

SPELLCHECK USING BK-TREE

submitted by

Bhagyashri Bhamare (181IT111)

IV SEM B. Tech. (I.T.)

under the guidance of

Dr. Biju R. Mohan

Dept. of I.T., NITK Surathkal

in partial fulfillment for the award of the degree

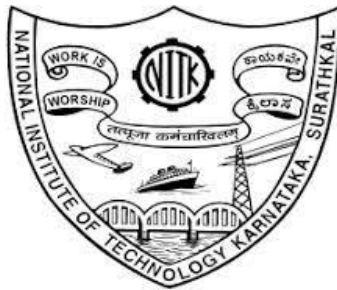
of

Bachelor of Technology

in

Information Technology

at



Department of Information Technology

National Institute of Technology Karnataka, Surathkal June 2020

DECLARATION

I hereby *declare* that the *Seminar (IT290) Report* entitled “Spellcheck Using BK-Tree” which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a ***bonafide report of the work carried out by me***. The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Bhagyashri Nilesh Bhamare - 181IT111

Signature of the Student

Place : NITK, Surathkal

Date :

CERTIFICATE

This is to certify that the Seminar entitled “ Spellcheck Using BK-Tree” has been presented by Bhagyashri Nilesh Bhamare, 181IT111, a student of IV semester B.Tech. (IT), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on Spellcheck Using B-K Tree during the even semester of the academic year 2019 – 2020 in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Examine Name: Bhagyashri N Bhamare

Signature of the Examiner with Date

Abstract

The project Spellcheck using BK-Tree is implemented using BK-Tree. I intended to develop a functioning of spellcheck system which will correct the errors while typing in the text file. This system also detects error in a word and also suggests the alternate word for the given word. This system will suggest the words which are in dictionary. If similar word doesn't exit in dictionary, system will alert that word doesn't exit.

The project aims to implement two concepts: BK-Tree and Levenshtein Distance. By using both these concepts, the project will create a "spellchecker". This spellchecker gives us the following information about the input string:

- 1) If the word is spelled correctly.
- 2) If the word is spelled incorrectly, in which case, it will suggest a list of words that the incorrect word can be corrected to.
- 3) If the word is spelled incorrectly to an extent that it cannot be corrected to, it will display that no suggestions were found.

This project has been implemented in C++ using a text file that contains nearly 150k words of the English language in the alphabetical order.

Table of Content

	Title	Page No.
1	Introduction	vi
2	Literature Review	ix
4	Technical Discussion	x
5	Conclusion and Future Trends	xxvi
6	References	xxvii

Chapter 1

Introduction

1)Levenshtein Distance

This concept is used to measure distance between two sequences. In simpler words, it is the minimum number of edits required to be done to a word character-wise to change one word into the other.

The Levenshtein distance between two strings a, b (length |a| and |b| respectively) is given by

$$\text{lev}_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1, j) + 1 \\ \text{lev}_{a,b}(i, j-1) + 1 \\ \text{lev}_{a,b}(i-1, j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

For example, if the two words are “Helmet” and “Held” then the Levenshtein distance is 3.

Helmet -> Helm (‘et’ deleted) - value = 2

Helm -> Held - value = 1

Hence the total Levenshtein distance is 3

The average time complexity can be reduced to $O(n + d^2)$, where n is the length of the longer string and d is the edit.

What is BK-Tree?

BK-Tree or Burkhard Keller Tree is a data structure that is used to perform spell check based on Edit Distance (Levenshtein distance) concept. BK trees are also used for approximate string matching. Various auto correct feature in many softwares can be implemented based on this data structure.

Now, let's look at the structure of our BK Tree. Like all other trees, BK Tree consists of nodes and edges. The nodes in the BK Tree will represent the individual words in our dictionary and there will be exactly the same number of nodes as the number of words in our dictionary. The edge will contain some integer weight that will tell us about the edit-distance from one node to another. Let's say we have an edge from node u to node v having some edge-weight w , then w is the edit-distance required to turn the string u to v .

Every node in the BK Tree will have exactly one child with same edit-distance. In case, if we encounter some collision for edit-distance while inserting, we will then propagate the insertion process down the children until we find an appropriate parent for the string node. Every insertion in the BK Tree will start from our root node. Root node can be any word from our dictionary.

In this particular implementation of BK Trees, we have assumed the structure of a “Left Child, Right Sibling Tree”. In this, each parent has one left child. Each left child may or may not have siblings. At each node, children of the same parent are linked from left to right.

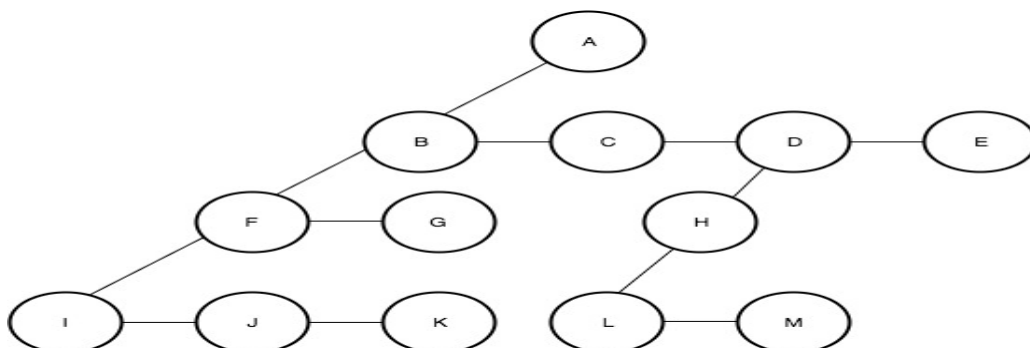


Figure 1.1: Left Child, Right Sibling tree

Above is an example of “Left Child, Right Sibling tree”. When this is combined with the string metric factor, i.e., Levenshtein distance, it yields a BK-Tree.

CHAPTER 2

Literature Review

[^ W. Burkhard and R. Keller. Some approaches to best-match file searching, CACM, 1973](#)

The article finds the problem of searching the set of keys in a file to find a key which is closest to a given query key is discussed. After “closest,” in terms of a metric on the the key space, is suitably defined, three file structures are presented together with their corresponding search algorithms, which are intended to reduce the number of comparisons required to achieve the desired result. These methods are derived using certain inequalities satisfied by metrics and by graph-theoretic concepts. Some empirical results are presented which compare the efficiency of the methods.

[^ Ricardo Baeza-Yates and Gonzalo Navarro. Fast Approximate String Matching in a Dictionary. Proc. SPIRE'98](#)

The article implements successful technique to search large textual databases allowing errors relies on an online search in the vocabulary of the text. To reduce the time of that online search, we index the vocabulary as a metric space. We show that with reasonable space overhead we can improve by a factor of two over the fastest online algorithms, when the tolerated error level is low (which is reasonable in text searching)

Chapter 3

Technical Discussion

System Specifications:

Software Requirements

- g++ installation to compile device program written in C++.
- Edit distance Problem

Hardware requirements:

The hardware requirements are very minimal.

- Processor: 64-bit, Above x86
- Processor speed: 500 MHz and above
- RAM: storage space 4GB and more
- Mouse pointer which is not completely faulty

3.1 Methodology

This project has been implemented using C++. A file “words.txt” containing nearly 150,000 words of the English language is used as the dictionary. A class named BK-Tree is created.

This class contains several functions in order to:

- Add words to the tree
- Return Levenshtein distance between entered string and current node
- Perform search operation to find the word or similar words in the tree
- Store and print the suggested words
- An object of the class BK-Tree is created and all the functions are called for implementation in the main() function

DATA-SET - A file “words.txt” containing nearly 150,000 words of the English language.

- *struct Node*
 - This will initialize every node in tree
- *class BKTree*
 - In this class we will define all function which are needed.
- *BKTree::BKTree()*
 - This is constructor where root=NULL is defined
- *BKTree::~~BKTree()*
 - This is destructor deletion operation on root is performed.
- *Node* BKTree::createNode(string w, size_t d)*
 - This function will create a node with left child and right child as node and store the distance between root and word
- *int BKTree::min(int a, int b, int c)*
 - The above function is used in levenshteinDistance function
 - This function find the minimum of three numbers
- *size_t BKTree::levenshteinDistance(string w1, string w2)*
 - The above function is used in levenshteinDistance function two strings.
 - This function used the idea of dynamic programming
 - Dynamic formula for this code is

- $dp[i][j] = \min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1)$
- void BKTree::recursiveSearch(Node* curNode, vector<string>& suggestions, string w, size_t t, bool& wordFound)
 - The above function is used to search the word in tree. w is word, curNode is root of the tree, t is 1, and wordFound is False.
 - If the word found in tree function will make wordFound=True
 - Else the function will store all the word whose difference from the word is -1 or +1 in suggestions
- void BKTree::add(string w)
 - The above function will add the string in tree.
 - If root is none it will define tree as None
 - Else if insert the node while find the proper parent of this node
- void BKTree::cleanString(basic_string<char>& s)
 - The above function will convert the string lowercase character
- void BKTree::search(string w, int t)
 - The above function will call search call the printSuggestions and recursiveSearch.
- void BKTree::printSuggestions(vector<string>& suggestions, bool wordFound)
 - The above function will tell if word is found in the tree or not
 - If word is found it will say word found
 - Else it will print the suggestions
- int main(int argc, const char * argv[])
 - this function will scan the words file line by line create the tree
 - it will ask for the words to be searched

Runtime of the Program

- If we iterate over all the words in the dictionary and collect the words which are within the tolerance limit the time complexity would be $O(n*m*n)$ (n is the number of words in dict[], m is average size of correct word and n is length of misspelled word) which times out for larger size of dictionary.
- As each node in BK Tree is constructed on basis of edit-distance measure from its parent, we will directly be going from root node to specific nodes that lie within the tolerance limit.
Let's, say our tolerance limit is TOL and the edit-distance of the current node from the misspelled word is dist. Therefore, now instead of iterating over all its children we will only iterate over its children that have edit distance in range $[dist-TOL, dist+TOL]$. This will reduce our complexity by a large extent.
- The time complexity majorly depends on the tolerance limit. We will be considering tolerance limit to be 1. Roughly estimating, the depth of BK Tree will be $\log n$, where n is the size of dictionary. At every level we are visiting 1 node in the tree and performing edit distance calculation.
- Therefore, our Time Complexity will be $O(L1*L2*\log n)$, here $L1$ is the average length of word in our dictionary and $L2$ is the length of misspelled. Generally $L1$ and $L2$ will be small.

Code Implementation

```
#include <iostream>
```

```
#include <fstream>
```

```
#include <time.h>
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <string>
```

```
#include <algorithm>
```

```
using namespace std;
```

struct Node

```
{  
    string word;  
    size_t distance;  
    Node* leftChild;  
    Node* rightSibling;  
};
```

class BKTree

```
{  
private:  
  
    Node* createNode(string w, size_t d);  
    int min(int a, int b, int c);  
    size_t levenshteinDistance(string w1, string w2);
```

```
void recursiveSearch(Node* node, vector<string>& suggestions, string w,  
size_t t, bool& wordFound);
```

```
bool inRange(size_t curDist, size_t minDist, size_t maxDist);
```

```
void printSuggestions(vector<string>& suggestions, bool wordFound);
```

```
public:
```

```
    Node* root;
```

```
BKTree();
```

```
~BKTree();
```

```
void add(string w);
```

```
void cleanString(basic_string<char>& s);
```

```
void search(string w, int t);
```

```
};
```

```
BKTree::BKTree()
```

```
{
```

```
    root = NULL;
```

```
}
```

```
BKTree::~~BKTree()
```

```
{
```

```
    delete root;
```

```
}
```

```
Node* BKTree::createNode(string w, size_t d)
```

```
{
```

```
    Node* node = new Node();
```

```

node->word = w;
node->distance = d;
node->leftChild = NULL;
node->rightSibling = NULL;

return node;
}

```

int BKTree::min(int a, int b, int c)

```

{
    int min = a;
    if (b < min)
        min = b;
    if (c < min)
        min = c;

    return min;
}

```

size_t BKTree::levenshteinDistance(string w1, string w2)

```

{
    if (w1.length() == 0)
        return w2.length();
    if (w2.length() == 0)
        return w1.length();
}

```



```

size_t n_w1 = w1.length();
size_t n_w2 = w2.length();
int cost;

int d[n_w1 + 1][n_w2 + 1];

for (int i = 0; i <= n_w1; i++)
    d[i][0] = i;
for (int i = 0; i <= n_w2; i++)
    d[0][i] = i;

for (int i = 1; i <= n_w1; i++)
{
    for (int j = 1; j <= n_w2; j++)
    {
        cost = (w1[i - 1] == w2[j - 1]) ? 0 : 1;

        d[i][j] = min(d[i - 1][j] + 1,
                      d[i][j - 1] + 1,
                      d[i - 1][j - 1] + cost);
    }
}

return d[n_w1][n_w2];
}

```

```

void BKTree::recursiveSearch(Node* curNode, vector<string>&
suggestions, string w, size_t t, bool& wordFound)
{
    size_t curDist = levenshteinDistance(curNode->word, w);
    size_t minDist = curDist - t;
    size_t maxDist = curDist + t;

    if (!curDist) {
        wordFound = true;
        return;
    }
    if (curDist <= t)
        suggestions.push_back(curNode->word);

    Node* child = curNode->leftChild;
    if (!child) return;

    while (child)
    {
        if (inRange(child->distance, minDist, maxDist))
            recursiveSearch(child, suggestions, w, t, wordFound);

        child = child->rightSibling;
    }
}

```

```
bool BKTree::inRange(size_t curDist, size_t minDist, size_t maxDist)
```

```
{  
    return (minDist <= curDist && curDist <= maxDist);  
}
```

```
void BKTree::printSuggestions(vector<string>& suggestions, bool  
wordFound)
```

```
{  
    if (wordFound)  
    {  
        cout << "Word is spelled correctly." << endl;  
    }  
    else if (suggestions.empty())  
    {  
        cout << "No suggestions found." << endl;  
    }  
    else  
    {  
        cout << "Did you mean: ";  
        for (int i = 0; i < suggestions.size() - 1; i++)  
        {  
            cout << suggestions[i] << ", ";  
        }  
        cout << suggestions[suggestions.size() - 1] << "?" << endl;  
    }  
}
```

```
}
```

```
void BKTree::add(string w)
```

```
{
```

```
    if (root == NULL)
```

```
    {
```

```
        root = createNode(w, -1);
```

```
        return;
```

```
    }
```

```
    Node* curNode = root;
```

```
    Node* child;
```

```
    Node* newChild;
```

```
    size_t dist;
```

```
    while (1)
```

```
    {
```

```
        dist = levenshteinDistance(curNode->word, w);
```

```
        if (!dist) //its the same word
```

```
            return;
```

```
        child = curNode->leftChild;
```

```
        while (child)
```

```
        {
```

```
            if (child->distance == dist)
```

```
            {
```

```
                curNode = child;
```

```

        break;
    }
    child = child->rightSibling;
}
if (!child)
{
    newChild = createNode(w, dist);
    newChild->rightSibling = curNode->leftChild;
    curNode->leftChild = newChild;
    break;
}
}
}

```

```

void BKTree::cleanString(basic_string<char>& s)
{
    for (basic_string<char>::iterator p = s.begin(); p != s.end(); ++p)
    {
        *p = tolower(*p);
    }
}

```

```

void BKTree::search(string w, int t)
{
    vector<string> suggestions;
    bool wordFound = false;

```

```

        recursiveSearch(root, suggestions, w, t, wordFound);

        printSuggestions(suggestions, wordFound);
    }

int main(int argc, const char * argv[])
{
    BKTREE* tree = new BKTREE();

    string line;

    ifstream myFile("words.txt"); // 154,937 English words

    if (myFile.is_open())
    {
        cout << "Building structure... " << endl;

        while (getline(myFile, line))
            tree->add(line);

        myFile.close();
    }

    cout << "Done." << endl << endl;

    string word;

```

```

while (word != "0")
{
    cout<<"(Enter 0 if you wish to exit)\n";
    cout << "Enter word: ";
    cin >> word;
    if(word == "0")
    {
        break;
    }
    tree->cleanString(word);

    tree->search(word, 1);

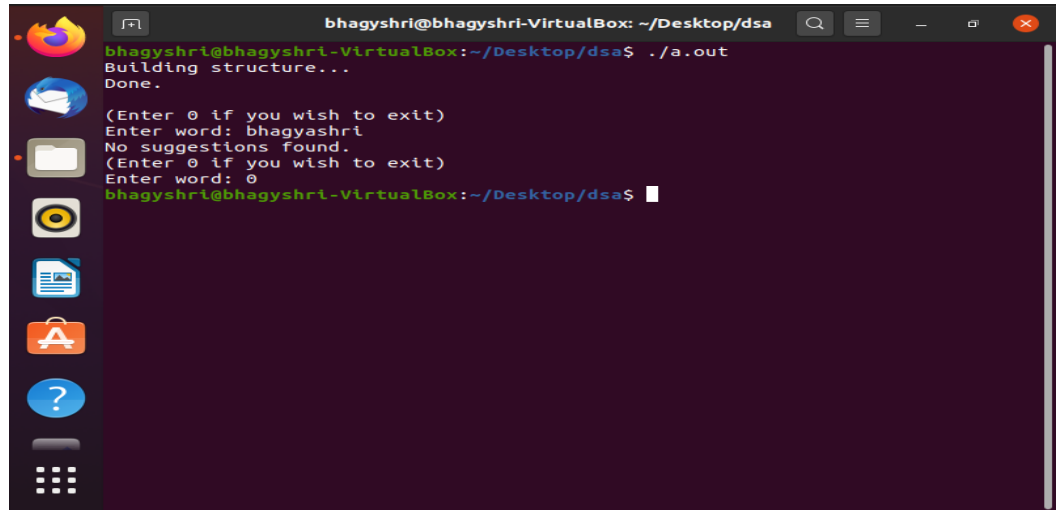
}
return 0;
}

```

How to run the code?

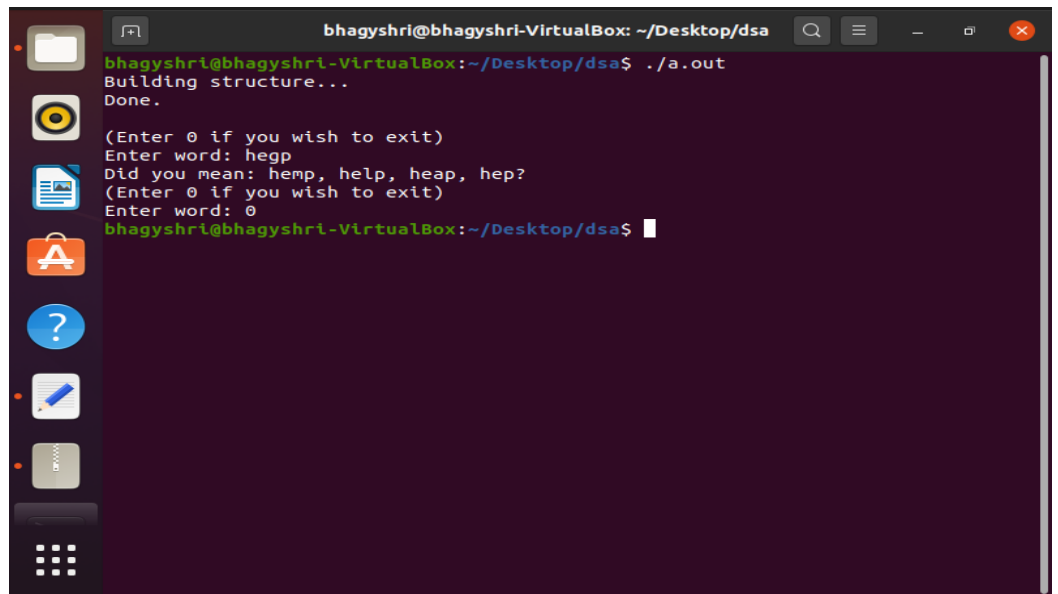
- compile the main.cpp file using `g++ main.cpp`
- Run the program using `./a.out`

3.2 EXPERIMENTAL RESULTS



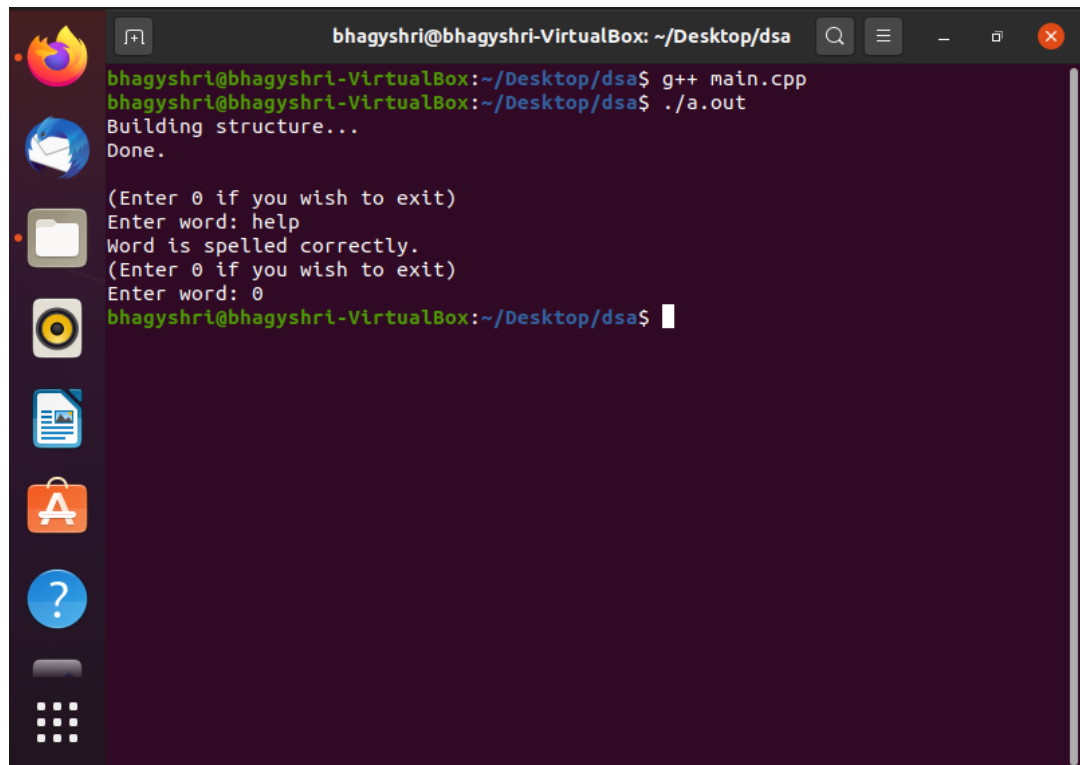
```
bhagyashri@bhagyashri-VirtualBox: ~/Desktop/dsa
bhagyashri@bhagyashri-VirtualBox:~/Desktop/dsa$ ./a.out
Building structure...
Done.
(Enter 0 if you wish to exit)
Enter word: bhagyashri
No suggestions found.
(Enter 0 if you wish to exit)
Enter word: 0
bhagyashri@bhagyashri-VirtualBox:~/Desktop/dsa$
```

Figure 3.2.1: no suggestion for Bhagyashri word



```
bhagyashri@bhagyashri-VirtualBox: ~/Desktop/dsa
bhagyashri@bhagyashri-VirtualBox:~/Desktop/dsa$ ./a.out
Building structure...
Done.
(Enter 0 if you wish to exit)
Enter word: hepp
Did you mean: hemp, help, heap, hep?
(Enter 0 if you wish to exit)
Enter word: 0
bhagyashri@bhagyashri-VirtualBox:~/Desktop/dsa$
```

Figure3.2.2: suggestion for word wrong word hemp



```
bhagyshri@bhagyshri-VirtualBox: ~/Desktop/dsa
bhagyshri@bhagyshri-VirtualBox:~/Desktop/dsa$ g++ main.cpp
bhagyshri@bhagyshri-VirtualBox:~/Desktop/dsa$ ./a.out
Building structure...
Done.
(Enter 0 if you wish to exit)
Enter word: help
Word is spelled correctly.
(Enter 0 if you wish to exit)
Enter word: 0
bhagyshri@bhagyshri-VirtualBox:~/Desktop/dsa$
```

Figure 3.2.3: help word found in the tree

Chapter 4

Conclusion and Future Trends

We proposed a method to determine spellcheck and approximate searching on the indexed text collection. Most present methods on a spellcheck use a classical online algorithm. This project instead aims to organize the vocabulary as a metric space, taking advantage of the fact that the edit distance models the approximate search is indeed a metric. This method can be applied to other problems also when a dictionary is searched allowing errors. Limitation of this project is that it can't be applied to small file of words.

Future work

- 1) Reduce space complexity
- 2) Use dictionary to store distance parameter of every node
- 3) Do search all small file also
- 4) Auto correct feature can be implemented with the help of BK Trees.

References

Given below are some online sources which helped me in progressing with the project implementation.

- [^ W. Burkhard and R. Keller. Some approaches to best-match file searching. CACM, 1973](#)
- [^ R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed queries trees. In M. Crochemore and D. Gusfield, editors, 5th Combinatorial Pattern Matching, LNCS 807, pages 198–212, Asilomar, CA, June 1994.](#)
- [^ Ricardo Baeza-Yates and Gonzalo Navarro. Fast Approximate String Matching in a Dictionary. Proc. SPIRE'98](#)