r) > (1))
    {
        return (-1);
    }

    return (1) + std::max(l, r);
}
/**
 * Return true if the BST is balanced.
 */
```

```
template <typename Key, typename Value>
bool BinarySearchTree<Key, Value>::isBalanced() const
{

    // root null check
    if (root_ == NULL)
    {
        return true;
    }

    // checking if no children
    if(root_->getRight() == NULL && root_->getLeft() == NULL) {
        return true;
    }

    // checking for one child
    if ((root_->getRight() == NULL && root_->getLeft() != NULL) || (root_->getRight() != NULL && root_->getLeft() == NULL)) {
        return false;
    }

    // recursive call
    int l = calculateHeight(root_->getLeft());
    int r = calculateHeight(root_->getRight());

    return (l != -1 && r != -1);
}

template <typename Key, typename Value>
void BinarySearchTree<Key, Value>::nodeSwap(Node<Key, Value> *n1, Node<Key, Value> *n2)
{
    if ((n1 == n2) || (n1 == NULL) || (n2 == NULL))
    {
        return;
    }
    Node<Key, Value> *n1p = n1->getParent();
    Node<Key, Value> *n1r = n1->getRight();
    Node<Key, Value> *n1lt = n1->getLeft();
    bool n1isLeft = false;
    if (n1p != NULL && (n1 == n1p->getLeft()))
        n1isLeft = true;
    Node<Key, Value> *n2p = n2->getParent();
    Node<Key, Value> *n2r = n2->getRight();
    Node<Key, Value> *n2lt = n2->getLeft();
    bool n2isLeft = false;
    if (n2p != NULL && (n2 == n2p->getLeft()))
        n2isLeft = true;

    Node<Key, Value> *temp;
    temp = n1->getParent();
    n1->setParent(n2->getParent());
    n2->setParent(temp);

    temp = n1->getLeft();
    n1->setLeft(n2->getLeft());
    n2->setLeft(temp);

    temp = n1->getRight();
    n1->setRight(n2->getRight());
    n2->setRight(temp);

    if ((n1r != NULL && n1r == n2))
    {
        n2->setRight(n1);
        n1->setParent(n2);
    }
    else if (n2r != NULL && n2r == n1)
    {
        n1->setRight(n2);
        n2->setParent(n1);
    }
    else if (n1lt != NULL && n1lt == n2)
    {
        n2->setLeft(n1);
        n1->setParent(n2);
    }
    else if (n2lt != NULL && n2lt == n1)
    {
        n1->setLeft(n2);
        n2->setParent(n1);
    }

    if (n1p != NULL && n1p != n2)
```

```
    {
        if (n1isLeft)
            n1p->setLeft(n2);
        else
            n1p->setRight(n2);
    }
    if (n1r != NULL && n1r != n2)
    {
        n1r->setParent(n2);
    }
    if (n1lt != NULL && n1lt != n2)
    {
        n1lt->setParent(n2);
    }

    if (n2p != NULL && n2p != n1)
    {
        if (n2isLeft)
            n2p->setLeft(n1);
        else
            n2p->setRight(n1);
    }
    if (n2r != NULL && n2r != n1)
    {
        n2r->setParent(n1);
    }
    if (n2lt != NULL && n2lt != n1)
    {
        n2lt->setParent(n1);
    }

    if (this->root_ == n1)
    {
        this->root_ = n2;
    }
    else if (this->root_ == n2)
    {
        this->root_ = n1;
    }
}

/*
---------------------------------------------------
End implementations for the BinarySearchTree class.
---------------------------------------------------
*/

#endif
```