SRM INSTITUTE OF SCIENCE & TECHNOLOGY

COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF SCHOOL OF COMPUTING

MINI PROJECT REPORT

ODD Semester, 2023-2024

Lab code & Sub Name : 21CSC201J & Data Structures and Algorithms

Year & Semester : III

Project Title : Contact Management using Binary Search Tree

Lab Supervisor : **Dr. V. Pandimurugan**

Team Members :

1. Mathesh M. (Reg.No:RA2211030010053)

2.Austin Powers Newton (Reg.No:RA2211030010057)

3. Yogesh K. R. (Reg.No:RA2211030010064)

4. Saravana (Reg.No:RA2211030010018)

TABLE OF CONTENTS

CONTACT MANAGEMENT USING BINARY SEARCH

PROBLEM DEFINITION:

Contact management is a common application where a binary search tree (BST) can be utilized to efficiently store and retrieve contact information such as names and phone numbers.

PROBLEM STATEMENT WITH EXPLANATION:

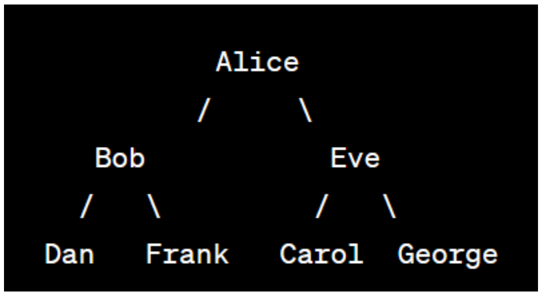
To develop a Contact Management System using a Binary Search Tree (BST) to efficiently store and manage contact information. The system should provide functionality for adding, searching, updating, and deleting contacts. Contacts are represented by their names and associated details, such as phone numbers, email addresses, and other relevant information. The goal is to create an efficient and organized system for managing contacts.

Example:

Imagine you have a contact management system implemented using a Binary Search Tree (BST). You want to find a specific contact, "Alice," in your contact list. Here's how the system performs the search operation:

1. Start at the root of the BST.
2. Compare "Alice" with the name of the contact stored at the current node.
3. Since "Alice" is lexicographically smaller than the current contact's name, move to the left child node.
4. Repeat steps 2 and 3 until you find "Alice" or reach a leaf node.
5. In this case, you find "Alice" as a contact in the tree.

This example demonstrates how a BST efficiently narrows down the search space, allowing you to find the desired contact, "Alice," quickly by exploiting the hierarchical structure and alphabetical ordering of the data.



```
          Alice
         /     \
    Bob            Eve
   /   \          /   \
 Dan    Frank   Carol  George
```

DESIGN TECHNIQUES USED:

Several design techniques and principles are applied to create an efficient and organized solution towards the problem. Some of the key design techniques used in this scenario include:

1. Data Structure Selection: The Binary Search Tree (BST) is chosen to enable efficient contact storage and retrieval.
2. Efficient Search and Retrieval: BSTs optimize contact search with logarithmic time complexity.
3. Data Organization: Contacts are automatically sorted alphabetically within the BST.
4. Dynamic Data Management: The system accommodates dynamic addition, update, and deletion of contacts while maintaining the tree structure.
5. Space Efficiency: Memory usage is optimized as the BST dynamically allocates memory for contacts.
6. Scalability: The system scales to handle a growing number of contacts without sacrificing performance.

7. Tree Balancing (Consideration): Balancing techniques may be considered to maintain tree efficiency in all scenarios.
8. Data Validation (Consideration): Data validation is critical to ensure accuracy and consistency in contact information.

ALGORITHM FOR THE PROBLEM:

Step 1: Define the Contact Structure

- Create a structure named `Contact` to represent a contact with fields for name, phone number, and pointers to left and right child nodes.

Step 2: Create a Contact

- Implement a function `createContact` that dynamically allocates memory for a new contact, initializes its fields with the provided name and phone number, and sets the child pointers to NULL. Return the newly created contact.

Step 3: Insert Contacts into the Binary Search Tree

- Implement a function `insertContact` to insert a contact into the binary search tree. This function recursively compares the name of the contact to be inserted with the names of contacts in the tree and inserts it in the appropriate position based on lexicographic order.

Step 4: Search for a Contact by Name

- Implement a function `searchContact` that allows searching for a contact by name in the binary search tree. The function recursively traverses the tree, comparing the search name with the names of contacts in the tree until it finds a match or determines that the contact does not exist in the tree.

Step 5: In-Order Traversal and Main Function

- Implement the `inOrderTraversal` function for an in-order traversal of the binary search tree, which prints the contacts in alphabetical order.

- In the `main` function, create the root of the binary search tree, insert several contacts, perform an in-order traversal to display the contacts in alphabetical order, and demonstrate searching for a specific contact by name.

EXPLANATION OF ALGORITM

This algorithm uses a Binary Search Tree (BST) to store contacts. It allows you to insert contacts in alphabetical order and efficiently search for specific contacts. The BST structure automatically organizes contacts for easy retrieval.

Example:

1. Create a contact for "Alice."

2. Insert "Bob" to the right of "Alice" in the tree.

3. Insert "Eve" to the right of "Bob."

4. Search for "Bob," which returns the contact information for "Bob" (Phone: 555-5678).

5. Perform an in-order traversal to display contacts in alphabetical order:

TEMPORARY OUTPUT:

```
Contacts in alphabetical order:

Name: Alice, Phone: 555-1234

Name: Bob, Phone: 555-5678

Name: Eve, Phone: 555-9876
```

COMPLEXITY ANALYSIS:

1. Insertion Complexity:

   - Average time complexity for inserting a contact is O(log N), but in the worst-case scenario, it can become O(N) if the tree becomes highly unbalanced.

2. Search Complexity:

   - Average time complexity for searching for a contact by name is O(log N), with the potential to degrade to O(N) in the worst case if the tree is unbalanced, but can be mitigated with balancing techniques.

3. Space Complexity:

   - The space complexity is O(N) because each contact consumes memory for their name, phone number, and two pointers, in addition to tree overhead.

4. In-Order Traversal Complexity:

   - In-order traversal to display contacts in alphabetical order has a time complexity of O(N), as it visits and prints all contacts once, with time proportional to the number of contacts.

PROGRAM & OUTPUT:

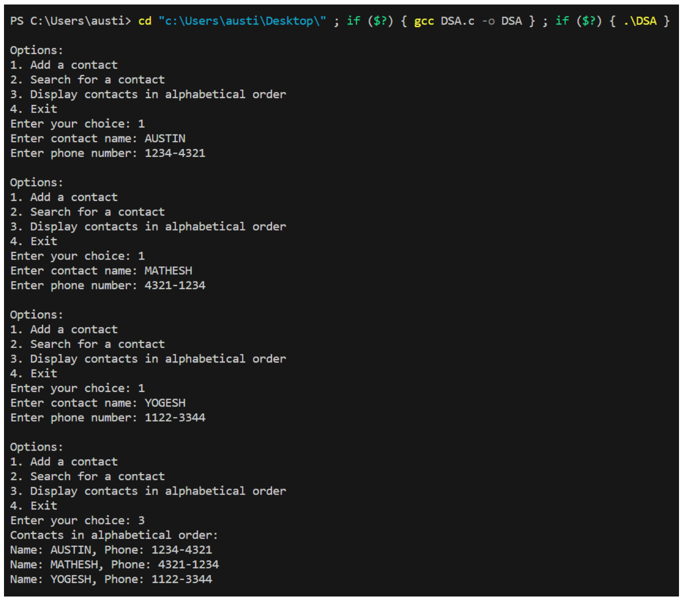
PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>
```

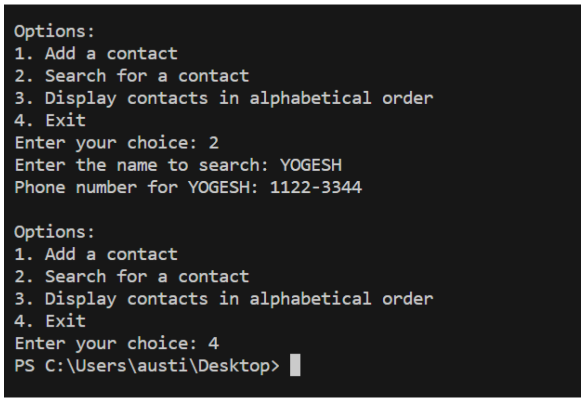
```
// Structure to represent a contact
struct Contact {
    char name[100];
    char phone_number[15];
    struct Contact* left;
    struct Contact* right;
};

// Function to create a new contact
struct Contact* createContact(char name[], char phone_number[]) {
    struct Contact* newContact = (struct Contact*)malloc(sizeof(struct
Contact));
    if (newContact == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(1);
    }
    strcpy(newContact->name, name);
    strcpy(newContact->phone_number, phone_number);
    newContact->left = NULL;
    newContact->right = NULL;
    return newContact;
}
```

```
// Function to insert a contact into the binary search tree
struct Contact* insertContact(struct Contact* root, char name[], char
phone_number[]) {
    if (root == NULL) {
        return createContact(name, phone_number);
    }

    if (strcmp(name, root->name) < 0) {
        root->left = insertContact(root->left, name, phone_number);
    } else {
        root->right = insertContact(root->right, name, phone_number);
    }

    return root;
}

// Function to search for a contact by name
struct Contact* searchContact(struct Contact* root, char name[]) {
    if (root == NULL || strcmp(name, root->name) == 0) {
        return root;
    }

    if (strcmp(name, root->name) < 0) {
        return searchContact(root->left, name);
    }

    return searchContact(root->right, name);
}

// In-order traversal to print contacts in alphabetical order
void inOrderTraversal(struct Contact* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("Name: %s, Phone: %s\n", root->name, root-
>phone_number);
        inOrderTraversal(root->right);
    }
}

// Function to free memory used by the BST
void freeBST(struct Contact* root) {
    if (root == NULL) {
        return;
    }
    freeBST(root->left);
    freeBST(root->right);
    free(root);
}

int main() {
    struct Contact* root = NULL;
    char input[100];

    while (1) {
        printf("\nOptions:\n");
        printf("1. Add a contact\n");
        printf("2. Search for a contact\n");
        printf("3. Display contacts in alphabetical order\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        fgets(input, sizeof(input), stdin);
        int choice = atoi(input);

        if (choice == 1) {
            char name[100];
            char phone[15];
            printf("Enter contact name: ");
            fgets(name, sizeof(name), stdin);
            name[strcspn(name, "\n")] = '\0';  // Remove newline character
            printf("Enter phone number: ");
            fgets(phone, sizeof(phone), stdin);
            phone[strcspn(phone, "\n")] = '\0';  // Remove newline character
            root = insertContact(root, name, phone);
        } else if (choice == 2) {
            char searchName[100];
            printf("Enter the name to search: ");
            fgets(searchName, sizeof(searchName), stdin);
            searchName[strcspn(searchName, "\n")] = '\0';  // Remove
newline character
            struct Contact* result = searchContact(root, searchName);
            if (result != NULL) {
                printf("Phone number for %s: %s\n", result->name, result-
>phone_number);
            } else {
                printf("%s not found.\n", searchName);
            }
        } else if (choice == 3) {
            printf("Contacts in alphabetical order:\n");
            inOrderTraversal(root);
        } else if (choice == 4) {
            break;  // Exit the loop
        }
    }
    // Free the memory used by the BST
    freeBST(root);

    return 0;
}
```

OUTPUT:



```
PS C:\Users\austi> cd "c:\Users\austi\Desktop\" ; if ($?) { gcc DSA.c -o DSA } ; if ($?) { .\DSA }

Options:
1. Add a contact
2. Search for a contact
3. Display contacts in alphabetical order
4. Exit
Enter your choice: 1
Enter contact name: AUSTIN
Enter phone number: 1234-4321

Options:
1. Add a contact
2. Search for a contact
3. Display contacts in alphabetical order
4. Exit
Enter your choice: 1
Enter contact name: MATHESH
Enter phone number: 4321-1234

Options:
1. Add a contact
2. Search for a contact
3. Display contacts in alphabetical order
4. Exit
Enter your choice: 1
Enter contact name: YOGESH
Enter phone number: 1122-3344

Options:
1. Add a contact
2. Search for a contact
3. Display contacts in alphabetical order
4. Exit
Enter your choice: 3
Contacts in alphabetical order:
Name: AUSTIN, Phone: 1234-4321
Name: MATHESH, Phone: 4321-1234
Name: YOGESH, Phone: 1122-3344
```



```
Options:
1. Add a contact
2. Search for a contact
3. Display contacts in alphabetical order
4. Exit
Enter your choice: 2
Enter the name to search: YOGESH
Phone number for YOGESH: 1122-3344

Options:
1. Add a contact
2. Search for a contact
3. Display contacts in alphabetical order
4. Exit
Enter your choice: 4
PS C:\Users\austi\Desktop>
```

CONCLUSION

In conclusion, the code presented provides a solid starting point for a contact management system using a Binary Search Tree, offering efficient organization and retrieval of contact information. While it's effective for educational purposes and simple applications, real-world scenarios would demand additional features and a more user-friendly interface. With further enhancements, it has the potential to serve as a practical component in address book applications, business contact management systems, or similar applications requiring organized and scalable contact storage and retrieval.

REFERENCES

1. https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm#:~:text=A%20Binary%20Search%20Tree%20(BST,to%20its%20parent%20node's%20key.
2. https://www.geeksforgeeks.org/binary-search-tree-data-structure/
3. https://www.javatpoint.com/binary-search-tree
4. https://www.geeksforgeeks.org/applications-of-bst/
5. https://www.geeksforgeeks.org/applications-of-bst/
6. https://dev.to/phuctm97/2-min-codecamp-binary-search-tree-and-real-world-applications-58cj
7. https://stackoverflow.com/questions/2130416/what-are-the-applications-of-binary-trees