```
/****************************************************************************
\
 * BST.h
 *
 *  Created on: March 13th 2025
 *      Author: Michael Jiang
 *
 * Implementation details: see comments below
\***************************************************************************/

#ifndef BST_H_
#define BST_H_

#include <string>
using namespace std;

/** Binary Search Tree node */
struct BSTNode {
    /** Pointer to the parent node. */
    BSTNode* parent;
    /** Pointer to the left child node. */
    BSTNode* left;
    /** Reference to the right child node. */
    BSTNode* right;
    /** Key used to search an element. */
    int key;
    /** Data associated to the element. */
    string data;
};


/** Binary Search Tree
 * In a Binary search tree:
 *     Every node y in the left subtree of x  must have y.key < x.key
 *      Every node y in the right subtree of x  must have y.key > x.key
 *      The root node has root.parent = null
 *      Here no keys can be duplicated
 */
class BST {
    private:
       /** Pointer to the root node of the tree, or nullptr if empty */
       BSTNode* m_root;

       /** Minimum: Finds and returns the node with the smallest key */
       BSTNode* _min(BSTNode* x);
       /** Successor: Returns the node following node x in key order in the
whole tree */
       BSTNode* _successor(BSTNode* x);

       /** Computes and returns the height of the node x. */
       int _height(BSTNode* x);

       /** Finds the node with the given key and return the node. */
        BSTNode* searchNode(int key);

       /** Helper function for recursive preOrder traversal. */
        void preOrderHelper(BSTNode* current);

       /** Helper function for recursive inOrder traversal. */
        void inOrderHelper(BSTNode* current);

       /** Helper function for recursive postOrder traversal. */
        void postOrderHelper(BSTNode* current);

       /** Helper function to delete the tree recursively. */
       void postOrderRemove(BSTNode* current);


    public:
       /** Construct an empty binary search tree. */
       BST();
       /** Destroy and free up the memory allocated by the binary search
tree. */
       virtual ~BST();
       /** Prints all the data from the tree in key order (using _min and
_successor). */
       void printTree();

       /** Finds a node with (key) from the tree and returns the associated
(data) or "" if not found. */
       string search(int key);
       /** Inserts a new node in the tree with given (key, data). */
       void insert(int key, string data);
       /** Deletes a the node with (key) from the tree and returns the
associated (data) or "" if not found. */
       string remove(int key);

       /** Computes and returns the height of the tree. */
       int height();

       /** Prints the keys (comma separated) of the tree using an preorder
```

```
traversal. */
       void preOrder();
       /** Prints the keys (comma separated) of the tree using an inorder
traversal. */
       void inOrder();
       /** Prints the keys (comma separated) of the tree using an postorder
traversal. */
       void postOrder();

};

#endif /* BST_H_ */


/***************************************************************************
\
 * BST.cpp
 *
 *  Created on: March 13th 2025
 *      Author: Michael Jiang
 *
 * Implementation details: See comments below
\**************************************************************************/

#include <iostream>
#include "BST.h"
using namespace std;

/** Minimum: Finds and returns the node with the smallest key */
BSTNode* BST::_min(BSTNode* x) {
    while (x->left != nullptr) {
        x = x->left;
    }
    return x;
}

/**Successor: Returns the node following node x in key order in the whole
tree */
BSTNode* BST::_successor(BSTNode* x) {
    if (x->right == nullptr) {
        int value = x->key;
        BSTNode* walker = x->parent;
        while (walker != nullptr && walker->key < value) {
            walker = walker->parent;
        }
        return walker;
    }
    return _min(x->right);

}

/** Computes and returns the height of the node x. */
int BST::_height(BSTNode* x) {
    if (x == nullptr) {
        return -1;
    }
    return 1 + max(_height(x->left), _height(x->right));
}

/** Construct an empty binary search tree. */
BST::BST() {
    m_root = nullptr;
}

/** Destroy and free up the memory allocated by the binary search tree. */
BST::~BST() {
    postOrderRemove(m_root);
}

/** Delete each node through post order traversal, this ensures that we don't
delete the parent before the child. */
void BST::postOrderRemove(BSTNode* current) {
    if (current == nullptr) {
        return;
    }
    postOrderRemove(current->left);
    postOrderRemove(current->right);
    delete current;
}

/** Prints all the data from the tree in key order (using _min and
_successor).
 *  The output should look like [(3,Hello), (8,RMC), (21,Student)] \n
 */

void BST::printTree() {
    if (m_root == nullptr) {
        cout << "[]" << endl;
        return;
    }
    BSTNode* current = _min(m_root);
    cout << "[";
    while (_successor(current) != nullptr) {
        cout << "(" << current->key << "," << current->data << ")" << ", ";
        current = _successor(current);
    }
    cout << "(" << current->key << "," << current->data << ")" << "]" <<
endl;
}


/** Finds a node with (key) from the tree and returns the associated (data)
or "" if not found. */
string BST::search(int key) {
    BSTNode* walker = m_root;
    while (walker != nullptr) {
        if (key == walker->key) {
            return walker->data;
        } if (key < walker->key) {
            walker = walker->left;
        } else {
            walker = walker->right;
        }
    }
    return "";
}

/** Finds a node with (key) from the tree and returns the node or NULL if not
found. */
BSTNode* BST::searchNode(int key) {
    BSTNode* walker = m_root;
    while (walker != nullptr) {
        if (key == walker->key) {
            return walker;
        } if (key < walker->key) {
            walker = walker->left;
        } else if (key > walker->key) {
            walker = walker->right;
        }
    }
    return walker;
}

/** Inserts a new node in the tree with given (key, data). */
void BST::insert(int key, string data) {
    BSTNode* x = new BSTNode();
    x->key = key;
    x->data = data;
    x->left = nullptr;
    x->right = nullptr;
    if (m_root == nullptr) {
        m_root = x;
        m_root->parent = nullptr;
        return;
    }
    BSTNode* walker = m_root;
    BSTNode* parent = walker->parent;
    while (walker != nullptr) {
        parent = walker;
        if (key < walker->key) {
            walker = walker->left;
        } else {
            walker = walker->right;
        }
    }
    if (key < parent->key) {
        parent->left = x;
    }
    else {
        parent->right = x;
    }
    x->parent = parent;
}

/** Deletes a the node with (key) from the tree and returns the associated
(data) or empty string "" if not found. */

string BST::remove(int key) {
    BSTNode* curr = searchNode(key);

    if (curr == nullptr) {
        return "";
    }
    string value = curr->data;
    BSTNode* parent = curr->parent;

    // Case 1: two children
    if (curr->left && curr->right) {
        BSTNode* successor = _successor(curr);
        BSTNode* successorParent = successor->parent;
        curr->key = successor->key;
        curr->data = successor->data;
        curr = successor;
        parent = successorParent;
    }

    // Case 2: One child or no children
    BSTNode* child;
    if (curr->left) {
        child = curr->left;
    } else {
        child = curr->right;
    }

    if (parent == nullptr) {
        m_root = child;
        if (child != nullptr) {
            m_root->parent = nullptr;
        }
    } else if (parent->left == curr) {
        parent->left = child;
    } else {
        parent->right = child;
    }

    delete curr;
    return value;
}

/** Computes and returns the height of the tree. */
int BST::height() {
    return _height(m_root);
}


/** Prints the keys (comma separated) of the tree using an preorder
traversal. */
void BST::preOrder() {
    preOrderHelper(m_root);
    cout << endl;
}

/** Helper function for recursive preOrder traversal. */
void BST::preOrderHelper(BSTNode *current) {
    if (current == nullptr) {
        return;
    }
    cout << current->key << ", ";
    preOrderHelper(current->left);
    preOrderHelper(current->right);
```

```
}

/** Prints the keys (comma separated) of the tree using an inorder traversal.
*/
void BST::inOrder() {
    inOrderHelper(m_root);
    cout << endl;
}


/** Helper function for recursive inOrder traversal. */
void BST::inOrderHelper(BSTNode *current) {
    if (current == nullptr) {
        return;
    }
    inOrderHelper(current->left);
    cout << current->key << ", ";
    inOrderHelper(current->right);
}

/** Prints the keys (comma separated) of the tree using an postorder
traversal. */
void BST::postOrder() {
    postOrderHelper(m_root);
    cout << endl;
}

/** Helper function for recursive postOrder traversal. */
void BST::postOrderHelper(BSTNode *current) {
    if (current == nullptr) {
        return;
    }
    postOrderHelper(current->left);
    postOrderHelper(current->right);
    cout << current->key << ", ";
}


/****************************************************************************
\
* main.cpp
*
*  Created on: March 13th 2025
*      Author: Michael Jiang
*
*
```

```
*
\**************************************************************************/

#include <iostream>
#include "BST.h"
using namespace std;

int main() {
    cout << "CSE250 - Lab4 " << endl;


    cout << "---------------- Question 1 ----------------" << endl << endl;
    BST* t = new BST();
    cout << "t->preOrder() outputs "; t->preOrder();
    cout << "t->postOrder() outputs "; t->postOrder();
    cout << "t->inOrder() outputs "; t->inOrder();
    cout << "t->search(4) outputs " << t->search(4) << endl;
    cout << "t->printTree() outputs "; t->printTree();
    cout << "t->height() outputs "<< t->height() << endl;
    cout << "t->remove(4) outputs " << t->remove(4) << endl << endl;
    delete t;

    cout << "---------------- Question 2 ----------------" << endl << endl;
    BST* t2 = new BST();
    t2->remove(5);
    t2->insert(5, "Data5");
    t2->remove(5);
    t2->insert(10, "Data10");

    cout << "t->preOrder() outputs "; t2->preOrder();
    cout << "t->postOrder() outputs "; t2->postOrder();
    cout << "t->inOrder() outputs "; t2->inOrder();
    cout << "t->search(4) outputs " << t2->search(4) << endl;
    cout << "t->search(5) outputs " << t2->search(5) << endl;
    cout << "t->search(10) outputs " << t2->search(10) << endl;
    cout << "t->printTree() outputs "; t2->printTree();
    cout << "t->height() outputs "<< t2->height() << endl;
    cout << "t->remove(10) outputs " << t2->remove(10) << endl << endl;
    delete t2;

    cout << "---------------- Question 3 ----------------" << endl << endl;
    BST* t3 = new BST();
    t3->insert(10, "Data10");
    t3->insert(5, "Data5");
    t3->insert(15, "Data15");
    t3->insert(6, "Data6");
```

```
t3->insert(3, "Data3");
t3->insert(16, "Data16");
t3->insert(13, "Data13");
t3->insert(2, "Data2");
t3->remove(5);

cout << "t->preOrder() outputs "; t3->preOrder();
cout << "t->postOrder() outputs "; t3->postOrder();
cout << "t->inOrder() outputs "; t3->inOrder();
cout << "t->search(13) outputs " << t3->search(13) << endl;
cout << "t->search(100) outputs " << t3->search(100) << endl;
cout << "t->search(5) outputs " << t3->search(5) << endl;
cout << "t->printTree() outputs "; t3->printTree();
cout << "t->height() outputs "<< t3->height() << endl << endl;

cout << "---------------- Question 4 ----------------" << endl << endl;
BST* t4 = new BST();
t4->insert(15, "Data15");
t4->insert(14, "Data14");
t4->insert(12, "Data12");
t4->insert(13, "Data13");
t4->insert(10, "Data10");
t4->remove(13);

cout << "t->preOrder() outputs "; t4->preOrder();
cout << "t->postOrder() outputs "; t4->postOrder();
cout << "t->inOrder() outputs "; t4->inOrder();
cout << "t->search(13) outputs " << t4->search(12) << endl;
cout << "t->search(100) outputs " << t4->search(10) << endl;
cout << "t->search(5) outputs " << t4->search(13) << endl;
cout << "t->printTree() outputs "; t4->printTree();
cout << "t->height() outputs "<< t4->height() << endl << endl;

cout << "---------------- Question 5 ----------------" << endl << endl;
BST* t5 = new BST();
t5->insert(15, "Data15");
t5->insert(16, "Data16");
t5->insert(19, "Data19");
t5->insert(17, "Data17");
t5->insert(20, "Data20");
t5->remove(19);
t5->remove(15);

cout << "t->preOrder() outputs "; t5->preOrder();
cout << "t->postOrder() outputs "; t5->postOrder();
cout << "t->inOrder() outputs "; t5->inOrder();
```

```
    cout << "t->search(15) outputs " << t5->search(15) << endl;
    cout << "t->search(20) outputs " << t5->search(20) << endl;
    cout << "t->printTree() outputs "; t5->printTree();
    cout << "t->height() outputs "<< t5->height() << endl << endl;

    cout << "---------------- Question 6 ----------------" << endl << endl;
    BST* t6 = new BST();
    t6->insert(-10, "Data-10");
    t6->insert(-25, "Data-25");
    t6->insert(-5, "Data-5");
    t6->insert(-30, "Data-30");
    t6->insert(-7, "Data-7");
    t6->insert(-3, "Data-3");
    t6->insert(-35, "Data-35");
    t6->insert(-8, "Data-8");
    t6->insert(-4, "Data-4");
    t6->insert(6, "Data6");
    t6->remove(-10);


    cout << "t->preOrder() outputs "; t6->preOrder();
    cout << "t->postOrder() outputs "; t6->postOrder();
    cout << "t->inOrder() outputs "; t6->inOrder();
    cout << "t->search(15) outputs " << t6->search(15) << endl;
    cout << "t->search(20) outputs " << t6->search(20) << endl;
    cout << "t->printTree() outputs "; t6->printTree();
    cout << "t->height() outputs "<< t6->height() << endl << endl;
    return 0;


}
```