```
#pragma once
#include <queue>
/*Implementation of Binary Search Tree class that has recursive
member functions. */


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

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

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

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

template <typename K, typename V>
class BinarySearchTree
{
private:
	TreeNode<K, V>* root;

	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)
		{
			delete_(key, ptr->lChild);
		}
		else if (key > ptr->key)
		{
			delete_(key, ptr->rChild);
		}
		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);
```

```
                }
        }
    }//end of delete

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

    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;
        }
    }
public:
    BinarySearchTree()
    {
        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);
        }

        ~BinarySearchTree()
        {
                this->deleteAll(this->root);
        }

};
```