Phone Directory

Members ---------- Muhammad Hammad Tahir /Aqsa Saeed /Noor Fatima

Group no ---------- 20

Course ---------- Data Structure and Algorithms (DSA)

Requirements:

Following 3 types of requirements are needed for this project:

Functional:

- Save information about different people.
- Give information of required person when asked
- Display list of all information
- Can delete information
- Save more than one numbers for a single person (Do not Duplicate Phone numbers)

Non-Functional:

- Fast in speed
- Efficient in space

Usability:

- System must be simple and easy to use

- Should not show the extra information (screen must display only running function to prevent confusion)

Overview:

This is a phone directory project which will manage the data of different peoples. The following are the key features for this system.

- Add the new contact information
- Search the contact information of specific person
- Remove the existing contact information
- View all contact's information

Data Structures:

•Binary search Tree (BST):

This data structure plays a pivotal role in this project because the main data of the phone directory (Contact information) is stored in a Binary Search Tree. As we have studied in this course, searching, and inserting an entity are more efficient in BST than in other data structures like arrays and linked lists. This data structure is studied in the Data Structures and Algorithms course, and we want to implement it in our project to gain more practice with this data structure.

•Set:

This is the additional data structure that we didn't study in this DSA course, but we have learned it and we're going to implement it in this project to meet the requirement of our term project. We'll be using this data structure to store more than one phone number for one person. The reason behind using this data structure is that the set data structure does not allow storing two identical entities, which is a requirement of our project as the repetition of the same phone number for the same person is not allowed.

What's new to us?

Previously, the programs we designed and wrote did not take into consideration the time and space complexity of the algorithms. We were unaware of the consequences of these

factors. However, in this course, we learned about different data structures and algorithms that are both time and space efficient. In the past projects, we were unaware of these advanced data structures and used only arrays or linked lists to store and find data. This required searching the entire data, which took order of N { O(N) } time, and was hectic in the case of large data. Now, by using these advanced algorithms (BST and set), we will increase the efficiency of the program because in BST, the longest time it takes to search and insert data is { O(log(N)) }. This will make our system faster by using fewer resources.

Comparison of BST and SET with other Data Structures:

Why we choose BST and not AVL ?

Simplicity and Lower Overhead:

One of the most significant advantages of AVL trees over BST is that AVL trees do not form linked lists. In contrast, there is a possibility of linked list formation in the case of BST. However, when we compare strings to insert new records in a phone directory, the likelihood of linked list formation is almost non-existent. So, using BSTs which have lower overheads in terms of both memory and computational resources is a confident and smart choice in this scenario.

Simplicity in Insertions and Deletions:

It is a well-established fact that insertions in a Binary Search Tree (BST) are simpler than in an AVL Tree for a phone directory. This is because AVL Trees require additional rotations to maintain balance, which can introduce overhead and making the insertion process more complex. Therefore, if the phone directory experiences frequent insertions and the tree remains reasonably balanced, it is safe to say that a BST would deliver better performance in terms of simpler insertions.

Memory Efficiency:

AVL trees maintain balance by ensuring that the heights of the left and right subtrees of any node differ by at most one. This balance requirement necessitates the storage of additional information (balance factors) for each node, which consumes extra memory. In particular, each node in an AVL tree typically requires an additional byte (or more) to store the balance factor. As discussed earlier the chance of imbalance in phone directory is almost negligible, so using AVL instead of BST will not be a smart choice in this case in which memory wastage is a primary concern.

Why we choose BST and not Hashes ?

No Load Factor Consideration:

It is a well-established fact that unlike hash tables, Binary Search Trees (BSTs) do not have a load factor to consider. This makes them an ideal choice for phone directories where the size of the dataset may vary over time, and dynamic operations are frequent. Therefore, it is confidently argued that BSTs are a more efficient and effective data structure for such scenarios.

Simple Implementation and Maintenance:

structure for implementing a phone directory. The inherent properties of a BST, such as its sorted nature and easy-to-follow structure, make it an ideal choice for managing phone directory data. Additionally, the straightforward implementation of a BST makes the codebase more manageable and easier to maintain in the long run, making it a preferred choice over hash functions and collision handling. Therefore, it is safe to say that implementing a BST for a phone directory is a confident and logical decision.

Why we choose BST and not Heap ?

Ordered Data:

BST is a better option than Heap when maintaining order based on keys is of utmost importance. While Heap is efficient in finding and removing maximum or minimum elements, it does not enforce any particular order, making it less useful in the phone directory scenario where order is a crucial factor. Therefore, to traverse data in a specific order, BST would be the ideal choice.

Ordered Insertions and Deletions:

BST allows efficient and arbitrary insertion and deletion operations, though balancing may be required. On the other hand, heaps efficiently handle the insertion and deletion of the maximum (or minimum) element and use less space but do not enforce a specific order and efficient support for arbitrary insertions and deletions. So, depending on the priorities of the phone directory, such as quick searches or maintaining a specific order, BST is preferred.

Why we choose SET and not any other Data Structure?

The main purpose behind using the SET data structure for the phone directory project is its uniqueness property. As no person could have two same phone numbers or two persons cannot have same number so incorporating the uniqueness property is essential that is why Set data structure is an intelligent choice in this regard.

Time Consumed:

Investigation: 4-5 hours

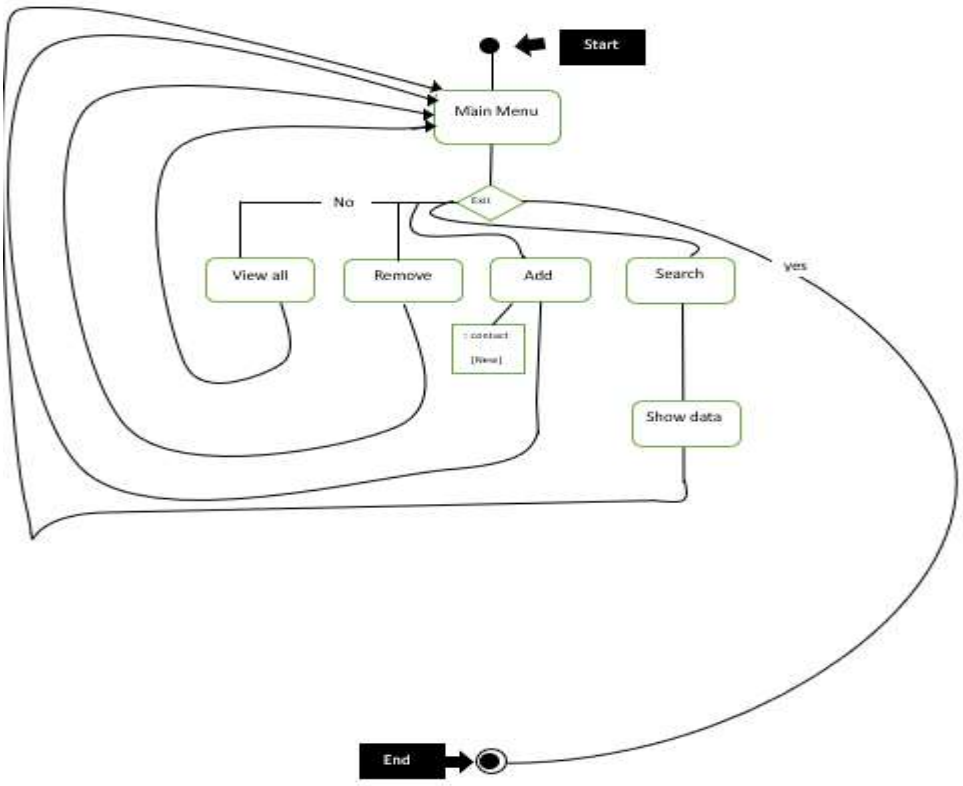
Experimentation: 2 hours

Design: 2 hours

Documentation: 3 hours

Coding: 6 hours

Working/User-Interaction:



Algorithms:

Insertion in BST:

Algorithm InsertContact(ContactName, ContactCity, phone_list):

    newNode = CreateNode(ContactName, ContactCity, phone_list)

    InsertNode(ContactName, ContactCity, phone_list, root)

Algorithm InsertNode(ContactName, ContactCity, phone_list, root):

    Input: ContactName, ContactCity, phone_list (set of phone numbers), root (current node in the BST)

    if root is NULL:

        // Create a new node and insert the data

        newNode = CreateNode(ContactName, ContactCity, phone_list)

        root = newNode

    else:

        if ContactName < root.name:

            // Insert in the left subtree

            InsertNode(ContactName, ContactCity, phone_list, root.left)

        else if ContactName > root.name:

            // Insert in the right subtree

            InsertNode(ContactName, ContactCity, phone_list, root.right)

Searching:

```
Algorithm SearchContact(ContactName):

Input: ContactName


value = SearchNode(ContactName, root)
   if value is NULL:
      DisplayContactNotFoundMessage()
   else:
      DisplayContactInformation(value)


Algorithm SearchNode(ContactName, root):
   Input: ContactName, root (current node in the BST)


   if root is NULL:
      return NULL
   else:
      if ContactName < root.name:
         // Check the left subtree (eliminate half elements from searching)
         return SearchNode(ContactName, root->left)
      else if ContactName > root->name:
```

```
        // Check the right subtree (eliminate half elements from searching)
        return SearchNode(ContactName, root.right)
    else:
        // Found the node with matching ContactName
        return root
```

Removing Contact:

```
Algorithm RemoveContact(ContactName):
  Input: ContactName

  RemoveNode(ContactName, root)

Algorithm RemoveNode(ContactName, root):
  Input: ContactName, root (current node in the BST)

  if root is not NULL:
    if ContactName < root->name:
      // Remove from the left sub-tree
      RemoveNode(ContactName, root->left)
    else if ContactName > root->name:
      // Remove from the right sub-tree
```

```
        RemoveNode(ContactName, root->right)
    else:
        // Two-child scenario
        if root.left is not NULL and root.right is not NULL:
            root.name = FindMin(root.right).name
            RemoveNode(root.name, root.right)
        else:
            // No child or one-child scenario
            oldNode = root
            root = (root.left is not NULL) ? root.left : root.right
            DeleteNode(oldNode)

Algorithm FindMin(root):
  Input: root (current node in the BST)

  if root is not NULL:
    while root.left is not NULL:
      root = root.left
  return root
```

Delete Node:

```
Algorithm DeleteNodes(root):
    Input: root (node to be deleted)


    if root is not NULL:
        DeleteNodes(root.left)          // Recursive call for the left subtree
        DeleteNodes(root.right)         // Recursive call for the right subtree
        Delete root
```

Add to Set:

```
Algorithm AddToSet( element):
    Input: element (data to be added)


    if not Contains( element):
        newNode = CreateNode(element)
        if set->head is NULL:
            set->head = newNode
        else:
            currentNode = set.head
            while currentNode->next is not NULL:
                currentNode = currentNode->next
```

```
        currentNode->next = newNode
    else:
        DisplayDuplicateElementMessage()
```

```
Algorithm Contains( element):
    Input: element (data to be checked)

    currentNode = set->head
    while currentNode is not NULL and currentNode.element is not equal to element:
        currentNode = currentNode->next
    if currentNode is not Null
return true
else
return false
```

Remove from set:

```
Procedure remove(x)
    // Input: x - element to be removed from the set
    // Output: None
```

```
// Initialize pointers
previous = null
curr = head

// Traverse the set until the element is found or end is reached
while curr is not null and curr.element is not equal to x:
    previous = curr
    curr = curr.next

// If the element is found, remove it
if curr is not null:
    previous.next = curr.next   // Skip the current node in the linked list
    delete curr                 // Deallocate memory for the current node
```

Display of Set:

```
Procedure display()
    // Output: Display data stored in the set

    // Initialize variables
    i = 1
```

```
curr = head


// Traverse the set and display each element
while curr is not null:
    output "Phone No " + i + ": " + curr.element  // Display the current element
    curr = curr.next  // Move to the next node
    i = i + 1  // Increment the sequence number
```

Analysis:

After the analysis of data structures and the algorithm following are the time and space complexities.

- Insertion and deletion operations also have an average time complexity of O(log n), ensuring efficient updates to the tree.
- BST allows efficient search operations with an average time complexity of O(log n).
- BST worst case will take O(N).
- SET insertion and deletion will take O(N)
- SET searching will take O(N)
- Space complexity for BST is O(N)
- Space complexity for SET is O(N).

PROGRAM CODE

BST.h:

```
#ifndef BST_H

#define BST_H

#include <string>

#include <set.h>

using namespace std;

class BST                  //This BST is specific for the phone directory so its structure node is modified so template can't be used

{

   public:



      //BST ADT  (Phone Directory Interface)

      BST();

      virtual ~BST();

      void search_contact(string ContactName);

      void insert_contact(string ContactName,string ContactCity ,set<long long> phone_list);

      void remove_contact(string ContactName);

      void print_contacts();
```

```
    private:
        struct node
        {
            string name;
            set<long long> phone_store;
            string city;
            node* left;
            node* right;
        };
        node*root;

        //Functions to create the layer of Abstraction
        void  remove_node( string name,node*& root);
        node* search_node(string name,node*& root);
        node* findMin(node* root);
        void insert_node(string name,string city,set<long long> phone_store,node*& root);
        void InorderTraversal(node*& root);
        void delete_nodes(node*& root);     // to free up space
};
```

```
#endif // BST_H
```

BST.cpp:

```
#include "BST.h"

#include <iostream>

#include <string>

#include <set.h>


using namespace std;


BST::BST() {

   root = NULL;              // Initialization of BST (Phone Directory)

}


BST::~BST() {

   delete_nodes(root);       // Free up space

}


void BST::search_contact(string ContactName) {

   node* value = search_node(ContactName, root);          // Searching function will return the required person's data

   if (value == NULL) {
```

```
        cout<<"\nContact not found\n";

    } else {

        cout<<"Name : "<<root->name<<"\n";

        cout<<"City : "<<root->city;

        root->phone_store.display();                    // Display function of Set ADT that will display
the data stored in set  (Phone Numbers)

        cout<<endl;



    }

}



void BST::insert_contact(string ContactName, string ContactCity,set<long long> phone_list)
{

    insert_node(ContactName,ContactCity,phone_list, root);

}

void BST::print_contacts()

{

    InorderTraversal(root);

}

BST::node* BST::search_node(string ContactName, node*& root) {            // Core Searching
function

    if (root == NULL) {
```

```
        return NULL;

    } else {

        if (ContactName < root->name) {                         // Check the left subtree (eliminate
half elements from searching)

            return search_node(ContactName, root->left);

        } else if (ContactName > root->name) {                  // Check the right subtree
(eliminate half elements from searching)

            return search_node(ContactName, root->right);

        } else {

            return root;

        }

    }

}
```

```
void BST::insert_node(string ContactName,string ContactCity,set<long long> phone_list,
node*& root) {      //Core Insertion Function

    if (root == NULL) {

        root = new node;

        root->name = ContactName;

        root->city = ContactCity;

        root->phone_store=phone_list;
```

```
      root->left = NULL;

      root->right = NULL;

   } else {

      if (ContactName < root->name)                                    // insertion in the left
subtree



         {

         insert_node(ContactName,ContactCity,phone_list, root->left);

         }

      else if (ContactName > root->name)                                 // insertion in the
right subtree

      {

         insert_node(ContactName,ContactCity,phone_list, root->right);

      }

   }

}



void BST::remove_contact(string ContactName) {

   remove_node(ContactName, root);

}
```

```
BST::node* BST::findMin(node* root) {                    // find the minimum element (used in remove node step)
   if (root != NULL) {
      findMin(root->left);
   } else {
      return root;
   }
}

void BST::remove_node(string ContactName, node*& root) {
   if (root != NULL) {
      if (ContactName < root->name) {                    // remove from the left sub-tree
         remove_node(ContactName, root->left);
      } else if (ContactName > root->name) {             // remove from the right sub-tree
         remove_node(ContactName, root->right);
      } else {                                           // Two child scenario
         if (root->left != NULL && root->right != NULL) {
            root->name = findMin(root->right)->name;
            remove_node(root->name, root->right);
         } else {                                        //no child or one child scenario
            node* oldNode = root;
```

```
            root = (root->left != NULL) ? root->left : root->right;
            delete oldNode;
          }
       }
    }
}

void BST::delete_nodes(node*& root) {          //delete the whole tree by recursive calls
   if (root != NULL) {
      delete_nodes(root->left);              //recursive calls
      delete_nodes(root->right);
      delete root;
   }
}
void BST::InorderTraversal(node* & x)          // Displaying data in tree in In-order traversal
{
   if (x!=NULL)
   {
      InorderTraversal(x->left);
      cout<<"NAME : "<<x->name<<endl<<"CITY : "<<x->city;
      x->phone_store.display();              //display function of set ADT to show Phone numbers
```

```
        cout<<endl<<endl;
        InorderTraversal(x->right);
    }
}
```

Set.h:

```
#ifndef SET_H
#define SET_H


template <class T>              //This set ADT implemented using Template because it contain Single type of data
class set                       // Implementation of the set using the list
{
    public:
        set();
        virtual ~set();
        void add(T x);          // Add data in set without duplication
        void remove(T x);       // Remove data from set
        void display();         // Display the data in the set

    private:
        struct node             // Set data node
```

```
    {
        T element;
        node* next;
    };
    node* head;
    bool contain(T x);      // This function give SET property(uniqueness) to link-list

};
#endif // SET_H
```

Set.cpp:

```
#include "set.h"
#include <iostream>

using namespace std;

template <class T>
set<T>::set()
{
   head =NULL;              // Initialization of set
}
```

```
template <class T>

set<T>::~set()                // Destructor to free up the space

{

   node* previous;

   node* curr = head;

   while(curr!=NULL)

   {

      previous=curr;

      curr=curr->next;

      delete previous;

   }

}



template <class T>

void set<T>::add(T x)             // Implementation of insertion function in the set ADT (maintaining the set property)

{

   if(!contain(x))              // This will give the set property (uniqueness) by preventing duplication

   {

      node* new_node= new node;

      new_node->element=x;
```

```
    if(head==NULL)
    {
       new_node->next=NULL;
       head=new_node;
    }
    else
    {
       node* curr = head;
       while(curr->next!=NULL)
       {
          curr=curr->next;
       }
       new_node->next=NULL;
       curr->next=new_node;
    }
}
else
{
  cout<<"\n!!!!!!!!!! You Entered Duplicate Number !!!!!!!!!\n";
}
```

```
}


template <class T>
void set<T>::remove(T x)          // Implementation of deletion function in the set ADT
{
   node* previous;
   node* curr=head;
   while(curr!=NULL && curr->element!=x)      // Check for the required object is found that is to be removed
   {
      previous=curr;
      curr=curr->next;;
   }
   if(curr!=NULL)
   {
      previous->next=curr->next;
      delete curr;
   }
}


template <class T>
```

```
void set<T>::display()                    // Display data stored in set

{



   int i=1;

   node* curr=head;

   while(curr!=NULL)

   {

      cout<<endl<<"Phone No "<<i<<": "<<curr->element;       // i is specifically used for the case of Phone Directory to show the sequence of phone numbers

      curr=curr->next;

      i++;

   }

}



template <class T>

bool set<T>::contain(T x)                 // The core function that will grant the set property (uniqueness) by checking already present data

{

   node* curr =head;

   while(curr!=NULL && curr->element!=x)

   {

      curr=curr->next;
```

```
    }
    if(curr!=NULL)
    {
        return true;
    }
    else
        return false;
}
```

Set_Implementation.cpp:

```
#include <iostream>
#include <string>
#include "set.cpp"

using namespace std;

//These data types are allowed to be used in SETs due to template
template class set<int>;
template class set<long long>;
template class set<float>;
template class set<double>;
template class set<string>;
```

```
template class set<char>;
```

Main.cpp:

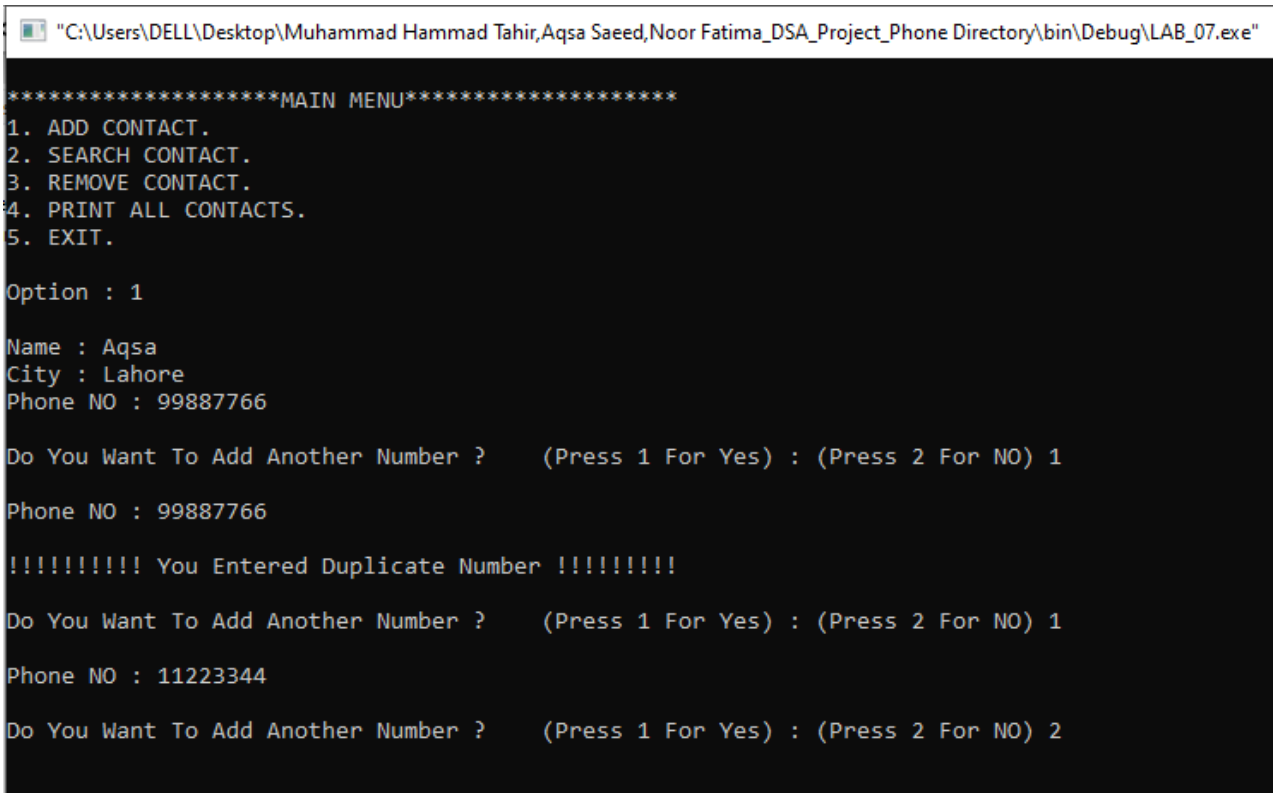
```
#include <iostream>

#include <string.h>

#include <cstdlib>

#include <BST.h>

#include <set.h>


using namespace std;


int main()

{

   BST PhoneDirectory;                          //BST object created

   char opt, op='1';

   string ContactName,ContactCity;

   long long phoneno;

   while(1)

   {

      cout<<"\n********************MAIN MENU********************\n";

      cout<<"1. ADD CONTACT.\n2. SEARCH CONTACT.\n3. REMOVE CONTACT.\n4. PRINT
ALL CONTACTS.\n5. EXIT.\n";
```

```
        cout<<"\nOption : ";
        cin>>opt;
        cout<<endl;
        set<long long> phone_list;
        switch(opt)
        {
        case '1' :
           cout<<"Name : ";
           cin.ignore();                          // To clear the Input stream
           std::getline(std::cin, ContactName);        //To take the Full name (multiple strings as input)
           cout<<"City : ";
           cin>>ContactCity;
           while(op!='2')
           {
              cout<<"Phone NO : ";
              cin>>phoneno;
              phone_list.add(phoneno);
              cout<<"\nDo You Want To Add Another Number ?    (Press 1 For Yes) : (Press 2 For NO) ";
              cin>>op;
              cout<<endl;
```

```
        }
        PhoneDirectory.insert_contact(ContactName,ContactCity,phone_list);     //Insertion In
BST
        op='1';
        break;
      case '2' :
        cout<<"Name : ";
        cin>>ContactName;
        cout<<endl;
        PhoneDirectory.search_contact(ContactName);       //Searching in BST
        break;
      case '3' :
        cout<<"Name : ";
        cin>>ContactName;
        PhoneDirectory.remove_contact(ContactName);       // Deletion in BST
        break;
      case '4' :
        PhoneDirectory.print_contacts();              //Print Tee
        break;
      case '5' :
        exit(0);
```

```
    }
  }
}
```

OUTPUT:


"C:\Users\DELL\Desktop\Muhammad Hammad Tahir,Aqsa Saeed,Noor Fatima_DSA_Project_Phone Directory\bin\Debug\LAB_07.exe"
**********MAIN MENU**********
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.
Option : 1
Name : Aqsa
City : Lahore
Phone NO : 99887766
Do You Want To Add Another Number ? (Press 1 For Yes) : (Press 2 For NO) 1
Phone NO : 99887766
!!!!!!!!!! You Entered Duplicate Number !!!!!!!!!
Do You Want To Add Another Number ? (Press 1 For Yes) : (Press 2 For NO) 1
Phone NO : 11223344
Do You Want To Add Another Number ? (Press 1 For Yes) : (Press 2 For NO) 2

```
********************MAIN MENU**********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 1

Name : noor
City : sahiwal
Phone NO : 22113355

Do You Want To Add Another Number ?    (Press 1 For Yes) : (Press 2 For NO) 2


********************MAIN MENU**********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 2

Name : hammad


Contact not found
```

```
********************MAIN MENU**********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 3

Name : noor

********************MAIN MENU**********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 1

Name : hammad
City : faislabad
Phone NO : 3366775522

Do You Want To Add Another Number ?    (Press 1 For Yes) : (Press 2 For NO) 2
```

```
********************MAIN MENU*********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 4

NAME : Aqsa
CITY : Lahore
Phone No 1: 99887766
Phone No 2: 11223344

NAME : hammad
CITY : faislabad
Phone No 1: 3366775522
```

```
********************MAIN MENU*********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 1

Name : noor
City : sahiwal
Phone NO : 332212112

Do You Want To Add Another Number ?    (Press 1 For Yes) : (Press 2 For NO) 2


********************MAIN MENU*********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 4

NAME : Aqsa
CITY : Lahore
Phone No 1: 99887766
Phone No 2: 11223344
```

```
NAME : hammad
CITY : faislabad
Phone No 1: 3366775522

NAME : noor
CITY : sahiwal
Phone No 1: 332212112


********************MAIN MENU********************
1. ADD CONTACT.
2. SEARCH CONTACT.
3. REMOVE CONTACT.
4. PRINT ALL CONTACTS.
5. EXIT.

Option : 5


Process returned 0 (0x0)   execution time : 156.339 s
Press any key to continue.
```