//============================================================================

// 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)

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

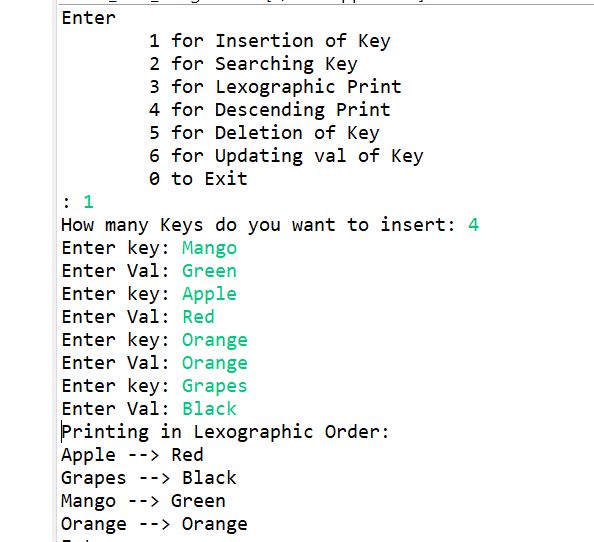
Set of Keys and Values Used:

Mango 🡪 Green

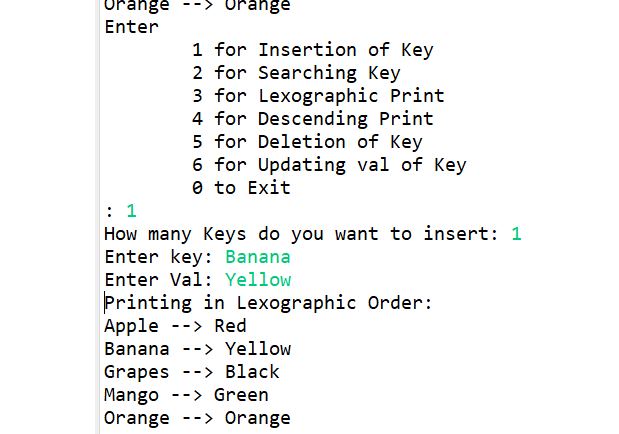
Apple 🡪 Red

Orange 🡪 Orange

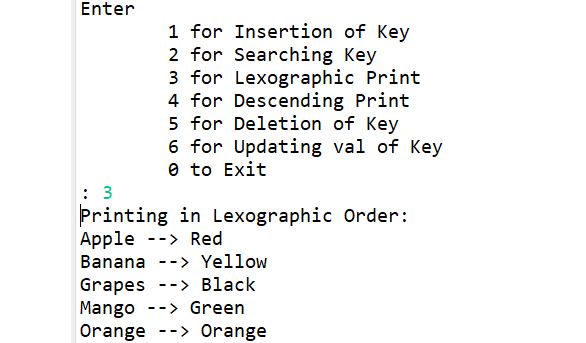
Grapes 🡪 Black



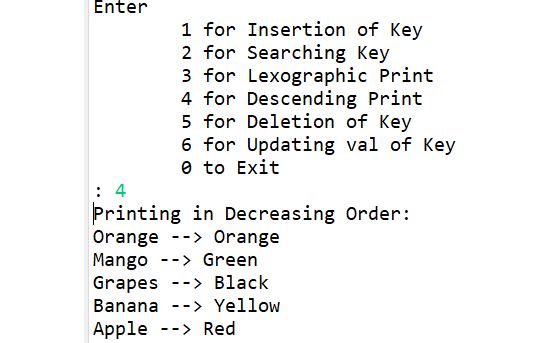
Testcase2: Inserting new key 🡪 Banana



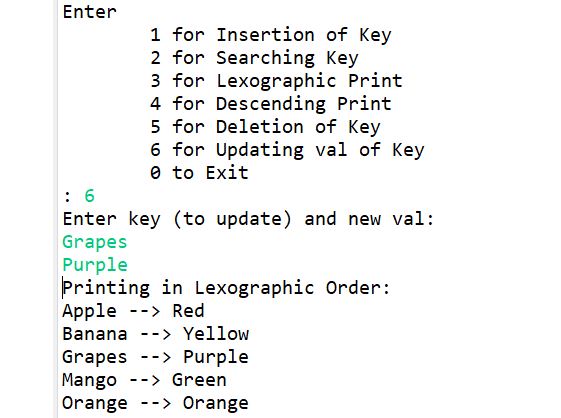
Testcase3: Increasing Print



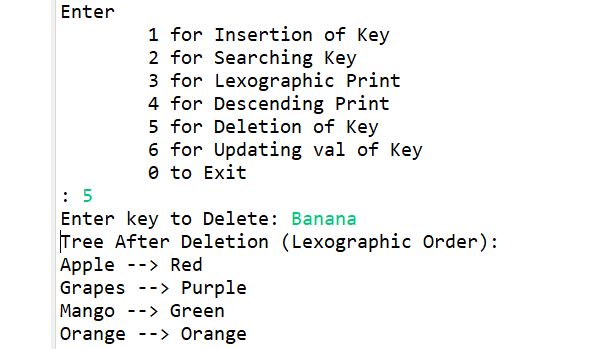
Testcase4: Decreasing Print



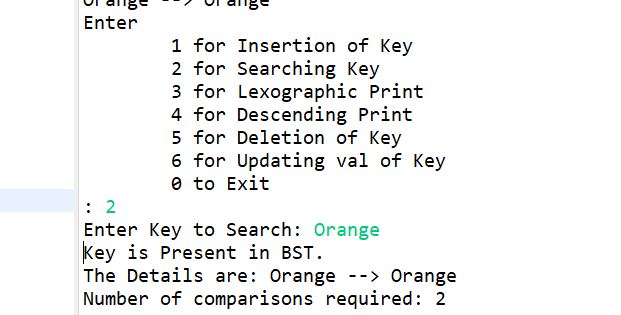
Testcase5: Updating value of key Grapes to Purple (previously Black)



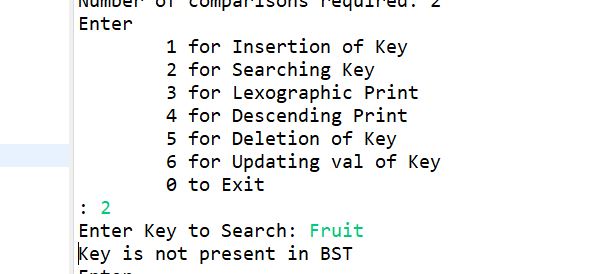
Testcase6: Deleting a key Banana



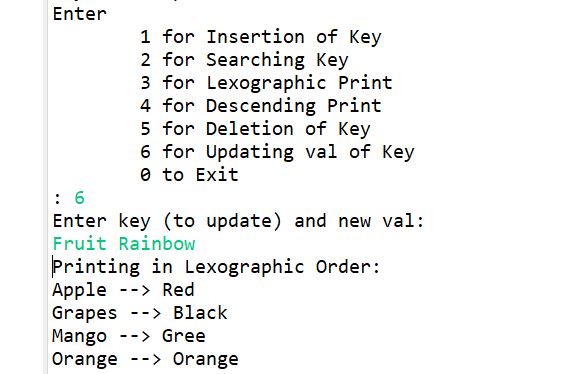
Testcase7: Searching a Key (Successful Search)



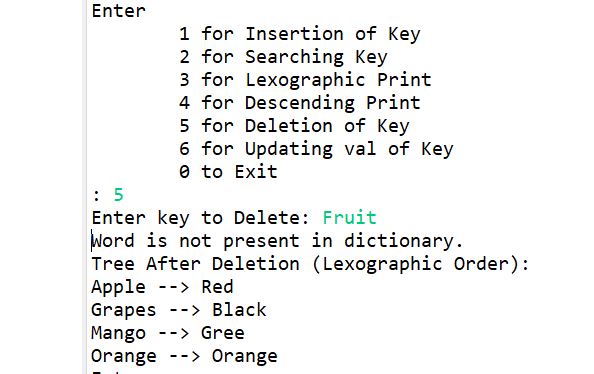
Testcase8: Searching a key (Unsuccessfull Search)



Testcase9: Updating Key which is not present in dictionary



Testcase10: Deleting a key which is not present in dictionary



------------------------------------------------------------------------------------------------------------------------------------------