PROBLEM STATEMENT:

Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry.Also,display whole data sorted in ascending or descending order. Find how many maximum comparisons may require for finding any keyword. Use Binary search tree implementation.

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

class Node {
   string word;
   string meaning;
   Node *left;
   Node *right;

public:
   Node() {
      word = "";
      meaning = "";
      left = NULL;
      right = NULL;

   }

   Node(string word, string meaning) {
      this->word = word;
      this->meaning = meaning;
      left = NULL;
      right = NULL;
   }
   friend class Dictionary;
};

class Dictionary {
   Node *root;

public:
   Dictionary() {
      root = NULL;
   }

   // add Node
   void insert() {

      Node *new_node = new Node();

      // input new node
      cout << "Enter new Word: ";
      cin >> new_node->word;
      cout << "Enter meaning: ";
      cin.ignore();
      getline(cin, new_node->meaning);

      // If root is null i.e dictionary is empty
      if (root == NULL) {
         root = new_node;
```

```
        cout << "Word Added to Dictionary !" << endl;
        return;
    }

    Node *current = root;

    while (current != NULL) {
        if (current->word > new_node->word) {
            if (current->left == NULL) {
                current->left = new_node;
                cout << "left" << current->word << endl;
                return;
            }
            current = current->left;
        } else {
            if (current->right == NULL) {
                current->right = new_node;
                cout << "right" << current->word << endl;
                return;
            }
            current = current->right;
        }
    }
}

//Traversal:
void inorder(Node *temp) {

    if (temp == NULL) {
        return;
    }
    inorder(temp->left);
    cout << temp->word << " =  " << temp->meaning << endl;
    inorder(temp->right);
}
void inorder() {
    inorder(root);
}

//Preorder:
void postorder(Node *temp) {
    if (temp == NULL) {
        return;
    }
    postorder(temp->left);
    postorder(temp->right);
    cout << temp->word << " =  " << temp->meaning << endl;

}
void postorder() {
    postorder(root);
}

//Searching a node
int search(string key) {
    int count = 0;
    Node *current = root;

    while (current != NULL) {

        if (current->word == key) {
            count = count + 1;
            cout << current->word << " = " << current->meaning << endl;
        }

        if (current->word > key) {
            current = current->left;
        } else {
            current = current->right;
        }
    }

    return count;

}

//Update Function

void update(string key) {
    Node *current = root;
    string meaning;

    while (current != NULL) {
        if (current->word == key) {
            cout << current->word << " = " << current->meaning << endl;
            cout << "Enter new meaning: " << endl;
            cin >> meaning;
            current->meaning = meaning;
        }

        if (current->word > key) {
            current = current->left;
        } else {
            current = current->right;
        }
    }
}
//Delete a Node

void deleteNode(string key) {
    Node *current = root;

    Node *parent = NULL;

    //3 Conditions for deletion
    //1. Node to be deleted is a leaf node
    //2. Node to be deleted have one child
    //3. Node to be deleted has two children

    while (current != NULL && current->word != key) {
        parent = current;
        if (current->word > key) {
            current = current->left;
    } else {
        current = current->right;
    }
}

//If current is null then value is not present
if (current == NULL) {
    cout << "Word is not present in the dictionary!!" << endl;
}

//Check the node to be deleted has 0 or 1 child

if (current->left == NULL || current->right == NULL) {

    //check if current have left or right child
    Node *new_node = new Node();

    if (current->left == NULL) {
        new_node = current->right;
    }
    if (current->right == NULL) {
        new_node = current->left;
    }

    //check if current is parents left or right

    if (parent->left == current) {
        parent->left = new_node;
    }

    if (parent->right == current) {
        parent->right = new_node;
    }
    delete current;
}

//if the current node has two childs
else {
    Node *prev = NULL;
    Node *temp;

    temp = current->right;

    while (temp->left != NULL) {
        prev = temp;
        temp = temp->left;
    }

    //To be Review------------------------------09/05/23
    /*
     before replacing the value check if the parent node of the inorder successor
     is the current node or not if it is not then make the left child of the parent
     equal to the inorder successor of the right child else if inorder successor was
     itself  the current then make the right child of the node to be deleted equal
     To the right child of inorder successor
     */
    prev != NULL ?
```

```
            prev->left = temp->right : current->right = temp->right;

        current->word = temp->word;
        delete temp;

      }

    }

  };

  int main() {

    Dictionary *dict = new Dictionary();

    cout << "Practical No 1: Dictionary using Trees" << endl;
    cout << "1. View New Words " << endl;
    cout << "2. Add Words" << endl;
    cout << "3. Search Word" << endl;
    cout << "4. Update Word" << endl;
    cout << "5. exit" << endl;

    int ch = 0;
    cout << "Enter Your choice: ";
    cin >> ch;

    string key;

    int flag = true;

    while (flag) {
      switch (ch) {

      case 1:
        cout << "1.Ascending Order.\n 2.Descending Order: " << endl;
        cin >> ch;
        if (ch == 1) {
          dict->inorder();
        } else if (ch == 2) {
          dict->postorder();
        } else {
          cout << "Invalid" << endl;
        }
        break;

      case 2:
        dict->insert();
        break;
      case 3:

        cout << "Enter key to search: " << endl;
        cin >> key;
        cout << dict->search(key) << " entries found!" << endl;
        break;
      case 4:
        cout << "Enter key to update: " << endl;
        cin >> key;
```

```
            dict->update(key);
            break;
        case 5:
            cout << "EXIT" << endl;
            flag = false;

            break;
        default:
            cout << "Invalid Input" << endl;

        }
    }

    return 0;
}

/*
 * Test Scfrit
 * Dictionary *dict = new Dictionary();

 dict->insert();
 dict->insert();
 dict->insert();
 dict->insert();
 cout << "Ascending Order" << endl;
 dict->inorder();

 cout << "Descending Order" << endl;
 dict->postorder();

 cout << dict->search("m") << " entries found!";

 dict->deleteNode("m");

 dict->inorder();

 */
```