```
//============================================================================
// Name        : 21118_DSA_Assign02.cpp
// Author      : Shubham (Roll No: 21118)
//============================================================================

#include <iostream>
#include <string>
using namespace std;

class Node {
private:
	string key, val;
	Node *lChild, *rChild;
public:
	Node(string k="", string v="") {
		key = k, val = v;
		lChild = rChild = NULL;
	}
	void setKeyVal(string k, string v) {key = k, val = v; }
	void printNode() { cout << key << " --> " << val << endl;}
	friend class BST;
};

class BST {
private:
	Node* root;
public:
	BST() {root = NULL;}
	Node* getRoot() {return root;}
	void setRoot(Node* rt) {root = rt;}
	void swapNodeVals(Node* n1, Node* n2) {
		swap (n1->key, n2->key);
		swap (n1->val, n2->val);
	}
	bool isEmpty() { return (root == NULL);}

//	Recursive Implementations
	Node* Insert(Node* curr_root, string key, string val);
	int Search(Node* root, string key, Node*&, Node*&); // returns number of 
comparisons
	void LexoPrint(Node* curr_root);
	void DescPrint(Node* curr_root);
	void deleteNode1(Node*& LOC, Node*& PAR);
	void deleteNode2(Node*& LOC, Node*& PAR);
	void deleteNode(string key);
	Node* deleteNodeSimple(Node* curr_root, string key);
	void Update(Node* curr_root, string, string);
	void UpdateUsingSearch(string key, string);
	void deleteTree(Node*);
};

//Recursive Implementations

Node* BST :: Insert(Node* curr_root, string key, string val) {
	if (curr_root == NULL)
```

```
        return new Node(key, val);

    if (key < curr_root->key)
        curr_root->lChild = Insert(curr_root->lChild, key, val);
    if (key > curr_root->key)
        curr_root->rChild = Insert(curr_root->rChild, key, val);

    return curr_root;
}

int BST :: Search(Node* curr_root, string key, Node*& curr, Node*& parent) {
    curr = curr_root;
    if (curr_root == NULL)
        return -1;

    if (curr_root->key == key)
        return 1;

    parent = curr;
    int l = -1, r = -1;
    if (key < curr_root->key)
        l = Search(curr_root->lChild, key, curr, parent);
    else
        r = Search(curr_root->rChild, key, curr, parent);

    if (l == -1 && r == -1)
        return -1;
    return 1 + ((l != -1) ? l : r);
}

void BST :: LexoPrint(Node* curr_root) {
    if (curr_root != NULL) {
        LexoPrint(curr_root->lChild);
        curr_root->printNode();
        LexoPrint(curr_root->rChild);
    }
}

void BST :: DescPrint(Node* curr_root) {
    if (curr_root != NULL) {
        DescPrint(curr_root->rChild);
        curr_root->printNode();
        DescPrint(curr_root->lChild);
    }
}

Node* BST :: deleteNodeSimple(Node* curr_root, string key) {
    if (curr_root == NULL)
        return curr_root;

    if (curr_root->key == key) {
        if (curr_root->lChild == NULL) {
            Node* temp = curr_root;
            curr_root = curr_root->rChild;
            delete temp;
        }
        else if (curr_root->rChild == NULL) {
            Node* temp = curr_root;
            curr_root = curr_root->lChild;
            delete temp;
        }
        else {
            Node* temp = curr_root->rChild;
            while (temp->lChild)
                temp = temp->lChild;
            swapNodeVals(temp, curr_root);
            curr_root->rChild = deleteNodeSimple(curr_root->rChild, key);
        }
    }
    else if (key < curr_root->key)
        curr_root->lChild = deleteNodeSimple(curr_root->lChild, key);
    else if (key > curr_root->key)
        curr_root->rChild = deleteNodeSimple(curr_root->rChild, key);

    return curr_root;
}

void BST :: Update(Node* curr_root, string key, string new_val) {
    if (curr_root == NULL)
        return;

    if (curr_root->key == key)
        curr_root->val = new_val;
    else if (key < curr_root->key)
        Update(curr_root->lChild, key, new_val);
    else if (key > curr_root->key)
        Update(curr_root->rChild, key, new_val);
}

void BST :: UpdateUsingSearch(string key, string new_val) {
    Node *LOC = NULL, *PAR = NULL;
    Search(root, key, LOC, PAR);

    if (LOC != NULL)
        LOC->val = new_val;
}

void BST :: deleteTree(Node* root) {
    if (root == NULL)
        return;
    deleteTree(root->lChild);
    deleteTree(root->rChild);
    delete root;
}

// When LOC node has no child or only one child
void BST :: deleteNode1(Node*& LOC, Node*& PAR) {
    Node* child = NULL;

    if (LOC->lChild == NULL && LOC->rChild == NULL)
```

```cpp
		child = NULL;
	else if (LOC->lChild != NULL)
		child = LOC->lChild;
	else
		child = LOC->rChild;

	if (PAR != NULL) {
		if (LOC == PAR->lChild)
			PAR->lChild = child;
		else
			PAR->rChild = child;
	}
	else
		root = child;
}

// When LOC has both children
void BST :: deleteNode2(Node*& LOC, Node*& PAR) {
	Node* ptr1 = LOC;
	Node* ptr2 = LOC->rChild;

	while (ptr2->lChild != NULL) {
		ptr1 = ptr2;
		ptr2 = ptr2->lChild;
	}

	deleteNode1(ptr2, ptr1);

	if (PAR != NULL) {
		if (LOC == PAR->lChild)
			PAR->lChild = ptr2;
		else
			PAR->rChild = ptr2;
	}
	else
		root = ptr2;

	ptr2->lChild = LOC->lChild;
	ptr2->rChild = LOC->rChild;

	delete LOC;
}

void BST :: deleteNode(string key) {
	Node *LOC, *PAR;
	LOC = PAR = NULL;

	Search(root, key, LOC, PAR);
	if (LOC == NULL) {
		cout << "Word is not present in dictionary.\n";
		return;
	}
	if (LOC->lChild != NULL && LOC->rChild != NULL)
		deleteNode2(LOC, PAR);
	else {
```

```
            deleteNode1(LOC, PAR);
            delete LOC;
        }
}

int main() {
//  Menu Of Program
    BST bst;

    while (true) {
        cout << "Enter\n\t1 for Insertion of Key\n"
                "\t2 for Searching Key\n"
                "\t3 for Lexographic Print\n"
                "\t4 for Descending Print\n"
                "\t5 for Deletion of Key\n"
                "\t6 for Updating val of Key\n"
                "\t0 to Exit\n: ";
        int choice; cin >> choice;
        if (choice == 0)
            break;

        switch(choice) {
        case 0:
            break;
        case 1: {
            cout << "How many Keys do you want to insert: ";
            int n; cin >> n;
            for (int i = 0; i < n; i++) {
                string key, val;
                cout << "Enter key: "; cin >> key;
                cout << "Enter Val: "; cin >> val;
                Node* root = bst.getRoot();
                root = bst.Insert(root, key, val);
                bst.setRoot(root);
            }
            cout << "Printing in Lexographic Order:\n";
            bst.LexoPrint(bst.getRoot());
            break;
        }
        case 2: {
            cout << "Enter Key to Search: ";
            string key; cin >> key;
            Node *curr, *parent;
            curr = parent = NULL;
            int camp = bst.Search(bst.getRoot(), key, curr, parent);
            if (curr == NULL)
                cout << "Key is not present in BST\n";
            else {
                cout << "Key is Present in BST.\n";
                cout << "The Details are: "; curr->printNode();
                cout << "Number of comparisons required: " << camp <<
endl;
            }
            break;
        }
        case 3: {
            cout << "Printing in Lexographic Order:\n";
            bst.LexoPrint(bst.getRoot());
            break;
        }
        case 4: {
            cout << "Printing in Decreasing Order:\n";
            bst.DescPrint(bst.getRoot());
            break;
        }
        case 5: {
            cout << "Enter key to Delete: ";
            string key; cin >> key;
            bst.deleteNode(key);
            cout << "Tree After Deletion (Lexographic Order):\n";
            bst.LexoPrint(bst.getRoot());
            break;
        }
        case 6: {
            cout << "Enter key (to update) and new val:\n";
            string key, new_val; cin >> key >> new_val;
            bst.UpdateUsingSearch(key, new_val);
            cout << "Printing in Lexographic Order:\n";
            bst.LexoPrint(bst.getRoot());
            break;
        }
        default:
            cout << "INVALID CHOICE.Try Again.\n";
        }
    }

    bst.deleteTree(bst.getRoot());

    return 0;
}
```

Output:

Testcase1: Creating Dictionary

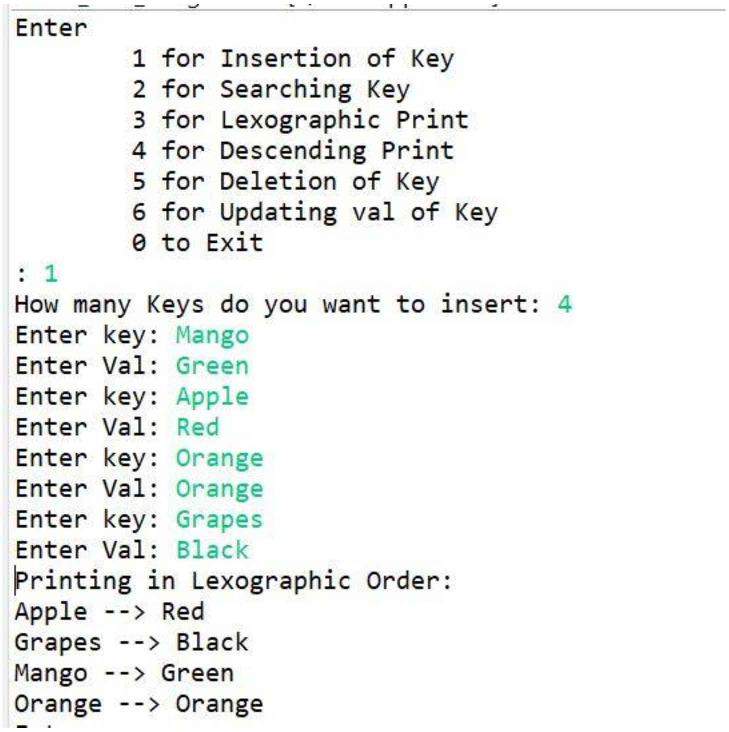
Set of Keys and Values Used:

Mango → Green

Apple → Red

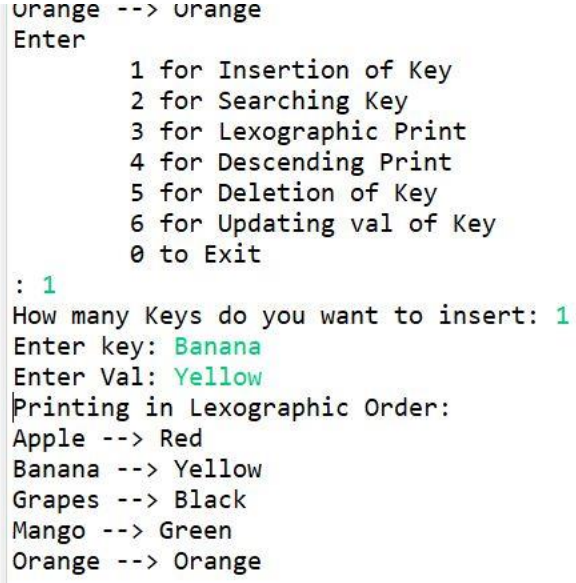
Orange → Orange

Grapes → Black



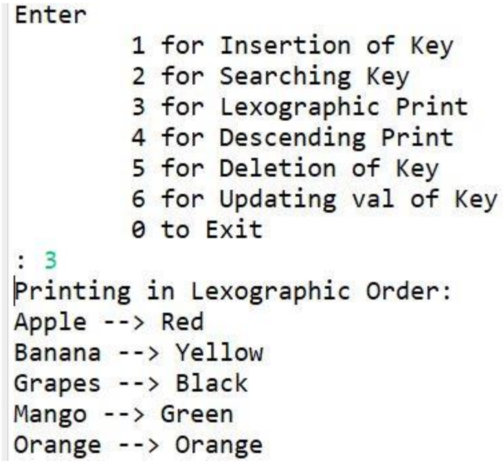
```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 1
How many Keys do you want to insert: 4
Enter key: Mango
Enter Val: Green
Enter key: Apple
Enter Val: Red
Enter key: Orange
Enter Val: Orange
Enter key: Grapes
Enter Val: Black
Printing in Lexographic Order:
Apple --> Red
Grapes --> Black
Mango --> Green
Orange --> Orange
```

Testcase2: Inserting new key → Banana



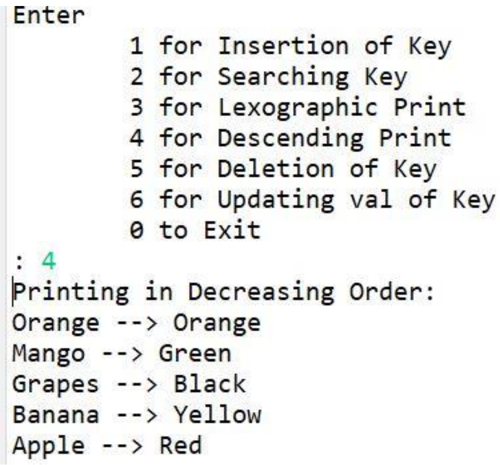
```
Orange --> Orange
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 1
How many Keys do you want to insert: 1
Enter key: Banana
Enter Val: Yellow
Printing in Lexographic Order:
Apple --> Red
Banana --> Yellow
Grapes --> Black
Mango --> Green
Orange --> Orange
```

Testcase3: Increasing Print



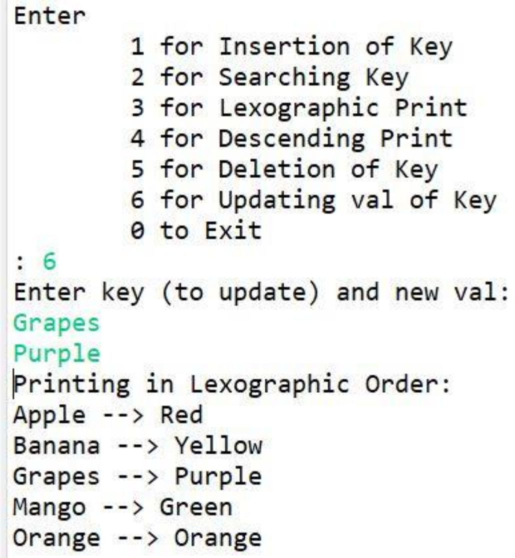
```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 3
Printing in Lexographic Order:
Apple --> Red
Banana --> Yellow
Grapes --> Black
Mango --> Green
Orange --> Orange
```

Testcase4: Decreasing Print



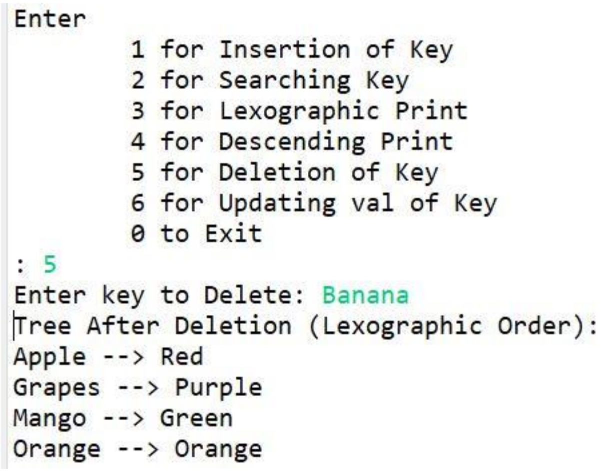
```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 4
Printing in Decreasing Order:
Orange --> Orange
Mango --> Green
Grapes --> Black
Banana --> Yellow
Apple --> Red
```

Testcase5: Updating value of key Grapes to Purple (previously Black)



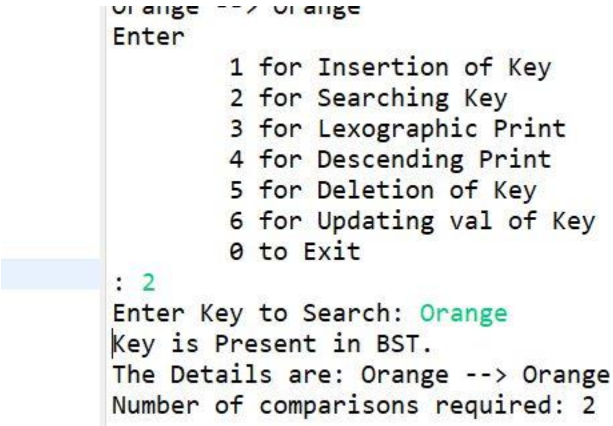
```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 6
Enter key (to update) and new val:
Grapes
Purple
Printing in Lexographic Order:
Apple --> Red
Banana --> Yellow
Grapes --> Purple
Mango --> Green
Orange --> Orange
```

Testcase6: Deleting a key Banana



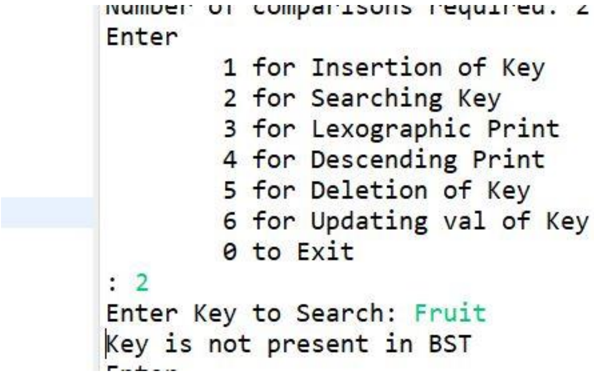
```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 5
Enter key to Delete: Banana
Tree After Deletion (Lexographic Order):
Apple --> Red
Grapes --> Purple
Mango --> Green
Orange --> Orange
```

Testcase7: Searching a Key (Successful Search)



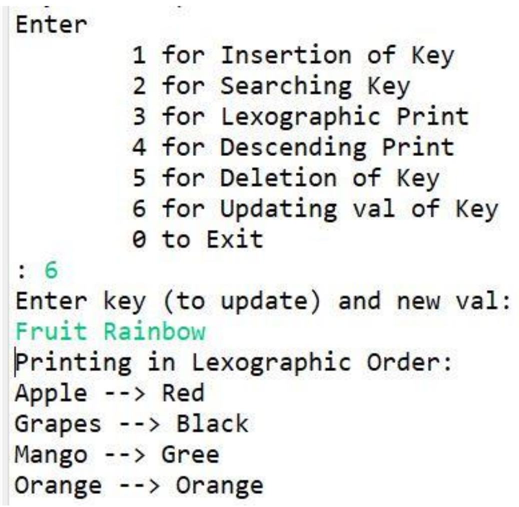
```
Orange --> Orange
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 2
Enter Key to Search: Orange
Key is Present in BST.
The Details are: Orange --> Orange
Number of comparisons required: 2
```

Testcase8: Searching a key (Unsuccessfull Search)



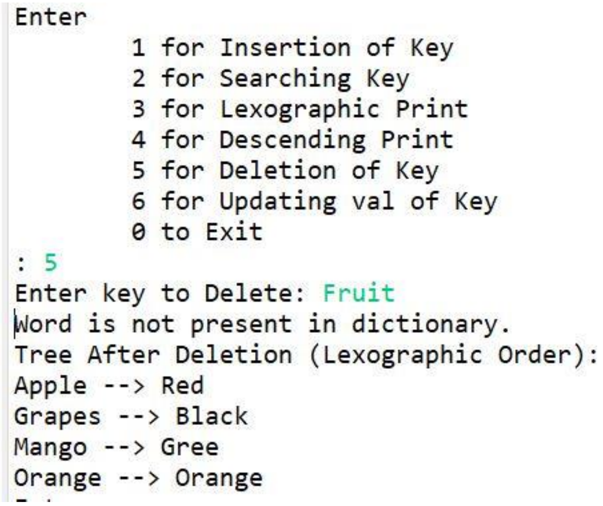
```
Number of comparisons required: 2
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 2
Enter Key to Search: Fruit
Key is not present in BST
Enter
```

Testcase9: Updating Key which is not present in dictionary



```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 6
Enter key (to update) and new val:
Fruit Rainbow
Printing in Lexographic Order:
Apple --> Red
Grapes --> Black
Mango --> Gree
Orange --> Orange
```

Testcase10: Deleting a key which is not present in dictionary



```
Enter
        1 for Insertion of Key
        2 for Searching Key
        3 for Lexographic Print
        4 for Descending Print
        5 for Deletion of Key
        6 for Updating val of Key
        0 to Exit
: 5
Enter key to Delete: Fruit
Word is not present in dictionary.
Tree After Deletion (Lexographic Order):
Apple --> Red
Grapes --> Black
Mango --> Gree
Orange --> Orange
```
