```
#pragma once
#include <queue>
#include <algorithm>
/*Implementation of AVL Tree class.*/

template <typename K, typename V>
struct TreeNode
{
        K key;
        V value;
        int height;
        TreeNode* lChild;
        TreeNode* rChild;

        TreeNode()
        {
                this->lChild = this->rChild = nullptr;
                this->height = 0;
        }

        TreeNode(K key, V value)
        {
                this->key = key;
                this->value = value;
                this->lChild = this->rChild = nullptr;
                this->height = 0;
        }

        bool isLeaf()
        {
                return !this->lChild && !this->rChild;
        }
};

template <typename K, typename V>
class AVLTree
{
private:
        TreeNode<K, V>* root;
        int getHeight(TreeNode<K, V>* ptr)
        {
                if (ptr == nullptr)
                        return -1;
                else return ptr->height;
        }

        void updateHeight(TreeNode<K, V>* ptr)
        {
                ptr->height = 1 + (max(getHeight(ptr->lChild),
getHeight(ptr->rChild)));
        }

        int getBalanceFactor(TreeNode<K, V>* ptr)
        {
                return getHeight(ptr->lChild) - getHeight(ptr-
>rChild);
```

```
	}

	void inorderPrintKeys(TreeNode<K, V>* ptr)
	{
		if (ptr)
		{
			inorderPrintKeys(ptr->lChild);
			cout << ptr->key << endl;
			inorderPrintKeys(ptr->rChild);
		}
	}

	void preOrderPrintKeys(TreeNode<K, V>* ptr)
	{
		if (ptr)
		{
			cout << ptr->key << endl;
			preOrderPrintKeys(ptr->lChild);
			preOrderPrintKeys(ptr->rChild);
		}
	}

	void postOrderPrintKeys(TreeNode<K, V>* ptr)
	{
		if (ptr)
		{
			postOrderPrintKeys(ptr->lChild);
			postOrderPrintKeys(ptr->rChild);
			cout << ptr->key << endl;
		}
	}
	void delete_(K key, TreeNode<K, V>*& ptr)
	{
		if (ptr == nullptr)
			return;

		else if (key < ptr->key || key>ptr->key)
		{
			if (key < ptr->key)
				delete_(key, ptr->lChild);

			else delete_(key, ptr->rChild);

			//balancing the node if required
			int balanceFactor = getBalanceFactor(ptr);

			//left left case
			if (balanceFactor > 1 && getBalanceFactor(ptr-
>lChild) >= 0)
			{
				this->rightRotate(ptr);
			}
			//right right case
			else if (balanceFactor < -1 &&
getBalanceFactor(ptr->rChild) <= 0)
			{
				this->leftRotate(ptr);
```

```
                        }
                        //left right
                        else if (balanceFactor > 1 &&
getBalanceFactor(ptr->lChild) < 0)
                        {
                                this->leftRotate(ptr->lChild);
                                this->rightRotate(ptr);
                        }
                        //right left
                        else if (balanceFactor < -1 && this-
>getBalanceFactor(ptr->rChild)>0)
                        {
                                this->rightRotate(ptr->rChild);
                                this->leftRotate(ptr);
                        }
                        else this->updateHeight(ptr);
                }
                else
                {
                        //case 0: leaf node
                        if (ptr->isLeaf())
                        {
                                delete ptr;
                                ptr=nullptr;
                        }
                        //case 1.1: only left child exists
                        else if (ptr->lChild && !ptr->rChild)
                        {
                                TreeNode<K, V>* delNode = ptr;
                                ptr = ptr->lChild;
                                delete delNode;
                        }
                        //case 1.2: only right child exists
                        else if (!ptr->lChild && ptr->rChild)
                        {
                                TreeNode<K, V>* delNode = ptr;
                                ptr = ptr->rChild;
                                delete delNode;
                        }
                        //case 2: both children exits
                        else
                        {
                                TreeNode<K, V>* successor = ptr-
>rChild;
                                while (successor->lChild)
                                        successor = successor->lChild;

                                ptr->key = successor->key;
                                ptr->value = successor->value;
                                delete_(successor->key, ptr->rChild);

                                //we need to perform balancing on ptr
here because we have performed deletion on ptr's right subtree,
                                //so ptr's balance may get disturbed
                                //balancing the node if required
                                int balanceFactor =
getBalanceFactor(ptr);
```

```
                                //left left case
                                if (balanceFactor > 1 &&
getBalanceFactor(ptr->lChild) >= 0)
                                {
                                        this->rightRotate(ptr);
                                }
                                //right right case
                                else if (balanceFactor < -1 &&
getBalanceFactor(ptr->rChild) <= 0)
                                {
                                        this->leftRotate(ptr);
                                }
                                //left right
                                else if (balanceFactor > 1 &&
getBalanceFactor(ptr->lChild) < 0)
                                {
                                        this->leftRotate(ptr->lChild);
                                        this->rightRotate(ptr);
                                }
                                //right left
                                else if (balanceFactor < -1 && this-
>getBalanceFactor(ptr->rChild)>0)
                                {
                                        this->rightRotate(ptr-
>rChild);
                                        this->leftRotate(ptr);
                                }
                                else this->updateHeight(ptr);
                        }
                }
        }//end of delete function

        void insert(K key, V value, TreeNode<K, V>*& ptr)
        {
                if (ptr == nullptr)
                {
                        ptr = new TreeNode<K, V>(key, value);
                }

                else if (key > ptr->key || key < ptr->key)
                {
                        if (key < ptr->key)
                                insert(key, value, ptr->lChild);
                        else
                                insert(key, value, ptr->rChild);

                        int balanceFactor = getBalanceFactor(ptr);

                        //left left case
                        if (balanceFactor > 1 && key < ptr->lChild-
>key)
                        {
                                this->rightRotate(ptr);
                        }
                        //right right case
                        else if (balanceFactor<-1 && key>ptr->rChild-
>key)
                        {
                                this->leftRotate(ptr);
                        }
                        //left right
                        else if (balanceFactor > 1 && key > ptr-
>lChild->key)
                        {
                                this->leftRotate(ptr->lChild);
                                this->rightRotate(ptr);
                        }
                        //right left
                        else if (balanceFactor < -1 && key < ptr-
>rChild->key)
                        {
                                this->rightRotate(ptr->rChild);
                                this->leftRotate(ptr);
                        }
                        else this->updateHeight(ptr);
                }
        }//end of insert function

        V const* search(K key, TreeNode<K, V>* ptr)
        {
                if (ptr == nullptr)
                        return nullptr;
                else if (key < ptr->key)
                        return this->search(key, ptr->lChild);
                else if (key > ptr->key)
                        return this->search(key, ptr->rChild);
                else return &ptr->value;
        }

        void deleteAll(TreeNode<K, V>* ptr)
        {
                if (ptr)
                {
                        deleteAll(ptr->lChild);
                        deleteAll(ptr->rChild);
                        delete ptr;
                }
        }

        /*
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
         y                            x
        / \   Left Rotation          / \
       T1  x  ---------------->     y   T3
          / \                        / \
         T2  T3                    T1  T2
*/
        void leftRotate(TreeNode<K, V>*& ptr)
        {
                TreeNode<K, V>* y = ptr;
                TreeNode<K, V>* x = y->rChild;
                TreeNode<K, V>* T2 = x->lChild;
                ptr = x;
                x->lChild = y;
                y->rChild = T2;
                this->updateHeight(y);
                this->updateHeight(x);
        }

        /*
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
         y                               x
        / \     Right Rotation          /  \
       x   T3   - - - - - - - >        T1   y
      / \                                  / \
     T1  T2                               T2  T3
        */
        void rightRotate(TreeNode<K, V>*& ptr)
        {
                TreeNode<K, V>* y = ptr;
                TreeNode<K, V>* x = y->lChild;
                TreeNode<K, V>* T2 = x->rChild;
                ptr = x;
                x->rChild = y;
                y->lChild = T2;

                this->updateHeight(y);
                this->updateHeight(x);
        }

public:
        AVLTree()
        {
                this->root = nullptr;
        }

        void inorderPrintKeys()
        {
                inorderPrintKeys(this->root);
        }

        void preOrderPrintKeys()
        {
                this->preOrderPrintKeys(this->root);
        }

        void postOrderPrintKeys()
        {
                this->postOrderPrintKeys(this->root);
        }

        void levelOrderPrintKeys()
        {
                if (!this->root)
                        return;
                queue<TreeNode<K, V>*> q;
                q.push(this->root);

                while (!q.empty())
                {
                        TreeNode<K, V>* ptr = q.front();
                        q.pop();
                        cout << ptr->key << endl;

                        if (ptr->lChild)
                                q.push(ptr->lChild);
                        if (ptr->rChild)
                                q.push(ptr->rChild);
                }
        }

        void insert(K key, V value)
        {
                insert(key, value, this->root);
        }

        void delete_(K key)
        {
                delete_(key, this->root);
        }

        V const* search(K key)
        {
                return this->search(key, this->root);
        }

        int getTreeHeight()
        {
                return getHeight(this->root);
        }
        ~AVLTree()
        {
                this->deleteAll(this->root);
        }

};
```