Introduction

We have implemented a complete Phone Book Management System in C++ which helps the users to manage the contacts efficiently. It allows adding, retrieving, searching, sorting and deleting contacts, as well as a few more advanced operations like binary and jump search for retrieval and sorting using the for Merge Sort and Quick Sort algorithms. But it was only a Partial implementation of B-Tree to list contacts by city.

The program covers several use cases related to contact management, making good use of STL Data structures and algorithms

Objectives

1) Contact Management:

Provide a robust and user-friendly interface for adding, viewing, searching, and deleting contacts.

2) Efficient Search Methods:

Implement Binary Search and Jump Search to retrieve contacts based on phone numbers, ensuring fast and accurate lookups.

3) Advanced Sorting Algorithms:

Include Merge Sort and Quick Sort for sorting contacts by their first names in an organized manner.

4)City-based Searching:

Allow users to find contacts residing in a specific city using a city-based search.

5) Data Organization:

Use an unordered map for fast duplicate checks and indexing, and partially implement a B-Tree

structure for future scalability.

6) Dynamic Deletion:

Support deletion of contacts using both array-based and linked list-based approaches, demonstrating algorithm flexibility.

Documentation:

Features

1. Add New Contact:

Prompts the user to enter contact details (first name, last name, phone number, address, city, and email) and adds it to the phone book if the phone number is unique.

1. Linear Search

- **Time Complexity**:
    - **Best Case**: O(1)O(1)O(1) → The target is found at the first element.
    - **Worst Case**: O(n)O(n)O(n) → The entire list is traversed.
    - **Average Case**: O(n)O(n)O(n) → Half of the list is checked on average.
- **Space Complexity**: O(1)O(1)O(1) → No additional data structures are required.

---

2. Hash Map

- **Time Complexity**:
    - **Search Time**: O(1)O(1)O(1) → Average case for searching in a hash map.
    - **Worst Case**: O(n)O(n)O(n) → In case of hash collisions.
- **Space Complexity**: O(n)O(n)O(n) → Storage for all keys and values in the hash map.

2. Retrieve Contact by Phone Number:

. Binary Search: A highly efficient method to search for contacts in a sorted list by their phone numbers.

. Jump Search: Another efficient method for searching in a sorted list, using a fixed step size for navigation.

3. Search Contacts by City:

Lists all contacts associated with a given city.

Linear Filtering : time : O(n) Space : O(1)

B-Tree : time : O(logn) Space : O(n)

4. Delete Contact:

. Array-based Deletion: Deletes a contact by phone number from the list stored in a vector.

. Linked-list-based Deletion: Deletes a contact in a dynamic linked list structure.

Search Time Complexity	**O(n)** (need to search for the phone number)	**O(n)** (traverse the list to find the contact)
Deletion Time Complexity	**O(n)** (shifting elements after deletion)	**O(1)** (adjust pointers after finding the node)

5. Sort Contacts by Name:

. Merge Sort: Divides the contact list into smaller parts, sorts them, and merges them back.

. Quick Sort: Selects a pivot, partitions the list, and recursively sorts partitions.

Worst-Case Time Complexity : Merge : O(n log n) Quick : O(n^2) (occurs when pivot selection is poor)

Average-Case Time Complexity : **Merge :** O(n log n) Quick : O(n log n)

Best-Case Time Complexity : Merge : O(n log n) Quick : O(n log n)

Space Complexity Merge : O(n) (additional space for merging) Quick : O(log n) (for recursion stack)

6. Display All Contacts:

Prints all contacts stored in the phone book.

7. City-based Contact Organization:

Includes a simplified implementation of a B-Tree structure for organizing and retrieving contacts by city.

8. Exit:

Ends the program.

Code Workflow

1. Data Structures Used:

. vector<Contact>: Stores the list of contacts.

. unordered_map: Maps phone numbers to contact indices for quick duplicate checks.

. BTreeNode and BTree: Provide a simplified city-based contact management system.

. Contact* (linked list): For demonstrating deletion via linked list.

2. Input/Output:

. Users interact via console inputs for adding, searching, and managing contacts.

. Outputs are displayed on the console.

3. Algorithms:

. Search: Binary and Jump Search for retrieval.

. Sort: Merge Sort and Quick Sort for sorting contacts by name.

4. Error Handling:

. Prevents duplicate contacts based on phone numbers.

. Handles cases where no contacts are found for a search or deletion operation.

. Gracefully handles empty phone books during sorting or display.

How to Use

1. Compile and Run:

Compile the program using a C++ compiler, e.g., g++ -o phonebook phonebook.cpp, and run the executable, ./phonebook.

2. Navigate the Menu:

Choose options from the displayed menu by entering the corresponding number.

3. Perform Operations:

. Add contacts by selecting option 1.

. Retrieve contacts by phone number using binary (2) or jump search (3).

. Search contacts by city (4).

. Delete a contact (5).

. Sort contacts using merge sort (6) or quick sort (7).

. View all contacts with option 8.

4. Exit:

Select option 9 to terminate the program.

Future Enhancements

1. Full B-Tree Implementation:

Expand the B-Tree structure to handle large-scale city-based contact storage.

2. File-Based Persistence:

Store and retrieve contacts from a file to ensure data persists between sessions.

3. GUI Integration:

Build a graphical interface for better usability.

4. Contact Groups:

Introduce functionality to group contacts for easier organization and management.

This code serves as a foundational implementation for a Phone Book Management System, showcasing various algorithms and data structures while being extendable for real-world applications.

THE FULL CODE

```
#include <iostream>
#include<vector>
#include<cmath>
#include<algorithm>
#include<deque>
#include<stack>
#include<queue>
```

```
#include<cstring>
#include<string>
#include<unordered_map>
using namespace std;
struct Contact {
	string phoneNumber;//primary key
	string firstName,lastName,address,city,email;
	Contact* next;
};
vector<Contact> contacts;
unordered_map<string, size_t> contactIndexMap;

// Define B-Tree Node
struct BTreeNode {
	bool isLeaf;
	vector<string> keys;			// Cities
	vector<vector<Contact>> values; // List of contacts mapped to city keys
	vector<BTreeNode*> children;	// Pointers to child nodes
};

// Define the BTree class
class BTree {
private:
	BTreeNode* root;
	int t; // Minimum degree of BTree

public:
	BTree(int degree) : t(degree) {
		root = new BTreeNode();
		root->isLeaf = true;
```

```
}

void insert(const string& city, const Contact& contact) {
    // Simplified: Inserts a contact into the appropriate city index.
    if (root->keys.empty()) {
        root->keys.push_back(city);
        root->values.push_back({ contact });
    }
    else {
        bool inserted = false;
        for (size_t i = 0; i < root->keys.size(); ++i) {
            if (root->keys[i] == city) {
                root->values[i].push_back(contact);
                inserted = true;
                break;
            }
        }
        if (!inserted) {
            root->keys.push_back(city);
            root->values.push_back({ contact });
        }
    }
}

void search(const string& city) {
    cout << "Searching for contacts in city: " << city << endl;
    for (size_t i = 0; i < root->keys.size(); ++i) {
        if (root->keys[i] == city) {
            for (auto& contact : root->values[i]) {
                cout << "Found Contact: " << contact.firstName << endl;
```

```
                }
                return;
            }
        }
        cout << "No contacts found in " << city << endl;
    }
};
```

```
// Function to add a contact
void addContact(vector<Contact>& contacts) {
    Contact NewContact;

    cout << "Enter First Name: ";
    cin >> NewContact.firstName;

    cout << "Enter Last Name: ";
    cin >> NewContact.lastName;

    cout << "Enter phone number: ";
    cin >> NewContact.phoneNumber;

    cout << "Enter The Address: ";
    cin >> NewContact.address;
```

```
        cout << "Enter The City: ";
        cin >> NewContact.city;

        cout << "Enter The Email: ";
        cin >> NewContact.email;

    // Check for duplicate phone number using hash map
        if (contactIndexMap.find(NewContact.phoneNumber) != contactIndexMap.end()) {
                cout << "Error:Contact with phone number     " << NewContact.phoneNumber << "
Already exists.\n";
                return;
        }

        // Add the contact
        contacts.push_back(NewContact);
        contactIndexMap[NewContact.phoneNumber] = contacts.size() - 1;
        cout << "Contact added successfully.\n";
}




// Function to display all contacts
void displayContacts(const vector<Contact>& contacts) {
        if (contacts.empty()) {
                cout << "Phone book is empty.\n";
                return;
        }
```

```
    cout << "\nContacts:\n";


    for (const auto& contact : contacts) {
        cout <<"\nName : "<< contact.firstName << ' ' << contact.lastName << '\n' <<
"PhoneNumber : " << contact.phoneNumber << '\n';
    }
}
```

```
void RetrieveContactByPhoneNumberBinary(string PhoneNumber) {// Search Method : (Binary
Search) , Time Complexity : (O(log n)) , Space Complexity : (O(1))


    int left = 0, right = contacts.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (contacts[mid].phoneNumber == PhoneNumber) {
            cout << "Phone Number: " << contacts[mid].phoneNumber << endl;
            cout << "Name: " << contacts[mid].firstName << " " << contacts[mid].lastName <<
endl;
            cout << "Address: " << contacts[mid].address << endl;
            cout << "City: " << contacts[mid].city << endl;
            cout << "Email: " << contacts[mid].email << endl;
            return;
        }
        else if (contacts[mid].phoneNumber < PhoneNumber) {
            left = mid + 1;
```

```
            }
            else {
                right = mid-1;
            }
        }
        cout << "Contact not found.\n";
}
```

```
void RetrieveContactByPhoneNumberJump(string PhoneNumber) {// Search Method : (Jump Search) , Time Complexity : (O(sqrt n)) , Space Complexity : (O(1))

        int n = contacts.size();
        int step = sqrt(n);
        int locate = 0;

        while (contacts[min(step, n) - 1].phoneNumber < PhoneNumber) {
            locate = step;
            step += sqrt(n);

            if (locate >= n) {
                cout << "Contact not found.\n";
                return;
            }
        }
```

```
        for (int i = locate;i < min(step, n);i++) {
                if (contacts[i].phoneNumber == PhoneNumber) {
                        cout << "Phone Number: " << contacts[i].phoneNumber << endl;
                        cout << "Name: " << contacts[i].firstName << " " << contacts[i].lastName << endl;
                        cout << "Address: " << contacts[i].address << endl;
                        cout << "City: " << contacts[i].city << endl;
                        cout << "Email: " << contacts[i].email << endl;
                        return;
                }
        }
        cout << "Contact not found.\n";
        return;
}




// Array-based Deletion :
// O(n)
void delete_contact_array(vector<Contact>& contacts,const string& phoneNumber) {
        for (int i = 0; i < contacts.size(); ++i) {
                if (contacts[i].phoneNumber == phoneNumber) {
                        contacts.erase(contacts.begin() + i);
                        contactIndexMap.erase(phoneNumber);
                        cout << "Contact Deleted successfully\n";
                        return;
                }
```

```
        }
        cout << "Contact Not Found !\n";
}




// Linked-list based Deletion :
// O(n)
void delete_contact_linked(Contact*& head, const string& phoneNumber) {
        if (head == nullptr) {
                cout << "No contacts available!" << endl;
                return;
        }

        if (head->phoneNumber == phoneNumber) {
                Contact* temp = head;
                head = head->next;
                delete temp;
                cout << "Contact with phone number " << phoneNumber << " deleted successfully." <<
endl;
                return;
        }

        Contact* current = head;
        while (current->next != nullptr && current->next->phoneNumber != phoneNumber) {
                current = current->next;
        }
```

```
        if (current->next != nullptr) {
            Contact* temp = current->next;
            current->next = current->next->next;
            delete temp;
            cout << "Contact with phone number " << phoneNumber << " deleted successfully." <<
endl;
        }
        else {
            cout << "Contact with phone number " << phoneNumber << " not found." << endl;
        }
}



void SearchByCity(string city) {
        bool found = false;
        for (const auto& contact : contacts) {
            if (contact.city == city) {
                found = true;
                cout << "\nName: " << contact.firstName << " " << contact.lastName
                    << "\nPhone Number: " << contact.phoneNumber << "\n";
            }
        }
        if (!found) {
            cout << "No contacts found in the city: " << city << "\n";
        }
}
```

```
void merge(vector<Contact>& phoneBook, int left, int mid, int right) {
	int n1 = mid - left + 1;
	int n2 = right - mid;

	vector<Contact> leftArray(n1), rightArray(n2);

	// Copy data to temporary arrays
	for (int i = 0; i < n1; ++i)
		leftArray[i] = phoneBook[left + i];
	for (int i = 0; i < n2; ++i)
		rightArray[i] = phoneBook[mid + 1 + i];

	// Merge the two arrays
	int i = 0, j = 0, k = left;
	while (i < n1 && j < n2) {
		if (leftArray[i].firstName <= rightArray[j].firstName) {
			phoneBook[k] = leftArray[i];
			++i;
		}
		else {
			phoneBook[k] = rightArray[j];
			++j;
		}
		++k;
	}

	// Copy remaining elements
	while (i < n1) {
		phoneBook[k] = leftArray[i];
```

```
            ++i;
            ++k;
        }

        while (j < n2) {
            phoneBook[k] = rightArray[j];
            ++j;
            ++k;
        }
}

// Merge Sort function
void mergeSort(vector<Contact>& phoneBook, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;

            // Sort first and second halves
            mergeSort(phoneBook, left, mid);
            mergeSort(phoneBook, mid + 1, right);

            // Merge sorted halves
            merge(phoneBook, left, mid, right);
        }

        else {

            if (contacts.empty()) {
                cout << "Phone book is empty.\n";
                return;
            }
```

```
        }

}



// Partition function for Quick Sort
int partition(vector<Contact>& phoneBook, int low, int high) {
        string pivot = phoneBook[high].firstName; // Last element as pivot
        int i = low - 1;

        for (int j = low; j < high; ++j) {
                if (phoneBook[j].firstName <= pivot) {
                        ++i;
                        swap(phoneBook[i], phoneBook[j]);
                }
        }
        swap(phoneBook[i + 1], phoneBook[high]);
        return i + 1;
}

// Quick Sort function
void quickSort(vector<Contact>& phoneBook, int low, int high) {
        if (low < high) {
                int pi = partition(phoneBook, low, high);

                // Recursively sort the partitions
                quickSort(phoneBook, low, pi - 1);
                quickSort(phoneBook, pi + 1, high);
```

```
        }
        else {
            if (contacts.empty()) {
                cout << "Phone book is empty.\n";
                return;
            }
        }
}

int main() {
        int choice;
        string phoneNumber;
        string city;

cout << "Phone Book Menu:\n";
            cout << "\n1. Add New Contact\n";
            cout << "2. Retrieve Contact by Phone Number (Binary Search)\n";
            cout << "3. Retrieve Contact by Phone Number (Jump Search)\n";
            cout << "4. Search Contact by City\n";
            cout << "5. Delete Contact by Phone Number (Array-based)\n";
            cout << "6. Sort Contacts by Name (Merge Sort)\n";
            cout << "7. Sort Contacts by Name (Quick Sort)\n";
            cout << "8. Display All Contacts\n";
            cout << "9. Exit\n";
            cout << "\nEnter your choice: ";

        do {
            cin >> choice;

            switch (choice) {
```

```
case 1:
        addContact(contacts);
        break;

case 2:
        cout << "Enter Phone Number to Retrieve (Binary Search): ";
        cin >> phoneNumber;
        RetrieveContactByPhoneNumberBinary(phoneNumber);
        break;
case 3:
        cout << "Enter Phone Number to Retrieve (Jump Search): ";
        cin >> phoneNumber;
        RetrieveContactByPhoneNumberJump(phoneNumber);
        break;

case 4:
        cout << "Enter City to Search Contacts: ";
        cin >> city;
        SearchByCity(city);
        break;

case 5:
        cout << "Enter Phone Number to Delete : ";
        cin >> phoneNumber;
        delete_contact_array(contacts, phoneNumber);
        break;

case 6:
        cout << "\nSorting Contacts by Name (Merge Sort):\n";
        mergeSort(contacts, 0, contacts.size() - 1);
```

```
            cout << "Contacts sorted successfully using Merge Sort.\n";
            break;

        case 7:
            cout << "\nSorting Contacts by Name (Quick Sort):\n";
            quickSort(contacts, 0, contacts.size() - 1);
            cout << "Contacts sorted successfully using Quick Sort.\n";
            break;

        case 8:
            displayContacts(contacts);
            break;

        case 9:
            cout << "Exiting Phone Book. Goodbye!\n";
            break;

        default:
            cout << "Invalid choice. Please try again.\n";
            break;
        }

        cout << "\n";
    } while (choice != 9);

    return 0;
}
```