Data Structures Lab
PROJECT
(15B17CI371)

IntelliSearch:
Using Ternary Search Trees for Searching and Auto-Completion

Submitted by:
Aditya Negi (21103068)
Akshita Khandelwal (21103072)
Akshit Tyagi (21103085)

B3 Batch | CSE | 3rd Sem

Submitted to:
Bharat Gupta

Department of CSE/IT
Jaypee Institute of Information Technology

Introduction

AIM: To provide autocompletion and spellchecking using Ternary Search Trees given data.

We have been storing large-scale data like phonebook data in the form of BSTs and Maps. However, this method is not so efficient since users may not get the desired results in an efficient manner. Compromises are to be made either in terms of time complexity or in terms of memory utilization.
This calls for a better design of a system wherein people can easily retrieve info without wasting much time.

This project has been designed to overcome this problem.

About the project

This system is developed keeping in view the general need required by the user while requiring searching and/or closest-record searching while using any given data. For example, a phone directory book or a database of movies. The users will have the ability to efficiently store and retrieve information with these functionalities.

Ternary Search Trees are a Data Structure similar to Binary Search Trees, the only difference being that each node includes three children. BST's require rearrangement of entire data which accounts for extreme inefficiency when a dataset is very large. To counter this, the data structures trie's can be used where each node has 26 children each corresponding to a possibility in the english alphabet. We would also need to include special characters and spaces which would only add to the **extremely** large amounts of space the trie would take up, be it only in storing the large amount of null pointers. Ternary Search Trees take care of both of these issues by not requiring any rearrangement, while not taking up too much space.

The project has a wide array of implementations, ranging from using it to search usernames to movie names from a database.

In order to demonstrate the power of this project, we will be implementing the user being able to search phrases / nearest-matching phrases in a directory containing a large number of .txt files.
If no such phrase is available, the closest-matching phrase along will be displayed as per user input provided to the system.

Project Objectives

To demonstrate the power of using ternary search trees for searching a large amount of data, very quickly while not taking up a lot of space as is the case with Tries. We plan to do this by using them for providing autocomplete and closest suggestion to a search term present in given files.

Dataset Description

The dataset can be as large as the phonebook of a city or a directory containing a large number of research papers. To provide a demonstration for a general case, we have implemented in-file searching for any given files. The files can be given in the form of a directory path and more than one files can be added (not necessarily in the same directory).

Features of the project

"IntelliSearch" has been designed to computerize the following functions that are performed by the system:

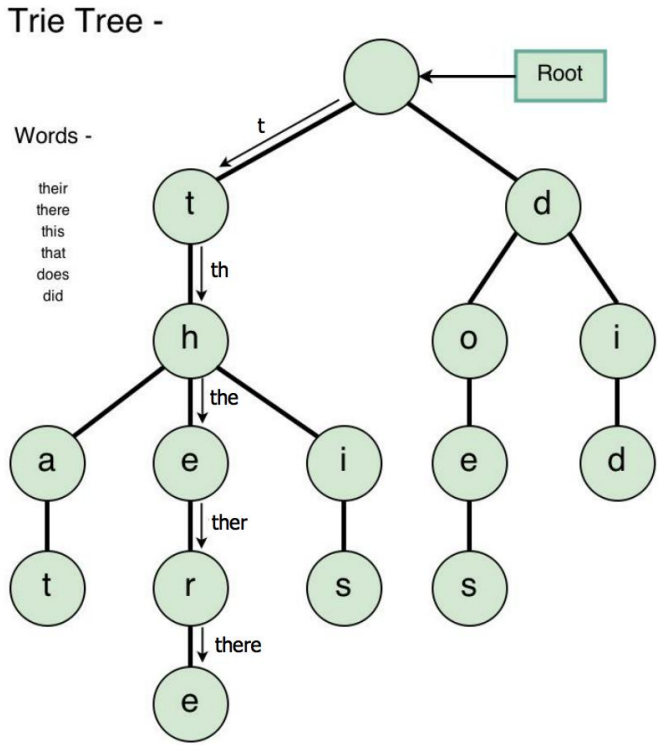
- **Search/Autocomplete for a given term *FAST*-** You will be able to find a given term provided a list of source files.
- **Memory Efficient-** Using ternary tree as the primary data structure ensures that we do not tradeoff heavily on memory for gains in speed of searching/autocompletion.
- **Specify Multiple Files-** You can specify multiple files among which you want to search your text from.
- **CLI-** You can pass all the filepaths as parameters while running the executable directly from the CLI using the -f flag.

Data Structures Used are -

- Ternary Search Trees
- Vectors
- Searching

Tries

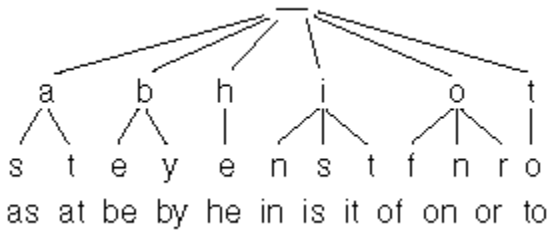
We know of Binary Search Trees where each node has at most two children/subtrees. Tries are k-ary trees. Unlike BST's a node in a trie doesn't store its associated key, instead its position in the tree determines that key.



A common application of Tries is seen in string-searching algorithms however they come with the drawback of taking up enormous amounts of space.

Ternary Search Tree

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree. It has three children, a left subtree, a right subtree and a mid or an equal subtree.



The information stored in a trie for performing fast string-searching can be stored in ternary search trees, making it more efficient by reducing the storage requirements considerably.

The Code:

```
#include <bits/stdc++.h>
using namespace std;

// Define the Node of the tree
struct Node {
  char data;
  int end;
  struct Node* left;
  struct Node* eq;
  struct Node* right;
};
```

```
// Function to create a Node
Node* createNode(char newData)
{
  struct Node* newNode = new Node();
  newNode->data = newData;
  newNode->end = 0;
  newNode->left = NULL;
  newNode->eq = NULL;
  newNode->right = NULL;
  return newNode;
}

// Function to insert a word in the tree
void insert(Node** root, string word, int pos = 0)
{

  // Base case
  if (!(*root))
        *root = createNode(word[pos]);

  // If the current character is less than root's data,
then it is inserted in the left subtree

  if ((*root)->data > word[pos])
        insert(&((*root)->left), word,
              pos);

  // If current character is more than root's data,
then it is inserted in the right subtree

  else if ((*root)->data < word[pos])
        insert(&((*root)->right), word,
              pos);

  // If current character is same as that of the root's
data

  else {
        // If it is the end of word

        if (pos + 1 == word.size())
              // Mark it as the end of word
              (*root)->end = 1;
```

```
        // If it is not the end of the string, then
current character inserted in the equal subtree

        else
              insert(&((*root)->eq), word, pos + 1);
  }
}

// Function to traverse the ternary search tree
void traverse(Node* root, vector<string>& ret, char*
buff, int depth = 0)
{
  // Base case
  if (!root)
        return;
  // The left subtree is traversed first
  traverse(root->left, ret, buff, depth);

  // Store the current character
  buff[depth] = root->data;

  // If the end of the string is detected, store it in
the final ans
  if (root->end) {
        buff[depth + 1] = '\0';
        ret.push_back(string(buff));
  }

  // Traverse the equal subtree
  traverse(root->eq, ret, buff, depth + 1);

  // Traverse the right subtree
  traverse(root->right, ret, buff, depth);
}

// Utility function to find all the words
vector<string> util(Node* root,
                        string pattern)
{
  // Stores the words to suggest
  char buffer[1001];

  vector<string> ret;
```

```
  traverse(root, ret, buffer);

  if (root->end == 1)
        ret.push_back(pattern);
  return ret;
}

// Function to autocomplete based on the given prefix
and return the suggestions
vector<string> autocomplete(Node* root, string pattern)
{
  vector<string> words;
  int pos = 0;

  // If pattern is empty return an empty list
  if (pattern.empty())
        return words;

  // Iterating over the characters of the pattern and
find it's corresponding node in the tree

  while (root && pos < pattern.length()) {

        // If current character is smaller
        if (root->data > pattern[pos]) {
              // Search the left subtree
              if (root->left==NULL) {
                    words.push_back("-NO WORD FOUND-");
                    return words;
              }
              root = root->left;
        }
        // current character is greater
        else if (root->data < pattern[pos]) {
              // Search right subtree
              if (root->right==NULL) {
                    cout << "Nothing Found. There is ";
                    return words;
              }
              root = root->right;
        }
        // If current character is equal
        else if (root->data == pattern[pos]) {
```

```
            // Search equal subtree since character is
found, move to the next character in the pattern
            if (root->eq==NULL) {
                  cout << "[✓]";
                  return words;
            }
            root = root->eq;
            pos++;
      }

      else
            return words;

  }

  words = util(root, pattern);

  return words;
}


void print(vector<string> sugg,
      string pat)
{
  for (int i = 0; i < sugg.size();
      i++)
      cout << pat << sugg[i].c_str()
            << "\n";
}

void getwords (string filename, vector<string> &vecstr)
{
    string line;
    ifstream myfile (filename);
    string word;
    if (myfile.is_open())
    {
        while (myfile >> word)
        {
            vecstr.push_back(word);
        }
        myfile.close();
    }
    else cout << "can't open the file";
```

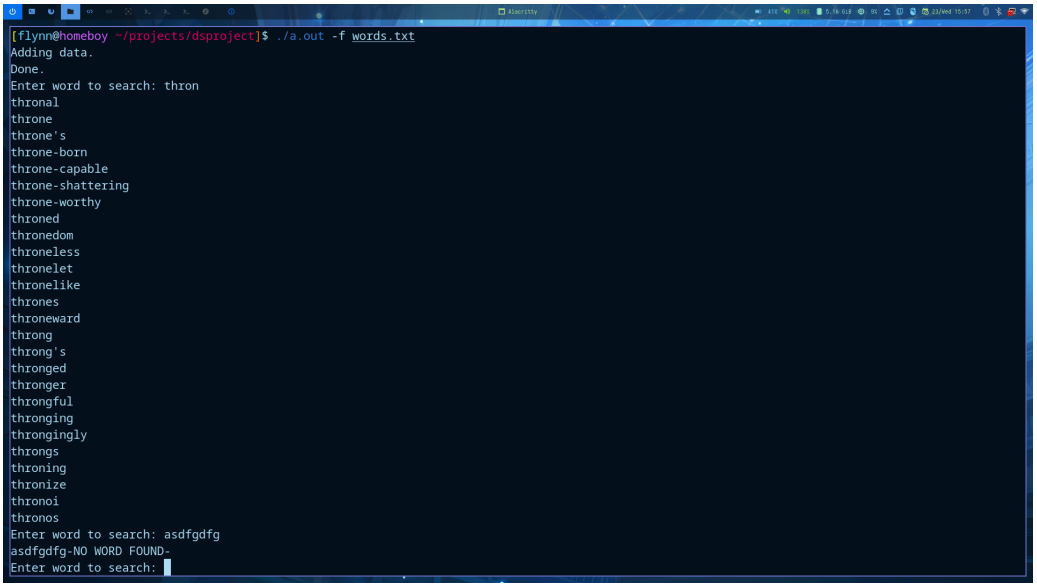
```
}

void searchingHandler(Node* tree) {

  cout << "Enter word to search: ";
  string pat;
  cin >> pat;
  vector<string> sugg
       = autocomplete(tree, pat);
  if (sugg.size() == 0) {
       cout << "None";
  }
  else {
       print(sugg, pat);
  };
};

void submain(string filename, vector <string> &V) {
  getwords(filename, V);
  Node* tree = NULL;
  cout << "Adding data. \n";
  for (string str : V)
       insert(&tree, str);
  cout << "Done. \n";
  while(true){
       searchingHandler(tree);
  }
}

int main(int argc, char* argv[]) {
  vector <string> V;
  if (strcmp(argv[1], "-f") == 0){
       for (int i = 0; i < argc-2; i++) {
            submain(argv[i+2],V);
       };
  };
  return 0;
}
```



References:

https://theoryofprogramming.wordpress.com/2015/01/16/trie-tree-implementation/
https://www.cs.upc.edu/~ps/downloads/tst/tst.html
https://iq.opengenus.org/autocomplete-with-ternary-search-tree/
https://en.wikipedia.org/wiki/Ternary_search_tree