MINI PROJECT
On

Contacts Management using Trie

In
Data Structure & Algorithms

BACHELOR OF TECHNOLOGY

IN
Artificial Intelligence and Machine Learning

SUBMITTED BY

Rhea Chainani
(PRN - 22070126086)

Under the Supervision of

Ms. Hema Karande,

Assistant Professor

SYMBIOSIS INSTITUTE OF TECHNOLOGY, PUNE – 412115

(A CONSTITUENT OF SYMBIOSIS INTERNATIONAL (DEEMED UNIVERSITY))

2023 - 24

Problem Statement

Design and implement a contacts management system using the trie data structure in C. The system aims to efficiently store, search for, update and delete contact information while providing a user-friendly interface with case-insensitive search capabilities for an improved user experience.

Motivation

The motivation behind this project stems from the need to simplify and enhance the management of contact information in a digital era. With the increasing reliance on digital devices, there is a compelling motivation to develop a streamlined and efficient contacts management system that caters to the user's need for quick access and organization of their contact details. This project is driven by the desire to offer a practical solution for users to efficiently store, search for, update and delete their contact information.

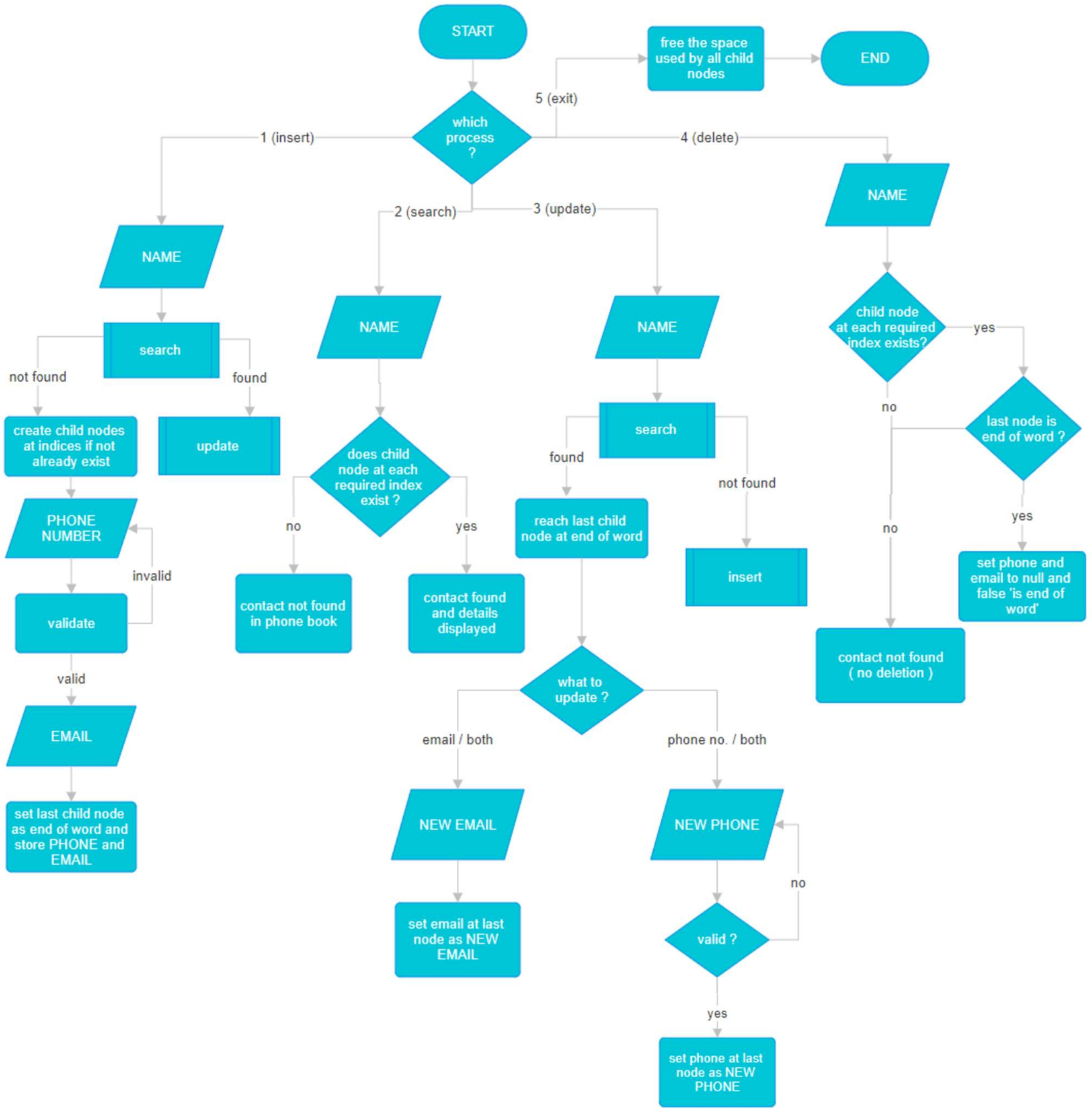
Objectives

- Efficient Contact Management: Create a system that efficiently stores a large number of contact details, including names, phone numbers, and email addresses, while minimizing memory usage.
- User-Friendly Interface: Develop an intuitive and user-friendly interface that simplifies the addition, search, update, and deletion of contact information, catering to users with varying technical proficiency.
- Case-Insensitive Search: Implement a case-insensitive search feature, allowing users to find contacts regardless of letter casing in their search queries, enhancing the user experience.
- Streamlined Contact Operations: Enable users to quickly access, update, and delete contact information, offering a comprehensive solution for effective digital contact management.

Software Used

Visual Studio Code

Methodology of Implementation


START
which process ?
1 (insert)
2 (search)
3 (update)
4 (delete)
5 (exit)
free the space used by all child nodes
END
NAME
search
not found
found
create child nodes at indices if not already exist
update
PHONE NUMBER
invalid
validate
valid
EMAIL
set last child node as end of word and store PHONE and EMAIL
NAME
does child node at each required index exist ?
no
yes
contact not found in phone book
contact found and details displayed
NAME
search
found
not found
reach last child node at end of word
insert
what to update ?
email / both
phone no. / both
NEW EMAIL
set email at last node as NEW EMAIL
NEW PHONE
valid ?
no
yes
set phone at last node as NEW PHONE
NAME
child node at each required index exists?
yes
no
last node is end of word ?
no
yes
set phone and email to null and false 'is end of word'
contact not found (no deletion)

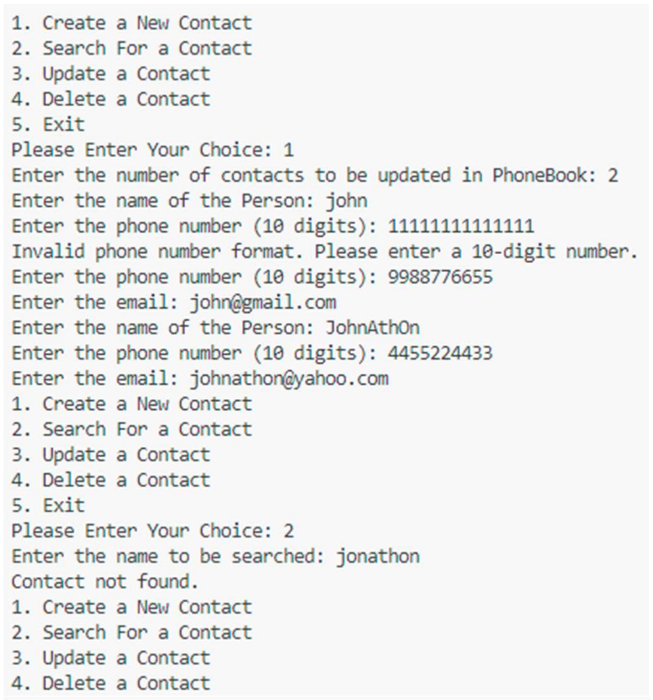
Executable Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define alphasize (26)
#define chartoind(c) ((int)(tolower(c)) - (int)'a')

struct TrieNode
{
    struct TrieNode *children[alphasize];
    char number[11];
    char email[100];
    int is_end_of_word;
};

struct TrieNode *getNode(void)
{
    struct TrieNode *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;
        }
        node->number[0] = '\0';
        node->email[0] = '\0';
    }
    return node;
}

int validatePhoneNumber(const char *phoneNumber)
{
    int length = strlen(phoneNumber);
    if (length != 10)
        return 0;
    for (int i = 0; i < length; i++)
    {
        if (!isdigit(phoneNumber[i]))
            return 0;
    }
    return 1;
}

void insert(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int ind;
    struct TrieNode *newnode = root;
    for (level = 0; level < length; level++)
    {
        ind = chartoind(key[level]);
        if (!newnode->children[ind])
            newnode->children[ind] = getNode();
        newnode = newnode->children[ind];
    }
    newnode->is_end_of_word = 1;

    while (1)
    {
        printf("Enter the phone number (10 digits): ");
        scanf("%s", newnode->number);

        if (validatePhoneNumber(newnode->number))
        {
            break;
        }
        else
        {
            printf("Invalid phone number format. Please enter a 10-digit 
number.\n");
        }
    }

    printf("Enter the email: ");
    scanf("%s", newnode->email);
}

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++)
```

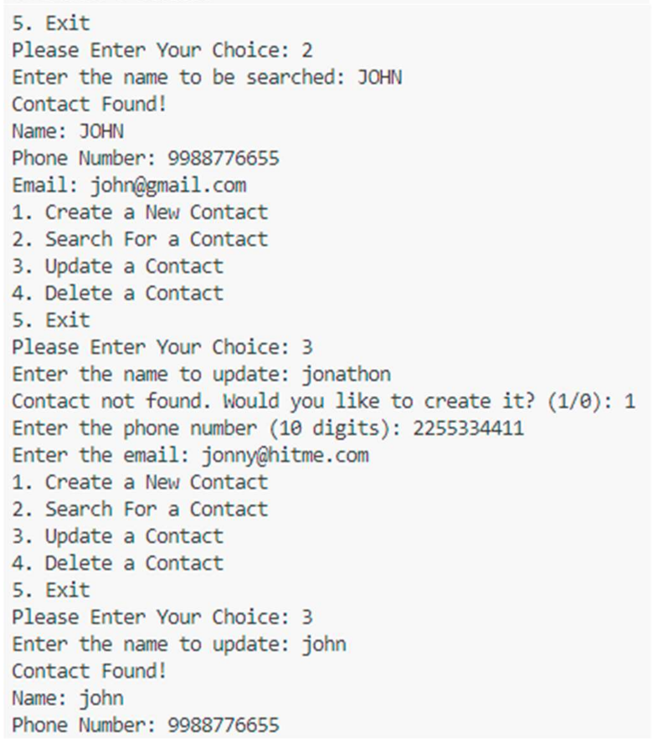
```
        {
            index = chartoind(key[level]);
            if (!newnode->children[index])
                return 0;
            newnode = newnode->children[index];
        }
        if (newnode->is_end_of_word)
        {
            printf("Contact Found!\n");
            printf("Name: %s\n", key);
            printf("Phone Number: %s\n", newnode->number);
            printf("Email: %s\n", newnode->email);
            return 1;
        }
        return 0;
    }

    void update(struct TrieNode *root, const char *key)
    {
        int level;
        int length = strlen(key);
        int ind;
        struct TrieNode *newnode = root;
        for (level = 0; level < length; level++)
        {
            ind = chartoind(key[level]);
            newnode = newnode->children[ind];
        }
        printf("What would you like to update?\n");
        printf("1. Phone Number\n2. Email\n3. Both\n");
        int updateOption;
        scanf("%d", &updateOption);

        if (updateOption == 1 || updateOption == 3)
        {
            printf("Enter the new phone number (10 digits): ");
            while (1)
            {
                scanf("%s", newnode->number);

                if (validatePhoneNumber(newnode->number))
                {
                    break;
                }
                else
                {
                    printf("Invalid phone number format. Please enter a 10-digit 
number.\n");
                }
            }
        }

        if (updateOption == 2 || updateOption == 3)
        {
            printf("Enter the new email: ");
            scanf("%s", newnode->email);
        }

        printf("Contact updated!\n");
    }

    void deleteContact(struct TrieNode *root, const char *key)
    {
        int level;
        int length = strlen(key);
        int ind;
        struct TrieNode *newnode = root;
        struct TrieNode *parentNode = root;
        int parentIndex = 0;
        for (level = 0; level < length; level++)
        {
            ind = chartoind(key[level]);
            if (!newnode->children[ind])
            {
                printf("Contact not found. Deletion failed.\n");
                return;
            }
            parentIndex = ind;
            parentNode = newnode;
            newnode = newnode->children[ind];
        }
        if (newnode->is_end_of_word)
        {
            newnode->is_end_of_word = 0;
            newnode->number[0] = '\0';
            newnode->email[0] = '\0';
            // Check if the parent node can be removed as well
            int hasChildren = 0;
            for (ind = 0; ind < alphasize; ind++)
            {
```

```
            if (parentNode->children[ind] != NULL)
            {
                hasChildren = 1;
                break;
            }
        }
        if (!hasChildren)
        {
            // The parent node has no other children, it can be removed
            free(parentNode);
        }
        printf("Contact deleted!\n");
    }
    else
    {
        printf("Contact not found. Deletion failed.\n");
    }
}

void deallocate(struct TrieNode *node)
{
    if (node == NULL)
        return;
    for (int i = 0; i < alphasize; i++)
    {
        if (node->children[i])
        {
            deallocate(node->children[i]);
        }
    }
    free(node);
}

int main()
{
    int ch, n;
    char name[100];
    char keys[100];
    char output[][32] = {"Not present in Phonebook", "Present in Phonebook"};
    struct TrieNode *root = getNode();
    do
    {
        printf("1. Create a New Contact\n2. Search For a Contact\n3. Update a
Contact\n4. Delete a Contact\n5. Exit\n");
        printf("Please Enter Your Choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter the number of contacts to be added in the PhoneBook:
    ");
            scanf("%d", &n);
            for (int i = 0; i < n; i++)
            {
                printf("Enter the name of the Person: ");
                scanf("%s", keys);
                if (!search(root, keys))
                {
                    insert(root, keys);
                }
                else
                {
                    printf("Would you like to update it ? (1/0): ");
                    int choice;
                    scanf("%d", &choice);
                    if (choice)
                    {
                        update(root, keys);
                    }
                }
            }
            break;
        case 2:
            printf("Enter the name to be searched: ");
            scanf("%s", name);
            if (!search(root, name))
            {
                printf("Contact not found.\n");
            }
            break;
        case 3:
            printf("Enter the name to update: ");
            scanf("%s", name);
            if (search(root, name))
            {
                update(root, name);
            }
            else
            {
                printf("Contact not found. Would you like to create it? (1/0):
");
                int create;
                scanf("%d", &create);
                if (create == 1)
                {
                    insert(root, name);
                }
            }
            break;
        case 4:
            printf("Enter the name to delete: ");
            scanf("%s", name);
            deleteContact(root, name);
            break;
        case 5:
            deallocate(root);
            printf("Exiting program...");
            break;
        default:
            printf("Invalid Choice\n");
        }
    } while (ch != 5);
    return 0;
}
```

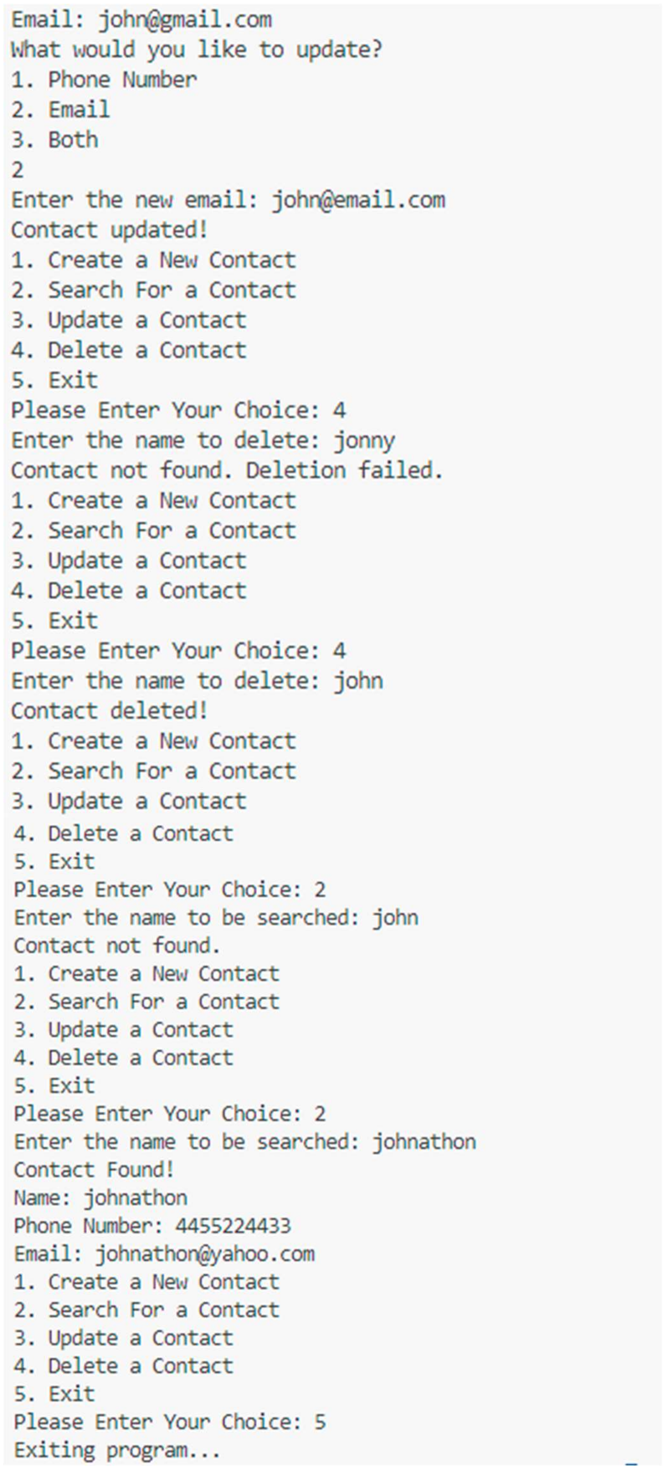
Result Analysis with Output Screen Shot



```
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 1
Enter the number of contacts to be updated in PhoneBook: 2
Enter the name of the Person: john
Enter the phone number (10 digits): 11111111111111
Invalid phone number format. Please enter a 10-digit number.
Enter the phone number (10 digits): 9988776655
Enter the email: john@gmail.com
Enter the name of the Person: JohnAthOn
Enter the phone number (10 digits): 4455224433
Enter the email: johnathon@yahoo.com
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 2
Enter the name to be searched: jonathon
Contact not found.
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
```



```
5. Exit
Please Enter Your Choice: 2
Enter the name to be searched: JOHN
Contact Found!
Name: JOHN
Phone Number: 9988776655
Email: john@gmail.com
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 3
Enter the name to update: jonathon
Contact not found. Would you like to create it? (1/0): 1
Enter the phone number (10 digits): 2255334411
Enter the email: jonny@hitme.com
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 3
Enter the name to update: john
Contact Found!
Name: john
Phone Number: 9988776655
```



```
Email: john@gmail.com
What would you like to update?
1. Phone Number
2. Email
3. Both
2
Enter the new email: john@email.com
Contact updated!
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 4
Enter the name to delete: jonny
Contact not found. Deletion failed.
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 4
Enter the name to delete: john
Contact deleted!
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 2
Enter the name to be searched: john
Contact not found.
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 2
Enter the name to be searched: johnathon
Contact Found!
Name: johnathon
Phone Number: 4455224433
Email: johnathon@yahoo.com
1. Create a New Contact
2. Search For a Contact
3. Update a Contact
4. Delete a Contact
5. Exit
Please Enter Your Choice: 5
Exiting program...
```

1. The system is case - insensitive.
2. It checks the validity of a phone number before adding it in contact details.
3. Before adding a contact it checks if it already exists and in that case gives the user an option to update the existing contact details instead.
4. Before updating a contact it checks whether the contact exists in phone book and if not, gives an option to create new contact instead.
5. Deleting a contact does not have any impact on other contacts with the same prefix.

Symbol Table and AVL Tree

A symbol table is a data structure that associates values (or symbols) with keys and supports two primary operations: insertion of a new key-value pair and retrieval of the value associated with a given key. Trie, in essence, acts as a symbol table:

1. Key-Value Association: A symbol table associates keys (such as names) with values (contact information like phone numbers and emails). In the context of a Trie, keys are the strings representing contact names, and values are the associated contact details (phone numbers and emails). When we search for a name in the Trie, it effectively retrieves the associated contact information (the value).
2. Efficient Retrieval: Symbol tables, including Tries, are designed for efficient retrieval of values based on keys. Tries excel at searching for keys character by character. In this contact management system, Trie allows us to quickly find contact information for a given name (key) by traversing the Trie structure.
3. Dynamic Data Structure: Symbol tables, including Tries, can dynamically grow and shrink as we add or remove key-value pairs. This is essential for managing a changing set of contacts in our system.
4. Support for Key-Based Operations: Symbol tables provide operations based on keys, such as inserting a new key-value pair, searching for a key, updating the value associated with a key, and deleting a key-value pair. Our contact management system uses these operations for adding, searching, updating, and deleting contacts.

So, in the implementation of Trie for a contact management system, we are essentially creating a symbol table tailored to our specific use case, where keys are contact names, and values are contact information.

An AVL Tree is a self-balancing binary search tree that maintains its balance by ensuring that the heights of its two child subtrees differ by at most one. While an AVL Tree is a powerful data structure, we are not using it in our contact management project because:

1. The primary data, contact names, are strings without inherent sorted order.
2. Tries are better suited for prefix-based searches, allowing efficient partial name lookups.
3. AVL Trees are more complex to implement and maintain.
4. Tries offer space efficiency by sharing common prefixes.

Learning Outcome

1. Data Structures Proficiency: Acquiring a strong grasp of Trie data structures and their application in efficiently managing contact information.
2. Input Validation and Error Handling: Learning to validate user input, particularly in the context of phone numbers, and handling scenarios where contacts already exist, improving error-handling skills.
3. Real-World Application: Applying theoretical knowledge to a practical context, demonstrating the ability to create functional, user-friendly software for everyday use.

References

[1] https://github.com/kakshak07/PhoneBook/blob/main/PhoneBook_Using_Trie_Trees.c
[2] https://youtu.be/YG6iX28hmd0?si=P85rfJoJGIV6IjZ6
[3] https://www.geeksforgeeks.org/trie-insert-and-search/