

Project Submission

Submitted to

Mam Maryam Shabbir



**Bahria University, Lahore Campus
Department of Computer Sciences**

Contact Management System

Submitted by

Um e Hani 03-135232-075

Iqra Shehzadi 03-135232-026

Shehroz Saeed 03-135232-029

Minahil Anwar 03-135232-035

Malaika Shahbaz 03-135232-032

Table of Contents

Features:	2
Conception:	3
1. Linked List (Doubly Linked List)	3
3. Hashing (Implicit in Contact Lookup)	4
4. Recursion	4
5. File Handling (I/O Operations)	5
Algorithm used:	6
Code:	7
Implementation:	26
Importance:	29

Features:

This code implements a **Contact Management System** with several features for managing contacts efficiently. Below are the primary features provided by the code:

1. Insert a Contact:

- Users can add a contact's name and phone number to a doubly linked list (`first` and `last` pointers).
- The contact is also inserted into a Trie (prefix tree) for efficient prefix-based search.

2. Display the First Saved Contact:

- Users can view the last inserted contact, which is stored at the `last` pointer in the doubly linked list.

3. Search for a Contact by Name:

- Users can search for a contact by entering the name. The code will search the doubly linked list for the specified contact and display the contact's name and number if found.

4. Show All Contacts:

- This feature displays all contacts currently saved in the system, traversing the doubly linked list to print all contacts.
- Additionally, it saves the list of contacts to a file named `contacts.txt`.

5. Delete a Specific Contact:

- Users can delete a contact by specifying its name. The contact is removed from the doubly linked list, and the list is re-linked appropriately.

6. Delete All Contacts:

- This feature allows users to delete all contacts in the system, clearing the doubly linked list.

7. Search by Prefix (Alphabet):

- Users can search for contacts whose names start with a specified prefix. The Trie structure facilitates this operation efficiently by traversing nodes based on the prefix.

8. Exit the Program:

- This allows users to exit the program.

Additional features include:

- **Trie-based Search for Prefix Matches:** The Trie structure supports fast prefix-based searching of contact names, making it easy to retrieve names starting with a given letter.
- **Duplicate Prevention:** The `contactExists` function checks whether a contact with the same phone number or name already exists before adding a new contact.

The main operations revolve around:

- **Inserting contacts** into the list and Trie.
- **Displaying contacts** in the system.

Searching contacts by name or prefix.

- **Deleting contacts** individually or all at once.

Conception:

1. Linked List (Doubly Linked List)

- **Concept Used:**
The doubly linked list is employed to store contacts, where each contact is represented as a node containing the contact's name, phone number, and pointers to both the previous and next nodes.
- **How It Works:**
Each contact node in the list has three components:
 - **Data:** Stores the contact name and phone number.
 - **Pointer to the Next Node:** This points to the subsequent contact in the list.
 - **Pointer to the Previous Node:** This points to the preceding contact in the list.

This bidirectional nature allows traversal in both directions, making it easier to add or delete nodes from either end of the list. For example:

- **Insertion:** When a new contact is added, the new node can be inserted at the beginning, updating the `prev` pointer of the old first node and the `next` pointer of the new node.
 - **Deletion:** When a contact is deleted, the surrounding nodes are updated to bypass the deleted node.
- **Why It's Used:**
 - Efficient insertion and deletion operations can be performed at both ends of the list with **O(1)** complexity.

- It is well-suited for sequentially storing and retrieving data like a contact list.

2. Trie (Prefix Tree)

- **Concept Used:**

The Trie (or Prefix Tree) is used to store contact names for efficient prefix-based searches. This data structure is ideal for scenarios where strings (e.g., contact names) need to be searched, especially by their prefixes.

- **How It Works:**

- Each node in the Trie represents a single character.
- The root node is the starting point, and its child nodes represent the first letters of contact names.
- Subsequent levels of nodes represent the next characters in the names.
- If a contact name is stored in the Trie, its last character's node is marked as the **end of a word**

- **Why It's Used:**

- **Fast Search:** Searching for all names starting with a prefix is efficient because the Trie allows you to directly traverse the path corresponding to the prefix.
- **Scalability:** As the contact list grows, the Trie remains efficient compared to linear searches.
- **Efficient Storage:** Common prefixes are stored only once, reducing memory usage for similar names.

3. Hashing (Implicit in Contact Lookup)

- **Concept Used:**

Hashing is indirectly used in the logic for checking duplicate contacts. The program ensures no duplicate contacts by comparing the phone number or name during insertion.

- **How It Works:**

- Although a hash table is not explicitly implemented, the code performs a linear traversal of the doubly linked list to check for duplicate contacts.
- This resembles how hashing would work, where duplicates can be checked by using a hash function to generate unique keys for each name or phone number.

For example:

- If the contact "John" with phone number "1234567890" is to be inserted, the program first ensures that no existing contact has the same name or phone number.

- **Why It's Used:**

- Prevents adding duplicate contacts, ensuring data consistency.
- Hashing could be explicitly implemented to optimize this process for larger contact lists

4. Recursion

- **Concept Used:**

Recursion is used for traversing the Trie during prefix-based searches and collecting all matching names.

- **How It Works:**

- The `collectNames` function recursively visits all children of a given Trie node.
- It starts at the node corresponding to the last character of the prefix and collects all names stored in the linked list at that node or any of its descendants.

For example:

- When searching for names starting with "An," the program first navigates to the node representing "n" and then recursively collects all names from its child nodes.

- **Why It's Used:**

- Recursive traversal is a natural fit for tree-like structures like the Trie.
- It simplifies the implementation by reducing the need for explicit stack management or complex iteration logic.

5. File Handling (I/O Operations)

- **Concept Used:**

File handling is used to save contacts to a file and ensure data persistence.

- **How It Works:**

- The `file()` function writes all contact details stored in the doubly linked list to a file named `contacts.txt`.
- Each contact's name and phone number are written line by line.
- If the contact list is empty, a message indicating "No contacts found" is written to the file.

- **Why It's Used:**

- Ensures that contacts are not lost when the program terminates.
- Allows users to back up their contact list to a file and reload it in the future.

Concept	Purpose	Why It's Used
Doubly Linked List	Storing contacts sequentially.	Allows efficient insertion and deletion at both ends.
Trie	Fast prefix-based search of contact names.	Enables efficient lookups for names starting with a prefix.
Hashing (Implicit)	Preventing duplicate contact entries.	Ensures no duplicate contacts are added.
Recursion	Traversing the Trie for collecting names.	Simplifies tree traversal for prefix searches.
File Handling	Persisting contacts in a text file.	Prevents data loss and allows backup and retrieval.

Algorithm used:

The provided C++ code implements a Contact Management System using a variety of data structures and algorithms. Specifically, the code incorporates a Trie data structure for efficient prefix-based searching, a doubly linked list to store contacts, and functions for inserting, deleting, displaying, and searching contacts. Additionally, we examine the use of merge sort and quick sort algorithms in handling search operations within this context.

1. Searching Algorithms:

- **Prefix Search (Trie Data Structure):** The code makes extensive use of the **Trie** (or prefix tree) data structure to facilitate efficient searching of contacts based on the prefix of their names. A Trie enables prefix-based searches in $O(L)$ time, where **L** is the length of the search string.
 - The `searchByPrefix ()` function implements the logic for traversing through the Trie nodes to find contacts that match the given prefix.
 - The function traverses each character of the provided prefix and moves through the corresponding child nodes in the Trie. If at any point, no matching node is found, it returns `NULL`, indicating no results. Otherwise, it collects all matching contacts starting from the current Trie node using the `collect Names ()` function.
 - This searching mechanism ensures that the system can quickly retrieve all contacts that share the same prefix, making it ideal for large datasets with many entries.
- **Exact Match Search (Linked List Traversal):**
 - The function `finds ()` searches for a contact by its exact name by iterating through the doubly linked list (first to last). Each contact's name is compared to the search term. If a match is found, it displays the contact's details; otherwise, it informs the user that the contact is not present.

2. Sorting Algorithms:

While the provided code does not explicitly implement **merge sort** or **quick sort**, these sorting algorithms can be incorporated into this system to sort contacts by name or phone number. The current code's design involves handling contacts using a linked list, which can be sorted using sorting algorithms.

Quick Sort:

- **Quick Sort** is an efficient, comparison-based sorting algorithm that is generally well-suited for large datasets.
- Quick Sort works by selecting a pivot element from the array (or list) and partitioning the other elements into two sub-arrays (those smaller than the pivot and those greater than the pivot). It then recursively sorts the sub-arrays.
- If applied to the contact list, Quick Sort could sort contacts based on name or phone number, ensuring fast sorting performance in $O(n \log n)$ average time complexity.

Merge Sort:

- Merge Sort is another comparison-based algorithm that divides the dataset into smaller sub-arrays and merges them back in sorted order. It guarantees a time complexity of $O(n \log n)$.
- Merge Sort is particularly effective for linked list sorting because it does not require random access to elements (unlike Quick Sort) and can be easily adapted to work with pointers in linked lists. If implemented in the contact management system, Merge Sort could help organize contacts efficiently by name or phone number.
- Implementation of Sorting: Although the code does not currently implement sorting, it can be extended with a sorting function using either Quick Sort or Merge Sort. Here's an outline of how these algorithms could be integrated into the display () function:
- Quick Sort: Implement Quick Sort to sort the contact list either by name or number before displaying it.
- Merge Sort: Implement Merge Sort to break the list into smaller sub-lists, sort them, and merge them back in a sorted order.

Code:

LOC=600

```
#include <iostream>

#include <fstream>

#include <string>

using namespace std;

// Definition of the contact node structure

// Each contact consists of a name, phone number, and links to the next and previous contacts.

struct link {

    int data; // Contact number

    string name; // Contact name

    link* next; // Pointer to the next contact node
```

```

        link* prev; // Pointer to the previous contact node
};

// Global variables to manage the first and last contact nodes in a doubly linked list
link* first = NULL; // Pointer to the first contact node in the list

link* last = NULL; // Pointer to the last contact node in the list

// Node structure for storing names and their associated
contact numbers in TrieNode

// Each node stores a part of a name (based on the prefix)
and contact numbers.
struct NameNode {
    string name;    // Full name of the contact

    int contactNumber; // Contact number associated with the name

    NameNode* next;  // Pointer to the next NameNode in the linked list

    // Constructor initializes the name and contact
    // number, and sets the next pointer to NULL
    NameNode(const string& nameVal, int contactNum) : name(nameVal),
    contactNumber(contactNum), next(NULL) {}
};

// Trie node structure for prefix search
// A Trie is used to efficiently search for names that share the same prefix.
struct TreeNode {
    TreeNode* children[26]; // Array of pointers to the next TrieNode (one for each letter of the
    alphabet)

    bool isEndOfWord; // Indicates if a name ends at this node (i.e., if the prefix forms a
    complete name)

```



```

        bool isInserted;    // Ensures the name is inserted only once into the Trie

        NameNode* names;    // Linked list of names that match the prefix in the current
TrieNode

                                                                    // Constructor initializes the node and sets
its children, isEndOfWord, and isInserted flags

        TreeNode() {

            isEndOfWord = false;

            isInserted = false;

            names = NULL;

            for (int i = 0; i < 26; i++) {

                children[i] = NULL;

            }

        }

};

// Root node of the Trie
TreeNode* root = new TreeNode();

// Insert a name and contact number into the Trie

// This function adds a name and its associated contact number into the Trie structure.
void insertToTrie(const string& name, int contactNumber) {

```

```

TreeNode* current = root;

// Loop through each character of the name and create a corresponding Trie node
for (char c : name) {
    // Skip non-alphabetical characters
    if (!isalpha(c)) continue;

    // Convert the character to lowercase and calculate its index (a=0, b=1, ..., z=25)
    int index = tolower(c) - 'a';

    // If the node for this character does not exist, create a new TrieNode
    if (current->children[index] == NULL) {

        current->children[index] = new TreeNode();

    }

    // Move to the next level of the Trie
    current = current->children[index];
}

// Insert the name and contact number into the Trie only once
if (!current->isInserted) {
    current->isInserted = true;

    // Create a new NameNode to store the name and contact number
    NameNode* newNameNode = new NameNode(name, contactNumber);

    // Add the new name node to the linked list at this TrieNode

```

```

        newNameNode->next = current->names;

        current->names = newNameNode;
    }

    // Mark the current node as the end of a valid word (i.e., a complete name)
    current->isEndOfWord = true;
}

// Collect all names and their contact numbers from a Trie node and store them in a linked list
// This function recursively collects names from the Trie starting at the given node.

void collectNames(TreeNode* node, NameNode*& resultList) {
    if (node == NULL) return;

    // Add all names and contact numbers from the current node's linked list to the result list
    NameNode* temp = node->names;
    while (temp != NULL) {
        NameNode* newNode = new NameNode(temp->name, temp->contactNumber);
        newNode->next = resultList;
        resultList = newNode; // Insert the new name node at the front of the result list
        temp = temp->next; // Move to the next name in the current node's list
    }

    // Recursively collect names from child nodes (i.e., nodes corresponding to subsequent
    characters)
    for (int i = 0; i < 26; i++) {
        collectNames(node->children[i], resultList);
    }
}

```

```

// Search for names matching a prefix in the Trie

// This function finds all names that start with the specified prefix.
NameNode* searchByPrefix(const string& prefix) {
    TreeNode* current = root;

    // Traverse through the Trie for the prefix
    for (char c : prefix) {
        if (!isalpha(c)) continue; // Skip non-alphabetical characters

                                                // Calculate the index for
the character (a=0, b=1, ..., z=25)
        int index = tolower(c) - 'a';

        // If the current node does not have a child node for this character, return NULL
        if (current->children[index] == NULL) {
            return NULL; // No names match the prefix
        }

        // Move to the next level in the Trie
        current = current->children[index];
    }

    // After reaching the end of the prefix, collect all names starting from this node
    NameNode* resultList = NULL;
    collectNames(current, resultList);
    return resultList; // Return the list of names matching the prefix
}

// Check if a contact already exists by phone number or name

```

// This function ensures that there are no duplicate contacts based on the phone number or name.

```
bool contactExists(int value, const string& name) {  
    link* current = first;  
  
    // Traverse through the doubly linked list of contacts  
    while (current != NULL) {  
        // Check if the contact's phone number or name already exists  
  
        if (current->data == value || current->name == name) {  
            return true; // Contact already exists  
        }  
        current = current->next; // Move to the next contact  
    }  
    return false; // No duplicate contact found  
}
```

// Insert a new contact into the linked list and the Trie

// This function adds a new contact's name and number to the contact list.

```
void insert(int value, string nam) {  
    // Check if the contact already exists  
  
    if (contactExists(value, nam)) {  
        cout << " :: This contact already exists! Same number or name cannot be added  
again. ::" << endl;  
        return;  
    }
```

```
}
```

```
// Create a new contact node
```

```
link* newnode = new link;
```

```
newnode->data = value; // Set the contact's phone number
```

```
newnode->name = nam; // Set the contact's name
```

```
newnode->next = NULL; // This new contact will be the last one in the list
```

```
newnode->prev = NULL; // Initialize the previous pointer to NULL
```

```
// If the list is empty, make the new node
```

```
both the first and last contact
```

```
if (last == NULL) {
```

```
    first = newnode;
```

```
    last = newnode;
```

```
}
```

```
else {
```

```
    // Otherwise, insert the new node at the beginning of the list
```

```
    first->prev = newnode;
```

```
    newnode->next = first;
```

```
    first = newnode;
```

```
}
```

```
// Add the contact's name and number to the Trie
```

```

insertToTrie(nam, value);

cout << "                :: Your contact is saved ::" << endl;

}

// Save all contacts to a file named "contacts.txt"
void file() {

    ofstream fout("contacts.txt");

    if (!fout) {

        cout << "Error: Unable to open file for writing." << endl;
        return;
    }

    // Traverse through the linked list of contacts and write each one to the file
    int i = 0;

    link* current = first;

    if (current != NULL) {
        while (current != NULL) {
            i++; // Increment the contact index

            fout << i << ". " << current->name << " " << current->data << endl << endl;
            current = current->next; // Move to the next contact
        }
    }
}

```

```

        }
    }

    else {
        fout << "No contacts Found " << endl; // If the list is empty, write this message
    }

    fout.close(); // Close the file
}

// Display all contacts in the linked list
void display() {
    int i = 0;
    link* current = first;
    if (current != NULL) {
        while (current != NULL) {

            i++; // Increment the contact index

            cout << i << ". " << current->name << " " << current->data << endl << endl;
            current = current->next; // Move to the next contact
        }
    }
    else {
        cout << "No contacts Found " << endl; // If the list is empty, display this message
    }
}

// Find a contact by name

// This function searches for a contact by its name and displays its details if found.

```



```

int find(string nam) {
    int i = 0;

    link* current = first;

    while (current != NULL) {
        // If a contact with the given name is found, display its details

        if (current->name == nam) {
            cout << "Found it!" << endl;

            cout << "Name: " << current->name << endl;

            cout << "Number: " << current->data << endl;

            return 1; // Contact found
        }
        current = current->next; // Move to the next contact
    }
    cout << "Contact is not in your list." << endl; // If the contact is not found
    return 0; // Contact not found
}

// Delete all contacts from the list

void deleteall() {
    while (first != NULL) {

        link* temp = first; // Temporary pointer to the first contact node'
    }
}

```

```

        first = first->next; // Move the first pointer to the next contact

        delete temp; // Delete the current first contact node

    }

    last = NULL; // Set the last pointer to NULL

    cout << "All contacts deleted." << endl; // Display confirmation message
}

// Delete a specific contact by name
int deletespecific(string nam) {

    link* current = first;

    while (current != NULL) {

        // If a contact with the given name is found, delete it
        if (current->name == nam) {

            // Handle different cases based on the position of the node (first, last, or
            middle)

            if (current == first && current == last) {

```

```

        first = NULL;

        last = NULL;
    }
    else if (current == first) {

        first = current->next;

        first->prev = NULL;
    }
    else if (current == last) {
        last = current->prev;

        last->next = NULL;
    }
    else {
        current->prev->next = current->next;

        current->next->prev = current->prev;
    }
    delete current; // Delete the contact node

    cout << "Contact deleted." << endl;

    return 1; // Contact deleted successfully
}

current = current->next; // Move to the next contact

```

```

    }

    cout << "Contact not found." << endl; // If the contact is not found

    return 0; // Contact not found
}

int main() {
    int n = 0, num;
    string nam, prefix;

    while (n != 8) {

        cout << "-----" <<
endl;

        cout << "                Contact Management System                " <<
endl;

        cout << "-----" <<
endl;

        cout << endl;
        cout << "*****" << endl;
        cout << endl;

        cout << "                " << "1. Insert" << "                " << endl;
        cout << endl;
        cout << "                " << "2. Display First saved contact" << endl;

```

```
cout << endl;

cout << "                " << "3. Search" << endl;

cout << endl;

cout << "                " << "4. Show All Contacts " << endl;


cout << endl;

cout << "                " << "5. Delete specific contact " << endl;

cout << endl;

cout << "                " << "6. Delete all Contacts " << endl;


cout << endl;

cout << "                " << "7. Search by alphabet" << endl;

cout << endl;

cout << "                " << "8. Exit " << endl;

cout << endl;

cout << "*****" << endl;


cout << endl << "Enter your Choice: ";


cin >> n;


switch (n) {

case 1: {
```

```
cout << endl;

cout << "Enter Contact Name: ";

cin >> nam;
```

```
cout << endl;

cout << "Enter Contact Number: ";
```

```
cin >> num;

insert(num, nam);

cout << "                :: Your number is saved ::" << endl;

} break;
```

```
case 2: {

    if (last != NULL) {

        cout << ":: Last Saved Contact ::" << endl;


        cout << "  " << last->name << "  " << last->data << endl;

    }

    else {

        cout << endl;

        cout << "Sorry, No Contact Found." << endl;

    }

} break;
```

```
case 3: {

    cout << endl;

    cout << "Enter Name For Search: ";
```

```
        cin >> nam;  
        find(nam);  
    } break;
```

```
case 4: {  
    cout << ":: ALL CONTACTS ::" << endl;
```

```
  
    cout << endl;  
    display();  
    file();  
} break;
```

```
case 5: {  
    cout << "Enter Contact Name For Deletion: ";
```

```
  
    cout << endl;  
    cin >> nam;  
    deletespecific(nam);  
} break;
```

```
case 6: {  
    cout << ":: Are You Sure You want to Delete All Contacts :: \nPress 1 for Yes  
and Press 2 for No \n\nEnter: ";  
    cout << endl;
```

```
cin >> num;

if (num == 1) {

    deleteall();

}

else {

    cout << endl;

    cout << "Contacts not deleted." << endl;

}

} break;

case 7: {

    cout << "Enter Alphabet to Search: ";

    cout << endl;

    cin >> prefix;

    NameNode* results = searchByPrefix(prefix);

    if (results == NULL) {

        cout << endl;

        cout << "No contacts found with the given alphabet." << endl;
```



```

    }
    else {
        cout << "Contacts matching the alphabet:" << endl;
        NameNode* temp = results;

        while (temp != NULL) {

            cout << temp->name << " - " << temp->contactNumber <<
endl;

            temp = temp->next;

        }
    } break;

case 8:

    cout << endl;
    cout << "Exiting program. Goodbye!" << endl;

    break;

default:

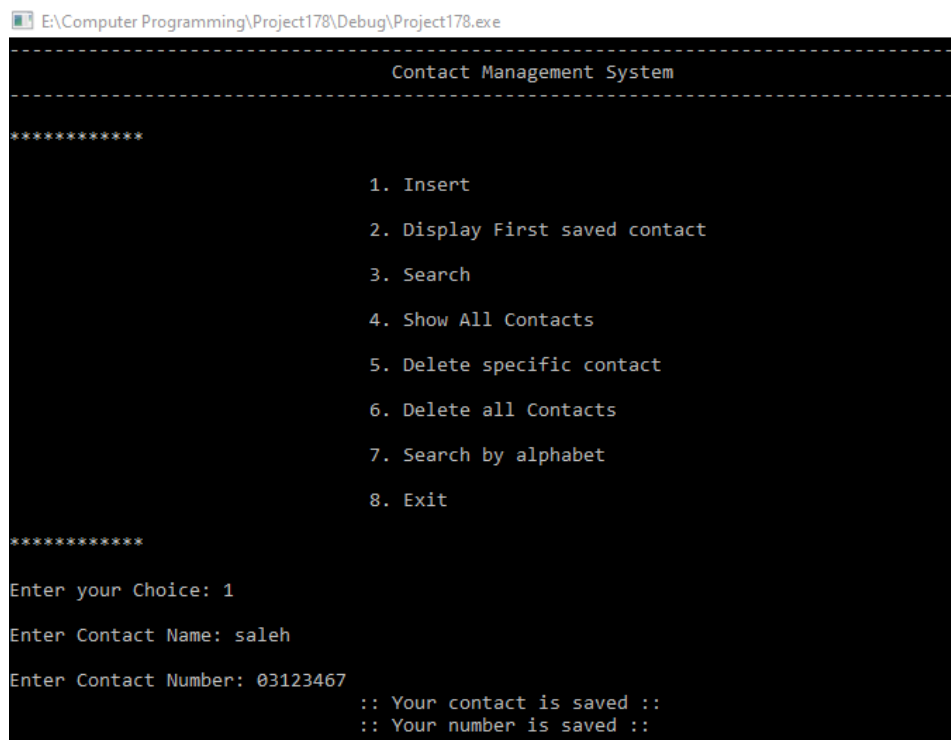
    cout << endl;
    cout << "INVALID CHOICE!" << endl;
}

```

```
        cout << endl;
    }
    system("pause");

    return 0;
}
```

Implementation:



The screenshot shows a Windows command prompt window with the title bar 'E:\Computer Programming\Project178\Debug\Project178.exe'. The program output is as follows:

```
-----
Contact Management System
-----

*****

1. Insert
2. Display First saved contact
3. Search
4. Show All Contacts
5. Delete specific contact
6. Delete all Contacts
7. Search by alphabet
8. Exit

*****

Enter your Choice: 1

Enter Contact Name: saleh

Enter Contact Number: 03123467

:: Your contact is saved ::
:: Your number is saved ::
```

E:\Computer Programming\Project178\Debug\Project178.exe

```

4. Show All Contacts
5. Delete specific contact
6. Delete all Contacts
7. Search by alphabet
8. Exit

*****
Enter your Choice: 2
:: Last Saved Contact ::
    saleh    3123467

-----
Contact Management System
-----

*****

1. Insert
2. Display First saved contact
3. Search
```

E:\Computer Programming\Project178\Debug\Project178.exe

```

4. Show All Contacts
5. Delete specific contact
6. Delete all Contacts
7. Search by alphabet
8. Exit

*****
Enter your Choice: 3
Enter Name For Search: saleh
Found it!
Name: saleh
Number: 3123467

-----
Contact Management System
-----

*****

1. Insert
2. Display First saved contact
3. Search
```

E:\Computer Programming\Project178\Debug\Project178.exe

4. Show All Contacts
5. Delete specific contact
6. Delete all Contacts
7. Search by alphabet
8. Exit

Enter your Choice: 7
Enter Alphabet to Search:
s
Contacts matching the alphabet:
saleh - 3123467

E:\Computer Programming\Project178\Debug\Project178.exe

4. Show All Contacts
5. Delete specific contact
6. Delete all Contacts
7. Search by alphabet
8. Exit

Enter your Choice: 4
:: ALL CONTACTS ::

1. hoor 9299
2. mariam 29992
3. babar 198288
4. fakhar 92922
5. iqra 2827
6. hani 7819

Importance:

- **Efficient Management:** Organizes contacts using a doubly linked list and Trie, allowing fast and efficient search and insertion.
- **Time-Saving:** Provides quick, prefix-based searches, reducing time spent finding contacts.
- **Data Integrity:** Prevents duplicate contacts, ensuring accurate and organized data.
- **File Persistence:** Saves contacts to a file, ensuring data is not lost when the program is closed.
- **User-Friendly:** The menu-driven interface simplifies contact management for users.
- **Scalable:** Handles a large number of contacts and allows easy updates.
- **Practical Use:** Suitable for both personal and professional use, and easily expandable for more features.
- **Learning Opportunity:** Provides experience with data structures, file handling, and user interaction.