**DATA STRUCTURES AND ALGORITHMS PRACTICALS**

**PARTH REWOO**

**ERP – 56**

**SE COMP - I**

**GROUP A**

Experiment No. 1

**Experiment Name:** Telephone Book Database using collision handling techniques in Hash Table.

**Aim:** Consider Telephone Book Database of N clients. Make use of hash table implementation to quickly look up client’s telephonic number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephonic numbers.

**Objective/Theory:**

Hash Table:

The Hash table data structure stores elements in key-value pairs where

* Key- unique integer that is used for indexing the values
* Value - data that are associated with keys.

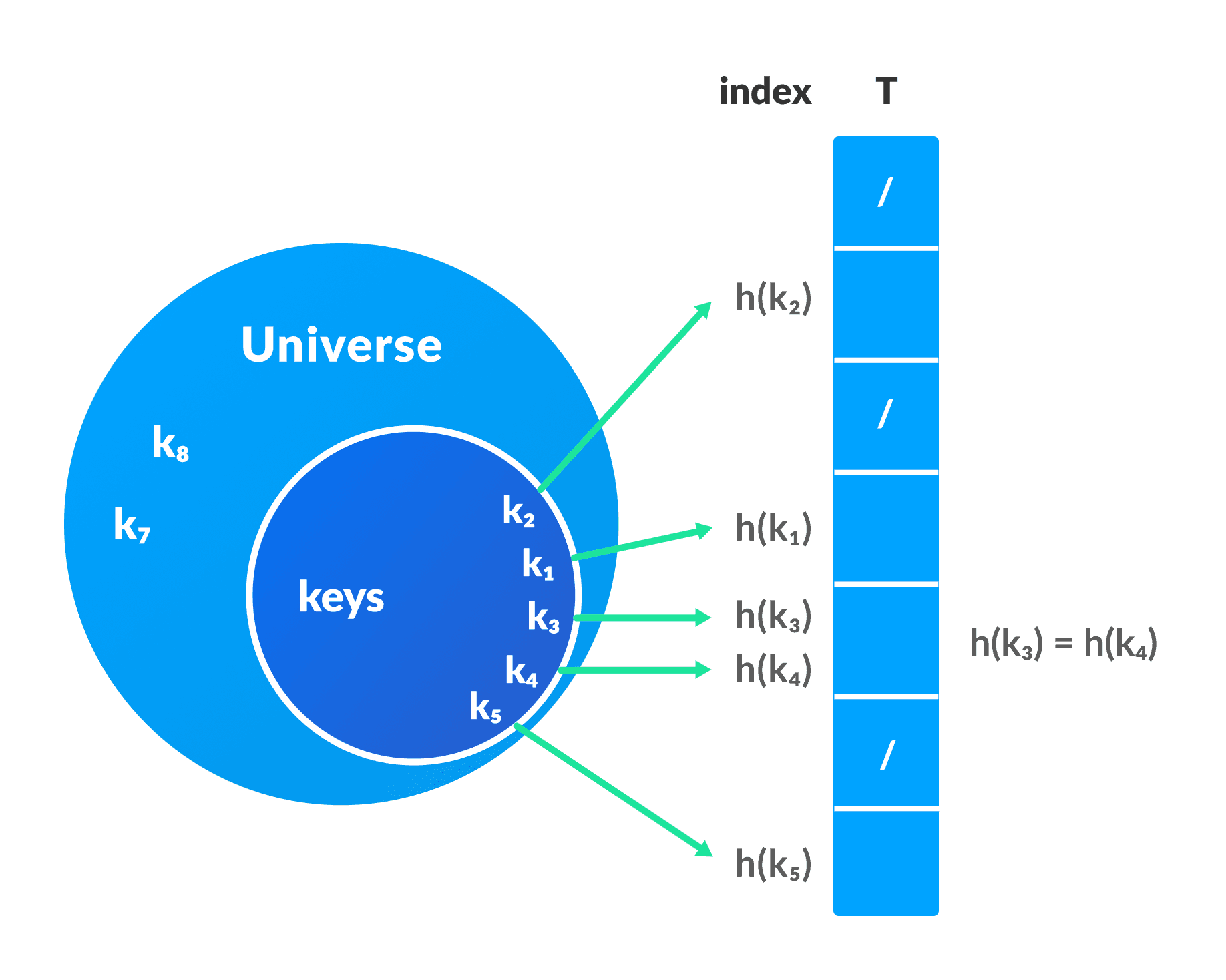
Hashing (Hash Function):

In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called hashing.

Let k be a key and *h(x)* be a hash function.

Here, *h(k)* will give us a new index to store the element linked with *k*.

Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function. It is a method for representing dictionaries for large datasets.



Hash Collision:

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision.

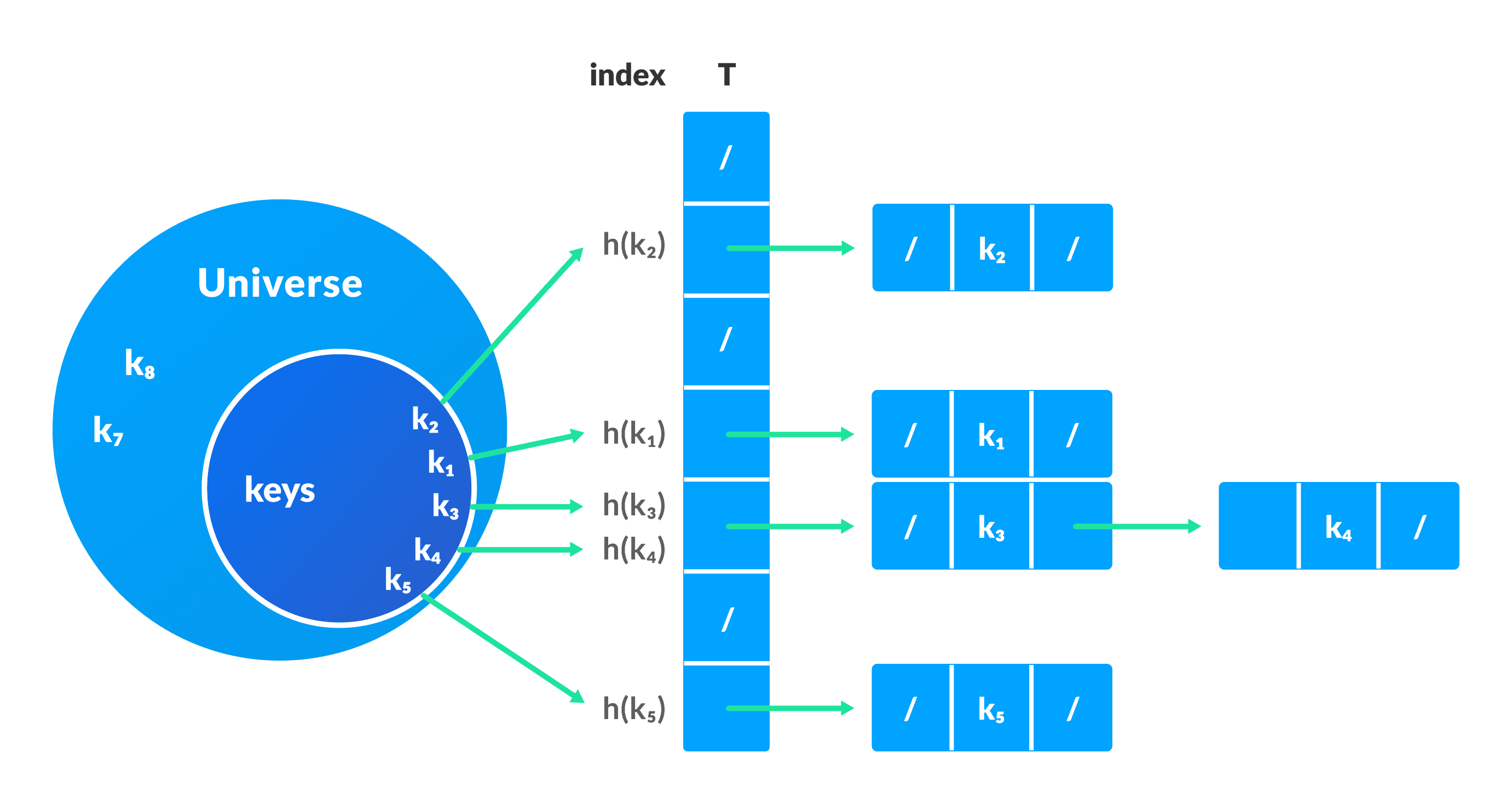
We can resolve the hash collision using one of the following techniques.

1. Collision resolution by chaining
2. Open Addressing: Linear/Quadratic Probing and Double Hashing

**Collision resolution by chaining**

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.

If *j* is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, *j* contains *NIL*.



**Open Addressing**

Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left *NIL*.

Different techniques used in open addressing are:

*• Linear Probing*

In linear probing, collision is resolved by checking the next slot.

*h(k, i) = (h′(k) + i) % m*

where *i = {0, 1, ….}*, *h'(k)* is a new hash function

If a collision occurs at *h(k, 0)*, then *h(k, 1)* is checked. In this way, the value of *i* is incremented linearly.

The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

*• Quadratic Probing*

It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

*h(k, i) = (h′(k) + c1i + c2i2) % m*

where, *c1* and *c2* are positive auxiliary constants, *i = {0, 1, ….}*

*• Double hashing*

If a collision occurs after applying a hash function *h(k)*, then another hash function is calculated for finding the next slot.

*h(k, i) = (h1(k) + ih2(k)) % m*

Applications of Hash Table:

Hash tables are implemented where

1. constant time lookup and insertion is required
2. cryptographic applications
3. indexing data is required

**Program:**

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()

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))

LinearProbing.py

# 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))

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*):

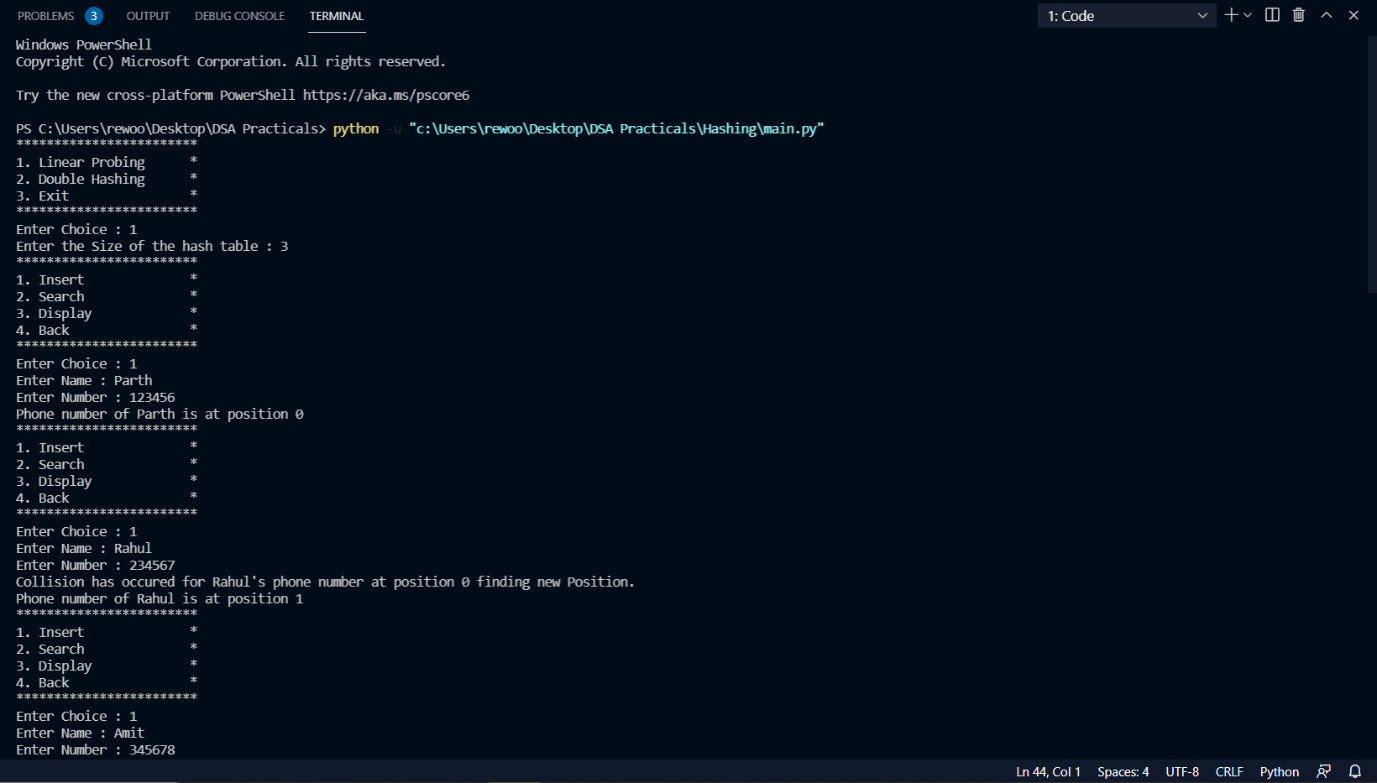
*self*.\_number = *number*

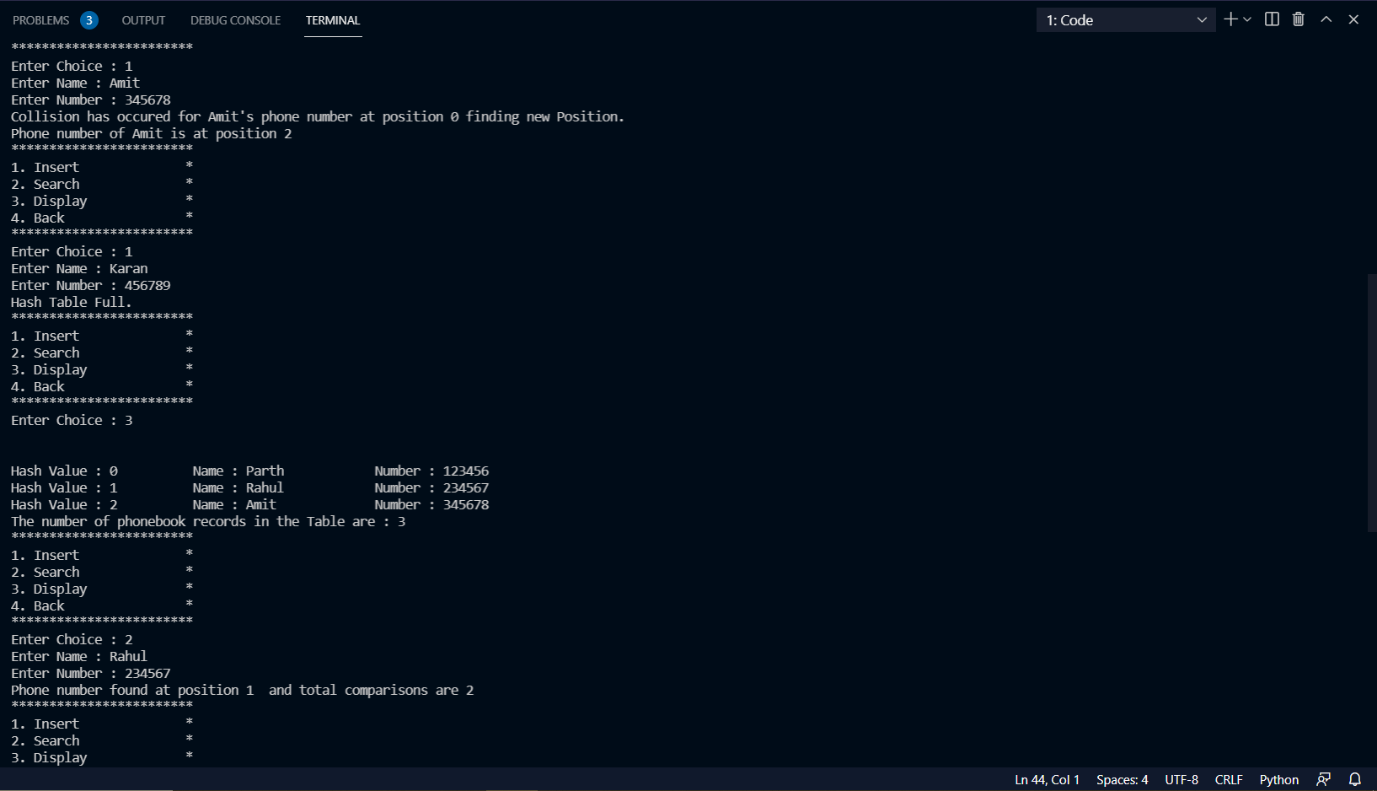
*def* \_\_str\_\_(*self*):

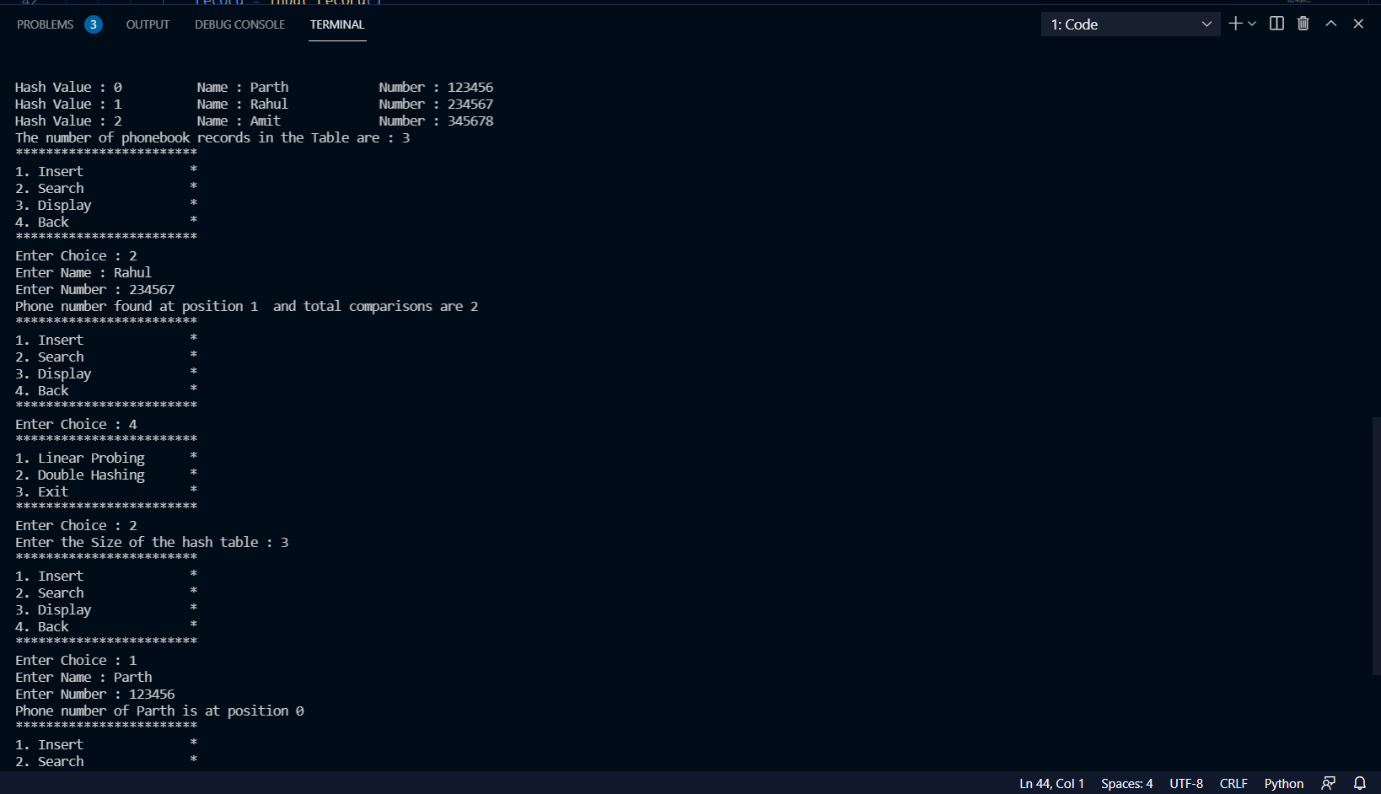
        record = "Name : "+str(*self*.get\_name())+"\t"+"\tNumber : "+str(*self*.get\_number())

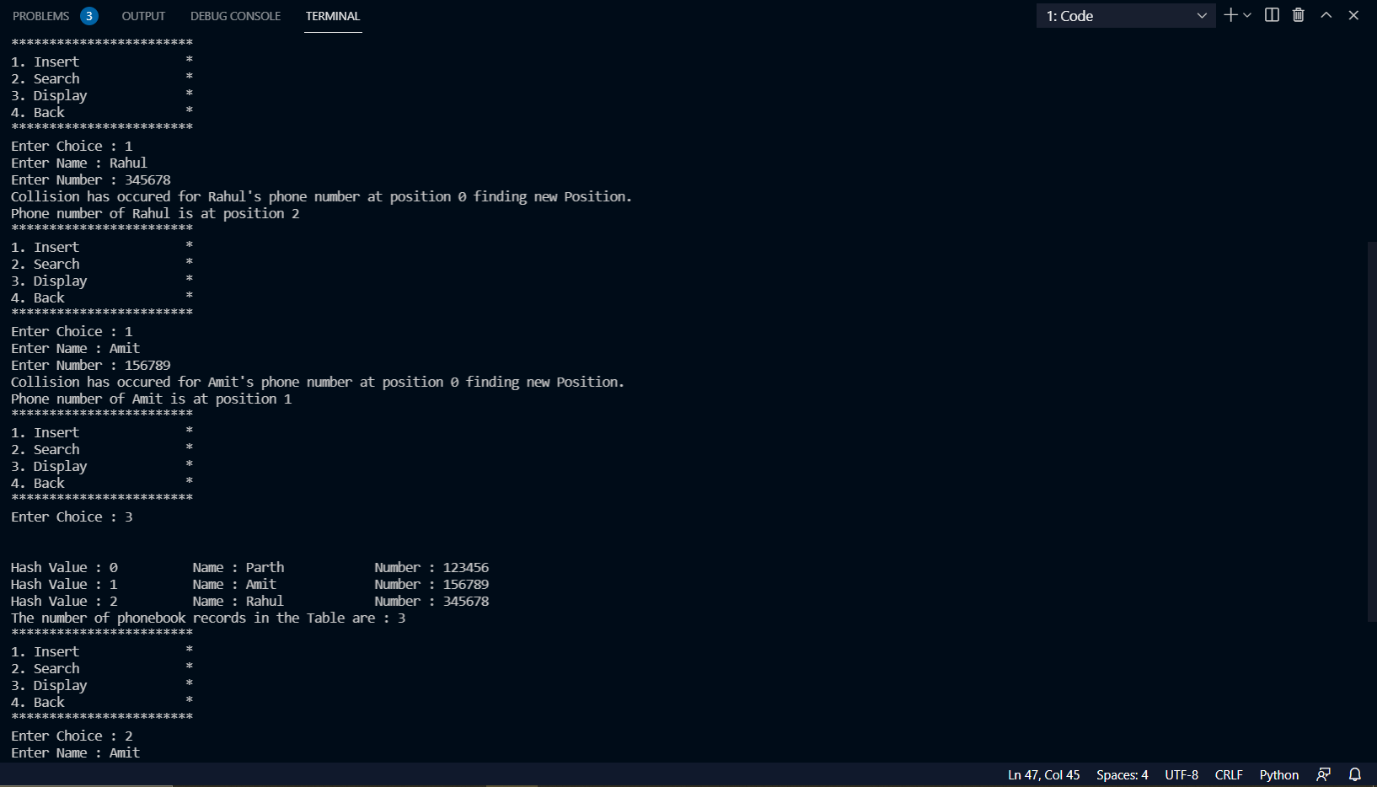
        return record

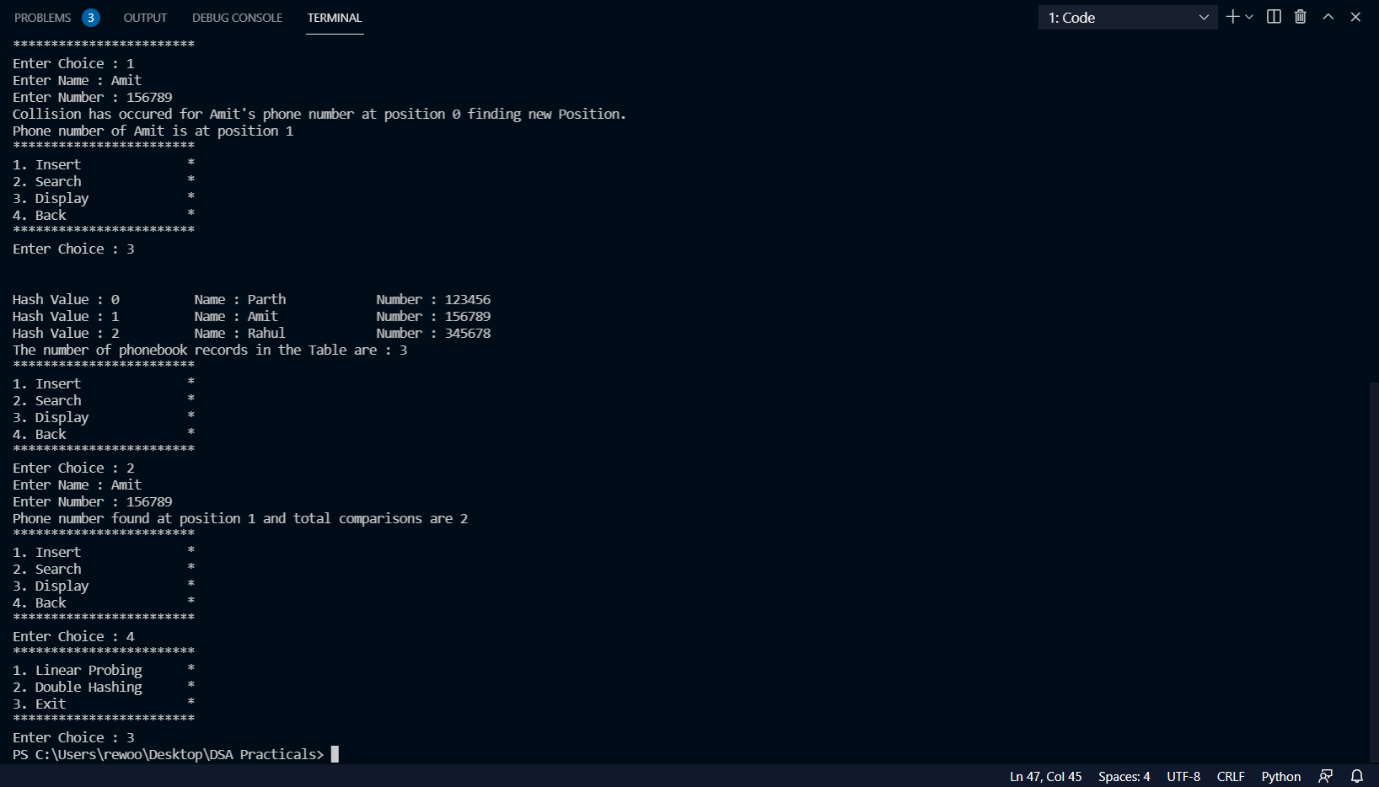
**Output:**











**Conclusion:**

Thus we have implemented a Telephone Book Database using collision handling techniques in Hash Table.

Experiment No. 4

**Experiment Name:** Set Operations

**Aim:** To create an ADT that implement the Set concept. a) Add (new element) – Place a value into the set. b) Remove (element) – Remove the value. c) Contains (element) – Return true if element is in collection. d) Size – Return number of values in collection. e) Iterator – Return an iterator used to loop over collection. f) Intersections of two sets. g) Union of two sets. h) Difference between two sets. i) Subset.

**Objective/Theory**:

Sets:

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Sets:

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

Modifying a set in Python:

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

Removing elements from a set:

A particular item can be removed from a set using the methods discard() and remove().

The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

Similarly, we can remove and return an item using the pop() method.

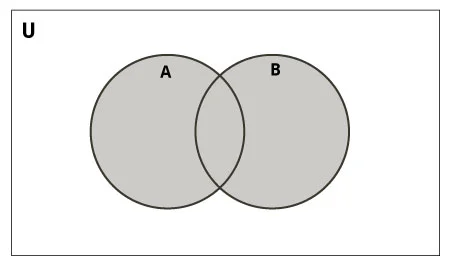
Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the clear() method.

Set Operations:

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

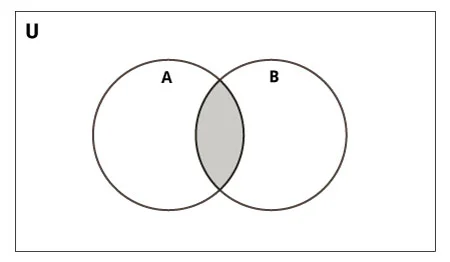
***Set Union:***



Union of A and B is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the union() method.

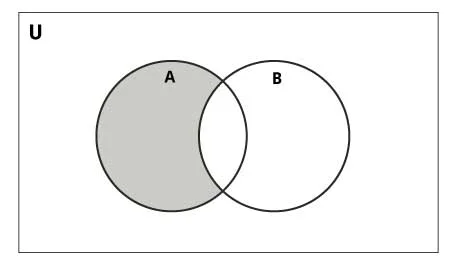
***Set Intersection:***



Intersection of A and B is a set of elements that are common in both the sets.

Intersection is performed using & operator. Same can be accomplished using the intersection() method.

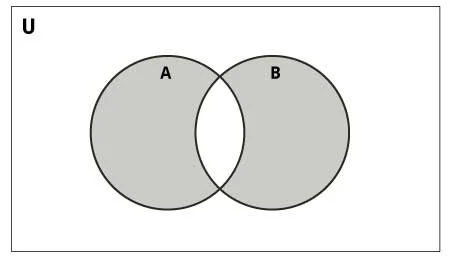
***Set Difference:***



Difference of the set B from set A(A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A.

Difference is performed using - operator. Same can be accomplished using the difference() method.

***Set Symmetric Difference:***



Symmetric Difference of A and B is a set of elements in A and B but not in both (excluding the intersection).

Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric\_difference().

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

*add()* - Adds an element to the set

*clear()* - Removes all elements from the set

*copy()* - Returns a copy of the set

*difference()* - Returns the difference of two or more sets as a new set

*difference\_update()* - Removes all elements of another set from this set

*discard()* - Removes an element from the set if it is a member. (Do nothing if the element is not in set)

*intersection()* - Returns the intersection of two sets as a new set

*intersection\_update()* - Updates the set with the intersection of itself and another

*isdisjoint()* - Returns True if two sets have a null intersection

*issubset()* - Returns True if another set contains this set

*issuperset()* - Returns True if this set contains another set

*pop()* - Removes and returns an arbitrary set element. Raises KeyError if the set is empty

*remove()* - Removes an element from the set. If the element is not a member, raises a KeyError

*symmetric\_difference()* - Returns the symmetric difference of two sets as a new set

*symmetric\_difference\_update()* - Updates a set with the symmetric difference of itself and another

*union()* - Returns the union of sets in a new set

*update()* - Updates the set with the union of itself and others

**Program**:

SetOperations.py

*class* Set :

    # Creates an empty set instance.

*def* \_\_init\_\_( *self*, *initElementsCount* ):

*self*.\_s = []

        for i in range(*initElementsCount*) :

            e = int(input("Enter Element {} : ".format(i+1)))

*self*.add(e)

*def* get\_set(*self*):

        return *self*.\_s

*def* \_\_str\_\_(*self*):

        string = "\n{ "

        for i in range(len(*self*.get\_set())):

            string = string + str(*self*.get\_set()[i])

            if i != len(*self*.get\_set())-1:

                string = string + " , "

        string = string + " }\n"

        return string

    # Returns the number of items in the set.

*def* \_\_len\_\_( *self* ):

        return len( *self*.\_s )

    # Determines if an element is in the set.

*def* \_\_contains\_\_( *self*, *e* ):

        return *e* in *self*.\_s

   # Determines if the set is empty.

*def* isEmpty( *self* ):

        return len(*self*.\_s) == 0

   # Adds a new unique element to the set.

*def* add( *self*, *e* ):

        if *e* not in *self* :

*self*.\_s.append( *e* )

   # Removes an e from the set.

*def* remove( *self*, *e* ):

        if *e* in *self*.get\_set():

*self*.get\_set().remove(*e*)

   # Determines if this set is equal to setB.

*def* \_\_eq\_\_( *self*, *setB* ):

        if len( *self* ) != len( *setB* ) :

            return False

        else :

            return *self*.isSubsetOf( *setB* )

   # Determines if this set is a subset of setB.

*def* isSubsetOf( *self*, *setB* ):

     for e in *setB*.get\_set() :

         if e not in *self*.get\_set() :

             return False

     return True

  # Determines if this set is a proper subset of setB.

*def* isProperSubset( *self*, *setB* ):

        if *self*.isSubsetOf(*setB*) and not *setB*.isSubsetOf(*self*):

            return True

        return False

   # Creates a new set from the union of this set and setB.

*def* union( *self*, *setB* ):

     newSet = *self*

     for e in *setB* :

         if e not in *self*.get\_set() :

             newSet.add(e)

     return newSet

   # Creates a new set from the intersection: self set and setB.

*def* intersect( *self*, *setB* ):

        newSet = Set(0)

        for i in range(len(*self*.get\_set())) :

            for j in range(len(*setB*.get\_set())) :

                if *self*.get\_set()[i] == *setB*.get\_set()[j] :

                    newSet.add(*self*.get\_set()[i])

        return newSet

    # Creates a new set from the difference: self set and setB.

*def* difference( *self*, *setB* ):

        newSet = Set(0)

        for e in *self*.get\_set() :

            if e not in *setB*.get\_set():

                newSet.add(e)

        return newSet

     # Creates the iterator for traversing the list of items

*def* \_\_iter\_\_( *self* ):

        return iter(*self*.\_s)

Menu.py

from SetOperations import Set

*def* createSet():

    n=int(input("Enter number of Elements in set : "))

    s = Set(n)

    return s

choice = 0

print("Create Set A : ")

s1 = createSet()

print(str(s1))

while choice != 10:

    print("|-------------------|")

    print("| Menu              |")

    print("| 1.Add             |")

    print("| 2.Remove          |")

    print("| 3.Contains        |")

    print("| 4.Size            |")

    print("| 5.Intersection    |")

    print("| 6.Union           |")

    print("| 7.Difference      |")

    print("| 8.Subset          |")

    print("| 9.Proper Subset   |")

    print("| 10.Exit           |")

    print("|-------------------|")

    choice = int(input("Enter Choice : "))

    if choice==1:

        e = int(input("Enter Number to Add : "))

        s1.add(e)

        print(str(s1))

    elif choice==2:

        e = int(input("Enter Number to Remove : "))

        s1.remove(e)

        print(str(s1))

    elif choice==3:

        e = int(input("Enter Number to Search : "))

        if e in s1:

            print("Number Present in Set.")

        else:

            print("Number is not Present in Set.")

        print(str(s1))

    elif choice==4:

        print("Set Contains {} elements".format(len(s1)))

    elif choice==5:

        print("Create a Set B for doing Intersection Operation : ")

        s2 = createSet()

        s3 = s1.intersect(s2)

        print("Set A = "+str(s1))

        print("Set B = "+str(s2))

        print("Intersection = "+str(s3))

    elif choice==6:

        print("Create a Set B for doing Union Operation : ")

        s2 = createSet()

        s3 = s1.union(s2)

        print("Set A = "+str(s1))

        print("Set B = "+str(s2))

        print("Union = "+str(s3))

    elif choice==7:

        print("Create a Set B for calculating Set Difference : ")

        s2 = createSet()

        s3 = s1.difference(s2)

        print("Set A = "+str(s1))

        print("Set B = "+str(s2))

        print("Difference = "+str(s3))

    elif choice==8:

        print("Create a Set B for checking Subset : ")

        s2 = createSet()

        isSubset = s1.isSubsetOf(s2)

        print("Set A = "+str(s1))

        print("Set B = "+str(s2))

        if isSubset:

            print("Set B is the Subset of Set A.")

        else:

            print("Set B is not a Subset of Set A.")

    elif choice==9:

        print("Create a Set B for checking ProperSubset : ")

        s2 = createSet()

        isProperSubset = s1.isProperSubset(s2)

        print("Set A = "+str(s1))

        print("Set B = "+str(s2))

        if isProperSubset:

            print("Set B is the Proper Subset of Set A.")

        else:

            print("Set B is not a Proper Subset of Set A.")

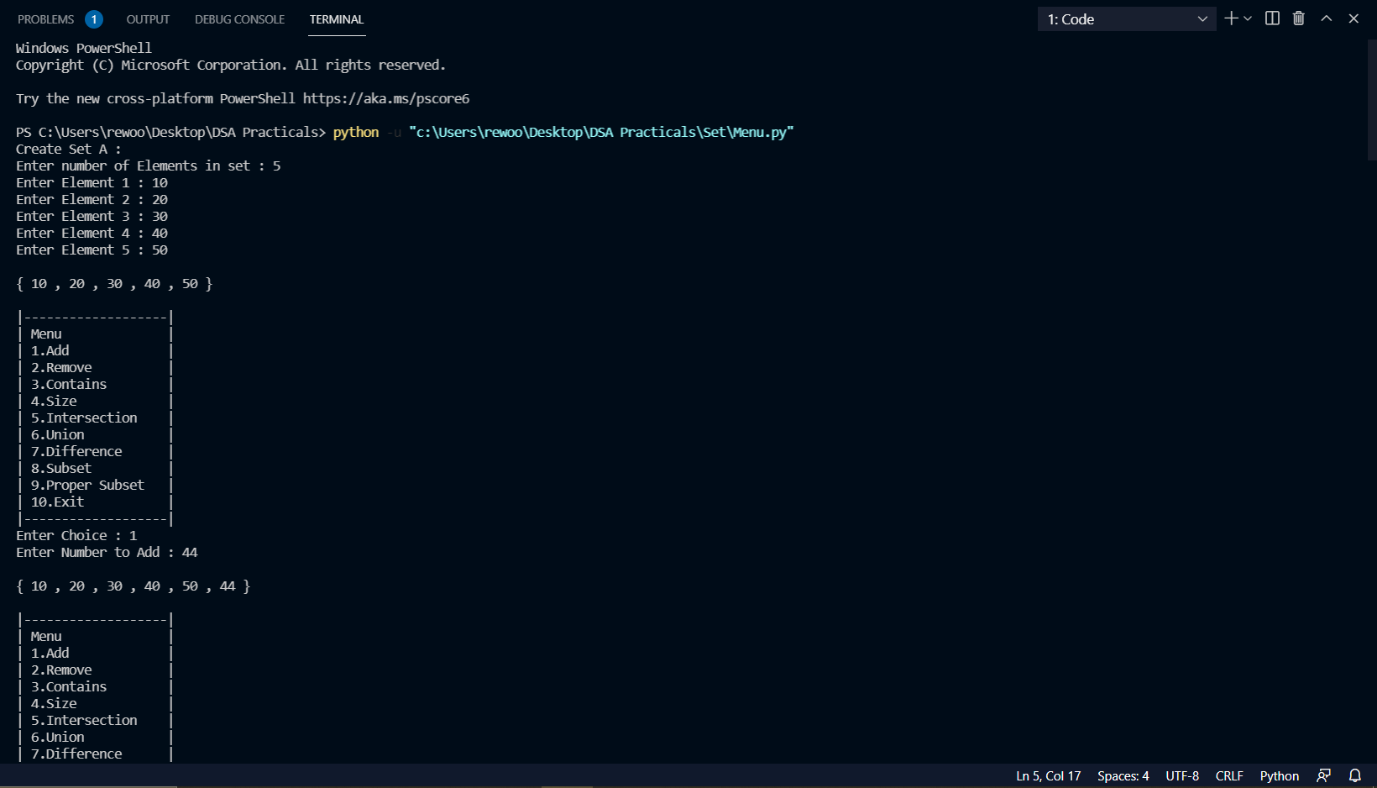
    elif choice==10:

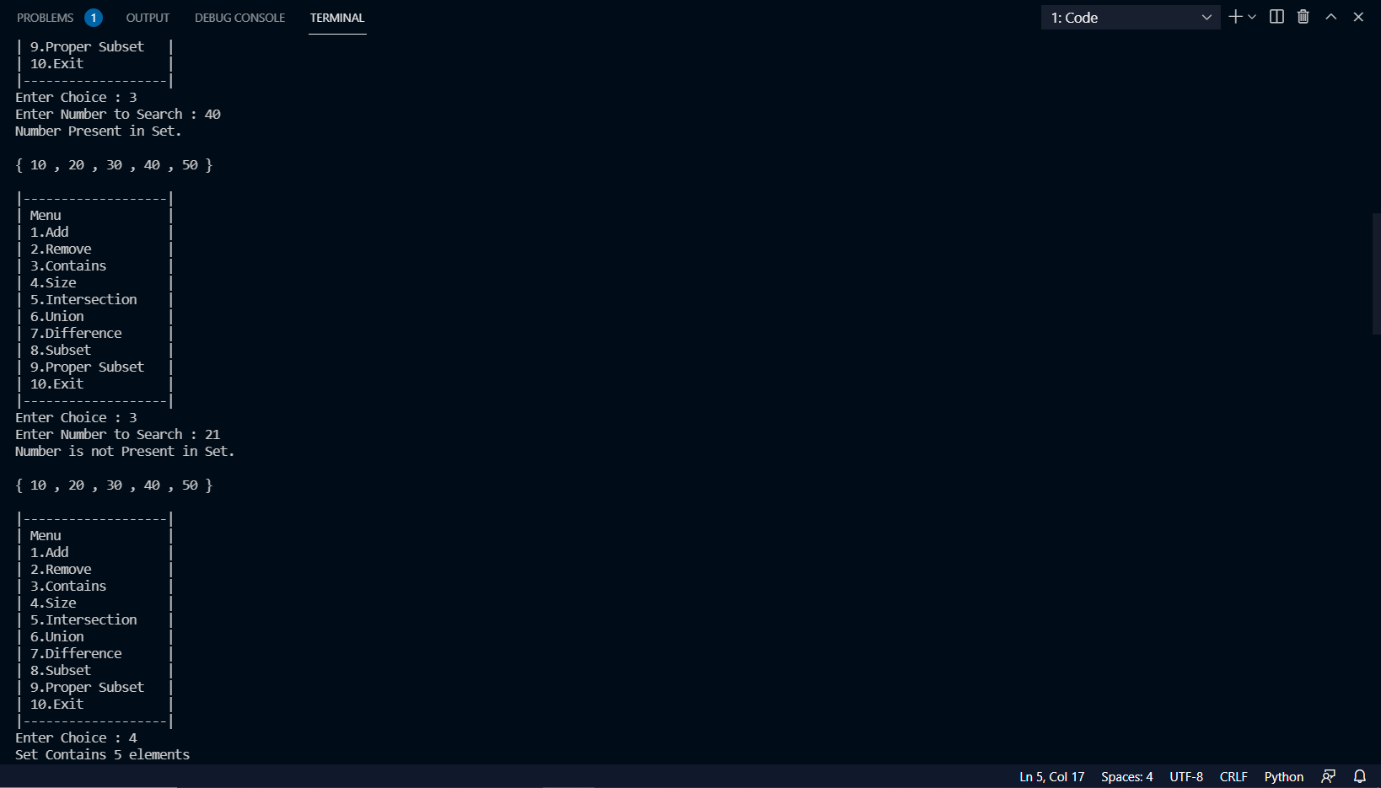
        break;

    elif choice<1 or choice>10:

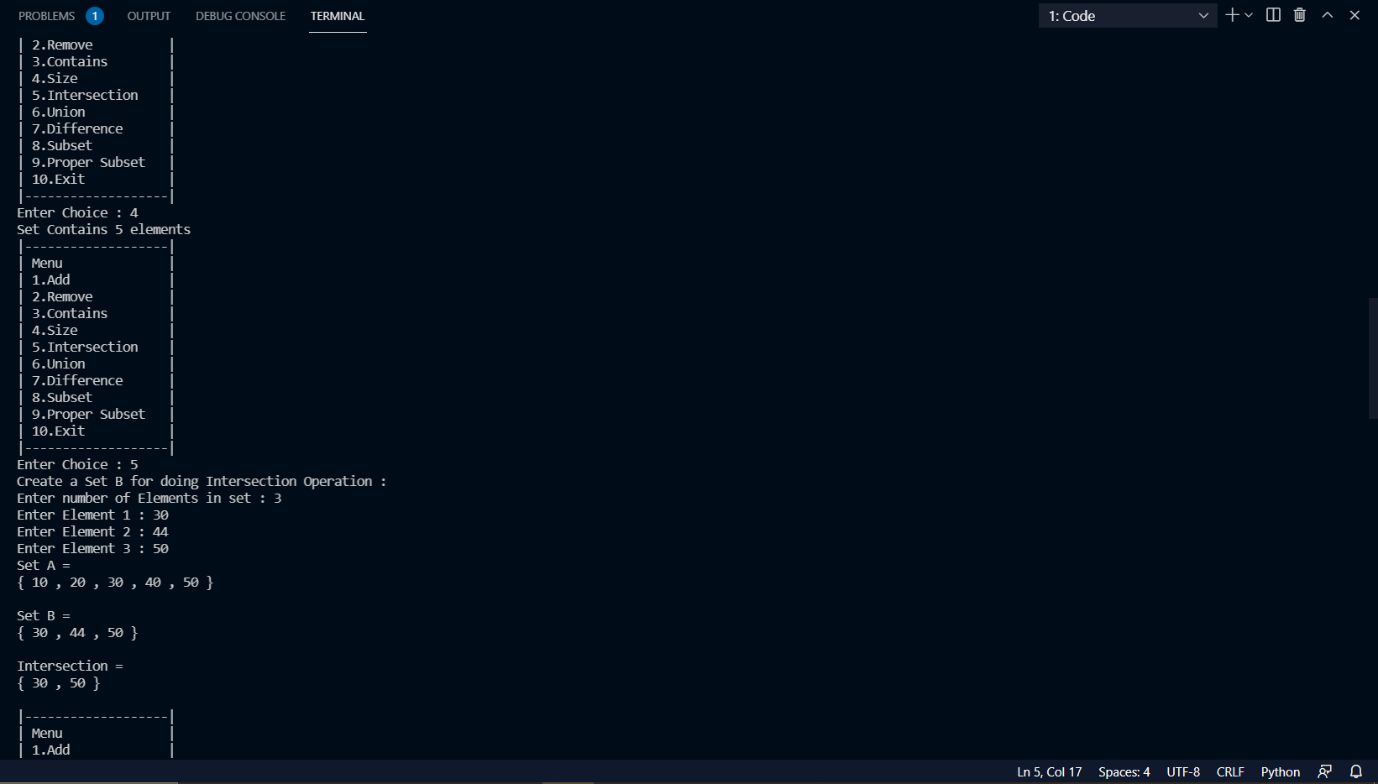
        print("Please Enter Valid Choice!!")

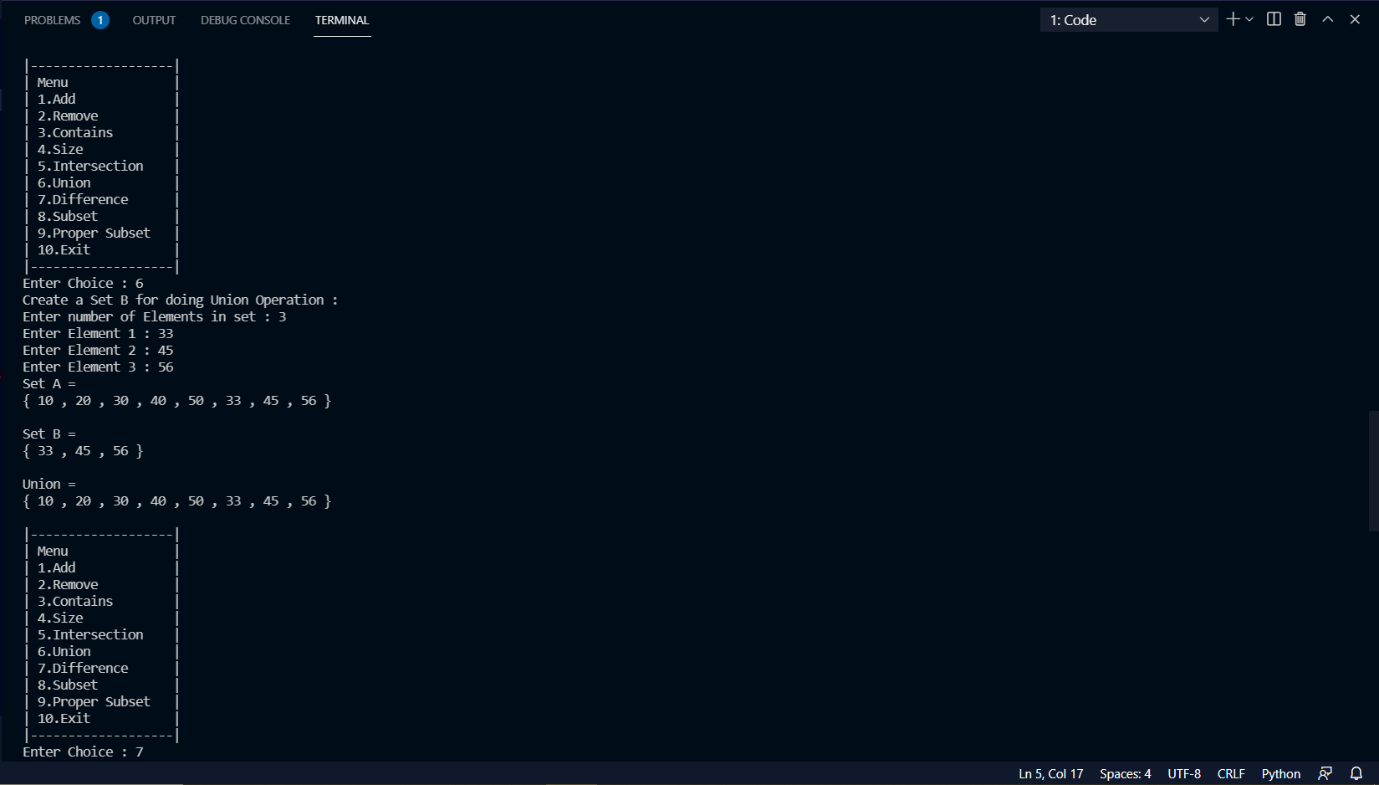
**Output:**

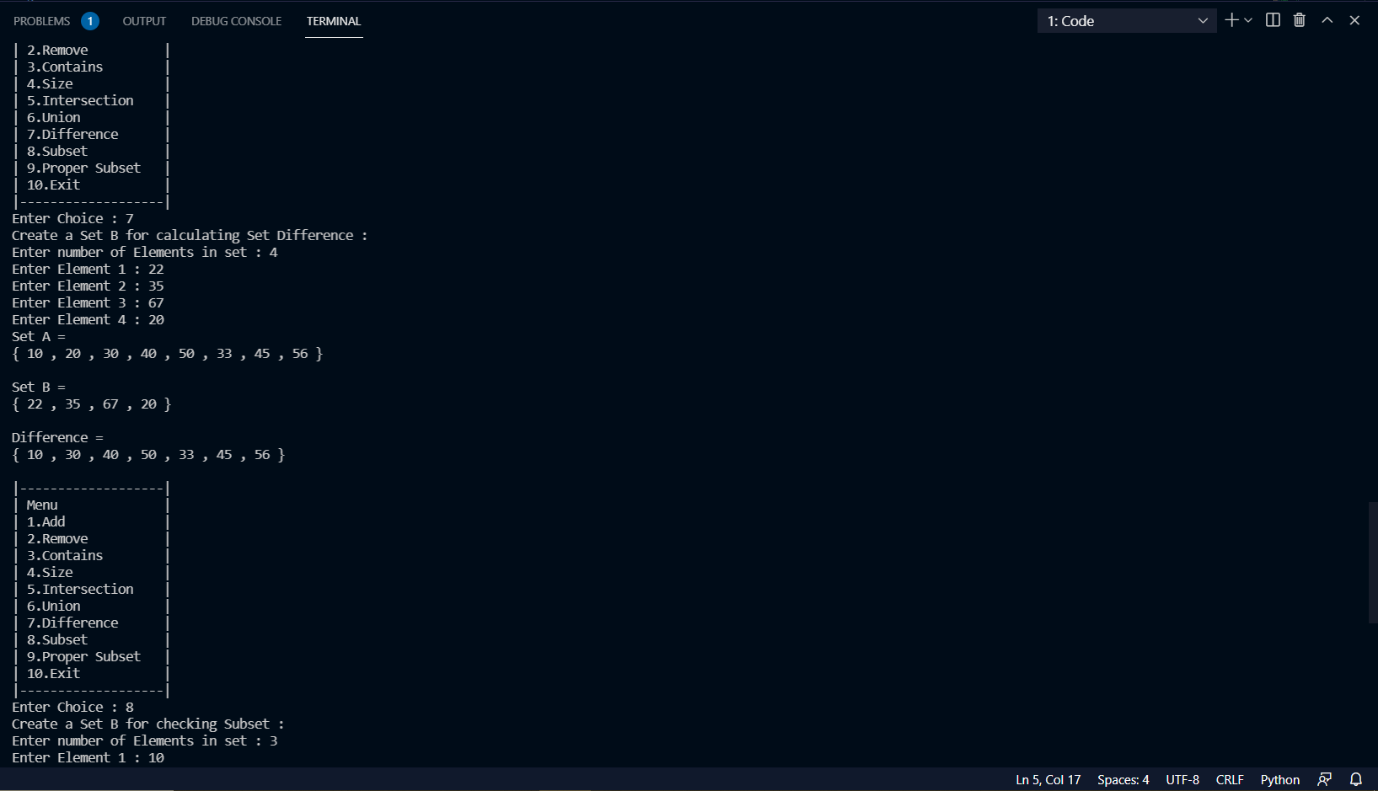


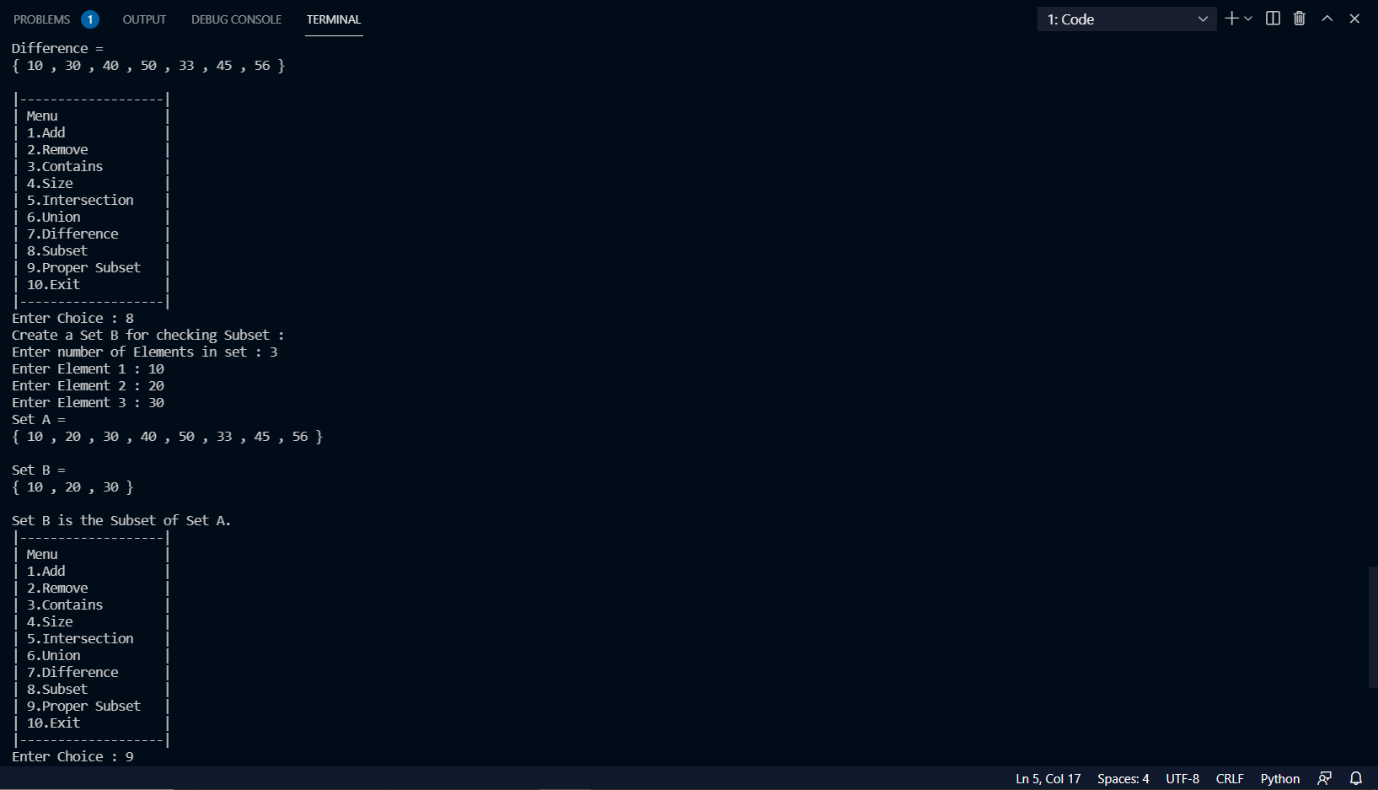


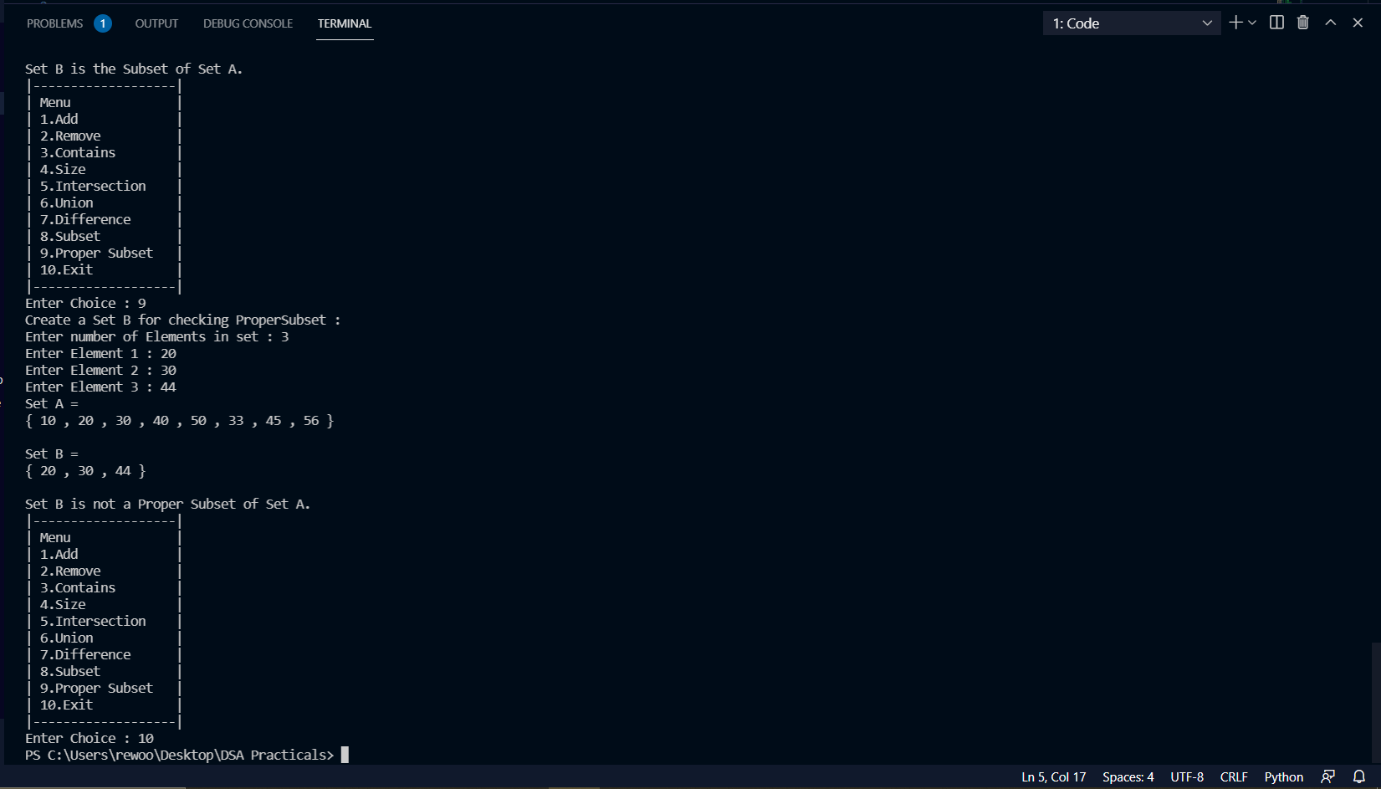












**Conclusion:**

Thus we have implemented Set operations.

**GROUP B**

Experiment No. 5

**Experiment Name**: Tree Creation and Display

**Aim**: C++ Program To read details of 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.

**Objective/Theory:**

*Tree Data Structure*:

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



*Tree Terminologies:*

**Node:**

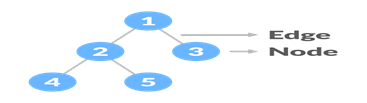
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes** or **external nodes** that do not contain a link/pointer to child nodes.

The node having at least a child node is called an **internal node**.

**Edge:**

It is the link between any two nodes.



**Root:**

It is the topmost node of a tree.

**Height of a Node:**

The height of a node is the number of edges from the node to the deepest leaf (the longest path from the node to a leaf node).

**Depth of a Node:**

The depth of a node is the number of edges from the root to the node.

**Height of a Tree:**

The height of a Tree is the height of the root node or the depth of the deepest node.

**Degree of a Node:**

The degree of a node is the total number of branches of that node.

**Forest:**

A collection of disjoint trees is called a forest.

Tree Traversal:

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

Tree Applications:

1. Binary Search Trees (BSTs) are used to quickly check whether an element is present in a set or not.
2. Heap is a kind of tree that is used for heap sort.
3. A modified version of a tree called Tries is used in modern routers to store routing information.
4. Compilers use a syntax tree to validate the syntax of every program you write.

**Program:**

/\*

Experiment 5 : C++ Program To read details of 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>

#include <string.h>

using *namespace* std;

*struct* node // Node Declaration

{

    string label;

    //char label[10];

*int* ch\_count;

*struct* node \*child[10];

} \* root;

*class* GT // Class Declaration

{

*public:*

*void* create\_tree();

*void* display(node \**r1*);

    GT()

    {

        root = NULL;

    }

};

*void* GT::create\_tree()

{

*int* tbooks, tchapters, i, j, k;

    root = new node;

    cout << "Enter name of book : ";

    cin.get();

    getline(cin, root->label);

    //cin>>root->label;

    cout << "Enter number of chapters in book : ";

    cin >> tchapters;

    root->ch\_count = tchapters;

    for (i = 0; i < tchapters; i++)

    {

        root->child[i] = new node;

        cout << "Enter the name of Chapter " << i + 1 << " : ";

        cin.get();

        getline(cin, root->child[i]->label);

        cout << "Enter number of sections in Chapter : " << root->child[i]->label << " : ";

        cin >> root->child[i]->ch\_count;

        for (j = 0; j < root->child[i]->ch\_count; j++)

        {

            root->child[i]->child[j] = new node;

            cout << "Enter Name of Section " << j + 1 << " : ";

            cin.get();

            getline(cin, root->child[i]->child[j]->label);

        }

    }

}

*void* GT::display(node \**r1*)

{

*int* i, j, k, tchapters;

    if (*r1* != NULL)

    {

        cout << "\n-----Book Hierarchy---";

        cout << "\n Book title : " << *r1*->label;

        tchapters = *r1*->ch\_count;

        for (i = 0; i < tchapters; i++)

        {

            cout << "\nChapter " << i + 1;

            cout << " : " << *r1*->child[i]->label;

            cout << "\nSections : ";

            for (j = 0; j < *r1*->child[i]->ch\_count; j++)

            {

                cout << "\n"

                     << *r1*->child[i]->child[j]->label;

            }

        }

    }

    cout << endl;

}

*int* main()

{

*int* choice;

    GT gt;

    while (1)

    {

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

        cout << "Book Tree Creation" << endl;

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

        cout << "1.Create" << endl;

        cout << "2.Display" << endl;

        cout << "3.Quit" << endl;

        cout << "Enter your choice : ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            gt.create\_tree();

        case 2:

            gt.display(root);

            break;

        case 3:

            cout << "Thanks for using this program!!!";

            exit(1);

        default:

            cout << "Wrong choice!!!" << endl;

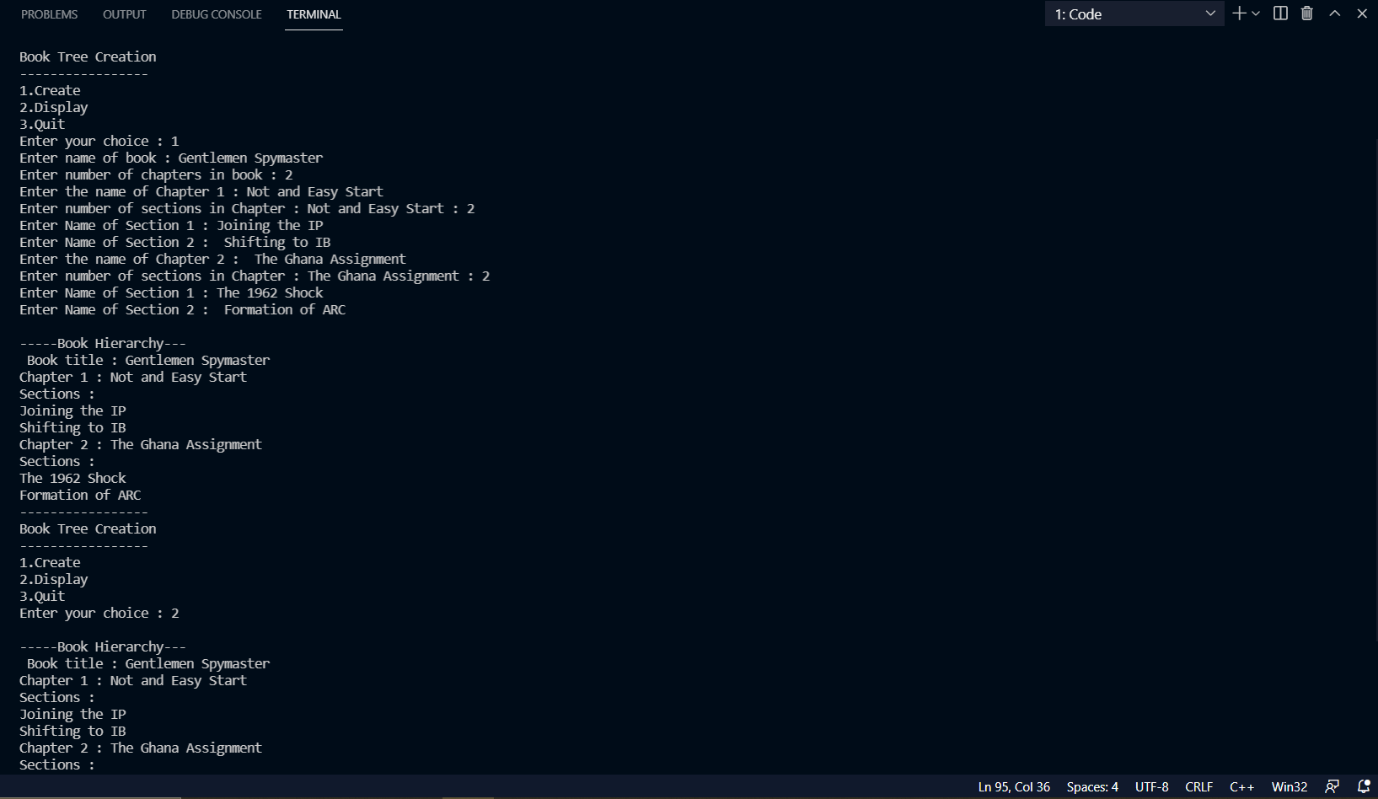
        }

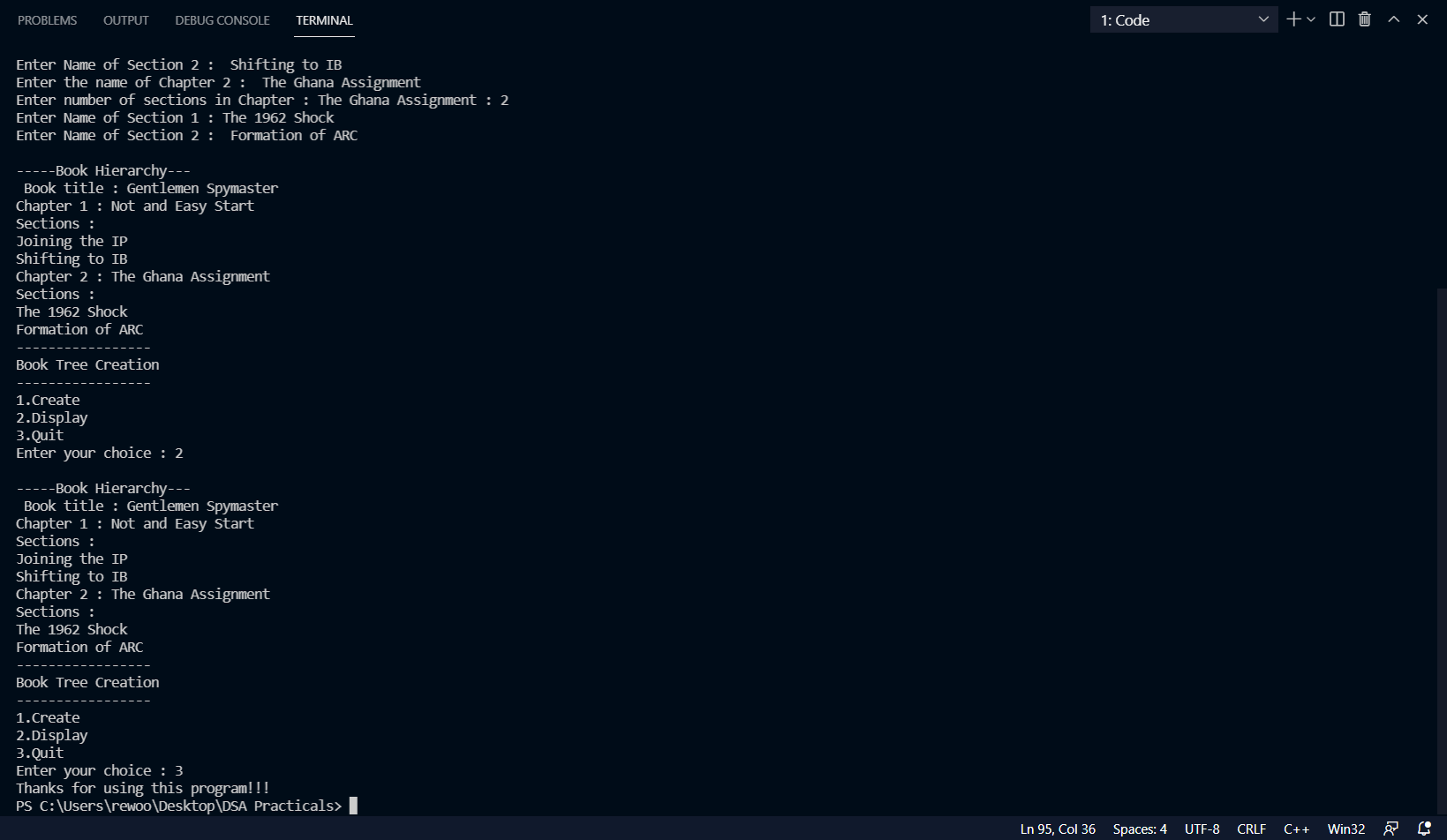
    }

    return 0;

}

**Output:**





**Conclusion:**

Thus we have implemented Tree data structure and performed operations on it.

**Experiment No. 6**

**Experiment Name:** Binary Search Tree and its operations.

**Aim:** Beginning with an empty binary search tree, construct binary search tree by inserting the values in the order given. After constructing a binary tree – a) Insert new node. b) Find number of nodes in longest path. c) Minimum data value found in the tree. d) Change a tree so that the roles of the left and right pointers are swapped at every node. e) Search a value.

**Objective/Theory:**

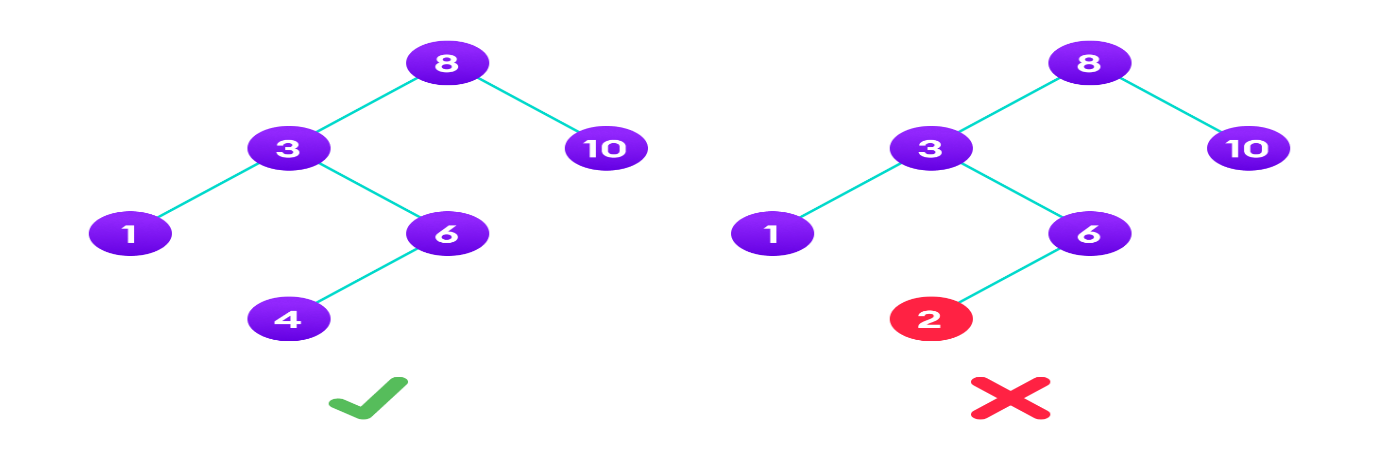
Binary Search Tree:

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers. It is called a binary tree because each tree node has a maximum of two children. It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node.
2. All nodes of right subtree are more than the root node.
3. Both subtrees of each node are also BSTs i.e. they have the above two properties.

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.



There are two basic operations that you can perform on a binary search tree:

**Search Operation:**

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

*Algorithm:*

If root == NULL

return NULL;

If number == root->data

return root->data;

If number < root->data

return search(root->left)

If number > root->data

return search(root->right)

**Insert Operation:**

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

*Algorithm:*

If node == NULL

return createNode(data)

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

**Deletion Operation:**

There are three cases for deleting a node from a binary search tree.

*Case I:*

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

*Case II:*

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

*Case III:*

In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the in-order successor of that node.
2. Replace the node with the in-order successor.
3. Remove the in-order successor from its original position.

**Traversal:**

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

*In-order Traversal:*

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.

Algorithm:

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Visit root node.

**Step 3** − Recursively traverse right subtree.

*Pre-order Traversal:*

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Algorithm:

Until all nodes are traversed −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

*Post-order Traversal:*

In this traversal method, first we traverse the left subtree, then the right subtree and finally the root node.

Algorithm:

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Recursively traverse right subtree.

**Step 3** − Visit root node.

**Binary Search Tree Applications:**

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

**Program:**

#include <iostream>

//#include <math.h>

using *namespace* std;

*struct* Bstnode

{

*int* data;

    Bstnode \*left = NULL;

    Bstnode \*right = NULL;

};

*class* Btree

{

*int* n;

*int* x;

*int* flag;

*public:*

    Bstnode \*root;

    Btree()

    {

        root = NULL;

    }

    Bstnode \*GetNewNode(*int* *in\_data*)

    {

        Bstnode \*ptr = new Bstnode();

        ptr->data = *in\_data*;

        ptr->left = NULL;

        ptr->right = NULL;

        return ptr;

    }

    Bstnode \*insert(Bstnode \**temp*, *int* *in\_data*)

    {

        if (*temp* == NULL)

        {

*temp* = GetNewNode(*in\_data*);

        }

        else if (*temp*->data > *in\_data*)

        {

*temp*->left = insert(*temp*->left, *in\_data*);

        }

        else

        {

*temp*->right = insert(*temp*->right, *in\_data*);

        }

        return *temp*;

    }

*void* input()

    {

        cout << "Enter number of elements in BST : ";

        cin >> n;

        for (*int* i = 0; i < n; i++)

        {

            cout << "Element "<<i+1<<" : ";

            cin >> x;

            root = insert(root, x);

        }

    }

   /\*int\*/*void* search(Bstnode \**temp*, *int* *in\_data*)

    {

        if (*temp* != NULL)

        {

            if (*temp*->data == *in\_data*)

            {

                cout<<"Element Found!!!"<<endl;

                //return 1;

            }

            else if (*in\_data* < *temp*->data)

            {

                this->search(*temp*->left, *in\_data*);

            }

            else if (*in\_data* > *temp*->data)

            {

                this->search(*temp*->left, *in\_data*);

            }

        }

        else

        {

            cout<<"Element Not Found!!!"<<endl;

            cout<<*temp*->data<<endl;

           // return 0;

        }

    }

*void* minvalue(Bstnode \**temp*)

    {

        while (*temp*->left != NULL)

        {

*temp* = *temp*->left;

        }

        cout << "Minimum Value = " << *temp*->data << endl;

    }

*void* mirror(Bstnode \**temp*)

    {

        if (*temp* == NULL)

        {

            return;

        }

        else

        {

            Bstnode \*ptr;

            mirror(*temp*->left);

            mirror(*temp*->right);

            ptr = *temp*->left;

*temp*->left = *temp*->right;

*temp*->right = ptr;

        }

    }

*void* display()

    {

        cout << endl

             << "--- INORDER TRAVERSAL ---" << endl;

        inorder(root);

        cout << endl;

        cout << endl

             << "--- POSTORDER TRAVERSAL ---" << endl;

        postorder(root);

        cout << endl;

        cout << endl

             << "--- PREORDER TRAVERSAL ---" << endl;

        preorder(root);

        cout << endl;

    }

*void* inorder(Bstnode \**temp*)

    {

        if (*temp* != NULL)

        {

            inorder(*temp*->left);

            cout << *temp*->data << "  ";

            inorder(*temp*->right);

        }

    }

*void* postorder(Bstnode \**temp*)

    {

        if (*temp* != NULL)

        {

            postorder(*temp*->left);

            postorder(*temp*->right);

            cout << *temp*->data << " ";

        }

    }

*void* preorder(Bstnode \**temp*)

    {

        if (*temp* != NULL)

        {

            cout << *temp*->data << " ";

            preorder(*temp*->left);

            preorder(*temp*->right);

        }

    }

*int* depth(Bstnode \**temp*)

    {

        if (*temp* == NULL)

            return 0;

        return (max((depth(*temp*->left)), (depth(*temp*->right))) + 1);

    }

};

*int* main()

{

    Btree obj;

*int* ch,s;

    do

    {

        cout<<"\n\n1.Create\n2.Display\n3.Search\n4.Minimum Value\n5.Mirror\n6.Height\n7.Exit\n";

        cout<<"Enter your choice : ";

        cin>>ch;

        switch (ch)

        {

        case 1:

            obj.input();

            cout<<endl<<obj.root->data<<endl;

            break;

        case 2:

            obj.display();

            break;

        case 3:

            //int a;

            cout<<"Enter the element to be searched : ";

            cin>>s;

            /\*a =\*/ obj.search(obj.root, s);

            // if (a == 0)

            // {

            //     cout << "!!!Element not found!!!" << endl;

            // }

            // else

            //     cout << "!!!Element Found!!!" << endl;

            break;

        case 4:

            obj.minvalue(obj.root);

            break;

        case 5:

            obj.mirror(obj.root);

            obj.inorder(obj.root);

            break;

        case 6:

            cout<<"Height : "<<obj.depth(obj.root)<<endl;

            break;

        case 7:

            cout<<"Thanks for using this program!!!";

            exit(0);

        default:

            cout<<"Wrong Choice!!!"<<endl;

            break;

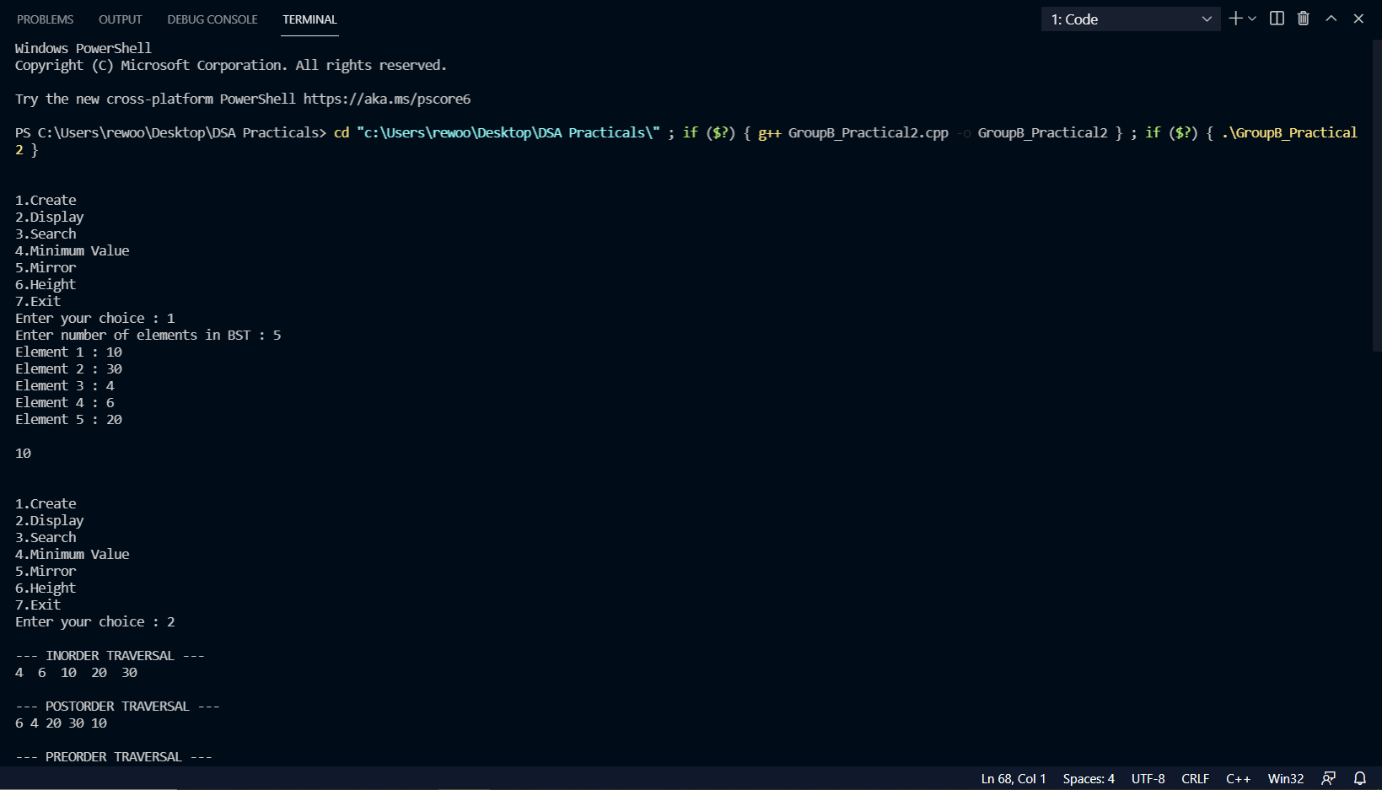
        }

    } while (ch!=8);

    return 0;

}

**Output:**







**Conclusion:**

Thus we have implemented Binary Search tree and performed operations on it.

**Experiment No. 7**

**Experiment Name:** Expression Tree and operations on it.

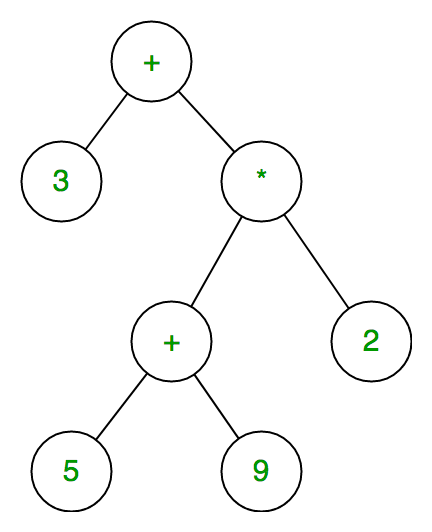
**Aim:** Construct an expression tree from the given prefix and traverse it using post order traversal and then delete the entire tree.

**Objective/Theory:**

Expression Tree:

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand.

For example expression tree for 3 + ((5+9)\*2) would be:



In-order traversal of expression tree produces infix version of given postfix expression and same with post-order traversal it gives postfix expression.

Construction of Expression Tree:

For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

**Program:**

/\*

Experiment 3 : Construct an expression tree from the given prefix and traverse it using post order traversal and then delete the entire tree.

\*/

#include <iostream>

#include <string.h>

using *namespace* std;

*struct* node

{

*char* data;

    node \*left;

    node \*right;

};

*class* tree

{

*char* prefix[20];

*public:*

    node \*top;

*void* expression(*char*[]);

*void* display(node \*);

*void* non\_rec\_postorder(node \*);

*void* del(node \*);

};

*class* stack1

{

    node \*data[30];

*int* top;

*public:*

    stack1()

    {

        top = -1;

    }

*int* empty()

    {

        if (top == -1)

            return 1;

        return 0;

    }

*void* push(node \**p*)

    {

        data[++top] = *p*;

    }

    node \*pop()

    {

        return (data[top--]);

    }

};

*void* tree::expression(*char* *prefix*[])

{

*char* c;

    stack1 s;

    node \*t1, \*t2;

*int* len, i;

    len = strlen(*prefix*);

    for (i = len - 1; i >= 0; i--)

    {

        top = new node;

        top->left = NULL;

        top->right = NULL;

        if (isalpha(*prefix*[i]))

        {

            top->data = *prefix*[i];

            s.push(top);

        }

        else if (*prefix*[i] == '+' || *prefix*[i] == '\*' || *prefix*[i] == '-' || *prefix*[i] == '/')

        {

            t2 = s.pop();

            t1 = s.pop();

            top->data = *prefix*[i];

            top->left = t2;

            top->right = t1;

            s.push(top);

        }

    }

    top = s.pop();

}

*void* tree::display(node \**root*)

{

    if (*root* != NULL)

    {

        cout << *root*->data;

        display(*root*->left);

        display(*root*->right);

    }

}

*void* tree::non\_rec\_postorder(node \**top*)

{

    stack1 s1, s2; /\*stack s1 is being used for flag . A NULL data implies that the right subtree has not been visited \*/

    node \*T = *top*;

    cout << "\n";

    s1.push(T);

    while (!s1.empty())

    {

        T = s1.pop();

        s2.push(T);

        if (T->left != NULL)

            s1.push(T->left);

        if (T->right != NULL)

            s1.push(T->right);

    }

    while (!s2.empty())

    {

*top* = s2.pop();

        cout << *top*->data;

    }

}

*void* tree::del(node \**node*)

{

    if (*node* == NULL)

        return;

    /\* first delete both subtrees \*/

    del(*node*->left);

    del(*node*->right);

    /\* then delete the node \*/

    cout <<endl<<"Deleting node : " << *node*->data<<endl;

    free(*node*);

}

*int* main()

{

*char* expr[20];

    tree t;

    cout <<"Enter prefix Expression : ";

    cin >> expr;

    cout << expr;

    t.expression(expr);

    //t.display(t.top);

    //cout<<endl;

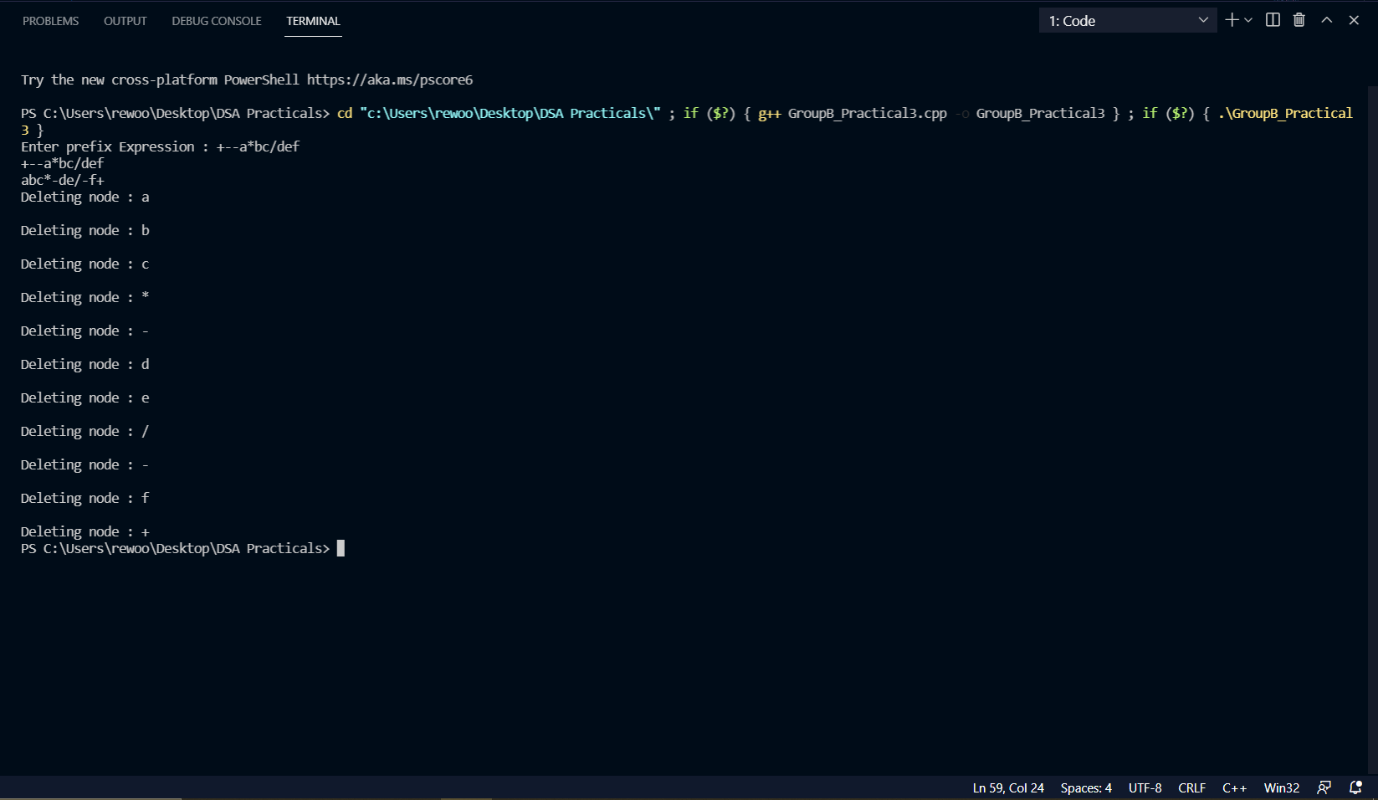
    t.non\_rec\_postorder(t.top);

    t.del(t.top);

    // t.display(t.top);

}

**Output:**



**Conclusion:**

Thus we have implemented and performed various operation on Expression Tree.

**GROUP C**

Experiment No. 13

**Experiment Name:** Graph using Adjacency Matrix and Adjacency List to perform DFS and BFS.

**Aim:** 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 a graph. Identify the prominent land marks as nodes and perform DFS and BFS on that. a) Adjacency Matrix b) Adjacency List.

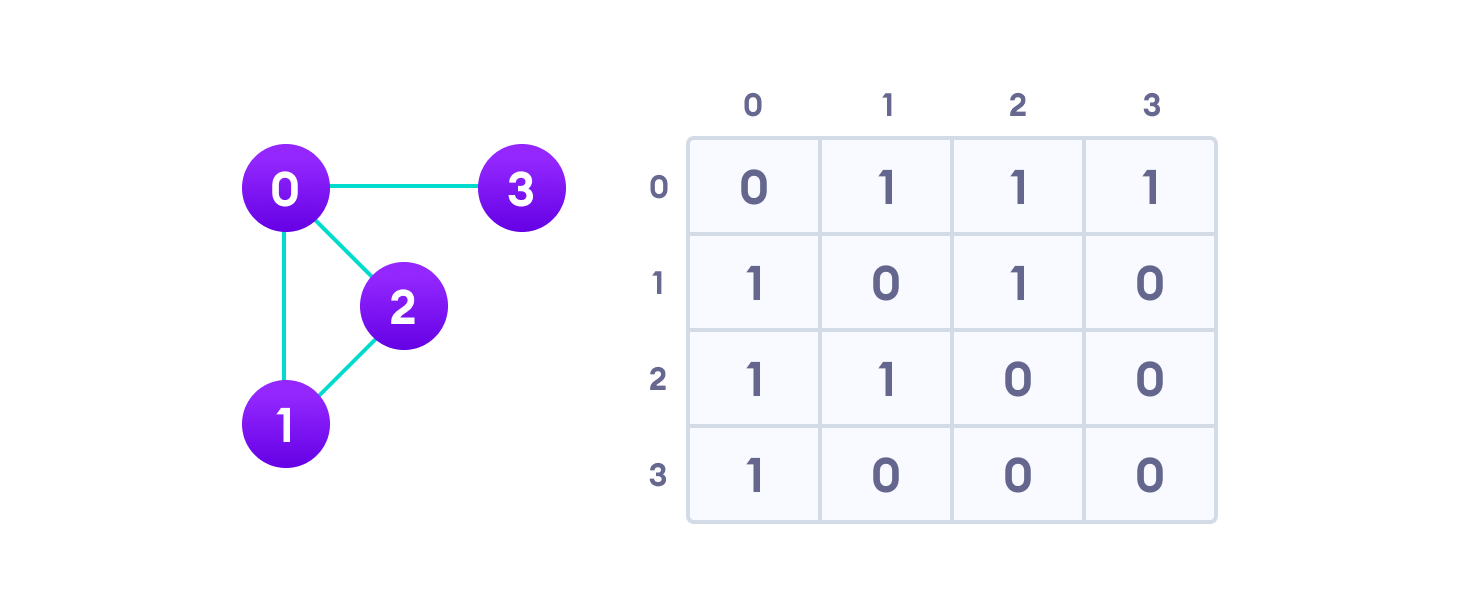
**Objective/Theory:**

Adjacency Matrix:

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

*Adjacency matrix representation:*

The size of the matrix is VxV where V is the number of vertices in the graph and the value of an entry Aij is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.

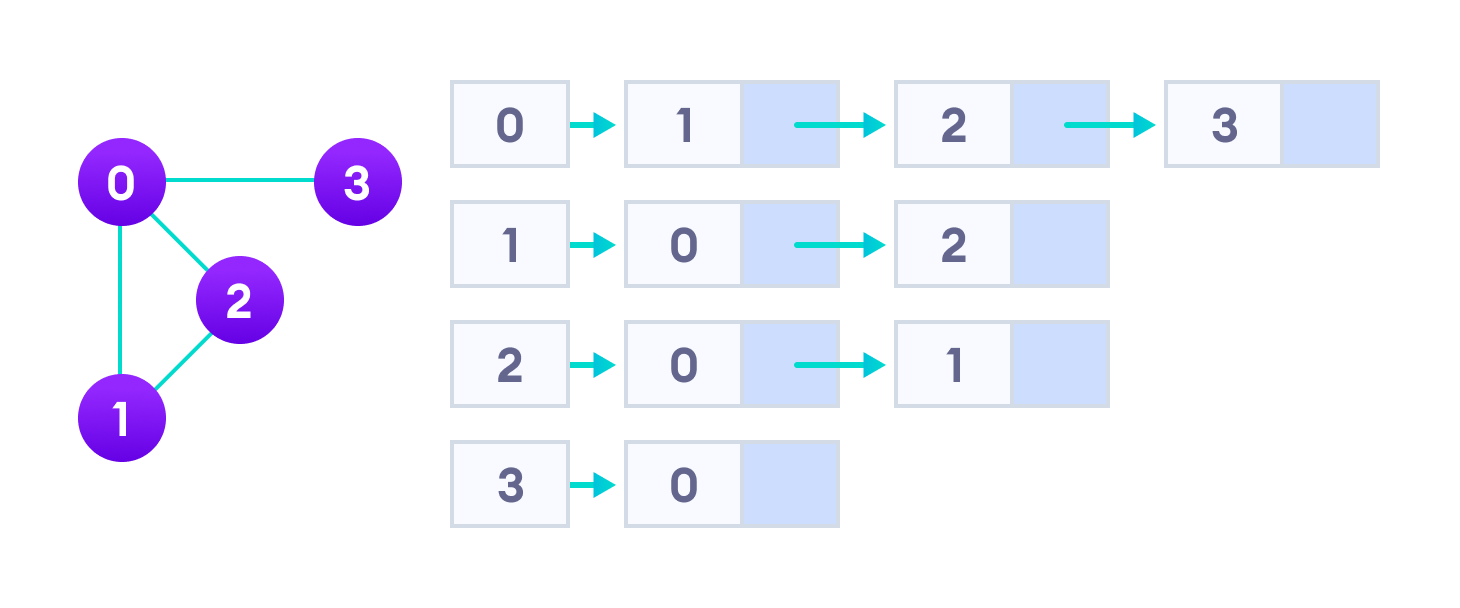


Adjacency List:

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

*Adjacency List representation:*



Depth First Search (DFS):

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

*Complexity of Depth First Search:*

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

*Application of DFS Algorithm:*

1. For finding the path.
2. To test if the graph is bipartite.
3. For finding the strongly connected components of a graph.
4. For detecting cycles in a graph.

Breadth First Search (BFS):

Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

*BFS Algorithm Complexity:*

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

*BFS Algorithm Applications:*

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

**Program:**

a)

/\*

Experiment 13 : Represent a given grapg using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as a graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

Adjacency Matrix : using adj matrix -BFS(Que)

\*/

#include <iostream>

#include <stdlib.h>

using *namespace* std;

*int* cost[10][10], i, j, k, n, qu[10], front, rear, v, visit[10], visited[10];

*int* stk[10], top, visit1[10], visited1[10];

*int* main()

{

*int* m;

    cout << "Enter number of vertices : ";

    cin >> n;

    cout << "Enter number of edges : ";

    cin >> m;

    cout << "\nEDGES :\n";

    for (k = 1; k <= m; k++)

    {

        cin >> i >> j;

        cost[i][j] = 1;

        cost[j][i] = 1;

    }

    //display function

    cout << "The adjacency matrix of the graph is : " << endl;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            cout << " " << cost[i][j];

        }

        cout << endl;

    }

    cout << "Enter initial vertex : ";

    cin >> v;

    cout << "The BFS of the Graph is\n";

    cout << v<<endl;

    visited[v] = 1;

    k = 1;

    while (k < n)

    {

        for (j = 1; j <= n; j++)

            if (cost[v][j] != 0 && visited[j] != 1 && visit[j] != 1)

            {

                visit[j] = 1;

                qu[rear++] = j;

            }

        v = qu[front++];

        cout << v << " ";

        k++;

        visit[v] = 0;

        visited[v] = 1;

    }

    cout <<endl<<"Enter initial vertex : ";

    cin >> v;

    cout << "The DFS of the Graph is\n";

    cout << v<<endl;

    visited[v] = 1;

    k = 1;

    while (k < n)

    {

        for (j = n; j >= 1; j--)

            if (cost[v][j] != 0 && visited1[j] != 1 && visit1[j] != 1)

            {

                visit1[j] = 1;

                stk[top] = j;

                top++;

            }

        v = stk[--top];

        cout << v << " ";

        k++;

        visit1[v] = 0;

        visited1[v] = 1;

    }

    return 0;

}

b)

/\*

Experiment 13 : Represent a given grapg using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as a graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

Adjecency List

\*/

#include <iostream>

using *namespace* std;

#define MAX 10

#define TRUE 1

#define FALSE 0

// declaring an adjacency list for storing the graph

*class* lgra

{

*private:*

*struct* node1

    {

*int* vertex;

*struct* node1 \*next;

    };

    node1 \*head[MAX];

*int* visited[MAX];

*public:*

    //static int nodecount;

    lgra();

*void* create();

*void* dfs(*int*);

};

//constructor

lgra::lgra()

{

*int* v1;

    for (v1 = 0; v1 < MAX; v1++)

        visited[v1] = FALSE;

    for (v1 = 0; v1 < MAX; v1++)

        head[v1] = NULL;

}

*void* lgra::create()

{

*int* v1, v2;

*char* ans;

    node1 \*N, \*first;

    cout << "Enter the vertices no. beginning with 0 : ";

    do

    {

        cout << "\nEnter the Edge of a graph : \n";

        cin >> v1 >> v2;

        if (v1 >= MAX || v2 >= MAX)

            cout << "Invalid Vertex Value!!\n";

        else

        {

            //creating link from v1 to v2

            N = new node1;

            if (N == NULL)

                cout << "Insufficient Memory!!\n";

            N->vertex = v2;

            N->next = NULL;

            first = head[v1];

            if (first == NULL)

                head[v1] = N;

            else

            {

                while (first->next != NULL)

                    first = first->next;

                first->next = N;

            }

            //creating link from v2 to v1

            N = new node1;

            if (N == NULL)

                cout << "Insufficient Memory!!\n";

            N->vertex = v1;

            N->next = NULL;

            first = head[v2];

            if (first == NULL)

                head[v2] = N;

            else

            {

                while (first->next != NULL)

                    first = first->next;

                first->next = N;

            }

        }

        cout << "\n Want to add more edges?(y/n) : ";

        cin >> ans;

    } while (ans == 'y');

}

//dfs function

*void* lgra::dfs(*int* *v1*)

{

    node1 \*first;

    cout << endl

         << *v1*;

    visited[*v1*] = TRUE;

    first = head[*v1*];

    while (first != NULL)

        if (visited[first->vertex] == FALSE)

            dfs(first->vertex);

        else

            first = first->next;

}

*int* main()

{

*int* v1;

    lgra g;

    g.create();

    cout << endl << "Enter the vertex from where you want to traverse : ";

    cin >> v1;

    if (v1 >= MAX)

        cout << "Invalid Vertex!!\n";

    else

    {

        cout << "The Dfs of the graph : ";

        g.dfs(v1);

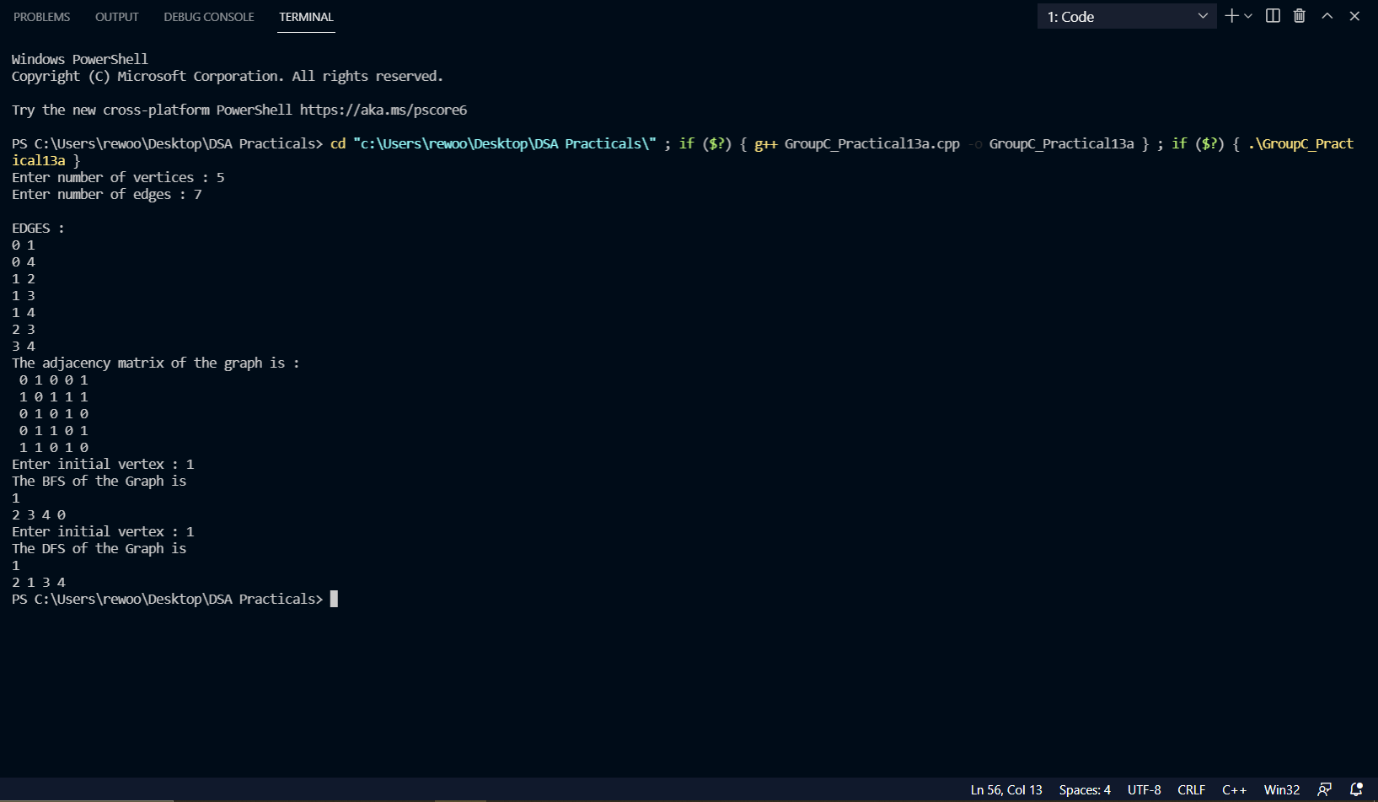
    }

    return 0;

}

**Output:**

a)



b)



**Conclusion:**

Thus we have implemented graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS.

Experiment No. 14

**Experiment Name**: Representation of Flights and Cities as a graph using Adjacency List or Adjacency Matrix.

**Aim:** 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 the airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.

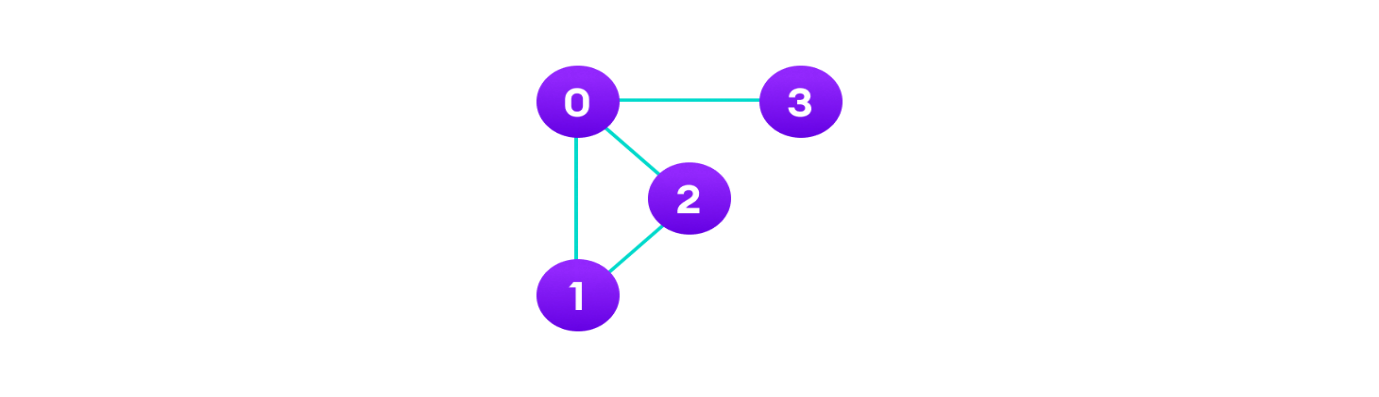
**Objective/Theory:**

Graph Data Structure:

A graph data structure is a collection of nodes that have data and are connected to other nodes.

More precisely, a graph is a data structure (V, E) that consists of

1. A collection of vertices V.
2. A collection of edges E, represented as ordered pairs of vertices (u,v).



In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

Graph Terminology:

1. *Adjacency*: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
2. *Path*: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
3. *Directed Graph*: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

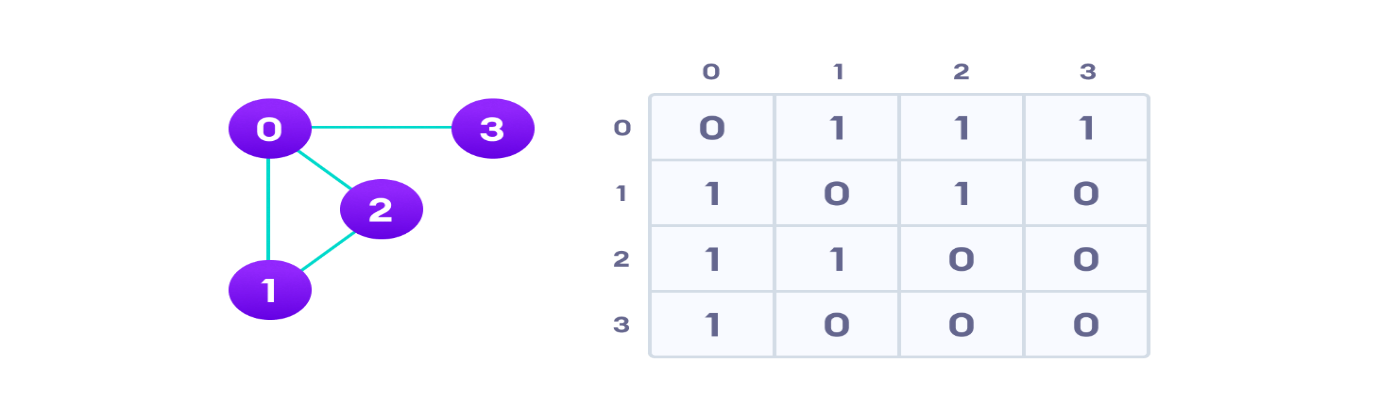
Graph Representation:

Graphs are commonly represented in two ways:

• *Adjacency Matrix:*

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

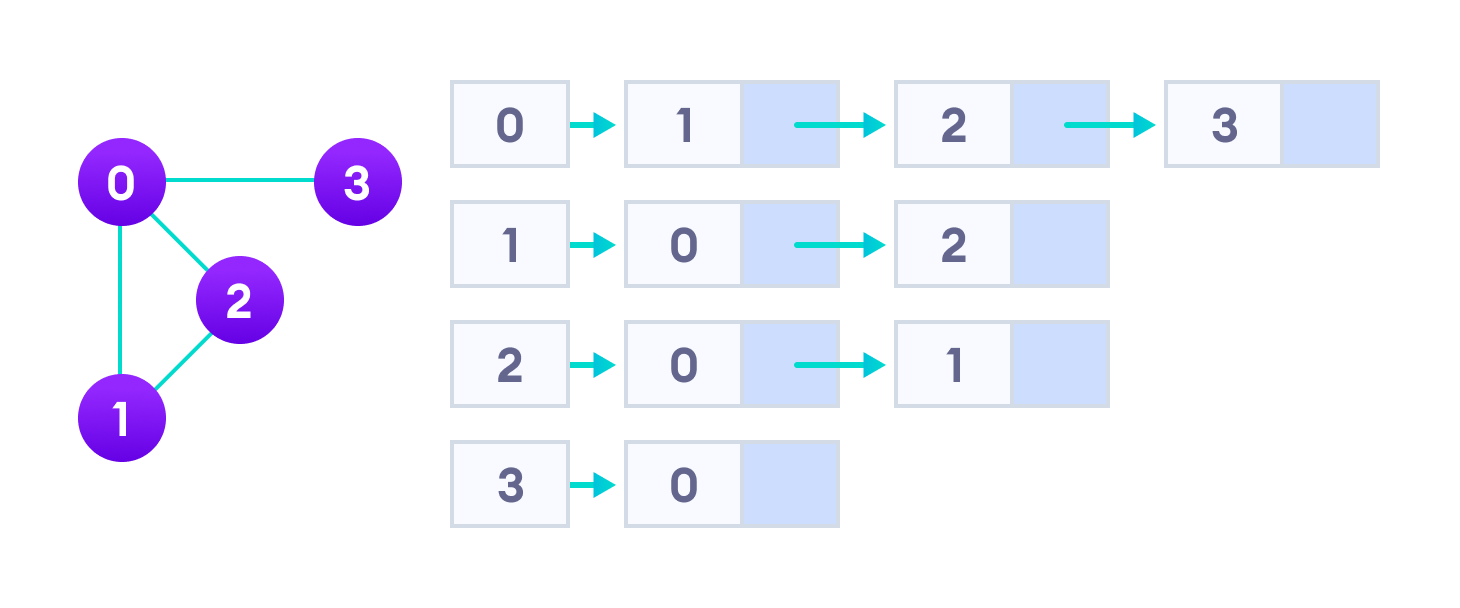
If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.



•*Adjacency List:*

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.



Graph Operations:

The most common graph operations are:

1. Check if the element is present in the graph
2. Graph Traversal
3. Add elements (vertex, edges) to graph
4. Finding the path from one vertex to another

**Program:**

/\*

Experiment 14 : 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 the airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.

\*/

#include <iostream>

#include <queue>

using *namespace* std;

*int* adj\_mat[50][50] = {0, 0};

*int* visited[50] = {0};

*void* dfs(*int* *s*, *int* *n*, string *arr*[])

{

    visited[*s*] = 1;

    cout << *arr*[*s*] << " ";

    for (*int* i = 0; i < *n*; i++)

    {

        if (adj\_mat[*s*][i] && !visited[i])

            dfs(i, *n*, *arr*);

    }

}

*void* bfs(*int* *s*, *int* *n*, string *arr*[])

{

*bool* visited[*n*];

    for (*int* i = 0; i < *n*; i++)

        visited[i] = false;

*int* v;

    queue<*int*> bfsq;

    if (!visited[*s*])

    {

        cout << *arr*[*s*] << " ";

        bfsq.push(*s*);

        visited[*s*] = true;

        while (!bfsq.empty())

        {

            v = bfsq.front();

            for (*int* i = 0; i < *n*; i++)

            {

                if (adj\_mat[v][i] && !visited[i])

                {

                    cout << *arr*[i] << " ";

                    visited[i] = true;

                    bfsq.push(i);

                }

            }

            bfsq.pop();

        }

    }

}

*int* main()

{

    cout << "Enter no. of cities: ";

*int* n, u;

    cin >> n;

    string cities[n];

    for (*int* i = 0; i < n; i++)

    {

        cout << "Enter city #" << i << " (Airport Code): ";

        cin >> cities[i];

    }

    cout << "\nYour cities are: " << endl;

    for (*int* i = 0; i < n; i++)

        cout << "city #" << i << ": " << cities[i] << endl;

    for (*int* i = 0; i < n; i++)

    {

        for (*int* j = i + 1; j < n; j++)

        {

            cout << "Enter distance between " << cities[i] << " and " << cities[j] << " : ";

            cin >> adj\_mat[i][j];

            adj\_mat[j][i] = adj\_mat[i][j];

        }

    }

    cout << endl;

    for (*int* i = 0; i < n; i++)

        cout << "\t" << cities[i] << "\t";

    for (*int* i = 0; i < n; i++)

    {

        cout << "\n"

             << cities[i];

        for (*int* j = 0; j < n; j++)

            cout << "\t" << adj\_mat[i][j] << "\t";

        cout << endl;

    }

    cout << "Enter Starting Vertex: ";

    cin >> u;

    cout << "DFS: ";

    dfs(u, n, cities);

    cout << endl;

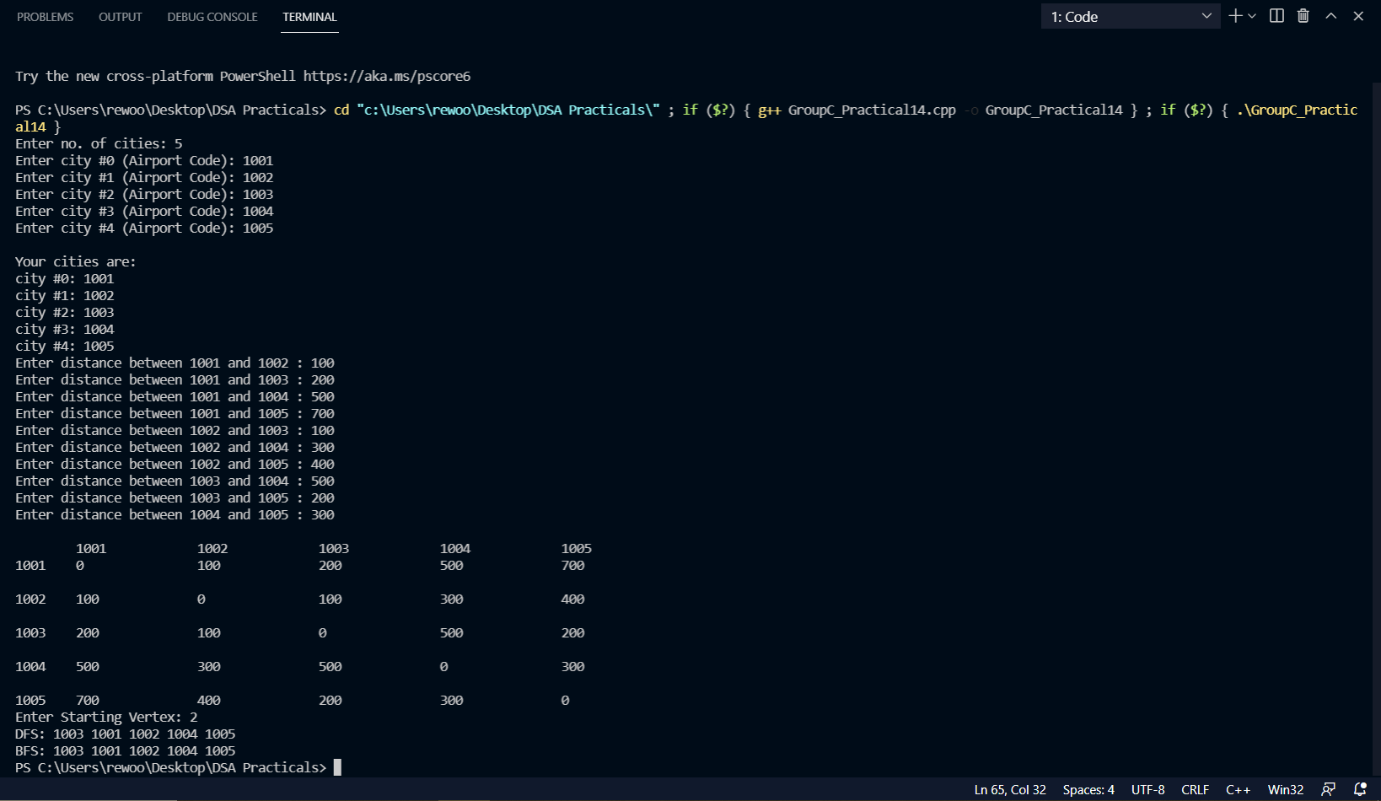
    cout << "BFS: ";

    bfs(u, n, cities);

    return 0;

}

**Output:**



**Conclusion:**

Thus we have represented Flights and Cities as a graph using Adjacency List or Adjacency Matrix.