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,Bye.\n";
            break;
        }

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

    return 0;
}
```