DATA STRUCTURES AND ALGORITHIMS

REVIEW-1

NAME: SRISHTI CHOPRAA

REG. NO. : 20BML0046

ABSTRACT

- In this busy world no one has time now. Technology is being developed every day to increase the efficiency. In this front, word predictor is a small step which increases our efficiency multifold times. Word predictor has applications in various areas like texting, search engine etc. To develop our word predictor program, we used the data structure Trie and also Binary Tree. Our program uses a stored file of words to predict the words which the user may think of thus helping a lot.

INTRODUCTION TO PROBLEM

- Word Predictor have application in messaging application like WhatsApp, web search engines, word processors, command like interpreters etc. The original purpose of word prediction software was to help people with physical disabilities increase their typing speed, as well as to help them decrease the number of keystrokes needed in order to complete a word or a sentence. Thus, in this front we developed our own program for word predictor using data structure trie which definitely increases efficiency of the user by at least 10%. Autocomplete, or word completion, is a feature in which an application predicts the rest of a word a user is typing. In graphical user interfaces, users can typically press the tab key to accept a suggestion or the down arrow key to accept one of several.

DETAILS ABOUT WORK

To use the dynamic data structure TRIE based on it develop a program having industrial application to predict words.

➢ To use the dynamic data structure tree in developing the program

➢ To use the data structure ‘trie’ being in the program.

➢ To construct a strong and efficient algorithm to develop the program which is editable and can be later used as a module for bigger software mechanism

➢ To develop a real time program which is efficient and has a fast processing and also has an industrial application

▸Autocomplete speeds up human-computer interactions when it correctly predicts the word a user intends to enter after only a few characters have been typed into a text input field. It works best in domains with a limited number of possible words (such as in command line interpreters), when some words are much more common (such as when addressing an email), or writing structured and predictable text (as in source code editors). Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user. Autocomplete or word completion works so that when the writer writes the first letter or letters of a word, the program predicts one or more possible words as choices. If the word he intends to –write is included in the list he can select it, for example by using the number keys.

▸If the word that the user wants is not predicted, the writer must enter the next letter of the word. At this time, the word choice(s) is altered so that the words provided begin with the same letters as those that have been selected. When the word that the user wants appears it is selected, and the word is inserted into the text. In another form of word prediction, words most likely to follow the just written one are predicted, based on recent word pairs used. Word prediction uses language modeling, where within a set vocabulary the words are most likely to occur are calculated. Along with language modeling, basic word prediction on AAC devices is often coupled with a regency model, where words that are used more frequently by the AAC user are more likely to be predicted. Word prediction software often also allows the user to enter their own words into the word prediction dictionaries either directly, or by "learning" words that have been written.

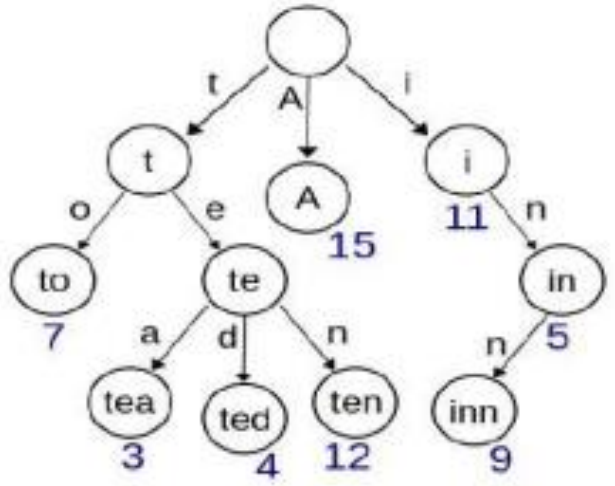
DATA STRUCTURES USED

- TRIE

In this program Data structure Trie is being used to search the data in an ordered fashion. In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest

A TRIE STRUCTURE

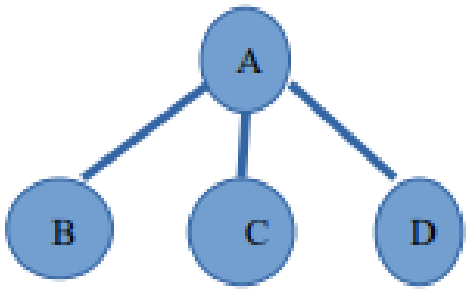
- For the space-optimized presentation of prefix tree, see compact prefix tree. In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a tree shaped deterministic finite automaton. Each finite language is generated by a trie automaton, and each trie can be compressed into a deterministic acyclic finite state automaton.

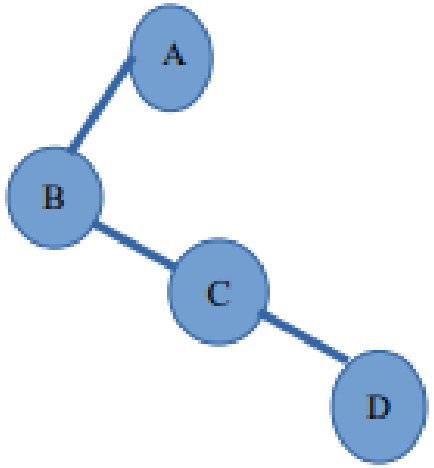


BINARY TREE

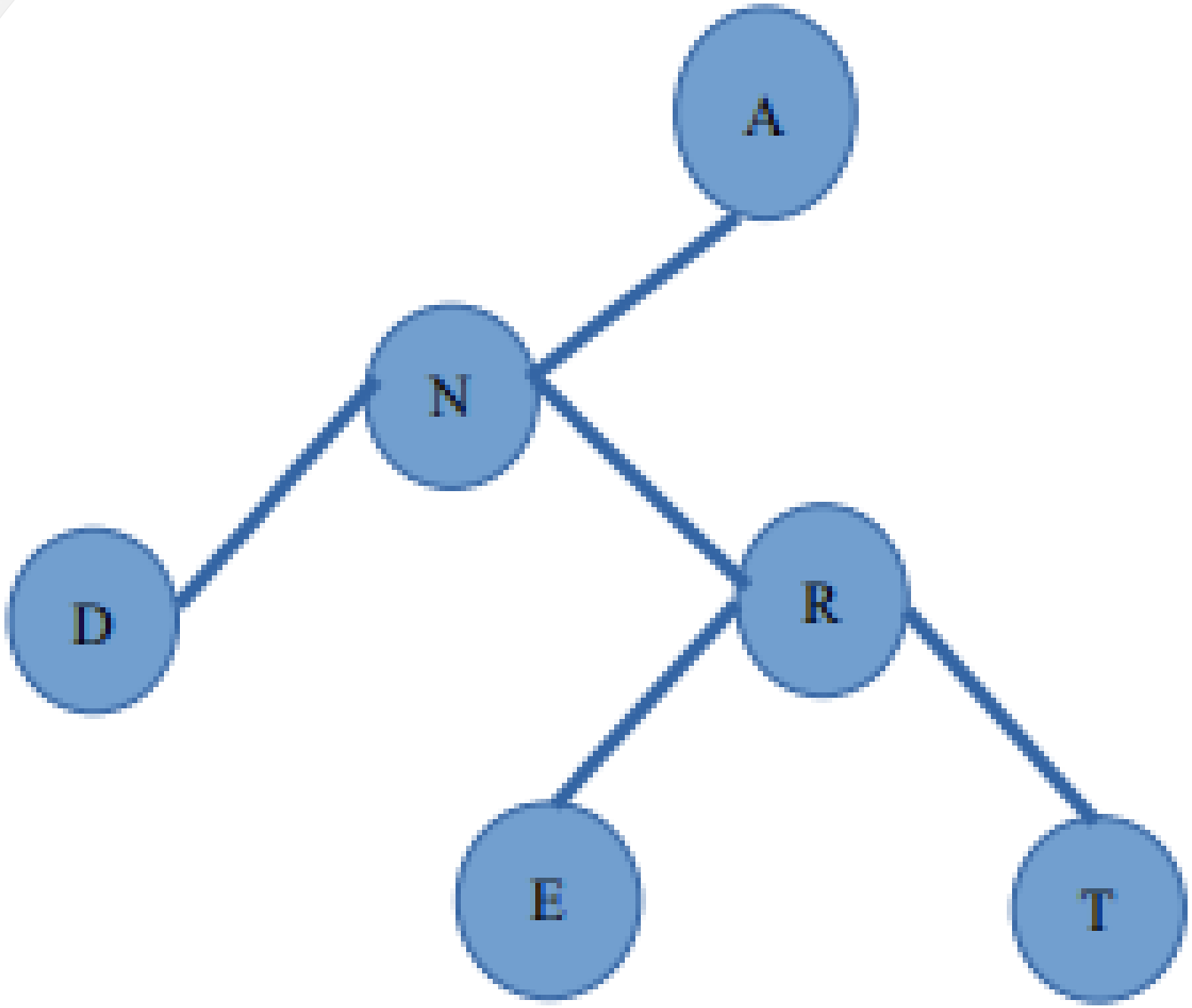
- A tree whose elements have at most two children is called binary tree. Since they have only two children they can be name as left child and right child. When general tree is converted to binary tree then left most child of parent will become left child and all remaining children will be right child to their siblings.
- To traverse through the binary tree, we have three different types of traversals. They are:
- • Post-order • In-order • Pre-order
- Post-order: It first traverse through left child node and then right child finally to the root (LRV).
- In-order: It first traverse through left child node and then root finally to right child (LVR).
- Pre-order: It first traverse through root and then left child finally to right child (VLR).

BINARY TREE STRUCTURE





- In auto-complete binary tree the traversal through the tree is it first traverse through the root and then to the left child and then to it's right child till we find the second letter if it is found then it traverse through the right child of the found node. To traverse for the word ARE: First visit the root node A and traverse to its left child N. Then compare to second letter of the word. Since it's not the same so we traverse to its right child R and compare. It is same therefore we traverse to its left child.


A
N
D
R
E
T

PROPOSED WORK

- We have used the data structure Trie and Binary Tree to autocomplete the words. As explained above, these two data structures enable us to autocomplete the word using various way of tree traversal. To demonstrate the working of the program, we have used a text file with the list of words imported from the internet and incorporated into the program. The program will then give out the results with the help of this list of words. The program will prompt an input from the user. Using the input, the program will traverse the tree. After traversing the maximum height with the help of the user given input, it will check if there are any further branches of the node. In case there are more branches, the program will display the top 20 options starting from leftmost root and so on. In case there are no further branches, the program will ask to save the input as a word and add it to the text file with the list of words which then the program will incorporate in the tree from that point. Using this project, we want to demonstrate an efficient way to guess the word the user is typing. This idea can be added to different programs to increase the productivity and save time as well. Also, different neural network and input datasets could be used to make this program more efficient.

THANKYOU

SOURCE CODE:

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

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
struct TrieNode
{
	struct TrieNode *children[ALPHABET_SIZE];

	// isWordEnd is true if the node represents
	// end of a word
	bool isWordEnd;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
	struct TrieNode *pNode = new TrieNode;
	pNode->isWordEnd = false;

	for (int i = 0; i < ALPHABET_SIZE; i++)
		pNode->children[i] = NULL;

	return pNode;
}

// If not present, inserts key into trie. If the
// key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const string key)
{
	struct TrieNode *pCrawl = root;

	for (int level = 0; level < key.length(); level++)
	{
		int index = CHAR_TO_INDEX(key[level]);
		if (!pCrawl->children[index])
			pCrawl->children[index] = getNode();
```

```
                pCrawl = pCrawl->children[index];
        }

        // mark last node as leaf
        pCrawl->isWordEnd = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const string key)
{
        int length = key.length();
        struct TrieNode *pCrawl = root;
        for (int level = 0; level < length; level++)
        {
                int index = CHAR_TO_INDEX(key[level]);

                if (!pCrawl->children[index])
                        return false;

                pCrawl = pCrawl->children[index];
        }

        return (pCrawl != NULL && pCrawl->isWordEnd);
}

// Returns 0 if current node has a child
// If all children are NULL, return 1.
bool isLastNode(struct TrieNode* root)
{
        for (int i = 0; i < ALPHABET_SIZE; i++)
                if (root->children[i])
                        return 0;
        return 1;
}

// Recursive function to print auto-suggestions for given
// node.
void suggestionsRec(struct TrieNode* root, string currPrefix)
{
        // found a string in Trie with the given prefix
        if (root->isWordEnd)
        {
                cout << currPrefix;
                cout << endl;
        }
```

```
        // All children struct node pointers are NULL
        if (isLastNode(root))
                return;

        for (int i = 0; i < ALPHABET_SIZE; i++)
        {
                if (root->children[i])
                {
                        // append current character to currPrefix string
                        currPrefix.push_back(97 + i);

                        // recur over the rest
                        suggestionsRec(root->children[i], currPrefix);
                }
        }
}

// print suggestions for given query prefix.
int printAutoSuggestions(TrieNode* root, const string query)
{
        struct TrieNode* pCrawl = root;

        // Check if prefix is present and find the
        // the node (of last level) with last character
        // of given string.
        int level;
        int n = query.length();
        for (level = 0; level < n; level++)
        {
                int index = CHAR_TO_INDEX(query[level]);

                // no string in the Trie has this prefix
                if (!pCrawl->children[index])
                        return 0;

                pCrawl = pCrawl->children[index];
        }

        // If prefix is present as a word.
        bool isWord = (pCrawl->isWordEnd == true);

        // If prefix is last node of tree (has no
        // children)
        bool isLast = isLastNode(pCrawl);

        // If prefix is present as a word, but
```

```
        // there is no subtree below the last
        // matching node.
        if (isWord && isLast)
        {
                cout << query << endl;
                return -1;
        }

        // If there are are nodes below last
        // matching character.
        if (!isLast)
        {
                string prefix = query;
                suggestionsRec(pCrawl, prefix);
                return 1;
        }
}

// Driver Code
int main()
{
        struct TrieNode* root = getNode();
        insert(root, "hello");
        insert(root, "dog");
        insert(root, "hell");
        insert(root, "cat");
        insert(root, "a");
        insert(root, "hel");
        insert(root, "help");
        insert(root, "helps");
        insert(root, "helping");
        int comp = printAutoSuggestions(root, "hel");

        if (comp == -1)
                cout << "No other strings found with this prefix\n";

        else if (comp == 0)
                cout << "No string found with this prefix\n";

        return 0;
}
```

SCREENSHOTS:

dsa.cpp

```cpp
// C++ program to demonstrate auto-complete feature
// using Trie data structure.
#include<bits/stdc++.h>
using namespace std;

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isWordEnd is true if the node represents
    // end of a word
    bool isWordEnd;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;
    pNode->isWordEnd = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie. If the
// key is prefix of trie node, just marks leaf node
```

dsa.cpp

```cpp
// key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const string key)
{
    struct TrieNode *pCrawl = root;

    for (int level = 0; level < key.length(); level++)
    {
        int index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isWordEnd = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const string key)
{
    int length = key.length();
    struct TrieNode *pCrawl = root;
    for (int level = 0; level < length; level++)
    {
        int index = CHAR_TO_INDEX(key[level]);

        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isWordEnd);
}

```

dsa.cpp

```cpp

// Returns 0 if current node has a child
// If all children are NULL, return 1.
bool isLastNode(struct TrieNode* root)
{
    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root->children[i])
            return 0;
    return 1;
}

// Recursive function to print auto-suggestions for given
// node.
void suggestionsRec(struct TrieNode* root, string currPrefix)
{
    // found a string in Trie with the given prefix
    if (root->isWordEnd)
    {
        cout << currPrefix;
        cout << endl;
    }

    // All children struct node pointers are NULL
    if (isLastNode(root))
        return;

    for (int i = 0; i < ALPHABET_SIZE; i++)
    {
        if (root->children[i])
        {
            // append current character to currPrefix string
            currPrefix.push_back(97 + i);

            // recur over the rest
            suggestionsRec(root->children[i], currPrefix);
        }
```

dsa.cpp

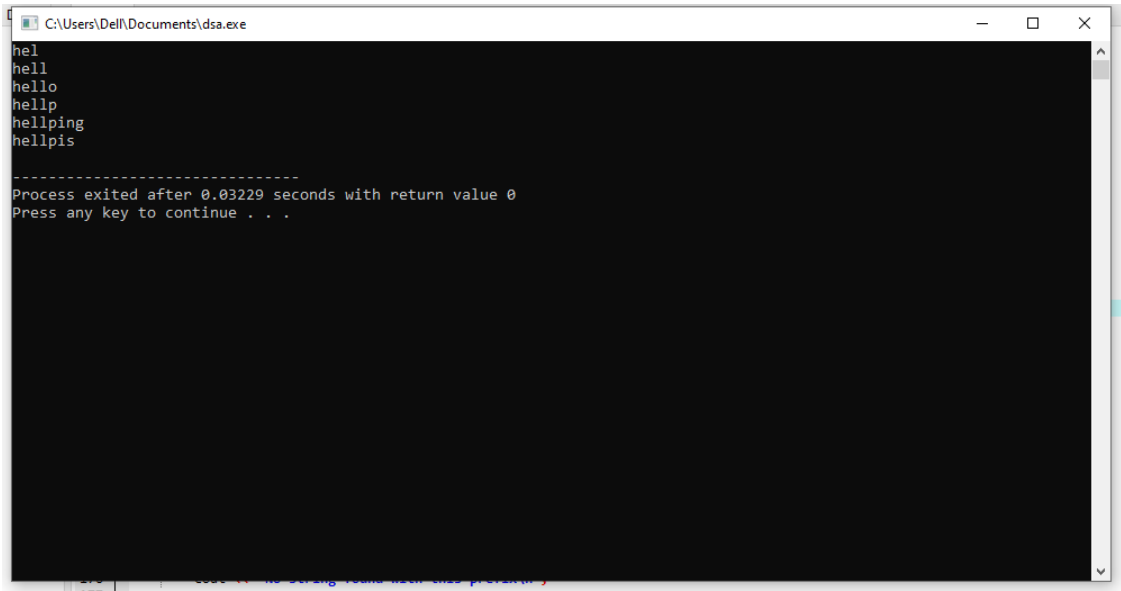
```
106             }
107         }
108     }
109
110     // print suggestions for given query prefix.
111     int printAutoSuggestions(TrieNode* root, const string query)
112     {
113         struct TrieNode* pCrawl = root;
114
115         // Check if prefix is present and find the
116         // the node (of last level) with last character
117         // of given string.
118         int level;
119         int n = query.length();
120         for (level = 0; level < n; level++)
121         {
122             int index = CHAR_TO_INDEX(query[level]);
123
124             // no string in the Trie has this prefix
125             if (!pCrawl->children[index])
126                 return 0;
127
128             pCrawl = pCrawl->children[index];
129         }
130
131         // If prefix is present as a word.
132         bool isWord = (pCrawl->isWordEnd == true);
133
134         // If prefix is last node of tree (has no
135         // children)
136         bool isLast = isLastNode(pCrawl);
137
138         // If prefix is present as a word, but
139         // there is no subtree below the last
140         // matching node.
141         if (isWord && isLast)
```

dsa.cpp

```
141         if (isWord && isLast)
142         {
143             cout << query << endl;
144             return -1;
145         }
146
147         // If there are are nodes below last
148         // matching character.
149         if (!isLast)
150         {
151             string prefix = query;
152             suggestionsRec(pCrawl, prefix);
153             return 1;
154         }
155     }
156
157     // Driver Code
158     int main()
159     {
160         struct TrieNode* root = getNode();
161         insert(root, "hello");
162         insert(root, "dog");
163         insert(root, "hell");
164         insert(root, "cat");
165         insert(root, "a");
166         insert(root, "hel");
167         insert(root, "help");
168         insert(root, "helps");
169         insert(root, "helping");
170         int comp = printAutoSuggestions(root, "hel");
171
172         if (comp == -1)
173             cout << "No other strings found with this prefix\n";
174
175         else if (comp == 0)
176             cout << "No string found with this prefix\n";
```

```
157     // Driver Code
158     int main()
159     {
160         struct TrieNode* root = getNode();
161         insert(root, "hello");
162         insert(root, "dog");
163         insert(root, "hell");
164         insert(root, "cat");
165         insert(root, "a");
166         insert(root, "hel");
167         insert(root, "help");
168         insert(root, "helps");
169         insert(root, "helping");
170         int comp = printAutoSuggestions(root, "hel");
171
172         if (comp == -1)
173             cout << "No other strings found with this prefix\n";
174
175         else if (comp == 0)
176             cout << "No string found with this prefix\n";
177
178         return 0;
179     }
```

OUTPUTS:




```
177
178     return 0;
```