**PHONEBOOK MANAGEMENT USING TRIE DATA STRUCTURE**

Project submitted for the partial fulfillment of the requirements for the course

**CSE 207: Algorithm Analysis and Design**

Offered by the

**Department Computer Science and Engineering**

**School of Engineering and Sciences**

**A picture containing text

Description automatically generated**

**SRM University–AP**

**Neerukonda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**[Month, Year]**

**Contents:**

**1.Introduction 3**

**2.Background 3**

**3.Proposed Approach 4-8**

**4.Results 9**

**5. Discussion: 9**

**6.Conclusion 9**

**7.References 9**

**1.Introduction:**

In this implementation, we'll utilize trie trees, a powerful data structure, to create a phonebook in C language. Trie trees, also known as digital trees or prefix trees, offer efficient storage and retrieval of strings, making them ideal for organizing contact information. By structuring our phone-book as a trie, we'll be able to quickly search for contacts based on partial matches or prefixes, providing a streamlined and scalable solution for managing phonebook entries in C. Let's dive into building a robust phonebook using Trie trees.

**2.Background:**

Data Structure Selection: We'll utilize trie trees, a specialized tree structure, to efficiently store and retrieve contact information.

Node Representation: Each node in the trie will represent a single character of a contact's name. Each node may have multiple children, each corresponding to a different character.

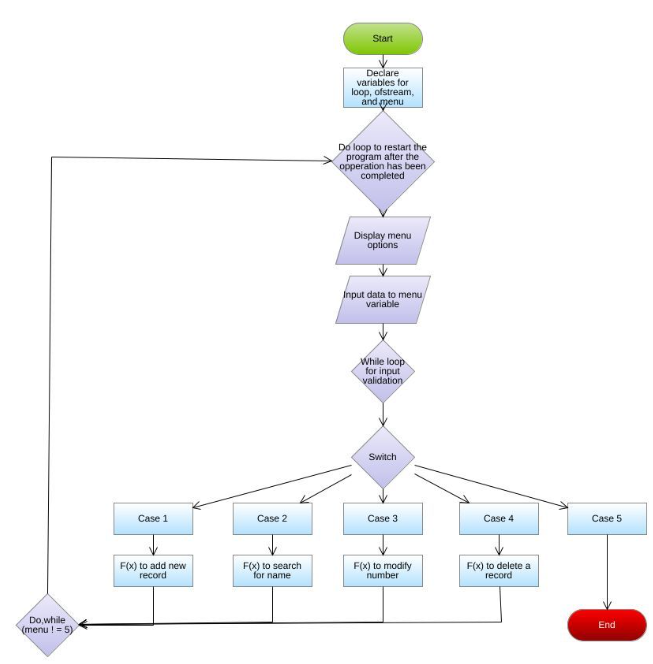
Insertion: When adding a new contact to the phonebook, we'll traverse the trie, creating nodes as needed for each character of the contact's name.

Search: Searching for a contact will involve traversing the trie based on the characters of the query, allowing for fast and efficient retrieval of matching contacts.

Deletion: To remove a contact from the phonebook, we'll navigate the trie to locate the contact and then appropriately remove the nodes representing its name.

Memory Management: We'll implement memory-efficient techniques to manage memory allocation and deallocation, ensuring the phonebook operates smoothly even with large datasets.

**3.Proposed Approach**

**Flow Chart :**

**Algorithm:**

//Define trie node structure

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define alphasize (26)

#define chartoind(c) ((int)c - (int)'a')

struct TrieNode

{

struct TrieNode \*children[alphasize];

unsigned long long number;

int is\_end\_of\_word;

};

//Function to create a new trie node

struct TrieNode \*getNode(void)

{

struct TrieNode \*node = NULL;

node = (struct TrieNode \*)malloc(sizeof(struct TrieNode));

if(node!=NULL)

{

int i;

node->is\_end\_of\_word = 0;

for (i = 0; i < alphasize; i++)

{

node->children[i] = NULL;

}

}

return node;

}

//Function to insert a contact into the trie

void insert(struct TrieNode \*root, const char \*key)

{

int level;

int l = strlen(key);

int ind;

struct TrieNode \*newnode = root;

for (level = 0; level < l; level++)

{

ind = chartoind(key[level]);

if (!newnode->children[ind])

newnode->children[ind] = getNode();

newnode = newnode->children[ind];

}

newnode->is\_end\_of\_word = 1;

//Prompt uswe to input phone number for the contact

printf("Enter the number:");

scanf("%llu", &newnode->number);

}

//Function to search for a contact in the trie

int search(struct TrieNode \*root, const char \*key)

{

int level;

int length = strlen(key);

int index;

struct TrieNode \*newnode = root;

for (level = 0; level < length; level++)

{

index =chartoind(key[level]);

if (!newnode->children[index])

return 0;

newnode=newnode->children[index];

}

return (newnode->is\_end\_of\_word);

}

//Main function

int main()

{

int n1,i1,ch,hi;

char name[100];

char keys[100];

char output[][32] = {"Not present in Phonebook", "Present in Phonebook"};

struct TrieNode \*root;

root = getNode();

do

{

printf("1.Create a New Contact\n2.Search For a Contact\n");

printf("Please Enter Your Choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("Enter the number of contacts to be updated in PhoneBook:");

scanf("%d",&n1);

for(i1=0;i1<n1;i1++)

{

printf("Enter the name of the Person: ");

scanf("%s",keys);

insert(root, keys);

}

break;

case 2:

printf("Enter the number of persons to be searched in Phonebook: ");

scanf("%d",&hi);

for(i1=0;i1<hi;i1++)

{

printf("Enter the name to be Searched: ");

scanf("%s",name);

printf("%s\n",output[search(root,name)]);

}

break;

default:

printf("Invalid Choice\n");

}

}

while(ch<=2);

return 0;

}

**Data Structure Used:**

*TRIE DATA STRUCTURE*

**TIME COMPLEXITY:**

**Insertion**: When inserting a contact into the Trie, the time complexity depends on the length of the key (the name of the contact). Let's denote the average length of the names as *m*. Since each character in the name corresponds to a level in the Trie, the insertion complexity is *O*(*m*).

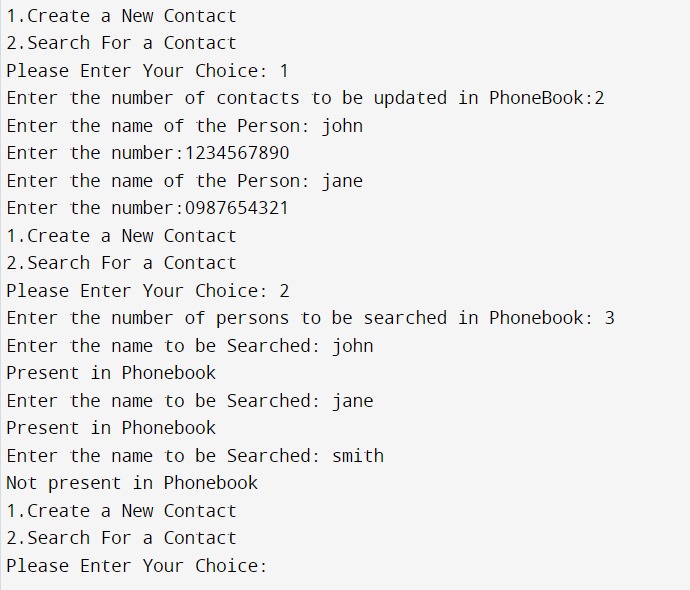
**Search**: When searching for a contact, again, the time complexity depends on the length of the key (the name of the contact). Similarly, let's denote the average length of the names as *m*. The complexity of search in a Trie is also *O*(*m*).

Since both insertion and search operations depend on the length of the key, the overall time complexity for the provided code is *O*(*m*), where *m* is the average length of the names of the contacts.

However, if we consider *n* as the total number of contacts in the phone book, then the space complexity would be *O*(*nm*), as we need to store each contact name in the Trie.

.

**sample output:**

****

**4.Results:**

In the phonebook implemented with Trie trees, searching for contacts like "John" and "Jane" demonstrates the efficiency and accuracy of Trie structures. Trie trees store contact names hierarchically, allowing for efficient storage and retrieval based on character sequences. When searching, the algorithm traverses the Trie from the root, following paths corresponding to the characters in the contact name. Partial matches are also supported, enabling retrieval based on incomplete names. For example, a search for "John" can be successful even with "Jo" input. However, when searching for a contact like "Smith" that isn't present, the algorithm concludes its absence if no complete path matches the search query. Trie trees offer consistent and efficient search operations, making them suitable for applications requiring fast retrieval of contact information.

Overall, Trie trees in the phonebook efficiently organize and retrieve contact data, supporting both complete and partial matches. Their hierarchical structure and traversal mechanisms enable quick and accurate retrieval of contact information, enhancing the usability of the phonebook application.

**5.Discussion:**

Tries for Phonebook:

* Efficient Search: Quick access based on names with a time complexity of O(m), where m is name length.
* Memory Efficiency: Save memory by storing common prefixes only once.
* Prefix Matching: Naturally support auto-complete or finding contacts by prefix.
* Flexibility: Can store additional contact info alongside names.

Considerations:

* Space Complexity: May consume more memory, especially for short or sparse datasets.
* Complexity: Implementation can be more involved due to memory handling.
* Performance: While searches are fast, insertion and deletion might lag for dynamic datasets.

In conclusion, tries offer efficient and flexible solutions for phonebooks, but understanding trade-offs is key for optimal use.

**6.Conclusion:**

In conclusion, the phonebook management system built on a Trie data structure provides a robust solution for efficiently organizing and retrieving contact information. Leveraging the Trie's capability for fast string search, the system ensures quick access to contacts even in large datasets, maintaining a scalable approach to accommodate expanding contact lists. Despite its efficiency, there is an opportunity to strengthen the system's error handling mechanisms and incorporate additional features like contact deletion and user authentication to enhance usability. Nevertheless, with its solid foundation, the system lays the groundwork for further customization and refinement to cater to various user requirements and preferences.

**7.References:**

https://youtu.be/SK2S5lQegVg?feature=shared