Experiment A1

```
# Program to implement Hashing with Linear Probing
from Record import Record
class hashTable:
    # initialize hash Table
    def _init_(self):
        self.size = int(input("Enter the Size of the hash table : "))
        # initialize table with all elements 0
        self.table = list(None for i in range(self.size))
        self.elementCount = 0
        self.comparisons = 0
    # method that checks if the hash table is full or not
    def isFull(self):
        if self.elementCount == self.size:
            return True
        else:
            return False
    # method that returns position for a given element
    def hashFunction(self, element):
        return element % self.size
    # method that inserts element into the hash table
    def insert(self, record):
        # checking if the table is full
        if self.isFull():
            print("Hash Table Full")
            return False
        isStored = False
        position = self.hashFunction(record.get_number())
        # checking if the position is empty
        if self.table[position] == None:
            self.table[position] = record
            print("Phone number of " + record.get_name() + " is at position " + str(position))
            isStored = True
            self.elementCount += 1
        # collision occured hence we do linear probing
        else:
            print("Collision has occured for " + record.get_name() + "'s phone number at position
" + str(
                position) + " finding new Position.")
            while self.table[position] != None:
                position += 1
                if position >= self.size:
                    position = 0
            self.table[position] = record
            print("Phone number of " + record.get_name() + " is at position " + str(position))
            isStored = True
            self.elementCount += 1
        return isStored
    # method that searches for an element in the table
    # returns position of element if found
    # else returns False
    def search(self, record):
        found = False
        position = self.hashFunction(record.get_number())
        self.comparisons += 1
        if (self.table[position] != None):
            if (self.table[position].get_name() == record.get_name() and self.table[
                position].get_number() == record.get_number()):
                isFound = True
                print("Phone number found at position {} ".format(position) + " and total
comparisons are " + str(1))
                return position
            # if element is not found at position returned hash function
            else:
                position += 1
                if position >= self.size - 1:
```

```
            position = 0
        while self.table[position] != None or self.comparisons <= self.size:
            if (self.table[position].get_name() == record.get_name() and self.table[
                position].get_number() == record.get_number()):
                isFound = True
                # i=0
                i = self.comparisons + 1
                print(
                    "Phone number found at position {} ".format(position) + " and total
comparisons are " + str(
                        i))
                return position
            position += 1
            # print(position)
            if position >= self.size - 1:
                position = 0
            # print(position)
            self.comparisons += 1
            # print(self.comparisons)
        if isFound == False:
            print("Record not found")
            return false


    # method to display the hash table
    def display(self):
        print("\n")
        for i in range(self.size):
            print("Hash Value: " + str(i) + "\t\t" + str(self.table[i]))
        print("The number of phonebook records in the Table are : " + str(self.elementCount))
```

2. DoubleHashing.py

```
from Record import Record
class doubleHashTable:
    # initialize hash Table
    def _init_(self):
        self.size = int(input("Enter the Size of the hash table : "))
        # initialize table with all elements 0
        self.table = list(None for i in range(self.size))
        self.elementCount = 0
        self.comparisons = 0
    # method that checks if the hash table is full or not
    def isFull(self):
        if self.elementCount == self.size:
            return True
        else:
            return False
    # First hash function
    def h1(self, element):
        return element % self.size
    # Second hash function
    def h2(self, element):
        return 5 - (element % 5)

    # method to resolve collision by double hashing method
    def doubleHashing(self, record):
        posFound = False
        # limit variable is used to restrict the function from going into infinite loop
        # limit is useful when the table is 80% full
        limit = self.size
        i = 1
        # start a loop to find the position
        while i <= limit:
            # calculate new position by quadratic probing
            newPosition = (self.h1(record.get_number()) + i * self.h2(record.get_number())) %
self.size
            # if newPosition is empty then break out of loop and return new Position
            if self.table[newPosition] == None:
```

```
            posFound = True
            break
        else:
            # as the position is not empty increase i
            i += 1
    return posFound, newPosition
# method that inserts element inside the hash table
def insert(self, record):
    # checking if the table is full
    if self.isFull():
        print("Hash Table Full")
        return False
    posFound = False
    position = self.h1(record.get_number())
    # checking if the position is empty
    if self.table[position] == None:
        # empty position found , store the element and print the message
        self.table[position] = record
        print("Phone number of " + record.get_name() + " is at position " + str(position))
        isStored = True
        self.elementCount += 1
    # If collision occured
    else:
        print("Collision has occured for " + record.get_name() + "'s phone number at position
" + str(
            position) + " finding new Position.")
        while not posFound:
            posFound, position = self.doubleHashing(record)
            if posFound:
                self.table[position] = record
                # print(self.table[position])
                self.elementCount += 1
                # print(position)
                # print(posFound)
                print("Phone number of " + record.get_name() + " is at position " + str(position))
    return posFound

# searches for an element in the table and returns position of element if found else
returns False
def search(self, record):
    found = False
    position = self.h1(record.get_number())
    self.comparisons += 1
    if (self.table[position] != None):
        if (self.table[position].get_name() == record.get_name()):
            print("Phone number found at position {}".format(position) + " and total
comparisons are " + str(1))
            return position

        # if element is not found at position returned hash function
        # then we search element using double hashing
        else:
            limit = self.size
            i = 1
            newPosition = position
            # start a loop to find the position
            while i <= limit:
                # calculate new position by double Hashing
                position = (self.h1(record.get_number()) + i * self.h2(record.get_number())) %
self.size
                self.comparisons += 1
                # if element at newPosition is equal to the required element

                if (self.table[position] != None):
                    if self.table[position].get_name() == record.get_name():
                        found = True
                        break
                    elif self.table[position].get_name() == None:
                        found = False
                        break
                    else:
                        # as the position is not empty increase i
                        i += 1
        if found:
            print("Phone number found at position {}".format(position) + " and total
comparisons are " + str(i + 1))
        # return position
        else:
            print("Record not Found")
            return found

            # method to display the hash table
def display(self):
    print("\n")
    for i in range(self.size):
        print("Hash Value: " + str(i) + "\t\t" + str(self.table[i]))
    print("The number of phonebook records in the Table are : " + str(self.elementCount))
```

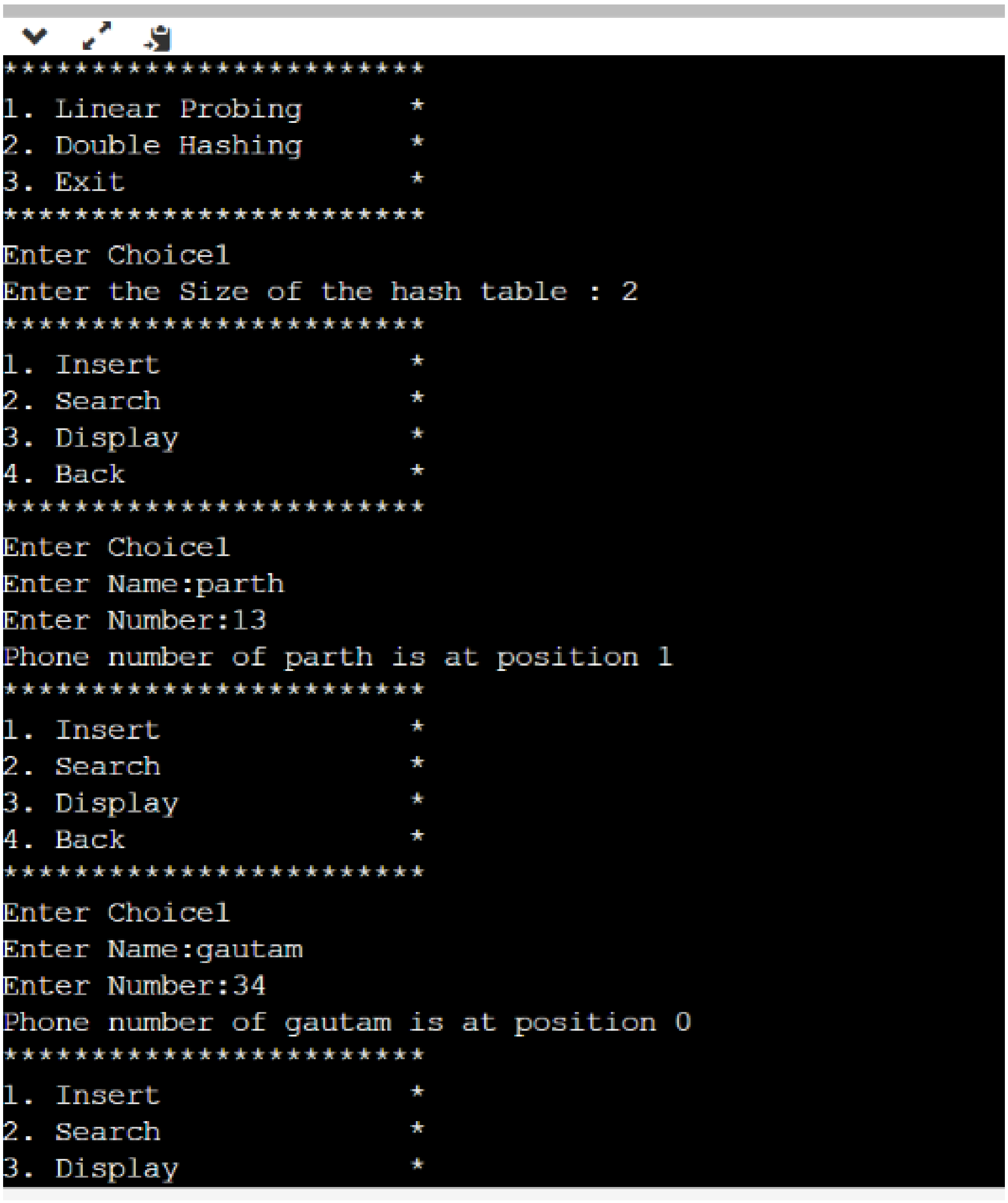
3.Record.py

```
class Record:
    def _init_(self):
        self._name = None
        self._number = None
    def get_name(self):
        return self._name
    def get_number(self):
        return self._number
    def set_name(self,name):
        self._name = name
    def set_number(self,number):
        self._number = number
    def _str_(self):
        record = "Name: "+str(self.get_name())+"\t"+"\tNumber: "+str(self.get_number())
        return record
```

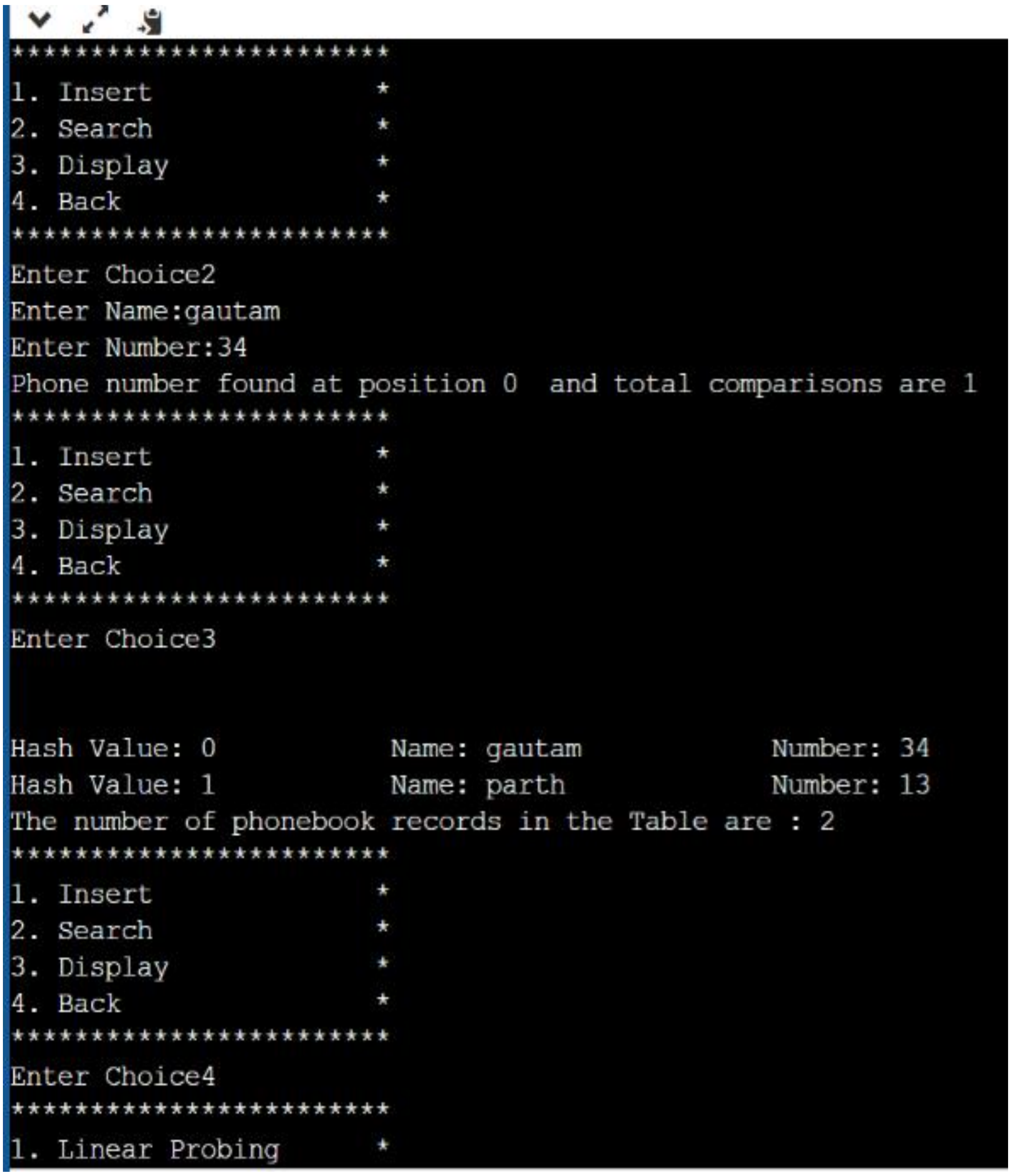
4. main.py

```
from LinearProbing import hashTable
from Record import Record
from DoubleHashing import doubleHashTable
def input_record():
    record = Record()
    name = input("Enter Name:")
    number = int(input("Enter Number:"))
    record.set_name(name)
    record.set_number(number)
    return record
choice1 = 0
while (choice1 != 3):
    print("********")
    print("1. Linear Probing      *")
    print("2. Double Hashing      *")
    print("3. Exit                *")
    print("********")
    choice1 = int(input("Enter Choice"))
    if choice1 > 3:
        print("Please Enter Valid Choice")
    if choice1 == 1:
        h1 = hashTable()
        choice2 = 0
        while (choice2 != 4):
            print("********")
            print("1. Insert              *")
            print("2. Search              *")
            print("3. Display             *")
            print("4. Back                *")
            print("********")
            choice2 = int(input("Enter Choice"))
            if choice2 > 4:
                print("Please Enter Valid Choice")
            if (choice2 == 1):
                record = input_record()
                h1.insert(record)


            elif (choice2 == 2):
                record = input_record()
                position = h1.search(record)
            elif (choice2 == 3):
                h1.display()
    elif choice1 == 2:
        h2 = doubleHashTable()
        choice2 = 0
        while (choice2 != 4):
            print("*********")
            print("1. Insert              *")
            print("2. Search              *")
            print("3. Display             *")
            print("4. Back                *")
            print("*********")
            choice2 = int(input("Enter Choice"))
            if choice2 > 4:
                print("Please Enter Valid Choice")
            if (choice2 == 1):
                record = input_record()
                h2.insert(record)
            elif (choice2 == 2):
                record = input_record()
                position = h2.search(record)
            elif (choice2 == 3):
                h2.display()
```

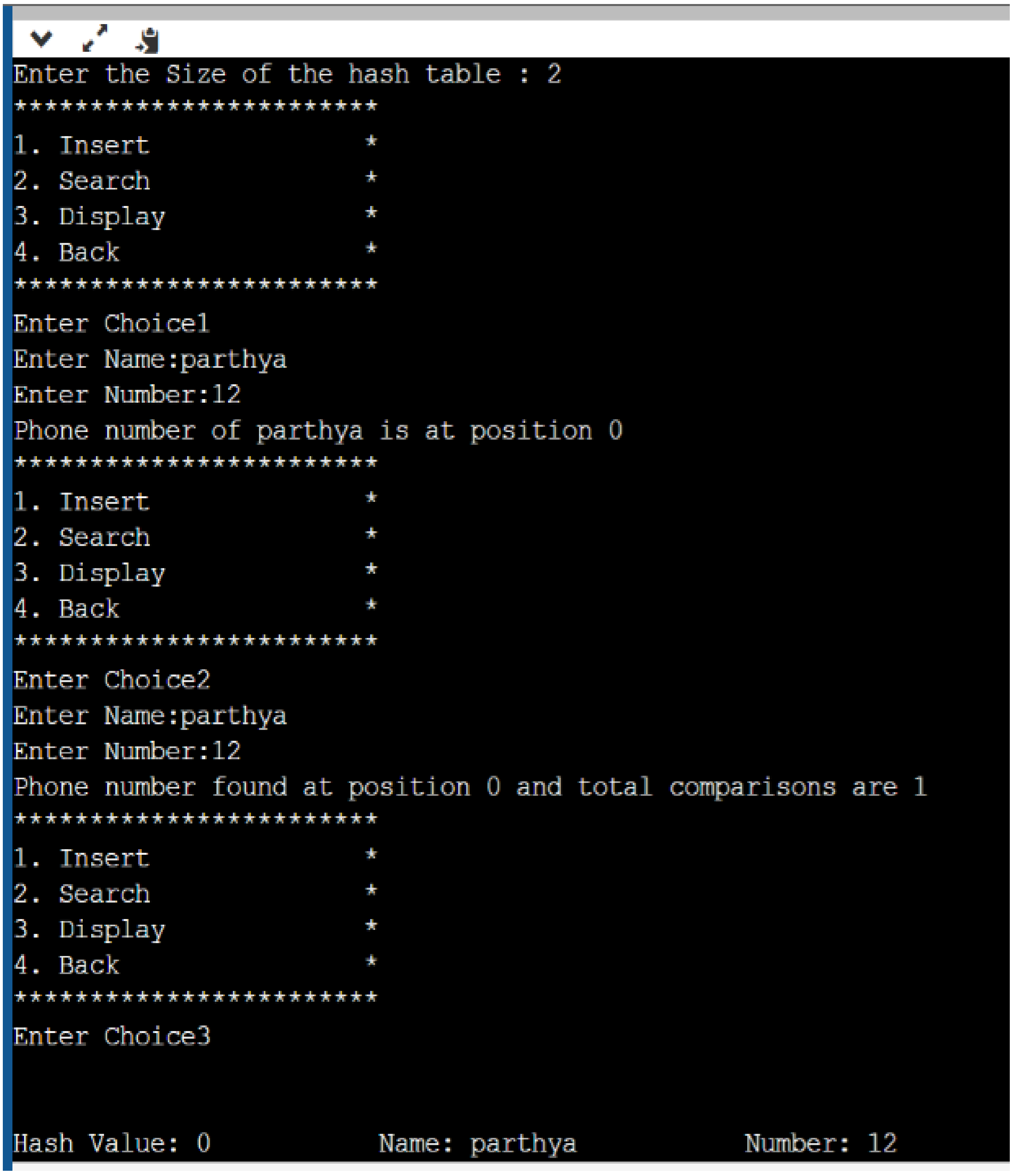
OUTPUT:



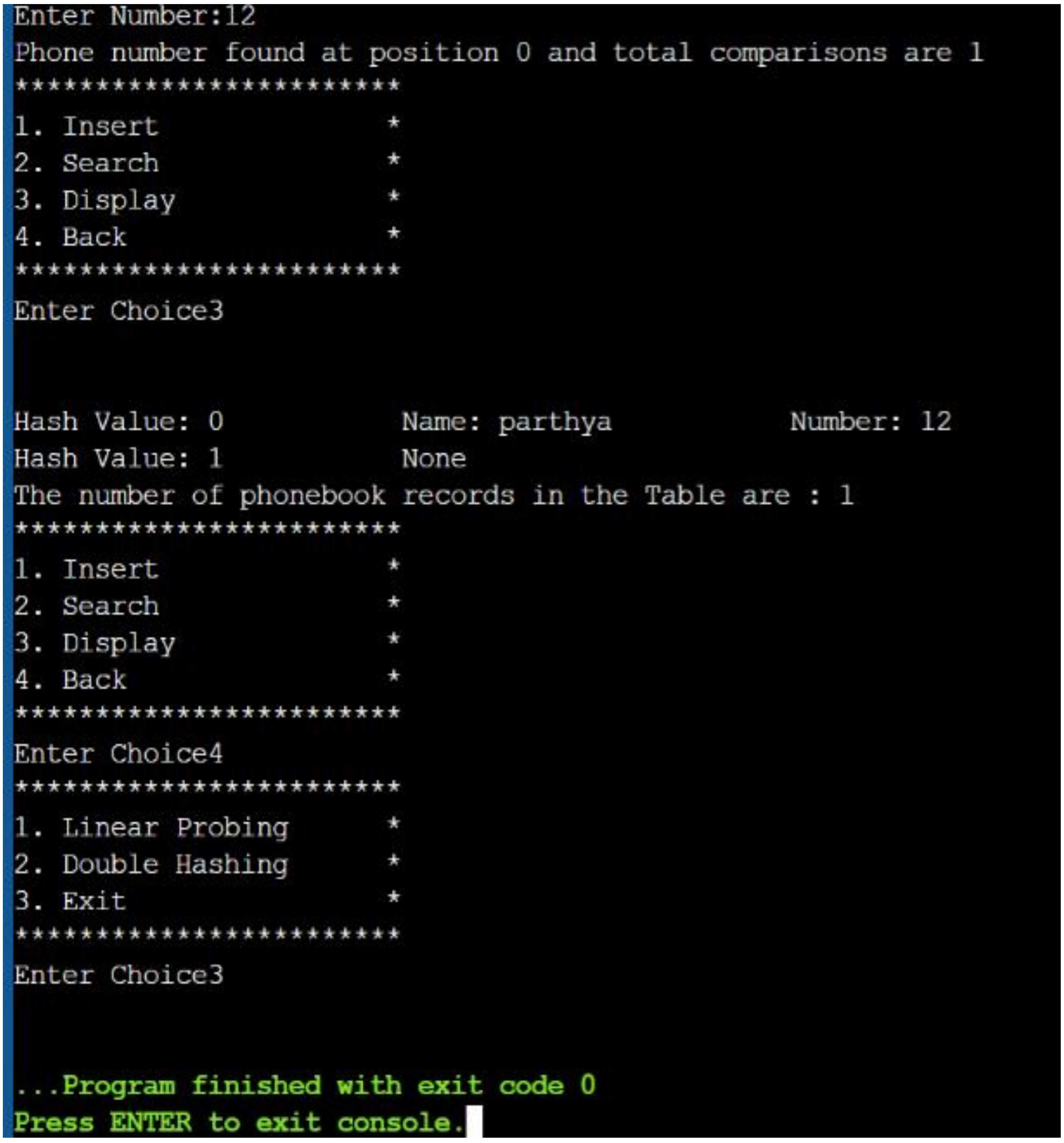
```
************************
1. Linear Probing      *
2. Double Hashing      *
3. Exit                *
************************
Enter Choice1
Enter the Size of the hash table : 2
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice1
Enter Name:parth
Enter Number:13
Phone number of parth is at position 1
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice1
Enter Name:gautam
Enter Number:34
Phone number of gautam is at position 0
************************
1. Insert              *
2. Search              *
3. Display             *
```



```
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice2
Enter Name:gautam
Enter Number:34
Phone number found at position 0  and total comparisons are 1
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice3


Hash Value: 0           Name: gautam            Number: 34
Hash Value: 1           Name: parth             Number: 13
The number of phonebook records in the Table are : 2
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice4
************************
1. Linear Probing      *
```



```
Enter the Size of the hash table : 2
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice1
Enter Name:parthya
Enter Number:12
Phone number of parthya is at position 0
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice2
Enter Name:parthya
Enter Number:12
Phone number found at position 0 and total comparisons are 1
************************
1. Insert              *
2. Search              *
3. Display             *
4. Back                *
************************
Enter Choice3


Hash Value: 0           Name: parthya           Number: 12
```



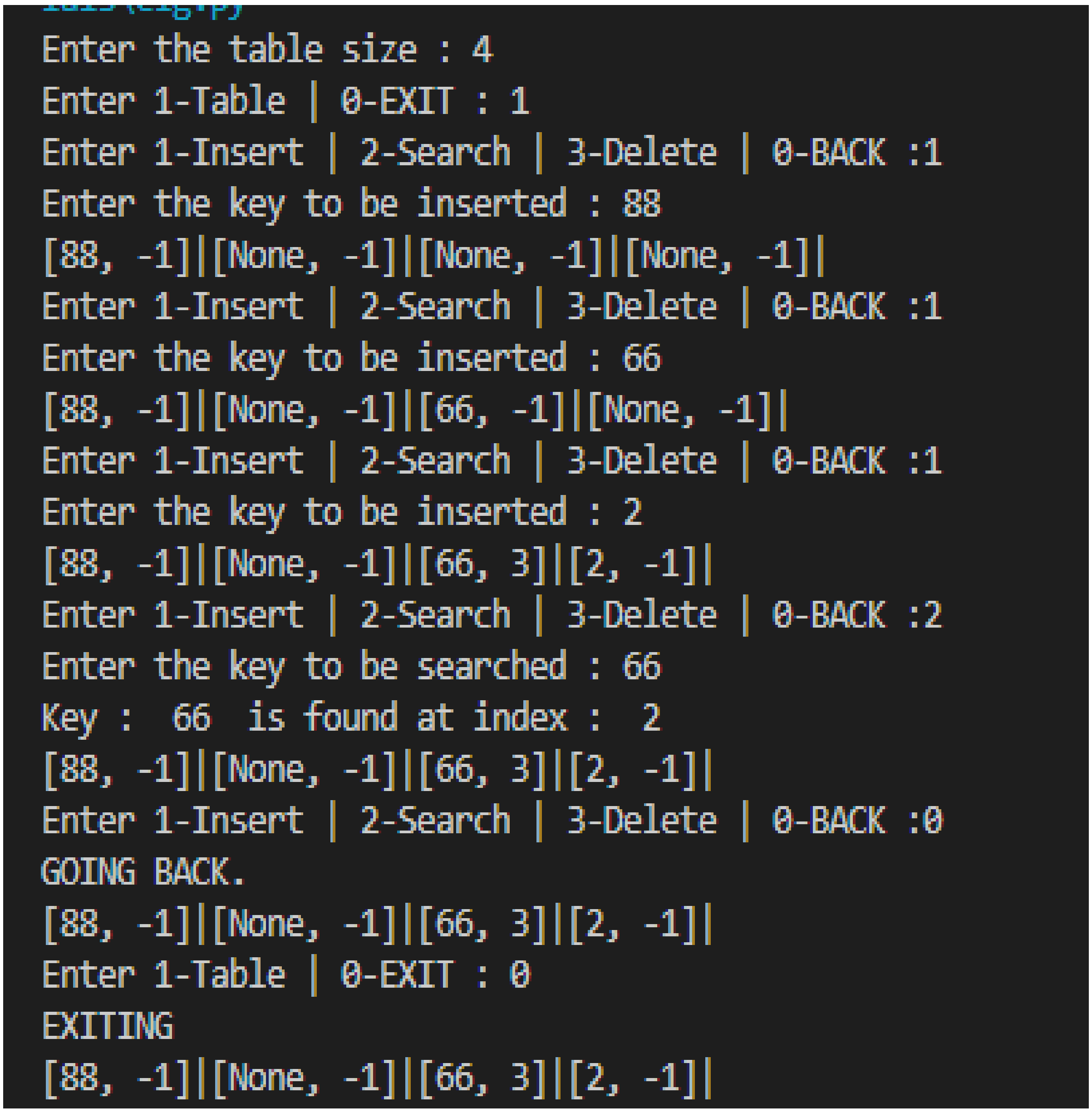
```
Enter Number:12
Phone number found at position 0 and total comparisons are 1
************************
1. Insert               *
2. Search               *
3. Display              *
4. Back                 *
************************
Enter Choice3


Hash Value: 0           Name: parthya           Number: 12
Hash Value: 1           None
The number of phonebook records in the Table are : 1
************************
1. Insert               *
2. Search               *
3. Display              *
4. Back                 *
************************
Enter Choice4
************************
1. Linear Probing       *
2. Double Hashing       *
3. Exit                 *
************************
Enter Choice3


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment A2

```
table = []
b,totl = 0,0
bucket = {}
def create():
    global b
    b = int(input("Enter the table size : "))
    for i in range(b):
        table.append([None,-1])
        bucket[i] = -1
def printtable():
 global b
 for i in range(b):
  print(table[i],end="|")
 print("")
def chaininsert(key):
    global b,totl
    hash = key%b
    if (table[hash][0]==None):
        table[hash][0] = key
        bucket[key%b] = hash
    else:
        flag = 0
        for i in range(0,b):
            hash = (key+i)%b
            if (table[hash][0]==None):
                totl += 1
                flag = 1
                if bucket[key%b]!=1:
                    table[bucket[key%b]][1] = hash
                bucket[key%b] = hash
                table[hash][0] = key
                break
        if(flag==0):
            print("Key : ",key," not inserted - table full .")
def chainsearch(key):
    global b
    hash = key%b
    if (table[hash][0]==key):
        print("Key : ",key," is found at index : ",hash)
    else:
        flag,i,chain = 0,0,table[hash][1]
        while(table[hash][0]!=None and table[hash][0]%b != key%b):
            hash = (key+i)%b
            chain = table[hash][1]
            if (table[hash][0]==key):
                print("Key : ",key," is found at index : ",hash)
                chain = -1
                flag = 1
                break
            i += 1
        while(chain!=-1):
            if (table[chain][0]==key):
                print("Key : ",key," is found at index : ",chain)
                flag = 1
                break
            chain = table[chain][1]
        if(flag==0):
            print("Key : ",key," not found.")
def chaindelete(key):
    global b
    hash = key%b
    if (table[hash][0]==key):
        table[hash][0],table[hash][1] = None,-1
        print("Key : ",key," was deleted from index : ",hash)
    else:
        flag,i,pchain,chain = 0,0,hash,table[hash][1]
        while(table[hash][0]!=None and table[hash][0]%b != key%b):
            hash = (key+i)%b
            pchain = chain
            chain = table[hash][1]
            if (table[hash][0]==key):
                table[pchain][1] = table[chain][1]
                table[chain][0],table[chain][1]=None,-1
                print("Key : ",key," was deleted from index : ",chain)            i += 1
        while(chain!=-1):
            if (table[chain][0]==key):
                table[pchain][1] = table[chain][1]
                table[chain][0],table[chain][1]=None,-1
                print("Key : ",key," was deleted from index : ",chain)
                flag = 1
                break
            pchain = chain
            chain = table[chain][1]
        if(flag==0):
            print("Key : ",key," not found.")
create()
while(1):
    ch = int(input("Enter 1-Table | 0-EXIT : "))
    if ch == 1 :
        while(1):
            ch2 = int(input("Enter 1-Insert | 2-Search | 3-Delete | 0-BACK :"))
            if ch2==1:
                key = int(input("Enter the key to be inserted : "))
                chaininsert(key)
                printtable()
            elif ch2==2:
```

```
            key = int(input("Enter the key to be searched : "))
            chainsearch(key)
            printtable()
        elif ch2==3:
            key = int(input("Enter the key to be searched : "))
            chaindelete(key)
            printtable()
        elif ch2==0:
            print("GOING BACK.")
            printtable()
            break
elif ch == 0:
    print("EXITING")
    printtable()
    break
else:
    printtable()
```

OUTPUT



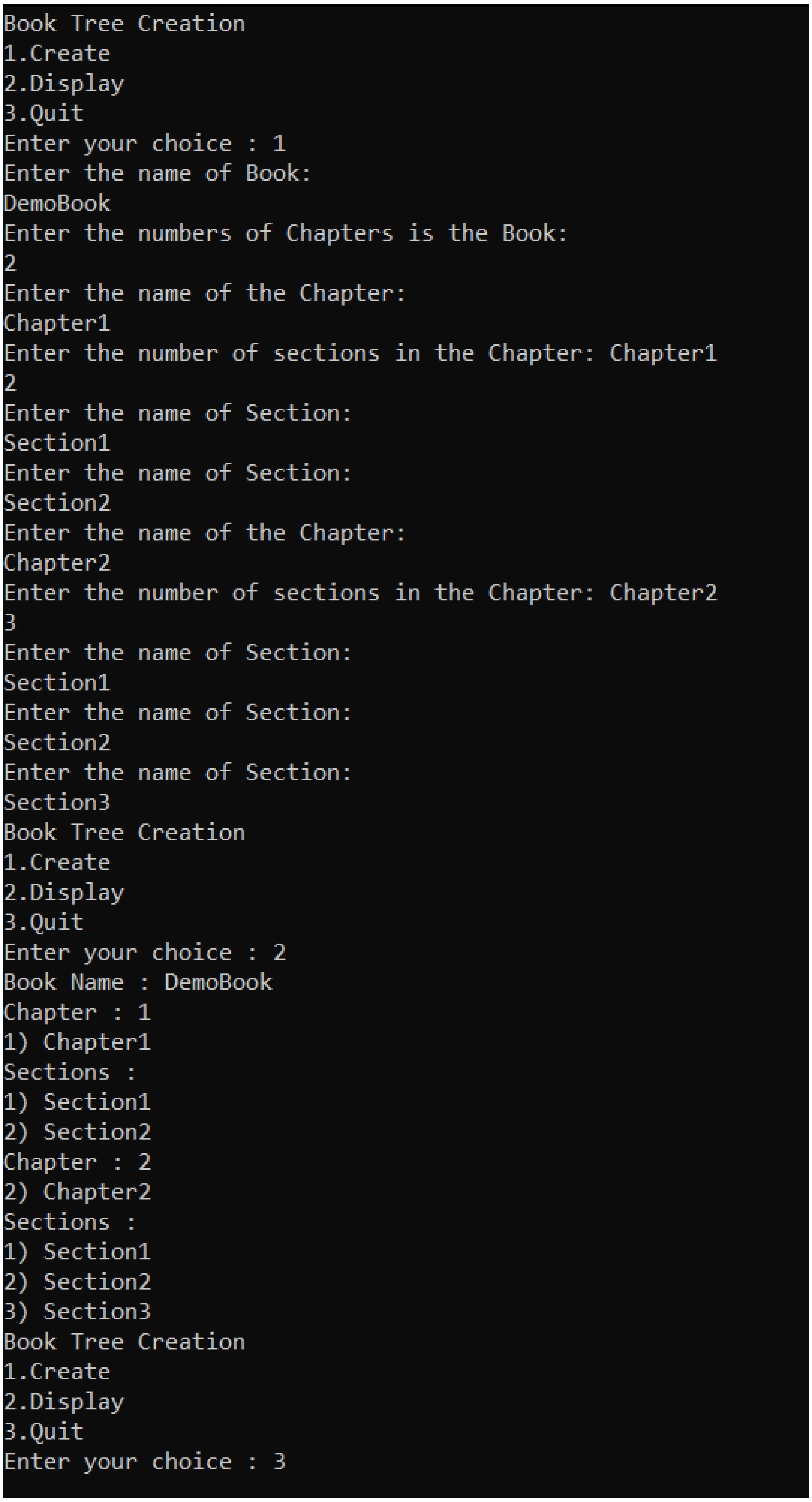
```
Enter the table size : 4
Enter 1-Table | 0-EXIT : 1
Enter 1-Insert | 2-Search | 3-Delete | 0-BACK :1
Enter the key to be inserted : 88
[88, -1]|[None, -1]|[None, -1]|[None, -1]|
Enter 1-Insert | 2-Search | 3-Delete | 0-BACK :1
Enter the key to be inserted : 66
[88, -1]|[None, -1]|[66, -1]|[None, -1]|
Enter 1-Insert | 2-Search | 3-Delete | 0-BACK :1
Enter the key to be inserted : 2
[88, -1]|[None, -1]|[66, 3]|[2, -1]|
Enter 1-Insert | 2-Search | 3-Delete | 0-BACK :2
Enter the key to be searched : 66
Key :  66  is found at index :  2
[88, -1]|[None, -1]|[66, 3]|[2, -1]|
Enter 1-Insert | 2-Search | 3-Delete | 0-BACK :0
GOING BACK.
[88, -1]|[None, -1]|[66, 3]|[2, -1]|
Enter 1-Table | 0-EXIT : 0
EXITING
[88, -1]|[None, -1]|[66, 3]|[2, -1]|
```

Experiment B5

```
/*
A book consists of chapters, chapters consist of sections and sections consist of
subsections.
Construct a tree and print the nodes. Find the time and space requirements of your method.
*/

#include<iostream>
using namespace std;
struct node
{
	char lable[30];
	int count;
	node* child[10];

}*root;

class Book
{
public:
	Book()
	{
		root = NULL;
	}
	void create()
	{
		int chapters, sections;
		root = new node;
		cout << "Enter the name of Book: " << endl;
		cin >> root->lable;
		cout << "Enter the numbers of Chapters is the Book: " << endl;
		cin >> chapters;
		root->count = chapters;
		for (int i = 0; i < chapters; i++) {
			root->child[i] = new node;
			cout << "Enter the name of the Chapter: " << endl;
			cin >> root->child[i]->lable;
			cout << "Enter the number of sections in the Chapter: " << root-
>child[i]->lable << endl;
			cin >> sections;
			root->child[i]->count = sections;
			for (int j = 0; j < root->child[i]->count; j++)
			{

				root->child[i]->child[j] = new node;
				cout << "Enter the name of Section: " << endl;
				cin >> root->child[i]->child[j]->lable;
```

```
                }
            }
        }
        void display(node* t)
        {
            if (t != NULL)
            {
                cout << "Book Name : " << root->lable << endl;
                int cha = root->count;
                for (int i = 0; i < cha; i++) {

                    cout << "Chapter : " << i + 1 << endl;
                    cout << i + 1 << ") " << root->child[i]->lable << endl;
                    cout << "Sections : " << endl;
                    int sec = root->child[i]->count;
                    for (int j = 0; j < sec; j++) {

                        cout << j + 1 << ") " << root->child[i]->child[j]->lable <<
endl;

                    }
                }
            }
        }
};
int main()
{

        Book k;
        int ch;
        do
        {
            cout << "Book Tree Creation" << endl;
            cout << "1.Create" << endl;
            cout << "2.Display" << endl;
            cout << "3.Quit" << endl;
            cout << "Enter your choice : ";
            cin >> ch;
            switch (ch)
            {
            case 1:
                k.create();
                break;
            case 2:
                k.display(root);
                break;
            }

        } while (ch != 3);

}
```

OUTPUT



```
Book Tree Creation
1.Create
2.Display
3.Quit
Enter your choice : 1
Enter the name of Book:
DemoBook
Enter the numbers of Chapters is the Book:
2
Enter the name of the Chapter:
Chapter1
Enter the number of sections in the Chapter: Chapter1
2
Enter the name of Section:
Section1
Enter the name of Section:
Section2
Enter the name of the Chapter:
Chapter2
Enter the number of sections in the Chapter: Chapter2
3
Enter the name of Section:
Section1
Enter the name of Section:
Section2
Enter the name of Section:
Section3
Book Tree Creation
1.Create
2.Display
3.Quit
Enter your choice : 2
Book Name : DemoBook
Chapter : 1
1) Chapter1
Sections :
1) Section1
2) Section2
Chapter : 2
2) Chapter2
Sections :
1) Section1
2) Section2
3) Section3
Book Tree Creation
1.Create
2.Display
3.Quit
Enter your choice : 3
```

Experiment B6

```
/*
Beginning with an empty binary search tree, construct binary search tree by inserting the
values in the order given.
After constructing a binary tree -
i. Insert new node,
ii. Find number of nodes in longest path from root,
iii. Minimum data value found in the tree,
iv. Change a tree so that the roles of the left and right pointers are swapped at every node,
v. Search a value
*/

#include <iostream>
using namespace std;

struct node
{
    int key;
    struct node* left, * right;
};

struct node* insert(struct node* root, int value)
{
    if (root == NULL)
    {
        root = new node;
        root->key = value;
        root->left = NULL;
        root->right = NULL;
        return root;
    }
    else if (value == root->key)
    {
        return root;
    }
    else
    {
        if (root->key < value)
            root->right = insert(root->right, value);
        else
        {
            if (root->key > value)
                root->left = insert(root->left, value);
        }
    }
    return root;
}
```

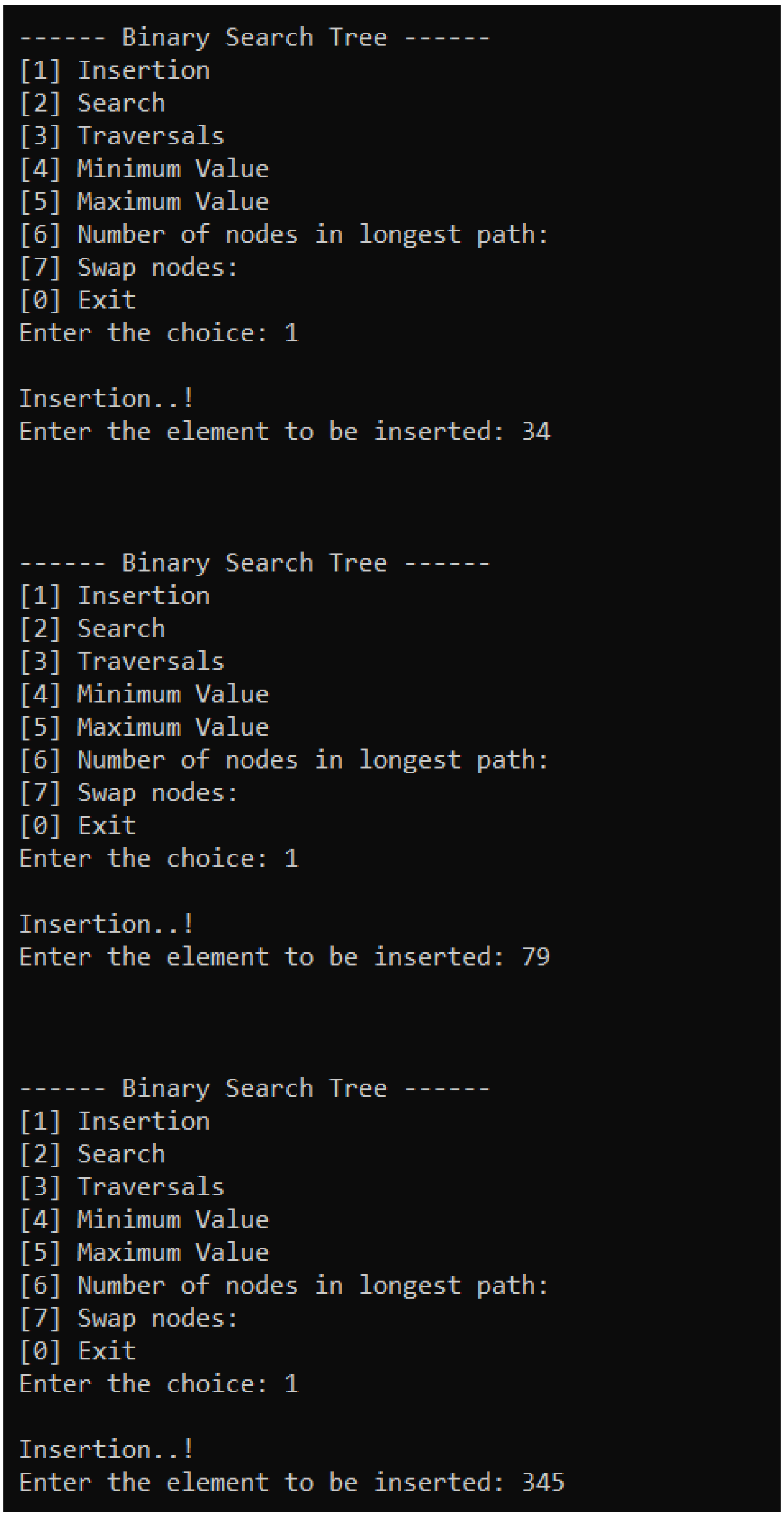
```
void inorder(struct node* root)
{
   if (root != NULL)
   {
      inorder(root->left);
      cout << "\t" << root->key;
      inorder(root->right);
   }
   return;
}

void postorder(struct node* root)
{
   if (root != NULL)
   {
      postorder(root->left);
      postorder(root->right);
      cout << "\t" << root->key;
   }
   return;
}

void preorder(struct node* root)
{
   if (root != NULL)
   {
      cout << "\t" << root->key;
      preorder(root->left);
      preorder(root->right);
   }
   return;
}

struct node* search(struct node* root, int value)
{
   if (root == NULL || root->key == value)
      return root;

   if (root->key < value)
      return search(root->right, value);

   return search(root->left, value);
}
struct node* minimumval(struct node* root)
{
   struct node* current = root;
   while (current->left != NULL)
      current = current->left;
   return current;
}

struct node* maximumval(struct node* root)
```

```
{
    struct node* current = root;
    while (current->right != NULL)
        current = current->right;
    return current;
}

struct node* swapnodes(struct node* root)
{
    node* temp;
    if (root == NULL)
        return NULL;
    temp = root->left;
    root->left = root->right;
    root->right = temp;
    swapnodes(root->left);
    swapnodes(root->right);
    return root;

}

int longestpath(node* root)
{
    if (root == NULL) {
        return 0;
    }
    int leftlong = longestpath(root->left);
    int rightlong = longestpath(root->right);
    return max(leftlong, rightlong) + 1;
}


int main()
{
    int choice = 0, value = 0, value1 = 0, d;
    struct node* root = NULL, * searchh = NULL, * position = NULL;
    do
    {

        cout << "\n\n";
        cout << "\n ------ Binary Search Tree ------";
        cout << "\n [1] Insertion ";
        cout << "\n [2] Search ";
        cout << "\n [3] Traversals ";
        cout << "\n [4] Minimum Value ";
        cout << "\n [5] Maximum Value";
        cout << "\n [6] Number of nodes in longest path:";
        cout << "\n [7] Swap nodes:";
        cout << "\n [0] Exit ";
        cout << "\n Enter the choice: ";
        cin >> choice;
```

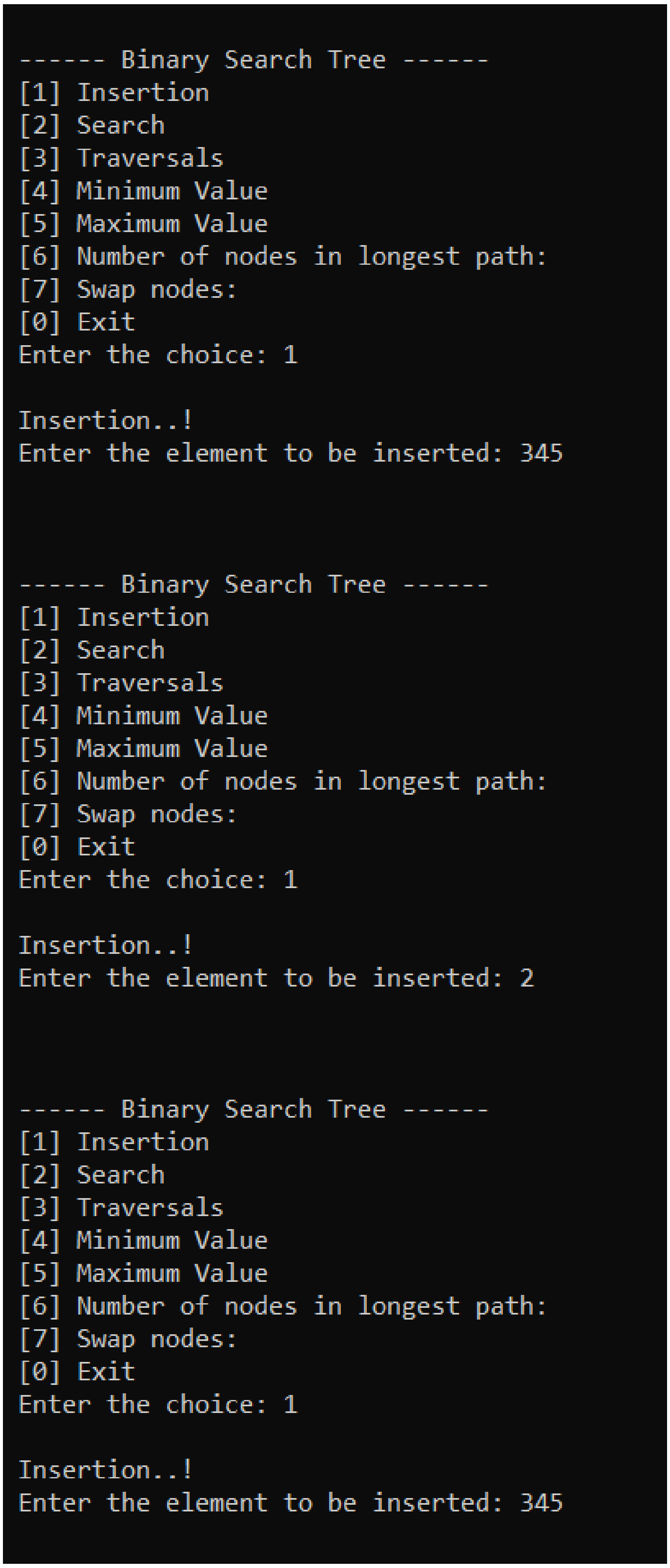
```
switch (choice)
{
case 1:
    cout << "\n Insertion..!";
    cout << "\n Enter the element to be inserted: ";
    cin >> value;
    root = insert(root, value);
    break;
case 2:
    cout << "\n Search..!";
    cout << "\n Enter the element to be searched: ";
    cin >> value;
    searchh = search(root, value);
    if (searchh == NULL)
        cout << "\n Key not found!";
    else
    {
        cout << "\n" << " Key " << searchh->key << " Found!";
    }
    break;
case 3:
    cout << "\n Traversals..!";
    cout << "\n Inorder: ";
    inorder(root);
    cout << "\n Preorder: ";
    preorder(root);
    cout << "\n Postorder: ";
    postorder(root);
    break;
case 4:
    cout << "\n Minimum Value..!";
    if (root == NULL)
        cout << "\n No minimum values in empty tree";
    else
    {
        value = minimumval(root)->key;
        cout << "\n Smallest value in the tree: " << value;
    }
    break;
case 5:
    cout << "\n Maximum Value..!";
    if (root == NULL)
        cout << "\n No maximum values in empty tree";
    else
    {
        value = maximumval(root)->key;
        cout << "\n Largest value in the tree: " << value;
    }
    break;
case 6:
    d = longestpath(root);
    cout << "The no. of nodes in the longest path are:" << d << endl;
```

```
            break;
        case 7:
            swapnodes(root);
            break;
        case 0:
            cout << "\n Exiting..!";
            break;
        default:
            cout << "\n Invalid Choice!";
            break;
        }
    } while (choice != 0);
    return 0;
}
```

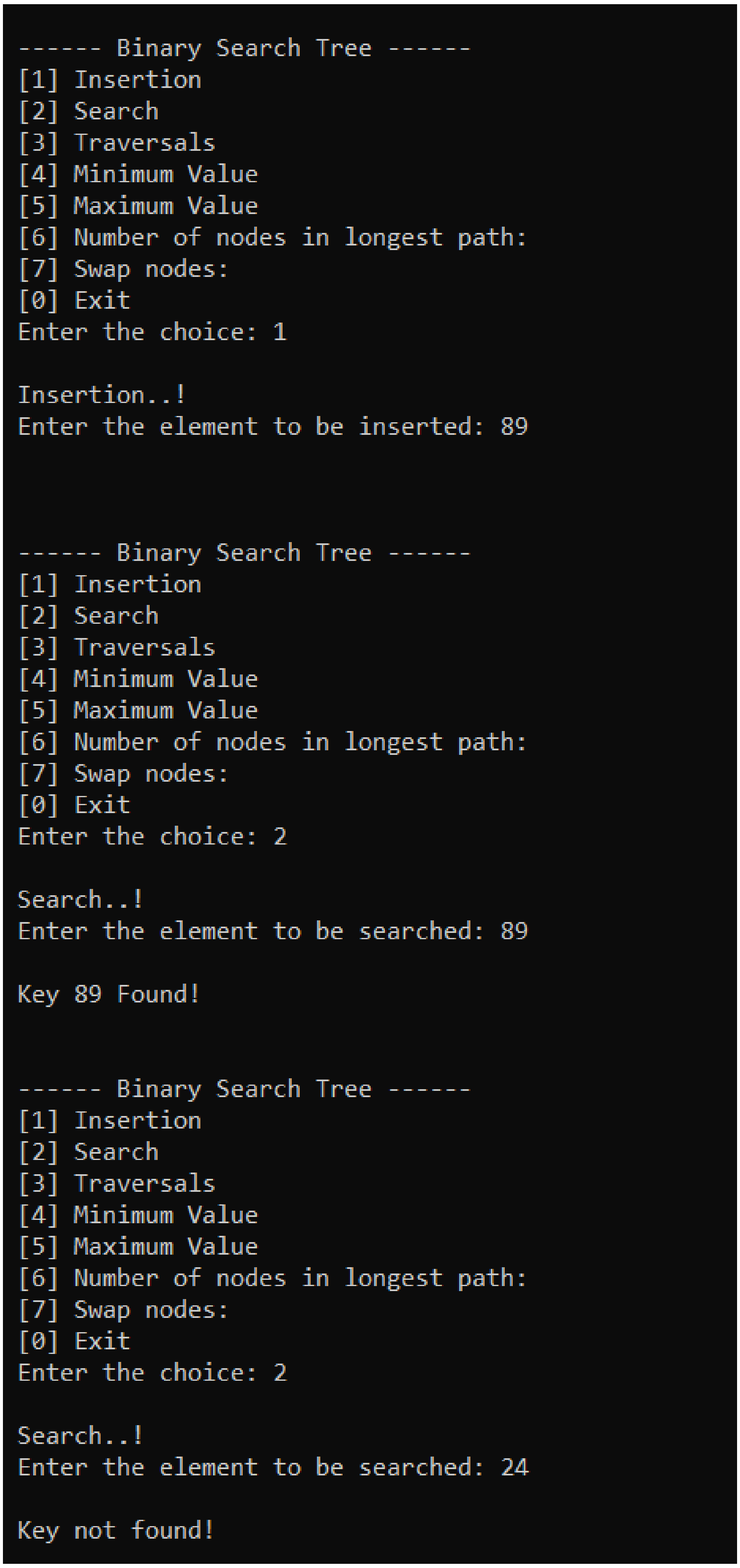
OUTPUT



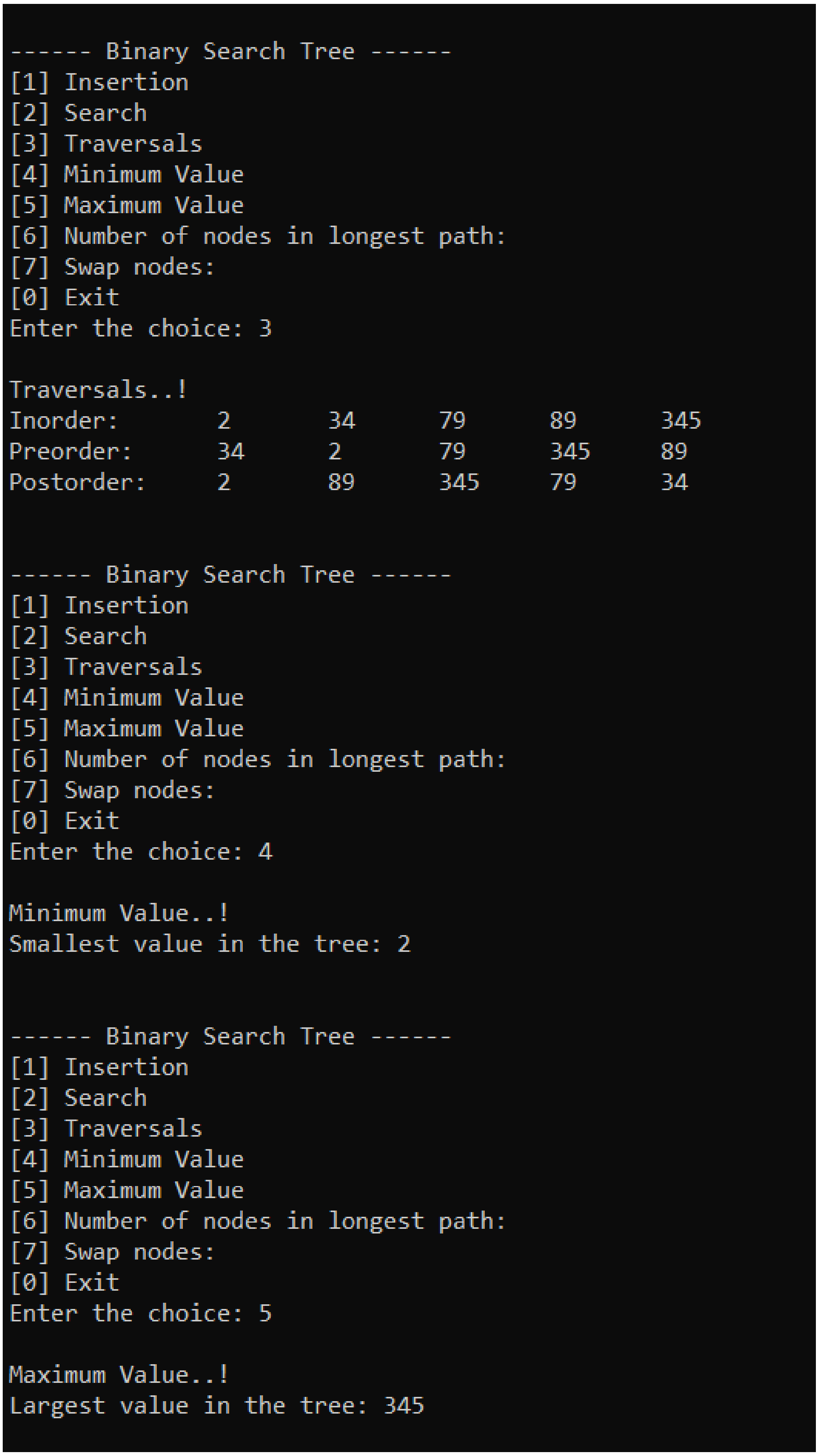
```
------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 34



------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 79



------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 345
```



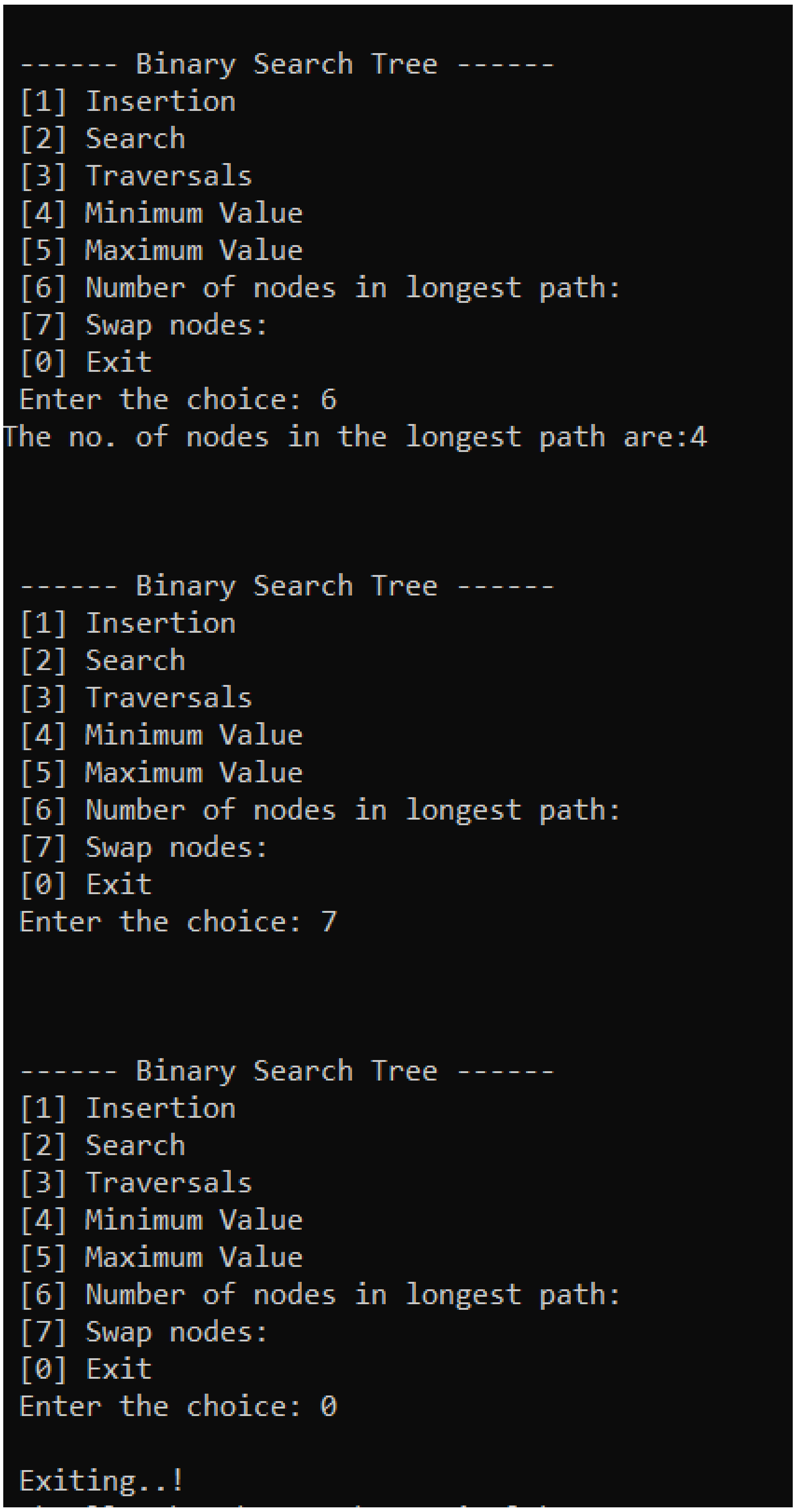
```
------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 345



------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 2



------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 345
```



```
------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 89



------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 2

Search..!
Enter the element to be searched: 89

Key 89 Found!



------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 2

Search..!
Enter the element to be searched: 24

Key not found!
```



```
------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 3

Traversals..!
Inorder:        2       34      79      89      345
Preorder:       34      2       79      345     89
Postorder:      2       89      345     79      34


------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 4

Minimum Value..!
Smallest value in the tree: 2


------ Binary Search Tree ------
[1] Insertion
[2] Search
[3] Traversals
[4] Minimum Value
[5] Maximum Value
[6] Number of nodes in longest path:
[7] Swap nodes:
[0] Exit
Enter the choice: 5

Maximum Value..!
Largest value in the tree: 345
```



```
 ------ Binary Search Tree ------
 [1] Insertion
 [2] Search
 [3] Traversals
 [4] Minimum Value
 [5] Maximum Value
 [6] Number of nodes in longest path:
 [7] Swap nodes:
 [0] Exit
 Enter the choice: 6
The no. of nodes in the longest path are:4



 ------ Binary Search Tree ------
 [1] Insertion
 [2] Search
 [3] Traversals
 [4] Minimum Value
 [5] Maximum Value
 [6] Number of nodes in longest path:
 [7] Swap nodes:
 [0] Exit
 Enter the choice: 7



 ------ Binary Search Tree ------
 [1] Insertion
 [2] Search
 [3] Traversals
 [4] Minimum Value
 [5] Maximum Value
 [6] Number of nodes in longest path:
 [7] Swap nodes:
 [0] Exit
 Enter the choice: 0

 Exiting..!
```

Experiment B11

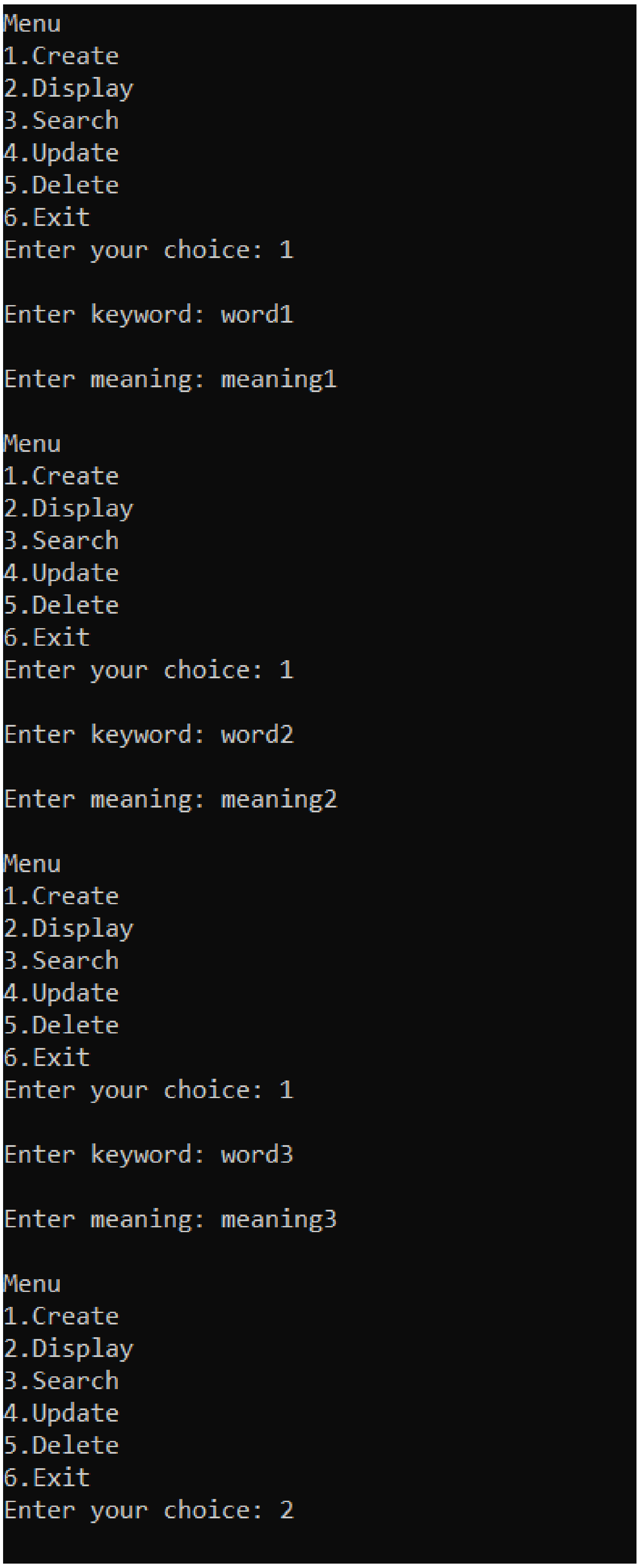
```
/*
A Dictionary stores keywords an d its meanings. Provide facility for adding new keywords,
deleting keywords, updating values of any entry.
Provide facility to display whole data sorted in ascending/ Descending order.
Also find how many maximum comparisons may require for finding any keyword. Use
Binary Search Tree for implementation.
*/
#include <iostream>
#include <cstring>
#include <algorithm>
#include <vector>
using namespace std;

struct node {
    char k[20]; //array to store keyword
    char m[20]; //array to store meaning;
    class node* left;
    class node* right;
};

class dict {
public:
    node* root; //root poitner
    void create(); //to create bst
    void disp(node*); //to display bst;

    void insert(node* root, node* temp); // to insert new node
    int search(node*, char[]); // to search any node;
    int update(node*, char[]); // to change  value of any node
    node* del(node*, char[]);
    node* min(node*);
};

void dict::create() {
    class node* temp;
    int ch;
    temp = new node;
    cout << "\nEnter keyword: ";
    cin >> temp->k;
    cout << "\nEnter meaning: ";
    cin >> temp->m;
    temp->left = NULL;
    temp->right = NULL;
    if (root == NULL) {
        root = temp; //if no root node then make temp as root node
    }
    else {
        insert(root, temp);// if root is present then call insert function to insrt node to
appropriate position
    }
```

```
}

void dict::insert(node* root, node* temp) {
   //to insert new node n bst when root node is available
   if (strcmp(temp->k, root->k) < 0)
      //compare keyboard of temp node root node & if it is less than 0 then need to inset left
   {

      if (root->left == NULL) //if left node is null then insert temp otherwise call insert
function to search position in left subtree
      {
         root->left = temp;
      }
      else {
         insert(root->left, temp);
      }
   }
   else {
      if (root->right == NULL) {
         root->right = temp;
      }
      else {
         insert(root->right, temp);
      }
   }
}
void dict::disp(node* root) //to display record
{
   if (root != NULL) {
      disp(root->left); // go towards extreme left node
      cout << "\n Key word: " << root->k; //print that node
      cout << "\t Meaning: " << root->m;
      disp(root->right);
   }
}

int dict::search(node* root, char k[20]) { // to search an element
   int c = 0; //to count no. of comparisons
   while (root != NULL) // until root becomes null
   {
      c++;
      if (strcmp(k, root->k) == 0) {
         cout << "\n No of comparisons: " << c; //if matches, return 1 and print count
         return 1;
      }
      if (strcmp(k, root->k) < 0)
         root = root->left;

      if (strcmp(k, root->k) > 0) //if comparison is greater than zero then search towards
right subtree
      {
         root = root->right;
```

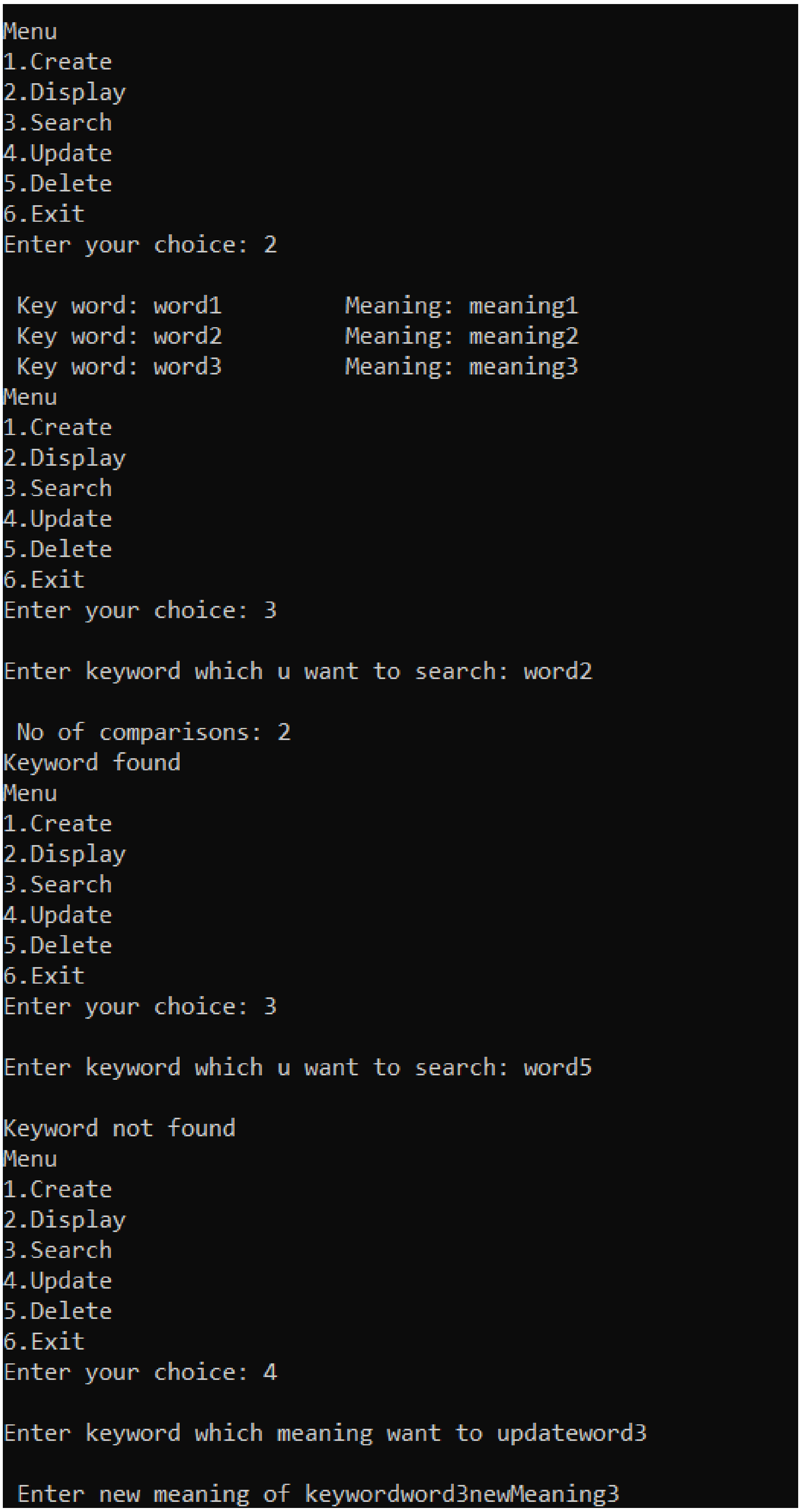
```
        }

    }
    return -1;
}

int dict::update(node* root, char k[20]) // to update any entry
{
    while (root != NULL) {
        if (strcmp(k, root->k) == 0) //compare search key with keyword of root node
        {
            cout << "\n Enter new meaning of keyword" << root->k;
            cin >> root->m; //if found then update the meaning of specified keyword & return 1 as
found
            return 1;
        }

        if (strcmp(k, root->k) < 0)
            root = root->left;
        if (strcmp(k, root->k) > 0)
            root = root->right;
    }
    return -1;
}

node* dict::del(node* root, char k[20])//to delete any entry
{
    node* temp;
    if (root == NULL) { //if root node is not present

        cout << "\nElement not found"; //no element is found
        return root;
    }

    if (strcmp(k, root->k) < 0) { //if keyword is less than root node

        root->left = del(root->left, k); //apply delete function on left element
        return root;
    }

    if (strcmp(k, root->k) > 0) { //if keyword is less than root node

        root->right = del(root->right, k); //apply delete function on right element
        return root;
    }

    if (root->right == NULL && root->left == NULL) //if that root node is leaf node
    {
        temp = root;
        delete temp;
        return NULL;
    }

    if (root->right == NULL) {
        temp = root;
        root = root->left;
        delete temp;
        return root;
    }
    else if (root->left == NULL) {
        temp = root;
        root = root->right;
        delete temp;
        return root;
    }
    temp = min(root->right); // if condition get unstatisfied that node is not a leaf node , node
is not having a left child, right child
    strcpy_s(root->k, temp->k);
    root->right = del(root->right, temp->k);
    return root;
}

node* dict::min(node* q) // find min element on extreme left //on the right subtree find the
min element
{
    while (q->left != NULL);
    {
        q = q->left; // until reach the leaf node
    }

    return q;
}

int main() {
    int ch;
    dict d; // object of class dictionary
    d.root = NULL; //setting initially root to the null
    bool flag = true;
    do {
        cout << "\nMenu\n1.Create\n2.Display\n3.Search\n4.Update\n5.Delete\n6.Exit\nEnter
your choice: ";
        cin >> ch;
        switch (ch)
        {
        case 1: d.create(); // to create bst;
            break;

        case 2: if (d.root == NULL) // nothing to display in a tree
        {
            cout << "\nNo any keywrod";
        }
              else {
            d.disp(d.root);
        }
              break;

        case 3: if (d.root == NULL) // nthing to display in a tree
        {
            cout << "\nDictionary is empty, first add keywords then try again ";
        }
              else {
            cout << "\nEnter keyword which u want to search: ";
            char k[20];
            cin >> k; // take a choice to search;
            if (d.search(d.root, k) == 1) {
              cout << "\nKeyword found";
            }
            else
              cout << "\nKeyword not found ";
        }
              break;

        case 4:
            if (d.root == NULL) {
              cout << "\ndictionary is empty, first add keywords then try again ";
            }
            else {
              cout << "\nEnter keyword which meaning want to update";
              char k[20];
              cin >> k;
              if (d.update(d.root, k) == 1) //if function returns meaning is updated
                cout << "\nmeaning updated";
              else
                cout << "\nMeaning Not Found";
            }
            break;

        case 5:
            if (d.root == NULL) {
              cout << "\nDictionary is emtpy , first add keywords then try again ";
            }
            else {
              cout << "\nEnter keyword which u want to delete: ";
              char k[20];
              cin >> k;
              if (d.root == NULL) {
                cout << "\nno any keyword";
              }
              else {
                d.root = d.del(d.root, k);
              }
            }
            break;
        case 6:
            cout << "Exiting";
```

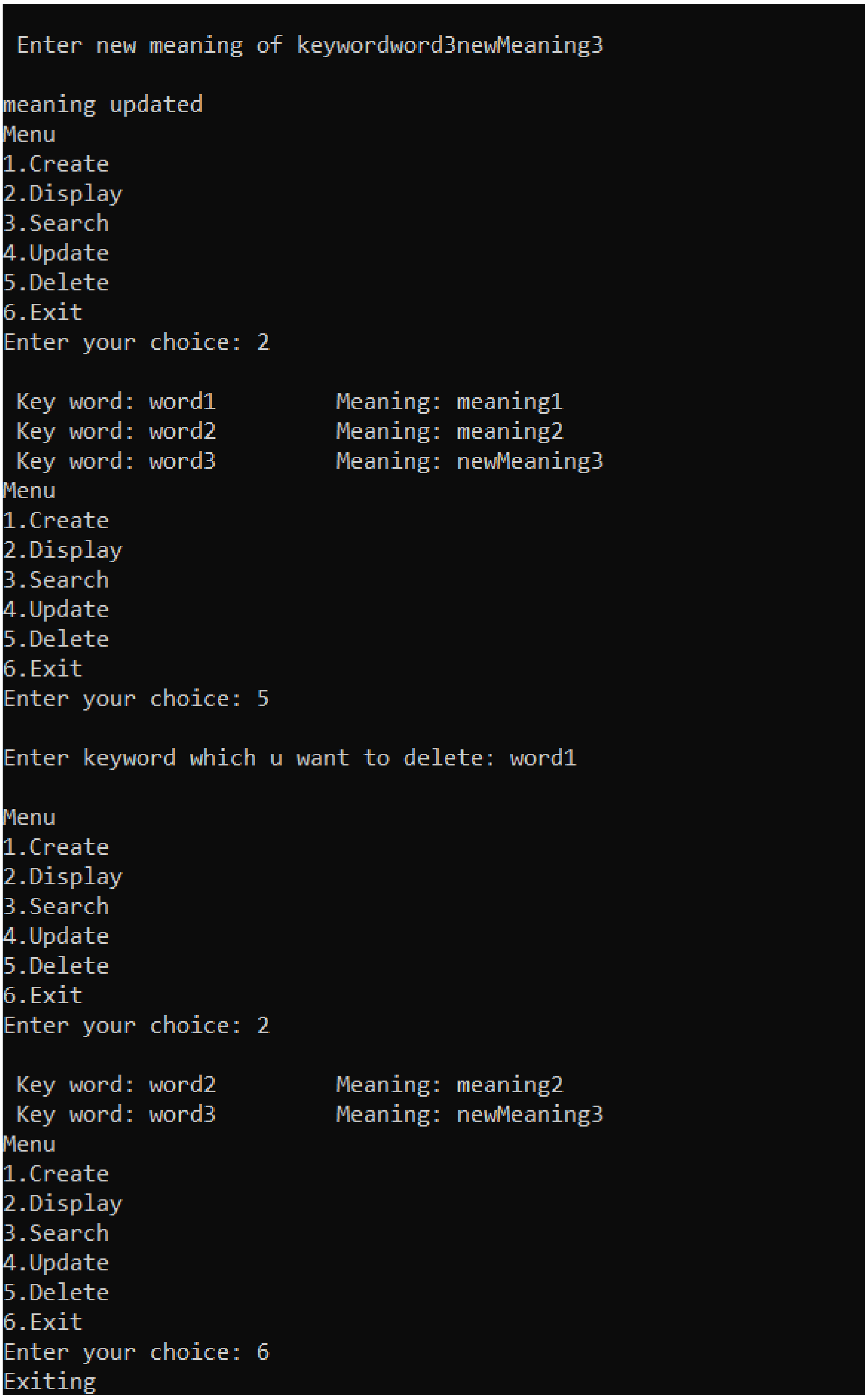
```
            flag = false;
            break;
        default:
            cout << "Invalid choice, try again";
        }

    } while (flag);


    return 0;
}
```

OUTPUT



```
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 1

Enter keyword: word1

Enter meaning: meaning1

Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 1

Enter keyword: word2

Enter meaning: meaning2

Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 1

Enter keyword: word3

Enter meaning: meaning3

Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 2
```


```
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 2

 Key word: word1         Meaning: meaning1
 Key word: word2         Meaning: meaning2
 Key word: word3         Meaning: meaning3
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 3

Enter keyword which u want to search: word2

 No of comparisons: 2
Keyword found
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 3

Enter keyword which u want to search: word5

Keyword not found
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 4

Enter keyword which meaning want to updateword3

 Enter new meaning of keywordword3newMeaning3
```



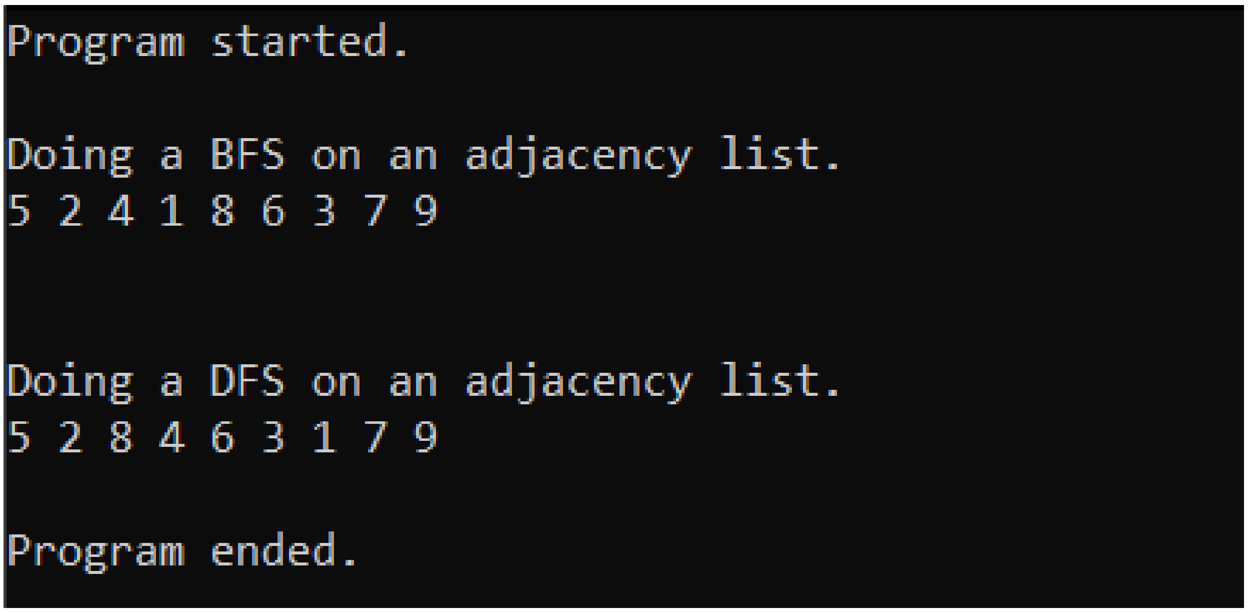
```
 Enter new meaning of keywordword3newMeaning3

meaning updated
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 2

 Key word: word1         Meaning: meaning1
 Key word: word2         Meaning: meaning2
 Key word: word3         Meaning: newMeaning3
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 5

Enter keyword which u want to delete: word1

Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 2

 Key word: word2         Meaning: meaning2
 Key word: word3         Meaning: newMeaning3
Menu
1.Create
2.Display
3.Search
4.Update
5.Delete
6.Exit
Enter your choice: 6
Exiting
```

Experiment C13

```
/*
Represent a given graph using adjacency matrix/list to perform DFS and using adjacency
list to perform BFS.
Use the map of the area around the college as the graph.
Identify the prominent land marks as nodes and perform DFS and BFS on that.
*/
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

// Adjacency List - Adding O(1), Lookup O(N), Space O(N^2) but usually better.
// Each vector position represents a node - the vector inside that position represents that
node's friends.
vector< vector<int> > FormAdjList()
{
   // Our adjacency list.
   vector< vector<int> > adjList;

   // We have 10 vertices, so initialize 10 rows.
   const int n = 9;

   for (int i = 0; i < n; i++)
   {
      // Create a vector to represent a row, and add it to the adjList.
      vector<int> row;
      adjList.push_back(row);
   }

   // Now "adjList[0]" has a vector<int> in it that represents the friends of vertex 1.
   // (Remember, we use 0-based indexing. 0 is the first number in our vector, not 1.

   // Now let's add our actual edges into the adjacency list.
   // See the picture here:
https://www.srcmake.com/uploads/5/3/9/0/5390645/adjl_4_orig.png

   adjList[0].push_back(2);
   adjList[0].push_back(4);
   adjList[0].push_back(6);

   adjList[1].push_back(4);
   adjList[1].push_back(7);

   adjList[2].push_back(0);
   adjList[2].push_back(5);

   adjList[3].push_back(4);
```

```
    adjList[3].push_back(5);

    adjList[4].push_back(1);
    adjList[4].push_back(3);
    adjList[4].push_back(0);

    adjList[5].push_back(2);
    adjList[5].push_back(3);
    adjList[5].push_back(8);

    adjList[6].push_back(0);

    adjList[7].push_back(1);

    adjList[8].push_back(5);

    // Our graph is now represented as an adjacency list.
    return adjList;
}

// Adjacency Matrix - Adding O(N), Lookup O(1), Space O(N^2)
vector< vector<int> > FormAdjMatrix()
{
    // We could use an array for the adjMatrix if we knew the size, but it's safer to use a
vector.
    vector< vector<int> > adjMatrix;

    // Initialize the adjMatrix so that all vertices can visit themselves.
    // (Basically, make an identity matrix.)
    const int n = 9;

    for (int i = 0; i < n; i++)
    {
        // Initialize the row.
        vector<int> row;
        adjMatrix.push_back(row);


        for (int j = 0; j < n; j++)
        {
            int value = 0;

            if (i == j)
            {
                value = 1;
            }

            adjMatrix[i].push_back(value);
        }
    }
    adjMatrix[0][2] = 1;
    adjMatrix[2][0] = 1;

    adjMatrix[0][4] = 1;
    adjMatrix[4][0] = 1;

    adjMatrix[0][6] = 1;
    adjMatrix[6][0] = 1;

    adjMatrix[1][4] = 1;
    adjMatrix[4][1] = 1;

    adjMatrix[1][7] = 1;
    adjMatrix[7][1] = 1;

    adjMatrix[2][5] = 1;
    adjMatrix[5][2] = 1;

    adjMatrix[3][4] = 1;
    adjMatrix[4][3] = 1;

    adjMatrix[3][5] = 1;
    adjMatrix[5][3] = 1;

    adjMatrix[5][8] = 1;
    adjMatrix[8][5] = 1;

    // Our adjacency matrix is complete.
    return adjMatrix;
}

// Given an Adjacency List, do a BFS on vertex "start"
void AdjListBFS(vector< vector<int> > adjList, int start)
{
    cout << "\nDoing a BFS on an adjacency list.\n";

     int n = adjList.size();
    // Create a "visited" array (true or false) to keep track of if we visited a vertex.
    vector<bool> visited(n);
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }

    // Create a queue for the nodes we visit.
    queue<int> q;

    // Add the starting vertex to the queue and mark it as visited.
    q.push(start);
    visited[start] = true;

    // While the queue is not empty..
    while (q.empty() == false)
    {
        int vertex = q.front();
        q.pop();

        // Doing +1 in the cout because our graph is 1-based indexing, but our code is 0-based.
        cout << vertex + 1 << " ";

        // Loop through all of it's friends.
        for (int i = 0; i < adjList[vertex].size(); i++)
        {
            // If the friend hasn't been visited yet, add it to the queue and mark it as visited
            int neighbor = adjList[vertex][i];

            if (visited[neighbor] == false)
            {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
    cout << endl << endl;
    return;
}

void AdjListDFS(vector< vector<int> >& adjList, int& vertex, vector<bool>& visited)
{
    // Mark the vertex as visited.
    visited[vertex] = true;

    // Outputting vertex+1 because that's the way our graph picture looks.
    cout << vertex + 1 << " ";

    // Look at this vertex's neighbors.
    for (int i = 0; i < adjList[vertex].size(); i++)
    {
        int neighbor = adjList[vertex][i];
        // Recursively call DFS on the neighbor, if it wasn't visited.
        if (visited[neighbor] == false)
        {
            AdjListDFS(adjList, neighbor, visited);
        }
    }
}
// Given an Adjacency Matrix, do a BFS on vertex "start"
void AdjListDFSInitialize(vector< vector<int> >& adjList, int start)
{
    cout << "\nDoing a DFS on an adjacency list.\n";

    int n = adjList.size();
    // Create a "visited" array (true or false) to keep track of if we visited a vertex.
    vector<bool> visited;
    for (int i = 0; i < n; i++)
    {
        visited.push_back(false);
    }

    AdjListDFS(adjList, start, visited);

    cout << endl << endl;
    return;
}

int main()
{
    cout << "Program started.\n";

    // Get the adjacency list/matrix.
    vector< vector<int> > adjList = FormAdjList();


    // Call BFS on Vertex 5. (Labeled as 4 in our 0-based-indexing.)
    AdjListBFS(adjList, 4);
    AdjListDFSInitialize(adjList, 4);

    cout << "Program ended.\n";

    return 0;
}
```

OUTPUT



```
Program started.

Doing a BFS on an adjacency list.
5 2 4 1 8 6 3 7 9


Doing a DFS on an adjacency list.
5 2 8 4 6 3 1 7 9

Program ended.
```

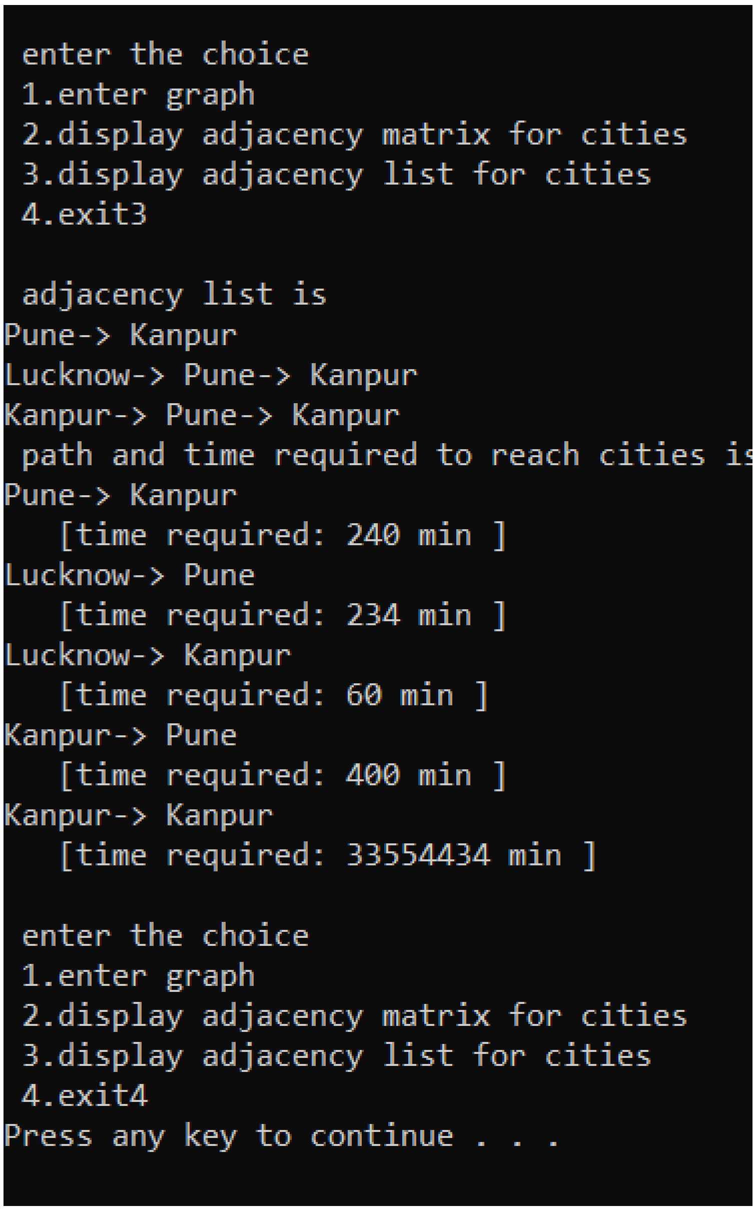
Experiment C14

```
/*
There are flight paths between cities. If there is a flight between city A and city B then there
is an edge between the cities.
The cost of the edge can be the time that flight take to reach city B from A, or the amount of
fuel used for the journey.
Represent this as a graph.
The node can be represented by airport name or name of the city.
Use adjacency list representation of the graph or use adjacency matrix representation of
the graph.
Check whether the graph is connected or not.
Justify the storage representation used.
*/
#include<iostream>
#include<stdlib.h>
#include<string.h>
using namespace std;
struct node
{
   string vertex;
   int time;
   node* next;
};
class adjmatlist
{
   int m[10][10], n, i, j; char ch;  string v[20];   node* head[20];  node* temp = NULL;

public:
   adjmatlist()
   {
      for (i = 0; i < 20; i++)
      {
         head[i] = NULL;
      }
   }
   void getgraph();
   void adjlist();

   void displaym();
   void displaya();
};
void adjmatlist::getgraph()
{
   cout << "\n enter no. of cities(max. 20)";
   cin >> n;
   cout << "\n enter name of cities";
   for (i = 0; i < n; i++)
      cin >> v[i];
   for (i = 0; i < n; i++)
```

```
    {
        for (j = 0; j < n; j++)
        {
            cout << "\n if path is present between city " << v[i] << " and " << v[j] << " then press
enter y otherwise n";
            cin >> ch;
            if (ch == 'y')
            {
                cout << "\n enter time required to reach city " << v[j] << " from " << v[i] << " in
minutes";
                cin >> m[i][j];
            }
            else if (ch == 'n')
            {
                m[i][j] = 0;
            }
            else
            {
                cout << "\n unknown entry";
            }
        }
    }
    adjlist();

}
void adjmatlist::adjlist()
{
    cout << "\n ****";
    for (i = 0; i < n; i++)
    {
        node* p = new(struct node);
        p->next = NULL;
        p->vertex = v[i];
        head[i] = p;      cout << "\n" << head[i]->vertex;
    }

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (m[i][j] != 0)
            {
                node* p = new(struct node);
                p->vertex = v[j];
                p->time = m[i][j];
                p->next = NULL;
                if (head[i]->next == NULL)
                {
                    head[i]->next = p;
                }
                else
                {
                    temp = head[i];
                    while (temp->next != NULL)
                    {
                        temp = temp->next;
                    }
                    temp->next = p;
                }

            }

        }
    }

}
void adjmatlist::displaym()
{
    cout << "\n";
    for (j = 0; j < n; j++)
    {
        cout << "\t" << v[j];
    }

    for (i = 0; i < n; i++)
    {
        cout << "\n " << v[i];
        for (j = 0; j < n; j++)
        {
            cout << "\t" << m[i][j];
        }
        cout << "\n";
    }
}
void adjmatlist::displaya()
{
    cout << "\n adjacency list is";

    for (i = 0; i < n; i++)
    {


        if (head[i] == NULL)
        {
            cout << "\n adjacency list not present";  break;
        }
        else
        {
            cout << "\n" << head[i]->vertex;
            temp = head[i]->next;
            while (temp != NULL)
            {
                cout << "-> " << temp->vertex;
                temp = temp->next;
            }

        }
    }

    cout << "\n path and time required to reach cities is";

    for (i = 0; i < n; i++)
    {


        if (head[i] == NULL)
        {
            cout << "\n adjacency list not present";  break;
        }
        else
        {

            temp = head[i]->next;
            while (temp != NULL)
            {
                cout << "\n" << head[i]->vertex;
                cout << "-> " << temp->vertex << "\n   [time required: " << temp->time << " min ]";
                temp = temp->next;
            }

        }




    }
}
int main()
{
    int m;
    adjmatlist a;

    while (1)
    {
        cout << "\n\n enter the choice";
        cout << "\n 1.enter graph";
        cout << "\n 2.display adjacency matrix for cities";
        cout << "\n 3.display adjacency list for cities";
        cout << "\n 4.exit";
        cin >> m;

        switch (m)
        {
        case 1: a.getgraph();
            break;
        case 2: a.displaym();
```

```
            break;

        case 3: a.displaya();
            break;
        case 4: exit(0);

        default:  cout << "\n unknown choice";
        }
    }
    return 0;
}
```

OUTPUT

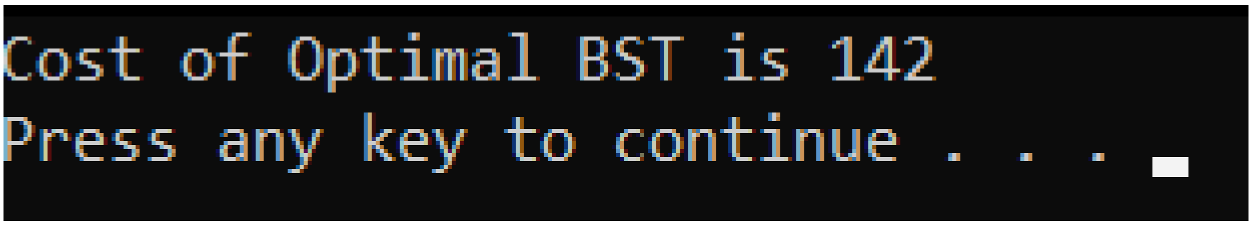


```
 enter the choice
 1.enter graph
 2.display adjacency matrix for cities
 3.display adjacency list for cities
 4.exit3

 adjacency list is
Pune-> Kanpur
Lucknow-> Pune-> Kanpur
Kanpur-> Pune-> Kanpur
 path and time required to reach cities is
Pune-> Kanpur
   [time required: 240 min ]
Lucknow-> Pune
   [time required: 234 min ]
Lucknow-> Kanpur
   [time required: 60 min ]
Kanpur-> Pune
   [time required: 400 min ]
Kanpur-> Kanpur
   [time required: 33554434 min ]

 enter the choice
 1.enter graph
 2.display adjacency matrix for cities
 3.display adjacency list for cities
 4.exit4
Press any key to continue . . .
```

Experiment D18

```
/*
Given sequence k = k1 < …<kn, of n sorted keys, with a search probability pi for each key ki.
Build the Binary search tree that has the least search cost given the access probability for
each key?
*/
#include <bits/stdc++.h>
using namespace std;

int sum(int frequency[], int i, int j)
{
    int sum = 0;
    for (int x = i; x <= j; x++)
        sum += frequency[x];
    return sum;
}

int optimalCost(int frequency[], int i, int j)
{
    if (j < i)
        return 0;
    if (j == i)
        return frequency[i];

    int frequencySum = sum(frequency, i, j);

    int min = INT_MAX;

    for (int r = i; r <= j; ++r)
    {
        int cost = optimalCost(frequency, i, r - 1) + optimalCost(frequency, r + 1, j);
        if (cost < min)
            min = cost;
    }

    return min + frequencySum;
}

int optimalSearchTree(int keys[], int frequency[], int n)
{
    return optimalCost(frequency, 0, n - 1);
}

int main()
{
    int keys[] = { 10, 12, 20 };
    int frequency[] = { 34, 8, 50 };

    int n = sizeof(keys) / sizeof(keys[0]);
    cout << "Cost of Optimal BST is " << optimalSearchTree(keys, frequency, n);

    return 0;
}
```

OUTPUT



```
Cost of Optimal BST is 142
Press any key to continue . . .
```

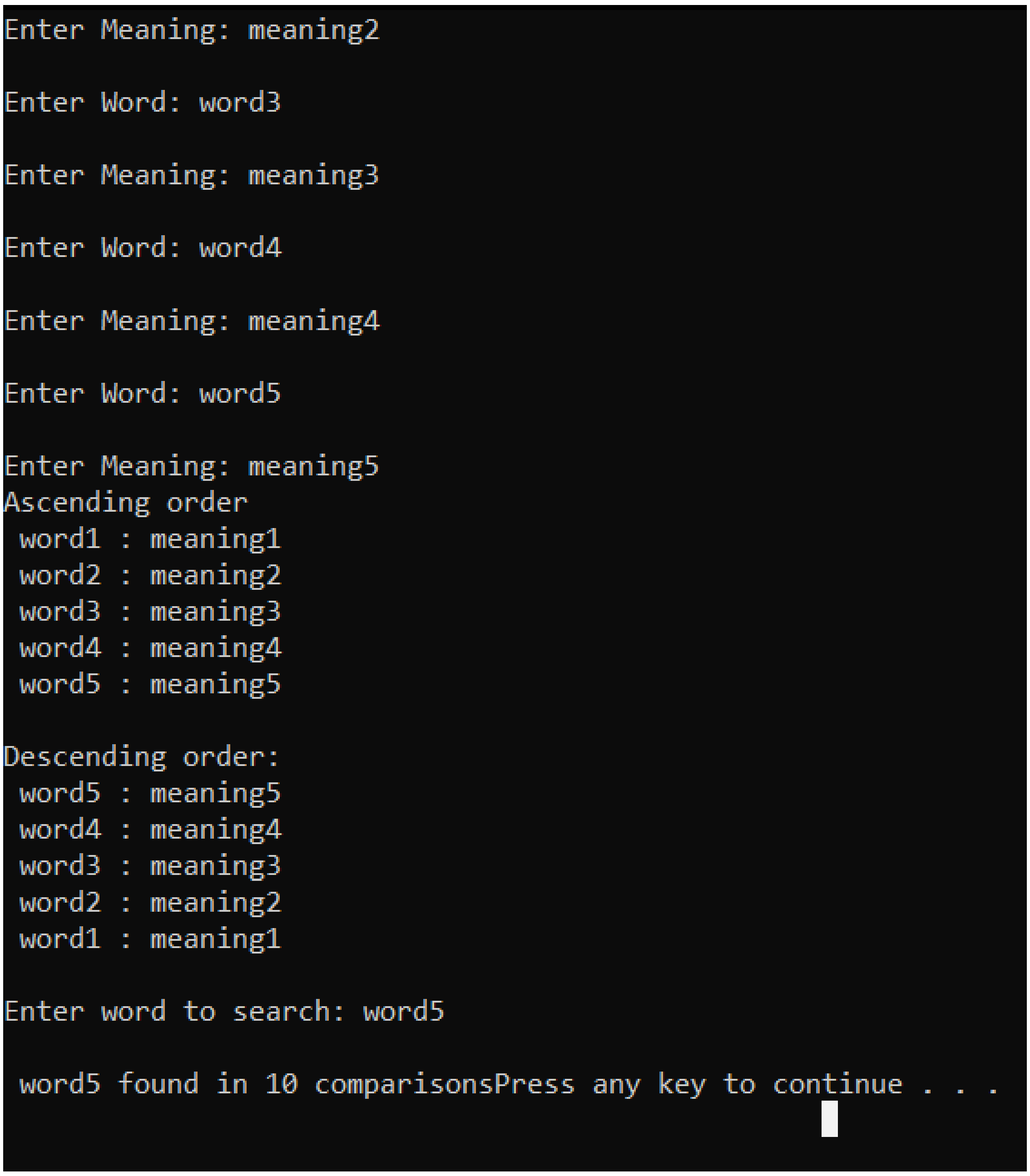
Experiment D19

```
/*
A Dictionary stores keywords and its meanings. Provide facility for adding new keywords,
deleting keywords, updating values of any entry.
Provide facility to display whole data sorted in ascending/Descending order. Also find how
many maximum comparisons may require for finding any keyword.
Use Height balance tree and find the complexity for finding a keyword.
*/
#include <iostream>
#include<string>
using namespace std;
class dictionary;
class node
{
	string word, meaning;
	node* left, * right;
public:
	friend class dictionary;
	node()
	{
		left = NULL;
		right = NULL;

	}
	node(string word, string meaning)
	{
		this->word = word;
		this->meaning = meaning;
		left = NULL;
		right = NULL;
	}
};

class dictionary
{
	node* root;
public:
	dictionary()
	{
		root = NULL;
	}
	void create();
	void inorder_rec(node* rnode);
	void postorder_rec(node* rnode);
	void inorder()
	{
		inorder_rec(root);
```

```
        }
        void postorder();

        bool insert(string word, string meaning);
        int search(string key);

};
int dictionary::search(string key)
{
        node* tmp = root;
        int count;
        if (tmp == NULL)
        {
                return -1;
        }
        if (root->word == key)
                return 1;
        while (tmp != NULL)
        {

                if ((tmp->word) > key)
                {
                        tmp = tmp->left;
                        count++;
                }
                else if ((tmp->word) < key)
                {
                        tmp = tmp->right;
                        count++;
                }
                else if (tmp->word == key)
                {
                        return ++count;
                }
        }
        return -1;

}
void dictionary::postorder()
{
        postorder_rec(root);
}
void dictionary::postorder_rec(node* rnode)
{
        if (rnode)
        {
                postorder_rec(rnode->right);
                cout << " " << rnode->word << " : " << rnode->meaning << endl;
                postorder_rec(rnode->left);
        }
}
void dictionary::create()
```

```
{
        int n;
        string wordI, meaningI;
        cout << "\nHow many Word to insert?:\n";
        cin >> n;
        for (int i = 0; i < n; i++)
        {
                cout << "\nEnter Word: ";
                cin >> wordI;
                cout << "\nEnter Meaning: ";
                cin >> meaningI;
                insert(wordI, meaningI);
        }
}
void dictionary::inorder_rec(node* rnode)
{
        if (rnode)
        {
                inorder_rec(rnode->left);
                cout << " " << rnode->word << " : " << rnode->meaning << endl;
                inorder_rec(rnode->right);
        }
}
bool dictionary::insert(string word, string meaning)
{
        node* p = new node(word, meaning);
        if (root == NULL)
        {
                root = p;
                return true;
        }
        node* cur = root;
        node* par = root;
        while (cur != NULL) //traversal
        {
                if (word > cur->word)
                {
                        par = cur;
                        cur = cur->right;
                }
                else if (word < cur->word)
                {
                        par = cur;
                        cur = cur->left;
                }
                else
                {
                        cout << "\nWord is already in the dictionary.";
                        return false;
                }
        }
        if (word > par->word) //insertion of node
        {
                par->right = p;
                return true;
        }
        else
        {
                par->left = p;

                return true;
        }
}

int main() {
        string word;
        dictionary months;
        months.create();
        cout << "Ascending order\n";
        months.inorder();

        cout << "\nDescending order:\n";
        months.postorder();

        cout << "\nEnter word to search: ";
        cin >> word;
        int comparisons = months.search(word);
        if (comparisons == -1)
        {
                cout << "\nNot found word";
        }
        else
        {
                cout << "\n " << word << " found in " << comparisons << " comparisons";
        }
        return 0;
}
```

OUTPUT



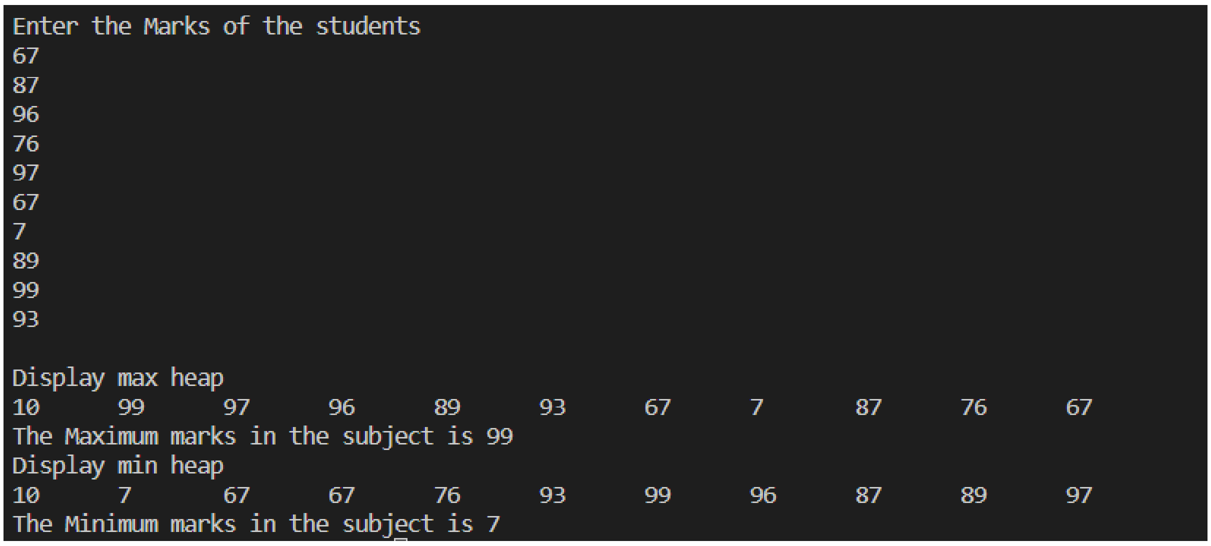
```
Enter Meaning: meaning2

Enter Word: word3

Enter Meaning: meaning3

Enter Word: word4

Enter Meaning: meaning4

Enter Word: word5

Enter Meaning: meaning5
Ascending order
 word1 : meaning1
 word2 : meaning2
 word3 : meaning3
 word4 : meaning4
 word5 : meaning5

Descending order:
 word5 : meaning5
 word4 : meaning4
 word3 : meaning3
 word2 : meaning2
 word1 : meaning1

Enter word to search: word5

 word5 found in 10 comparisonsPress any key to continue . . .
```

Experiment E22

```
/*
Read the marks obtained by students of second year in an online examination of particular
subject.
Find out maximum and minimum marks obtained in that subject.
Use heap data structure. Analyze the algorithm.
*/
#include<iostream>
#define SIZE 100
using namespace std;

int heap[SIZE];
void create(int heap[], int n);
void buildheapmax(int heap[], int i);
void create1(int heap[], int n);
void buildheapmin(int heap[], int i);

int main()

{
   int n, i, j, k;

   cout << "Enter the no of Students appeared For ADS online examination" << endl;
   cin >> n;
   heap[0] = n;
   cout << "\nEnter the Marks of the students" << endl;
   for (k = 1; k <= n; k++)
   {
      cin >> heap[k];
   }
   create(heap, n);
   cout << "\nDisplay max heap" << endl;
   for (k = 0; k <= n; k++)
   {
      cout << heap[k] << "\t";
   }

   cout << "\nThe Maximum marks in the subject is ";
   cout << heap[1];
   create1(heap, n);
   cout << "\nDisplay min heap" << endl;
   for (k = 0; k <= n; k++)
   {
      cout << heap[k] << "\t";
   }

   cout << "\nThe Minimum marks in the subject is ";
   cout << heap[1];
```

```
    return 0;
}

void create(int heap[], int n)
{
    int i, k;

    for (i = n / 2; i >= 1; i--)
    {
        buildheapmax(heap, i);
    }

}

void buildheapmax(int heap[], int i)
{


    int j, temp, m;
    int q = 1;
    m = heap[0];
    while (2 * i <= m && q == 1)
    {
        j = 2 * i;
        if (j + 1 <= m && heap[j + 1] > heap[j])
        {
            j = j + 1;
        }

        if (heap[i] > heap[j])
        {
            q = 0;
        }
        else
        {
            temp = heap[i];
            heap[i] = heap[j];
            heap[j] = temp;
            i = j;
        }

    }


}
void create1(int heap[], int n)
{
    int i, k;

    for (i = n / 2; i >= 1; i--)
    {
        buildheapmin(heap, i);
    }

}

void buildheapmin(int heap[], int i)
{


    int j, temp, m;
    int q = 1;
    m = heap[0];
    while (2 * i <= m && q == 1)
    {
        j = 2 * i;
        if (j + 1 <= m && heap[j + 1] < heap[j])
        {
            j = j + 1;
        }

        if (heap[i] < heap[j])
        {
            q = 0;
        }
        else
        {
            temp = heap[i];
            heap[i] = heap[j];
            heap[j] = temp;
            i = j;
        }

    }


}
```

OUTPUT



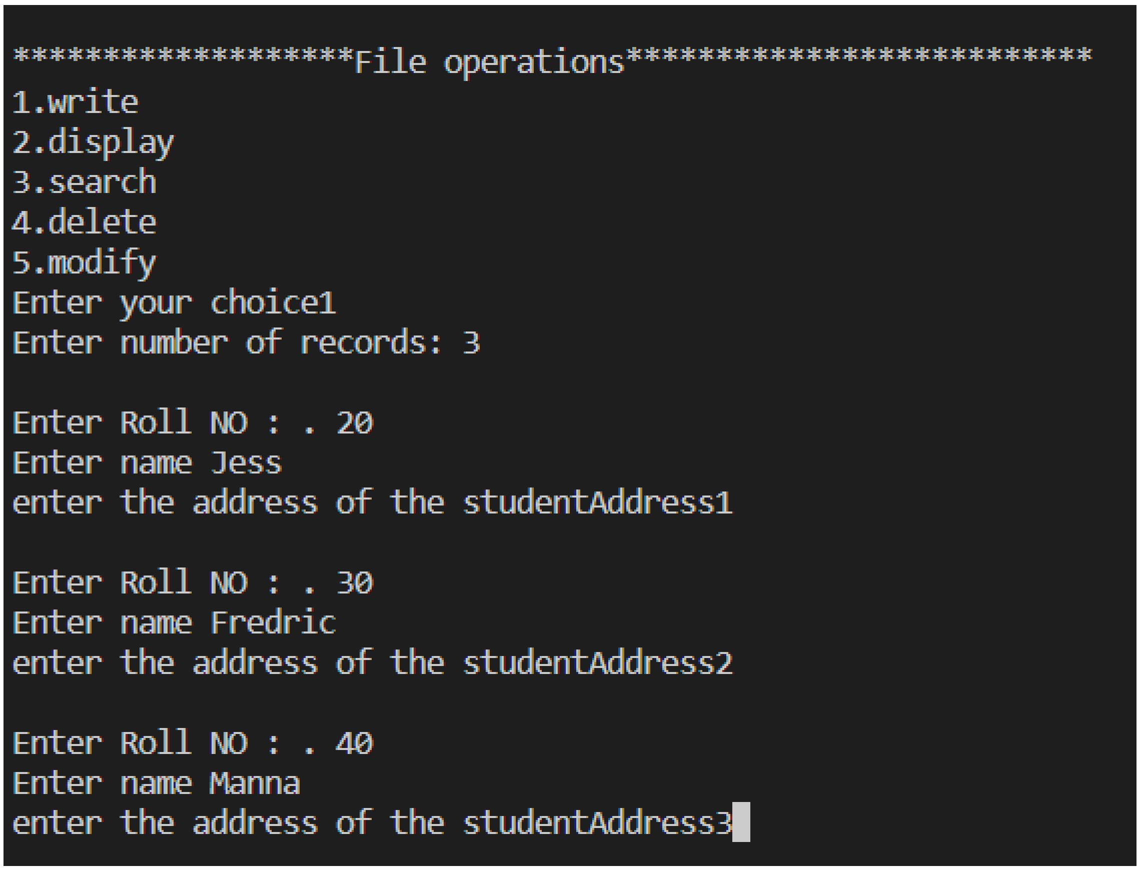
```
Enter the Marks of the students
67
87
96
76
97
67
7
89
99
93

Display max heap
10      99      97      96      89      93      67      7       87      76      67
The Maximum marks in the subject is 99
Display min heap
10      7       67      67      76      93      99      96      87      89      97
The Minimum marks in the subject is 7
```

Experiment F23

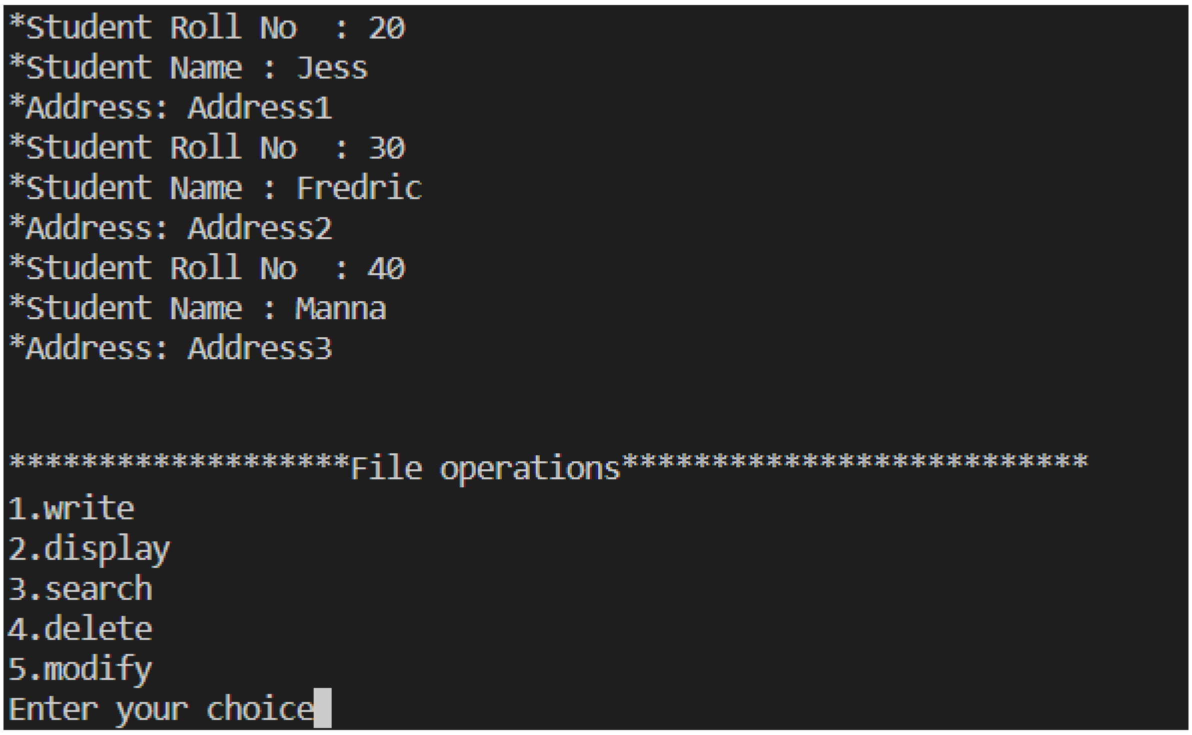
```
/*
Department maintains a student information. The file contains roll number, name, division
and address.
Allow user to add, delete information of student. Display information of particular
employee.
If record of student does not exist an appropriate message is displayed. If it is, then the
system displays the student details.
Use sequential file to main the data.
*/
#include<iostream>
#include<fstream>
#include<cstdio>
using namespace std;

class employee
{
   int admno;
   char name[50];
   char addr[50];
public:
   void setData()
   {
      cout << "\nEnter Roll NO : . ";
      cin >> admno;
      cout << "Enter name ";
      cin >> name;
      cout << "enter the address of the student";
      cin >> addr;


   }

   void showData()
   {
      cout << "\n*Student Roll No  : " << admno;
      cout << "\n*Student Name : " << name;
      cout << "\n*Address: " << addr;
   }

   int retAdmno()
   {
      return admno;
   }
};


void write_record()
{
   ofstream outFile;
   outFile.open("employee.dat", ios::binary | ios::app);

   employee obj;
   obj.setData();

   outFile.write((char*)&obj, sizeof(obj));

   outFile.close();
}

void display()
{
   ifstream inFile;
   inFile.open("employee.dat", ios::binary);

   employee obj;

   while (inFile.read((char*)&obj, sizeof(obj)))
   {
      obj.showData();
   }

   inFile.close();
}

void search(int n)
{
   ifstream inFile;
   inFile.open("employee.dat", ios::binary);

   employee obj;

   while (inFile.read((char*)&obj, sizeof(obj)))
   {
      if (obj.retAdmno() == n)
      {
          obj.showData();
          break;
      }
   }

   inFile.close();
}

void delete_record(int n)
{
   employee obj;
   ifstream inFile;
   inFile.open("employee.dat", ios::binary);

   ofstream outFile;
   outFile.open("temp.dat", ios::out | ios::binary);

   while (inFile.read((char*)&obj, sizeof(obj)))
   {
      if (obj.retAdmno() != n)
      {
          outFile.write((char*)&obj, sizeof(obj));
      }
   }

   inFile.close();
   outFile.close();

   remove("employee.dat");
   rename("temp.dat", "employee.dat");
}

void modify_record(int n)
{
   fstream file;
   file.open("employee.dat", ios::in | ios::out);

   employee obj;

   while (file.read((char*)&obj, sizeof(obj)))
   {
      if (obj.retAdmno() == n)
      {
          cout << "\nEnter the new details of Student";
          obj.setData();

          int pos = -1 * sizeof(obj);
          file.seekp(pos, ios::cur);

          file.write((char*)&obj, sizeof(obj));
      }
   }

   file.close();
}

int main()
{
   int ch;
   do {
      cout << "\n\n\n*******************File operations**************************
\n1.write\n2.display\n3.search\n4.delete\n5.modify";
      cout << "\nEnter your choice";
      cin >> ch;
      switch (ch) {

      case 1: cout << "Enter number of records: ";
```

```
            int n;
            cin >> n;
            for (int i = 0; i < n; i++)
                write_record();
            break;

        case 2:

            cout << "\nList of records";
            display();
            break;



        case 3:
            cout << "Enter Student Roll No : ";
            int s;
            cin >> s;
            search(s);
            break;
        case 4:
            cout << "enter no to be deleted";
            int d;
            cin >> d;

            delete_record(d);
            cout << "\nRecord Deleted";
            break;
        case 5:


            cout << "enter rno to be modified";
            int m;
            cin >> m;


            modify_record(m);
            break;
        case 6:

            return 0;
        }

    } while (ch != 6);
}
```

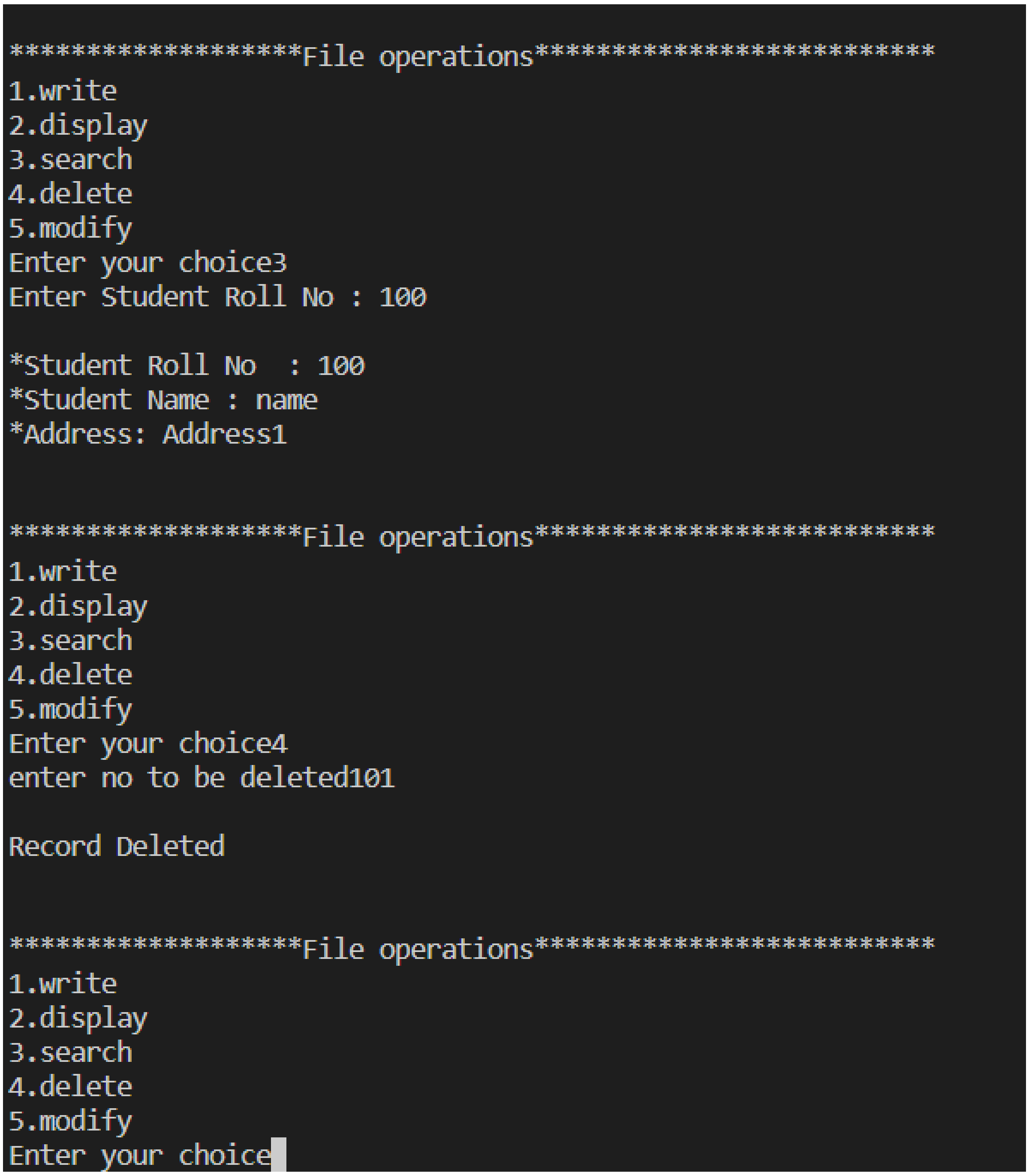
OUTPUT



```
*******************File operations***************************
1.write
2.display
3.search
4.delete
5.modify
Enter your choice1
Enter number of records: 3

Enter Roll NO : . 20
Enter name Jess
enter the address of the studentAddress1

Enter Roll NO : . 30
Enter name Fredric
enter the address of the studentAddress2

Enter Roll NO : . 40
Enter name Manna
enter the address of the studentAddress3
```



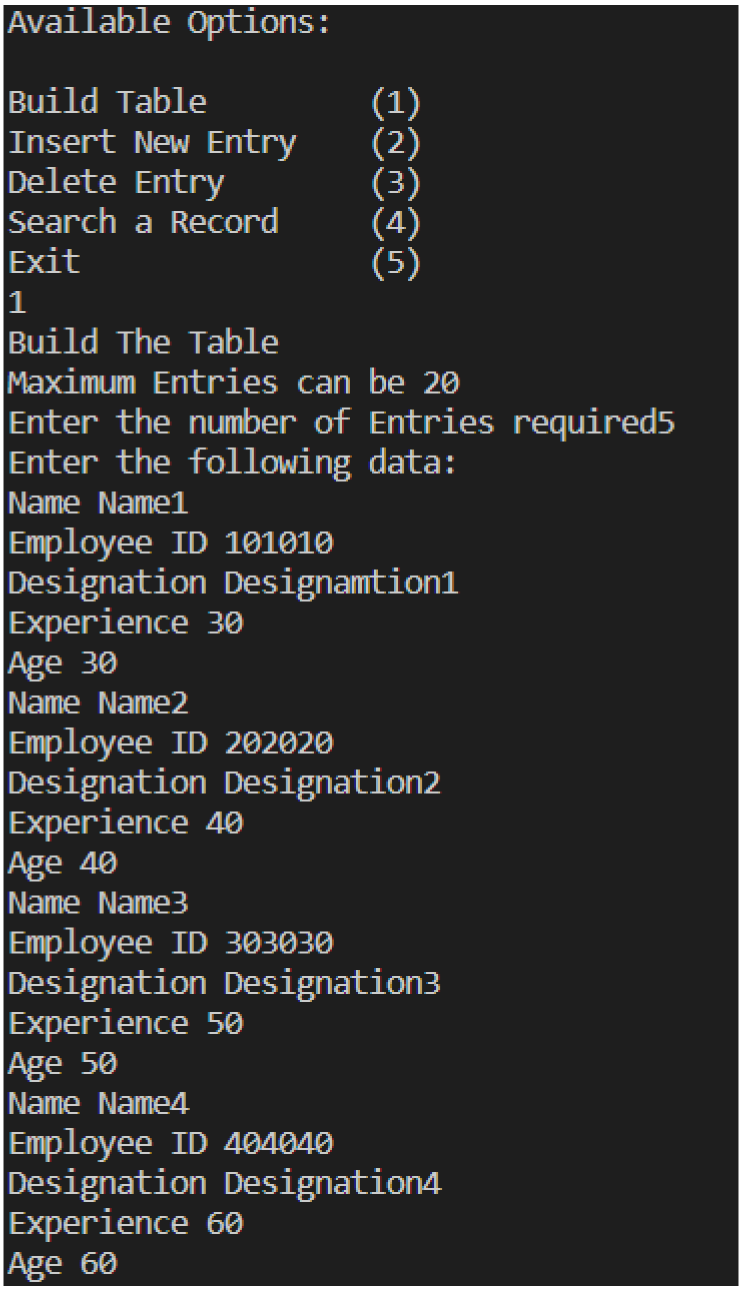
```
*Student Roll No  : 20
*Student Name : Jess
*Address: Address1
*Student Roll No  : 30
*Student Name : Fredric
*Address: Address2
*Student Roll No  : 40
*Student Name : Manna
*Address: Address3


*******************File operations***************************
1.write
2.display
3.search
4.delete
5.modify
Enter your choice
```


```
*******************File operations**************************
1.write
2.display
3.search
4.delete
5.modify
Enter your choice3
Enter Student Roll No : 100

*Student Roll No  : 100
*Student Name : name
*Address: Address1


*******************File operations**************************
1.write
2.display
3.search
4.delete
5.modify
Enter your choice4
enter no to be deleted101

Record Deleted


*******************File operations**************************
1.write
2.display
3.search
4.delete
5.modify
Enter your choice
```

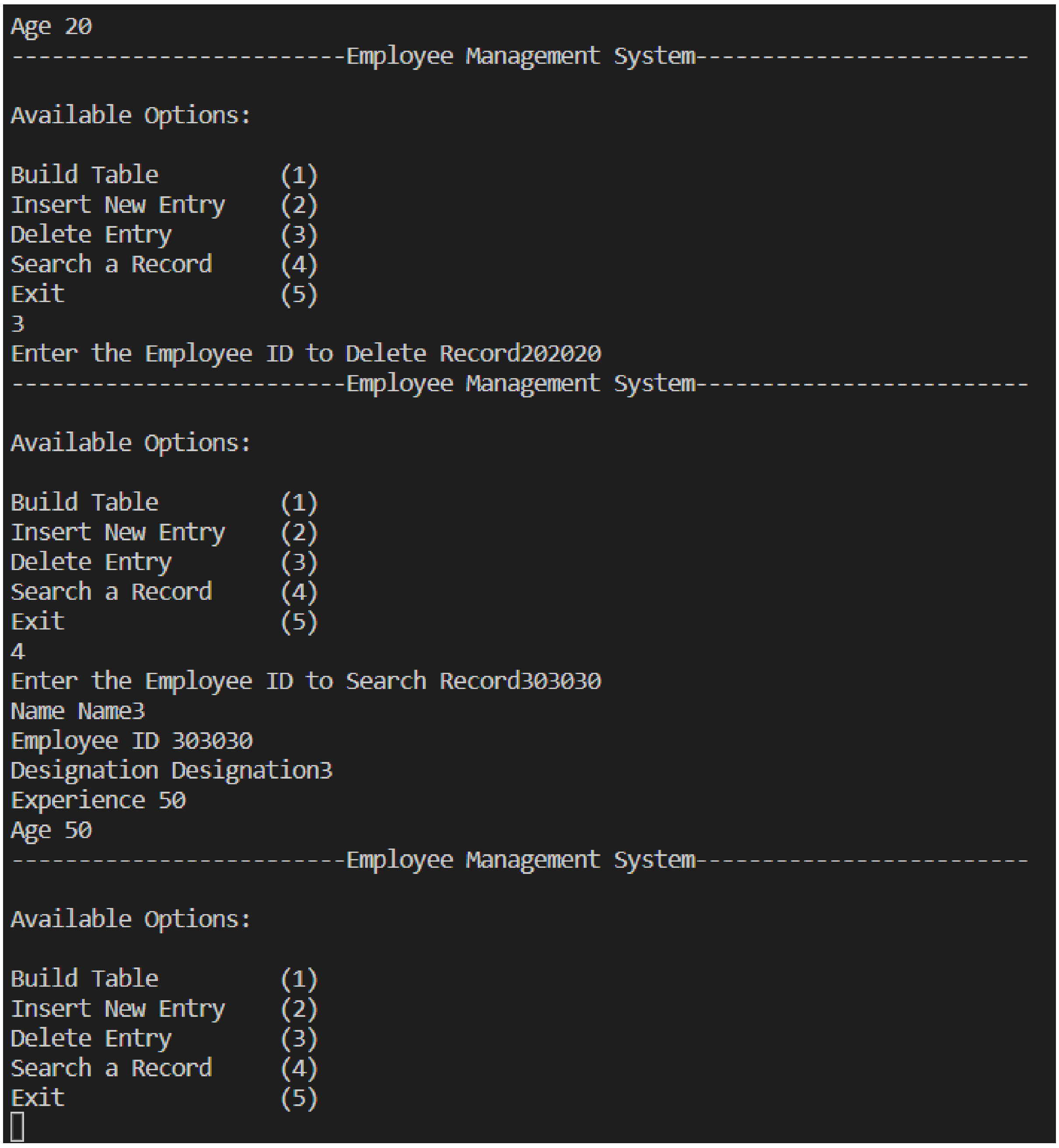
Experiment F24

```
/*
Company maintains employee information as employee ID, name, designation and salary.
Allow user to add, delete information of employee.
Display information of particular employee. If employee does not exist an appropriate
message is displayed.
If it is, then the system displays the employee details.
Use index sequential file to maintain the data.
*/
#include <iostream>
#include <stdlib.h>
#define max 20
using namespace std;

struct employee {
    string name;
    long int code;
    string designation;
    int exp;
    int age;
};
int num;
void showMenu();
employee emp[max], tempemp[max],
sortemp[max], sortemp1[max];
void build()
{
    cout << "Build The Table\n";
    cout << "Maximum Entries can be "
        << max << "\n";

    cout << "Enter the number of "
        << "Entries required";
    cin >> num;

    if (num > 20) {
        cout << "Maximum number of "
            << "Entries are 20\n";
        num = 20;
    }
    cout << "Enter the following data:\n";

    for (int i = 0; i < num; i++) {
        cout << "Name ";
        cin >> emp[i].name;
        cout << "Employee ID ";
        cin >> emp[i].code;
        cout << "Designation ";
        cin >> emp[i].designation;
        cout << "Experience ";
        cin >> emp[i].exp;
        cout << "Age ";
        cin >> emp[i].age;
    }
    showMenu();
}
void insert()
{
    if (num < max) {
        int i = num;
        num++;

        cout << "Enter the information "
            << "of the Employee\n";
        cout << "Name ";
        cin >> emp[i].name;

        cout << "Employee ID ";
        cin >> emp[i].code;

        cout << "Designation ";
        cin >> emp[i].designation;

        cout << "Experience ";
        cin >> emp[i].exp;

        cout << "Age ";
        cin >> emp[i].age;
    }
    else {
        cout << "Employee Table Full\n";
    }

    showMenu();
}
void deleteIndex(int i)
{
    for (int j = i; j < num - 1; j++) {
        emp[j].name = emp[j + 1].name;
        emp[j].code = emp[j + 1].code;
        emp[j].designation
            = emp[j + 1].designation;
        emp[j].exp = emp[j + 1].exp;
        emp[j].age = emp[j + 1].age;
    }
    return;
}
void deleteRecord()
{
    cout << "Enter the Employee ID "
        << "to Delete Record";
    int code;

    cin >> code;
    for (int i = 0; i < num; i++) {
        if (emp[i].code == code) {
            deleteIndex(i);
            num--;
            break;
        }
    }
    showMenu();
}
void searchRecord()
{
    cout << "Enter the Employee"
        << " ID to Search Record";

    int code;
    cin >> code;
    for (int i = 0; i < num; i++) {
        if (emp[i].code == code) {
            cout << "Name "
                << emp[i].name << "\n";

            cout << "Employee ID "
                << emp[i].code << "\n";

            cout << "Designation "
                << emp[i].designation << "\n";

            cout << "Experience "
                << emp[i].exp << "\n";

            cout << "Age "
                << emp[i].age << "\n";
            break;
        }
    }

    showMenu();
}
void showMenu()
{
    cout << "-------------------------"
        << "Employee"
        << " Management System"
        << "-------------------------\n\n";

    cout << "Available Options:\n\n";
    cout << "Build Table         (1)\n";
    cout << "Insert New Entry    (2)\n";
    cout << "Delete Entry        (3)\n";
    cout << "Search a Record     (4)\n";
    cout << "Exit                (5)\n";

    int option;

    // Input Options
    cin >> option;

    // Call function on the bases of the
    // above option
    if (option == 1) {
        build();
    }
    else if (option == 2) {
        insert();
    }
    else if (option == 3) {
        deleteRecord();
    }
    else if (option == 4) {
        searchRecord();
    }
    else if (option == 5) {
        return;
    }
    else {
        cout << "Expected Options"
            << " are 1/2/3/4/5";
        showMenu();
    }
}

// Driver Code
int main()
{

    showMenu();
    return 0;
}
```

OUTPUT



```
Available Options:

Build Table          (1)
Insert New Entry     (2)
Delete Entry         (3)
Search a Record      (4)
Exit                 (5)
1
Build The Table
Maximum Entries can be 20
Enter the number of Entries required5
Enter the following data:
Name Name1
Employee ID 101010
Designation Designamtion1
Experience 30
Age 30
Name Name2
Employee ID 202020
Designation Designation2
Experience 40
Age 40
Name Name3
Employee ID 303030
Designation Designation3
Experience 50
Age 50
Name Name4
Employee ID 404040
Designation Designation4
Experience 60
Age 60
```



```
Age 20
-------------------------Employee Management System-------------------------

Available Options:

Build Table         (1)
Insert New Entry    (2)
Delete Entry        (3)
Search a Record     (4)
Exit                (5)
3
Enter the Employee ID to Delete Record202020
-------------------------Employee Management System-------------------------

Available Options:

Build Table         (1)
Insert New Entry    (2)
Delete Entry        (3)
Search a Record     (4)
Exit                (5)
4
Enter the Employee ID to Search Record303030
Name Name3
Employee ID 303030
Designation Designation3
Experience 50
Age 50
-------------------------Employee Management System-------------------------

Available Options:

Build Table         (1)
Insert New Entry    (2)
Delete Entry        (3)
Search a Record     (4)
Exit                (5)
```