Phonebook Algorithm Design - Documentation

1. Introduction

This document provides an overview of the mobile phonebook application designed using basic data structures such as **arrays** and **linked lists**. The solution allows for standard operations like adding, deleting, updating, searching, displaying, and sorting contacts in a simple and efficient manner.

The phonebook system is divided into several modules, each of which is responsible for a specific task, such as managing contacts or searching the phonebook. A singly linked list data structure is chosen to dynamically manage the memory and efficiently handle operations. The solution has been designed to be simple yet modular, enabling flexibility and maintainability.

2. Problem Statement

The goal is to design a mobile phonebook system that supports the following operations:

1. **Insert new contacts** into the phonebook.
2. **Search for contacts** by name or phone number.
3. **Display the contacts** currently stored in the phonebook.
4. **Delete contacts** from the phonebook.
5. **Update contact details**, such as name or phone number.
6. **Sort contacts** alphabetically by name for easier browsing.

3. Data Structures Used

- **Singly Linked List**: A linked list is chosen as the primary data structure due to its dynamic nature and efficient memory usage. Each contact is stored in a node, which contains:
  - `name` : The name of the contact.
  - `phoneNumber` : The phone number of the contact.
  - `next` : A pointer to the next contact in the list.

The linked list allows easy insertion and deletion of contacts and avoids memory reallocation issues that can occur with fixed-size arrays.

---

4. Solution Overview

The phonebook solution is divided into well-defined modules to organize the functionality and make the code easier to manage and extend in the future.

---

5. Modules Overview

The solution is structured into several modules, each with its own responsibilities:

Module 1: Contacts Management Module

This module is responsible for managing all contact-related operations such as insertion, deletion, and updating. It interacts directly with the linked list to perform these actions.

- **insertContact()**: Adds a new contact to the phonebook. If the phonebook is empty, it creates the first contact. Otherwise, it appends the contact to the end of the linked list.

- **deleteContact()**: Removes a contact from the phonebook by name. It traverses the linked list and deletes the node that matches the given contact name.
- **updateContact()**: Allows for the modification of an existing contact's name or phone number. It searches for the contact by name and updates the relevant details.

Module 2: Search Module

This module provides the ability to search for contacts in the phonebook by either the contact’s name or phone number.

- **searchContact()**: Performs a linear search through the linked list to find a contact based on either the name or phone number. It returns the contact details if found or indicates that the contact does not exist.

Module 3: Display Module

The display module handles the presentation of all stored contacts in the phonebook. It traverses the linked list and prints the name and phone number of each contact.

- **displayContacts()**: Displays all contacts in the phonebook in a neat format. If there are no contacts, it informs the user that the phonebook is empty.

Module 4: Sorting Module

This module provides functionality to sort the contacts alphabetically by their name, making the list easier to browse.

- **sortContacts()**: Implements a sorting algorithm (e.g., Bubble Sort) to rearrange the contacts alphabetically by name.

---

6. Algorithm Description

Below is a general description of how the key operations are performed in the phonebook:

1. **Insert Contact**:
    - Create a new node with the provided name and phone number.

- If the phonebook is empty, make the new node the head of the linked list.
- Otherwise, traverse the linked list to the end and insert the new contact.

2. **Search Contact**:
    - Start from the head of the list and compare each contact's name or phone number with the search query.
    - If a match is found, return the contact details.
    - If the end of the list is reached without finding a match, indicate that the contact is not found.
3. **Display Contacts**:
    - Start from the head of the list.
    - Traverse the linked list and print each contact's name and phone number.
    - If the phonebook is empty, notify the user that no contacts are stored.
4. **Delete Contact**:
    - Search for the contact to be deleted by its name.
    - If the contact is found, adjust the pointers in the linked list to remove the node.
    - If the contact is not found, notify the user.
5. **Update Contact**:
    - Search for the contact by name.
    - If found, modify the name and/or phone number as provided by the user.
6. **Sort Contacts**:
    - Use a basic sorting algorithm (like Bubble Sort) to rearrange the nodes in the linked list alphabetically by name.
    - Swap nodes' data until the list is in the correct order.

---

7. High-Level Algorithm for Sorting Contacts

pseudocode

```
function sortContacts():
    if head == null OR head.next == null:
        return // No need to sort if list is empty or contains only one contact
    swapped = True
    while swapped:
        swapped = False
        current = head
        while current.next != null:
            if current.name > current.next.name:
                // Swap contacts
                tempName = current.name
                tempPhoneNumber = current.phoneNumber
                current.name = current.next.name
                current.phoneNumber = current.next.phoneNumber
                current.next.name = tempName
                current.next.phoneNumber = tempPhoneNumber
                swapped = True
            current = current.next
```

8. Performance Considerations

- **Time Complexity**:
  - Insertion: O(n) in the worst case (traverse to the end of the list).
  - Deletion: O(n) to find and remove a contact.
  - Searching: O(n) for a linear search.
  - Sorting: O(n^2) with bubble sort (suitable for smaller datasets).
- **Space Complexity**:
  - O(n), where `n` is the number of contacts in the phonebook (since each contact is stored as a node in the linked list).

9. Conclusion

The designed phonebook solution is simple, scalable, and modular. The use of a linked list allows for efficient dynamic memory management, and the modular approach makes the application easy to extend and maintain. By dividing the solution into clear modules, it is easier to modify individual functionalities, such as replacing the sorting algorithm with a more efficient one in the future.