AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department Of Computer Engineering

DSA Lab Experiments

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AISSMS IOIT

SE COMPTER ENGINEERING

SUBMITTED BY
Ashish Patil
ERP NO – 51

2020 -2021

Home

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.

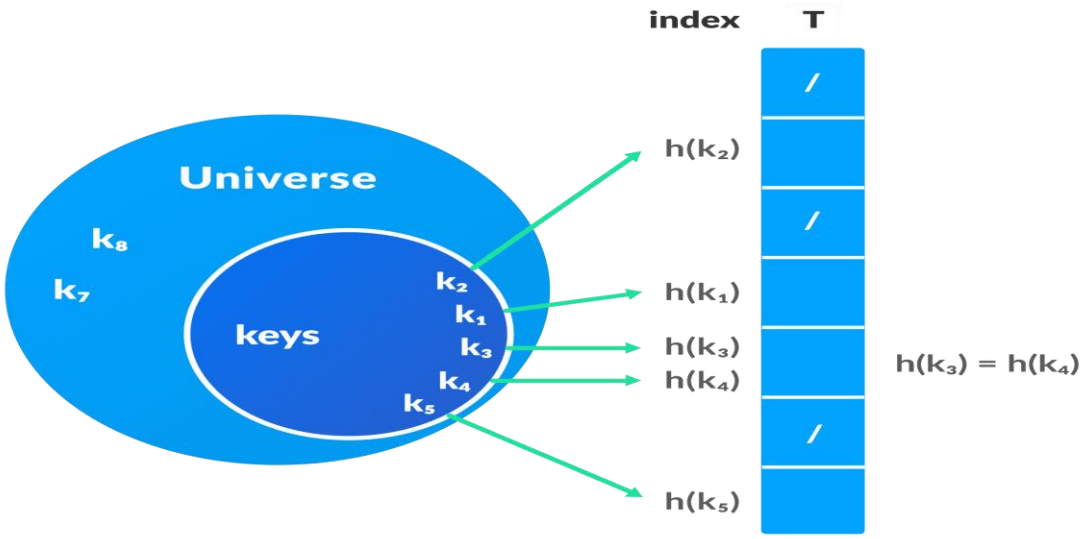
Hashing (Hash Function):

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.

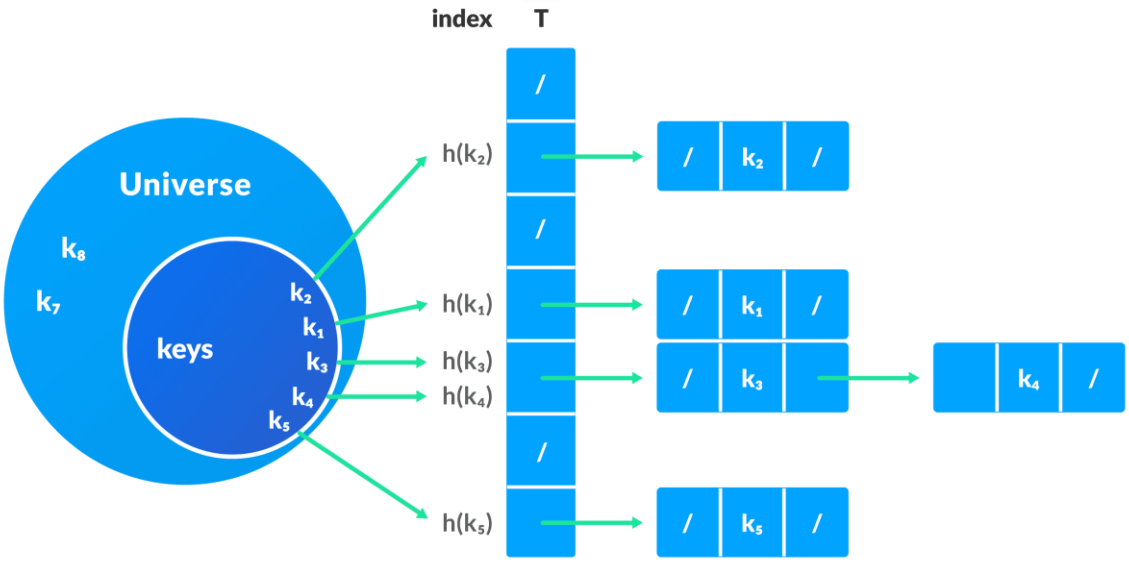
We can resolve the hash collision using one of the following techniques.

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:

```
comparison = 0  # Global Variable

class Chaining(object):
    def __init__(self, size):
        self.hashTable = [None] * size
        self.length = 0

    def __del__(self):
        pass

    def push(self, name, TeleNumber):
        hashCode = hash(name) % size
        lis = [(name, TeleNumber)]

        if self.hashTable[hashCode] == None:
            self.hashTable[hashCode] = lis
```

```
            self.length += 1

        else:
            tup = (name, TeleNumber)
            i = 0
            for j in range(size):
                if i == len(self.hashTable[hashCode]):
                    self.length += 1
                    self.hashTable[hashCode].append(tup)
                    return

                if self.hashTable[hashCode][i][0] == name:
                    self.hashTable[hashCode][i] = tup
                    return
                i += 1

            else:
                self.length += 1
                self.hashTable[hashCode].append(tup)

    def get(self, key):
        global comparison
        comparison = 0
        hashCode = hash(key) % size
        i = 0

        if self.hashTable[hashCode] == None:
            return -1

        elif self.hashTable[hashCode][i][0] == key:
            comparison += 1
            return self.hashTable[hashCode][i][1]

        else:
            i += 1
            for j in range(size):
                comparison += 1
                if i == len(self.hashTable[hashCode]):
                    return -1

                if self.hashTable[hashCode][i][0] == key:
                    return self.hashTable[hashCode][i][1]
                i += 1

            else:
                return -1

    def printDict(self):
        print("\n{", end="")
        i = 1
        for lis in self.hashTable:
            if lis != None:
                for ele in lis:
                    if i == self.length:
```

```
                        print(f"{ele[0]} : {ele[1]}", end="")
                    else:
                        print(f"{ele[0]} : {ele[1]}, ", end="")
                    i += 1
        print("}")

class Addressing(object):
    def __init__(self, size):
        self.hashTable = [None] * size
        self.length = 0

    def __del__(self):
        pass

    def push(self, name, TeleNumber):
        hashCode = hash(name) % size
        lis = (name, TeleNumber)

        if self.hashTable[hashCode] == None:
            self.hashTable[hashCode] = lis
            self.length += 1

        elif self.hashTable[hashCode][0] == name:
            self.hashTable[hashCode] = lis

        else:
            self.length += 1
            for i in range(num):
                hashCode = (hashCode + 1) % size
                if self.hashTable[hashCode] == None:
                    self.hashTable[hashCode] = lis
                    return

    def get(self, key):
        global comparison
        comparison = 0
        hashCode = hash(key) % size
        if self.hashTable[hashCode] == None:
            return -1

        elif self.hashTable[hashCode][0] == key:
            comparison += 1
            return self.hashTable[hashCode][1]

        else:
            for i in range(num):
                comparison += 1
                hashCode = (hashCode + 1) % size
                if self.hashTable[hashCode][0] == key:
                    return self.hashTable[hashCode][1]
            else:
                return -1

    def printDict(self):
```

```
        print("\n{", end="")
        i = 1
        for ele in self.hashTable:
            if ele != None:
                if i == self.length:
                    print(f"{ele[0]} : {ele[1]}", end="")
                else:
                    print(f"{ele[0]} : {ele[1]}, ", end="")
                i += 1
        print("}")

if __name__ == '__main__':
    try:
        size = int(input("\nEnter The Size of Telephone Book: "))

        while True:
            userInput = int(input("\n1. Press 1 For Chaining\n2. Press 2 For Addressin
g\n3. Press 3 To Exit\n>>>> "))
            if userInput == 1:
                ch = Chaining(size)
                while True:
                    userInput = int(input("\n1. Press 1 To Push Data\n2. Press 2 To Ge
t Data\n3. Press 3 To Print Data\n4. Press 4 To Go Back\n>>>> "))
                    if userInput == 1:
                        num = int(input("\nEnter The Number of Clients: "))
                        for i in range(num):
                            name = input(f"\nEnter The Name of Client {i + 1}: ")
                            TeleNumber = int(input(f"\nEnter The Telephone Number of {
name}: "))
                            ch.push(name, TeleNumber)

                    elif userInput == 2:
                        if ch.hashTable == [None] * size:
                            print("\nNo Data Available to Get!!")
                            continue
                        name = input("\nEnter The Name of Client: ")
                        print("Telephone Number:", ch.get(name))
                        print("\nNo. of Comparisons:", comparison)

                    elif userInput == 3:
                        ch.printDict()

                    elif userInput == 4:
                        break

                    else:
                        print("\nPlease Enter Correct Input!!")

            elif userInput == 2:
                ad = Addressing(size)
                while True:
                    userInput = int(input("\n1. Press 1 To Push Data\n2. Press 2 To Ge
t Data\n3. Press 3 To Print Data\n4. Press 4 To Go Back\n>>>> "))
                    if userInput == 1:
```

```
                        num = int(input("\nEnter The Number of Clients: "))
                        for i in range(num):
                            name = input(f"\nEnter The Name of Client {i + 1}: ")
                            TeleNumber = int(input(f"\nEnter The Telephone Number of {
name}: "))
                            ad.push(name, TeleNumber)

                    elif userInput == 2:
                        if ad.hashTable == [None] * size:
                            print("\nNo Data Available to Get!!")
                            continue
                        name = input("\nEnter The Name of Client: ")
                        print("Telephone Number:", ad.get(name))
                        print("\nNo. of Comparisons:", comparison)

                    elif userInput == 3:
                        ad.printDict()

                    elif userInput == 4:
                        break

                    else:
                        print("\nPlease Enter Correct Input!!")

            elif userInput == 3:
                exit()

            else:
                print("\nPlease Enter Correct Input!!")

    except Exception as e:
        print("\nWrong Input,", e)
```

Output:

```
# Output

# Enter The Size of Telephone Book: 10

# 1. Press 1 For Chaining
# 2. Press 2 For Addressing
# 3. Press 3 To Exit
# >>>> 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 1

# Enter The Number of Clients: 4
```

```
# Enter The Name of Client 1: Ashish

# Enter The Telephone Number of Ashish: 1234

# Enter The Name of Client 2: Mayur

# Enter The Telephone Number of Mayur: 2345

# Enter The Name of Client 3: Rohit

# Enter The Telephone Number of Rohit: 3456

# Enter The Name of Client 4: Virat

# Enter The Telephone Number of Virat: 4567

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 3

# {Rohit : 3456, Ashish : 1234, Virat : 4567, Mayur : 2345}

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Ashish
# Telephone Number: 1234

# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Mayur
# Telephone Number: 2345

# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Rohit
# Telephone Number: 3456
```

```
# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Virat
# Telephone Number: 4567

# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Rushikesh
# Telephone Number: -1

# No. of Comparisons: 0

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 3

# {Rohit : 3456, Ashish : 1234, Virat : 4567, Mayur : 2345}

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 1

# Enter The Number of Clients: 1

# Enter The Name of Client 1: Virat

# Enter The Telephone Number of Virat: 1245

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 3

# {Rohit : 3456, Ashish : 1234, Virat : 1245, Mayur : 2345}

# 1. Press 1 To Push Data
```

```
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 4

# 1. Press 1 For Chaining
# 2. Press 2 For Addressing
# 3. Press 3 To Exit
# >>>> 2

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 1

# Enter The Number of Clients: 4

# Enter The Name of Client 1: Ashish

# Enter The Telephone Number of Ashish: 1234

# Enter The Name of Client 2: Mayur

# Enter The Telephone Number of Mayur: 2345

# Enter The Name of Client 3: Rohit

# Enter The Telephone Number of Rohit: 3456

# Enter The Name of Client 4: Virat

# Enter The Telephone Number of Virat: 4567

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 3

# {Virat : 4567, Rohit : 3456, Ashish : 1234, Mayur : 2345}

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Ashish
# Telephone Number: 1234

# No. of Comparisons: 1

# 1. Press 1 To Push Data
```

```
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Mayur
# Telephone Number: 2345

# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Rohit
# Telephone Number: 3456

# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 2

# Enter The Name of Client: Virat
# Telephone Number: 4567

# No. of Comparisons: 1

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 3

# {Virat : 4567, Rohit : 3456, Ashish : 1234, Mayur : 2345}

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 1

# Enter The Number of Clients: 1

# Enter The Name of Client 1: Rohit

# Enter The Telephone Number of Rohit: 1245

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
```

```
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 3

# {Virat : 4567, Rohit : 1245, Ashish : 1234, Mayur : 2345}

# 1. Press 1 To Push Data
# 2. Press 2 To Get Data
# 3. Press 3 To Print Data
# 4. Press 4 To Go Back
# >>>> 4

# 1. Press 1 For Chaining
# 2. Press 2 For Addressing
# 3. Press 3 To Exit
# >>>> 3
```

Conclusion:

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

Experiment No. 2

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.

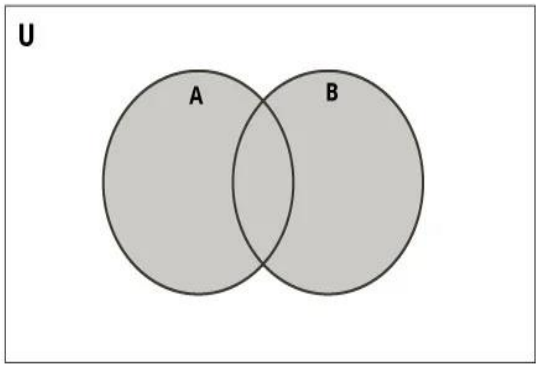
Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

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.

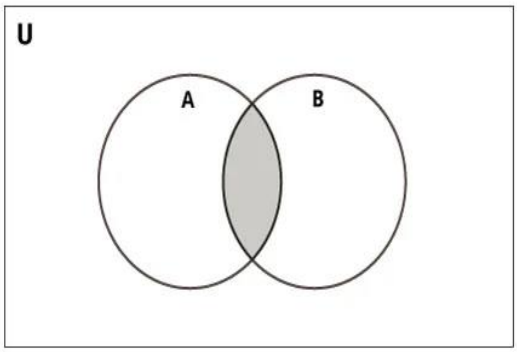
Set Union:



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.

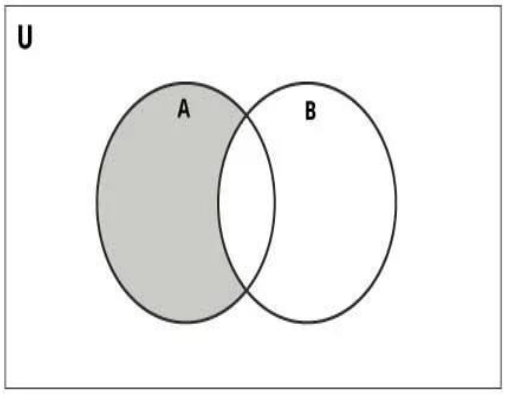
Set Intersection:



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.

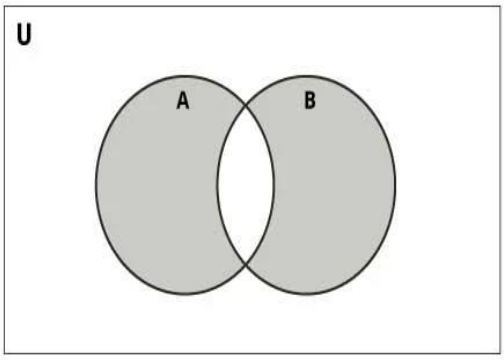
Set Difference:



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)
```

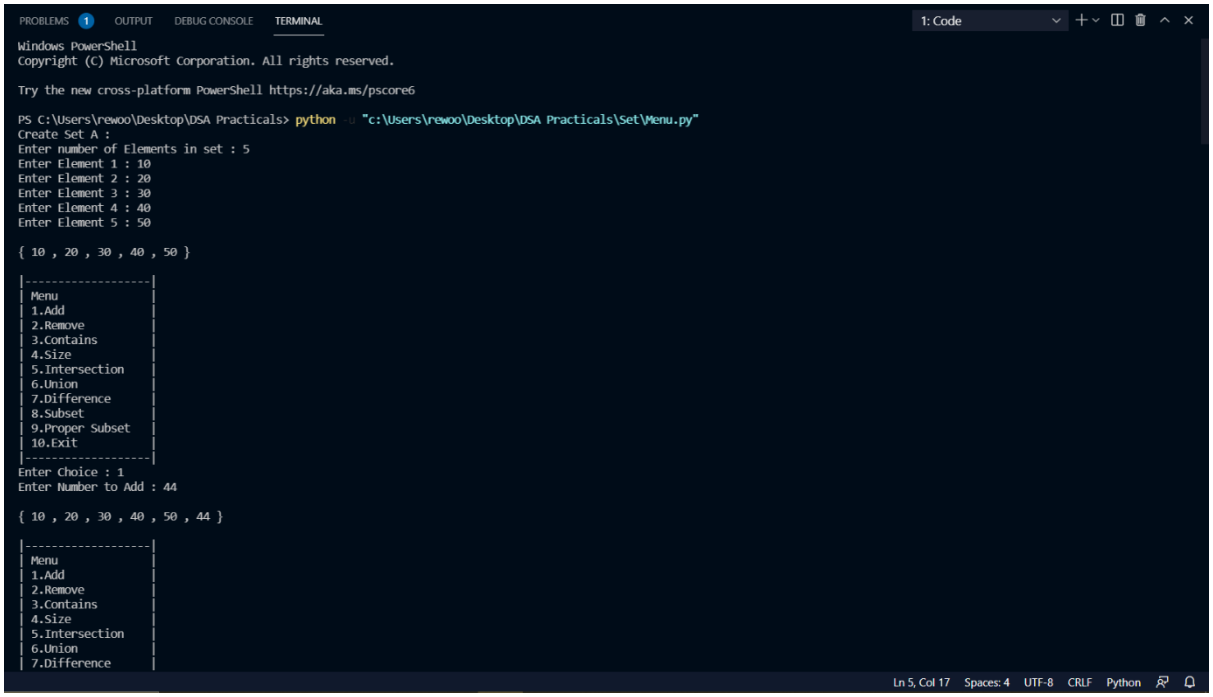
Menu.py

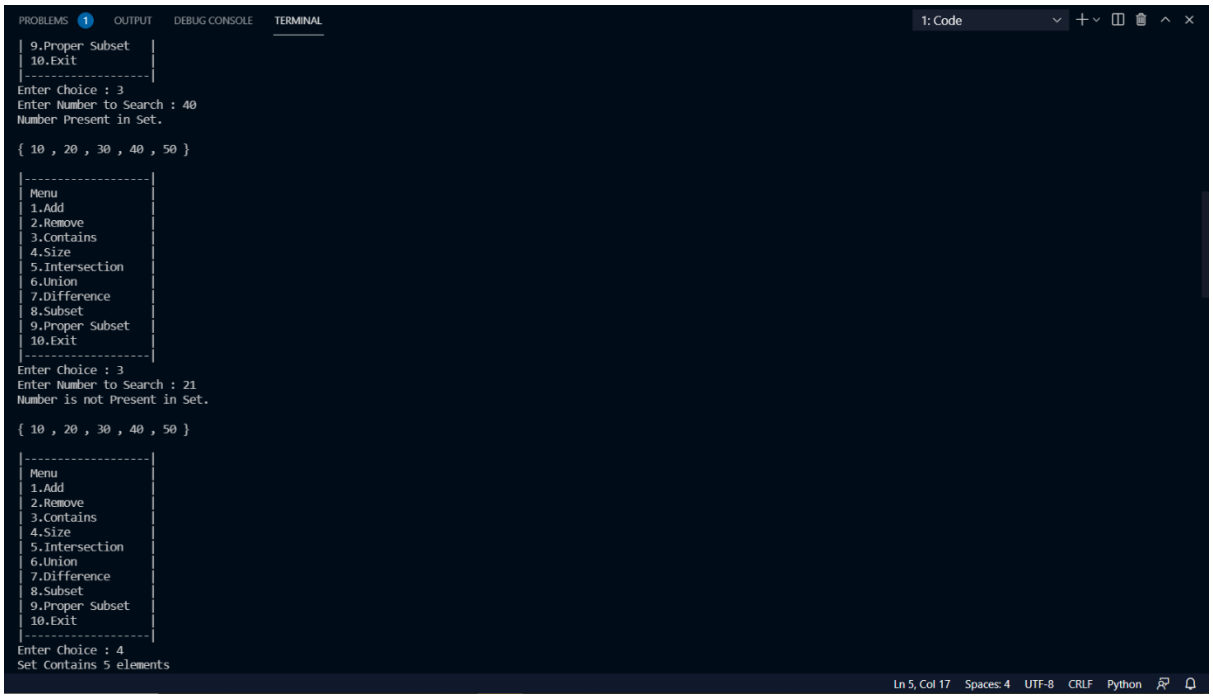
```
from SetOperations import Set

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        |")
```

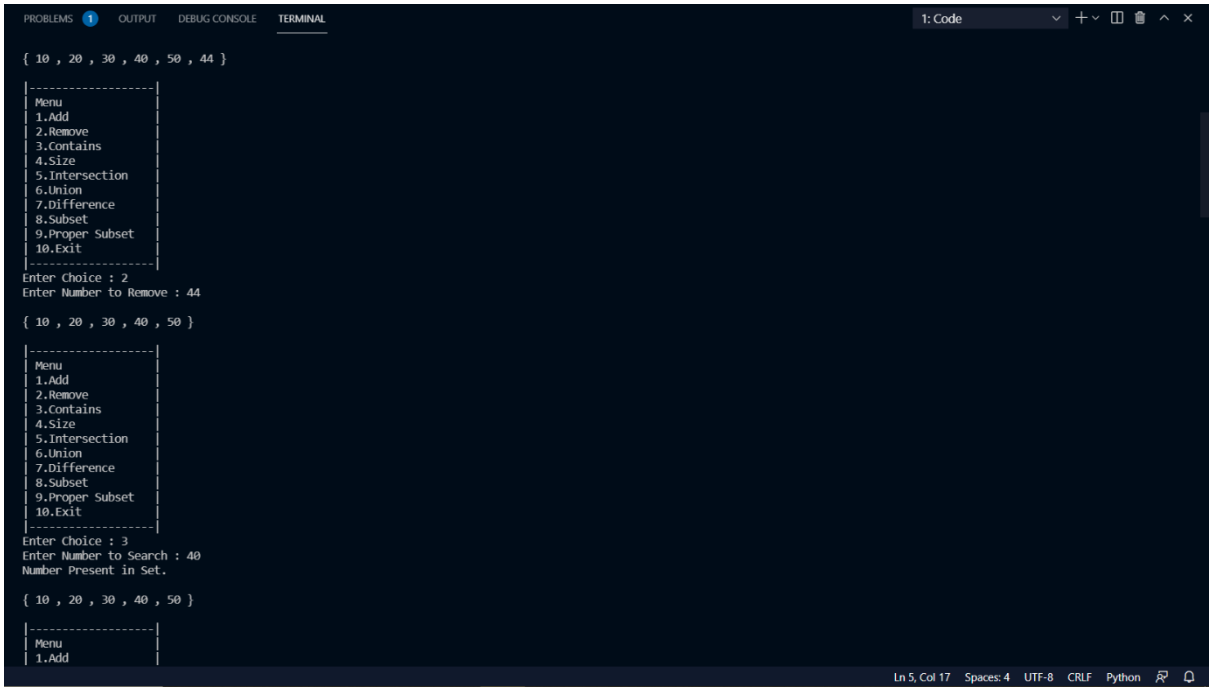
```
    print("| 4.Size            |")
    print("| 5.Intersection    |")
    print("| 6.Union           |")
    print("| 7.Difference      |")
    print("| 8.Subset          |")
    print("| 9.Proper Subset   |")
    print("| 10.Exit           |")
    print("|-------------------|")

    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))
```

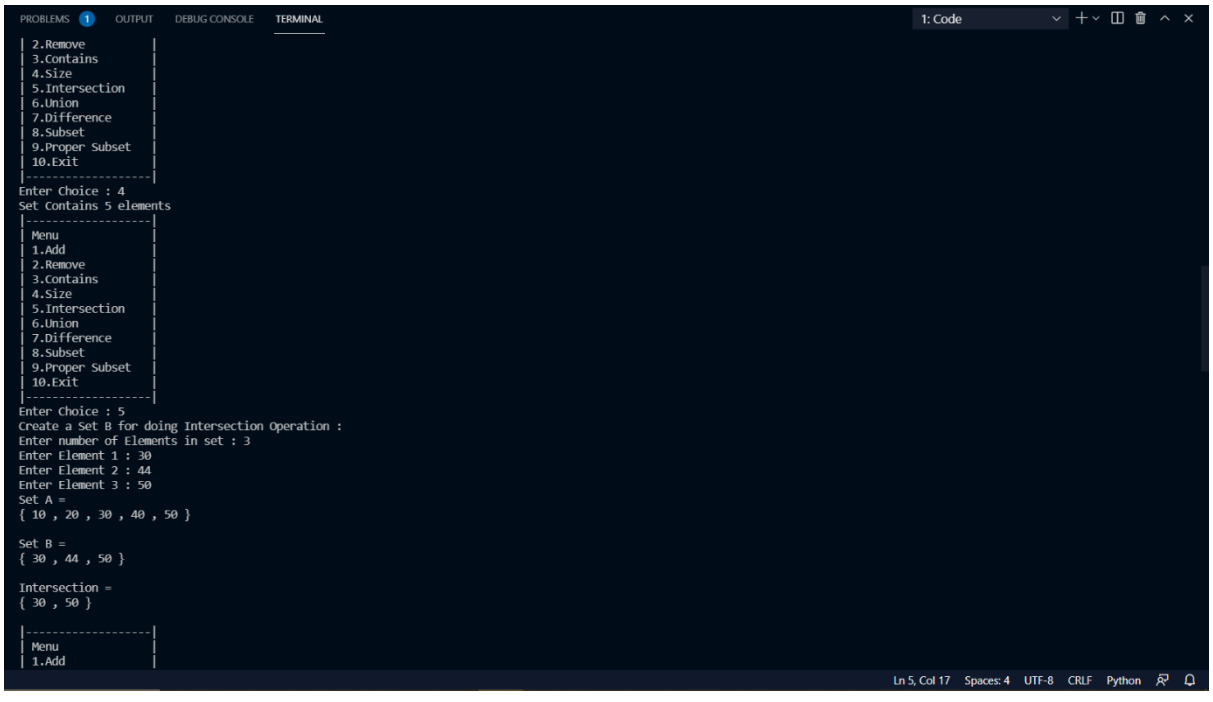
```
elif choice==7:
    print("Create a Set B for calculating Set Difference : ")
    s2 = createSet()
    s3 = s1.difference(s2)
    print("Set A = "+str(s1))
    print("Set B = "+str(s2))
    print("Difference = "+str(s3))

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!!")
```

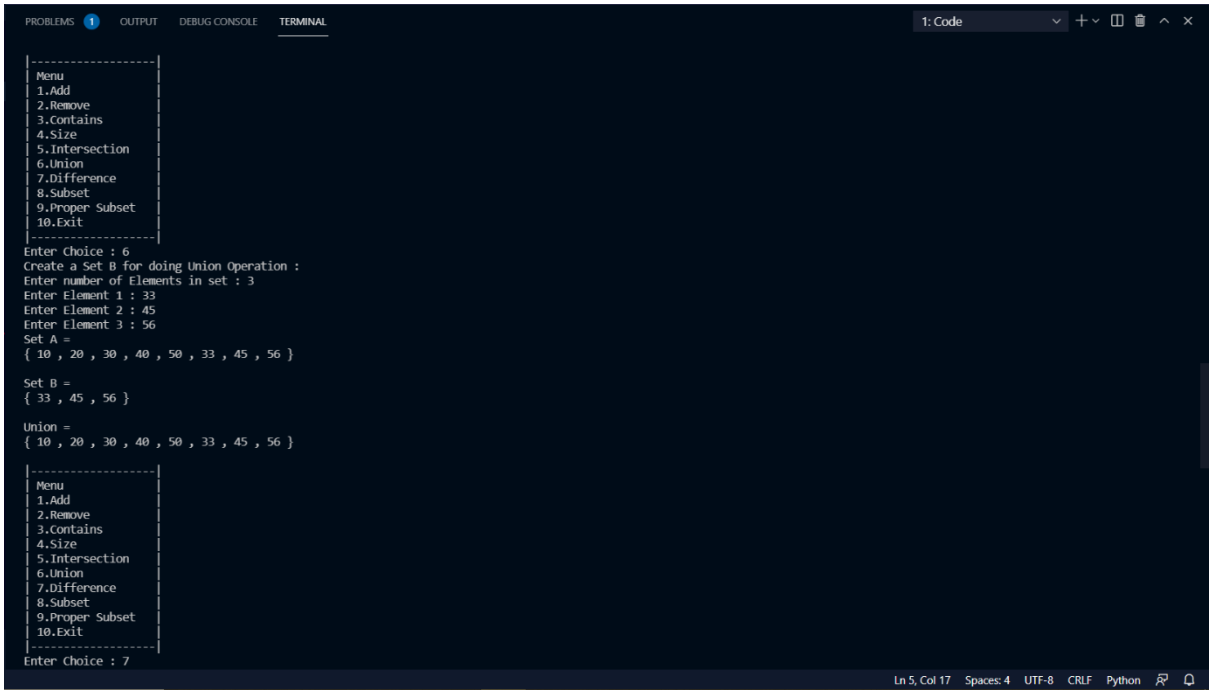
Output:

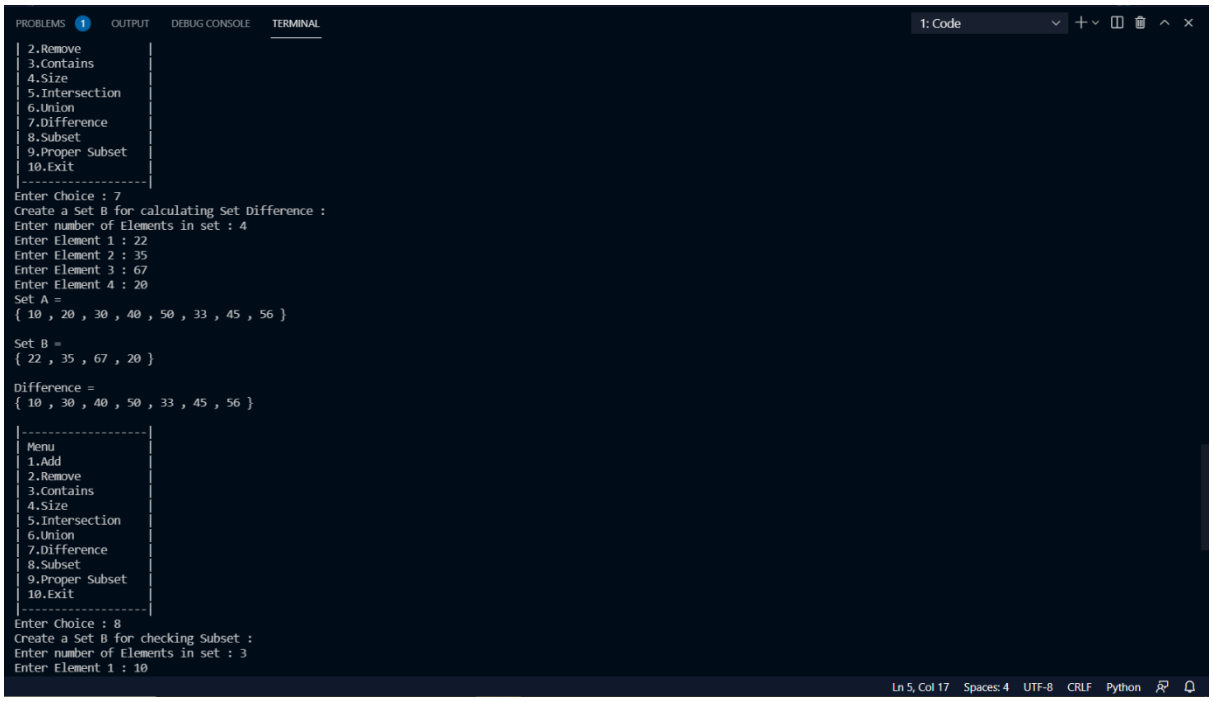

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
1: Code
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
Try the new cross-platform PowerShell https://aka.ms/pscore6
PS C:\Users\rewoo\Desktop\DSA Practicals> python "c:\Users\rewoo\Desktop\DSA Practicals\Set\Menu.py"
Create Set A :
Enter number of Elements in set : 5
Enter Element 1 : 10
Enter Element 2 : 20
Enter Element 3 : 30
Enter Element 4 : 40
Enter Element 5 : 50
{ 10 , 20 , 30 , 40 , 50 }
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
| 10.Exit |
Enter Choice : 1
Enter Number to Add : 44
{ 10 , 20 , 30 , 40 , 50 , 44 }
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
Ln 5, Col 17 Spaces: 4 UTF-8 CRLF Python

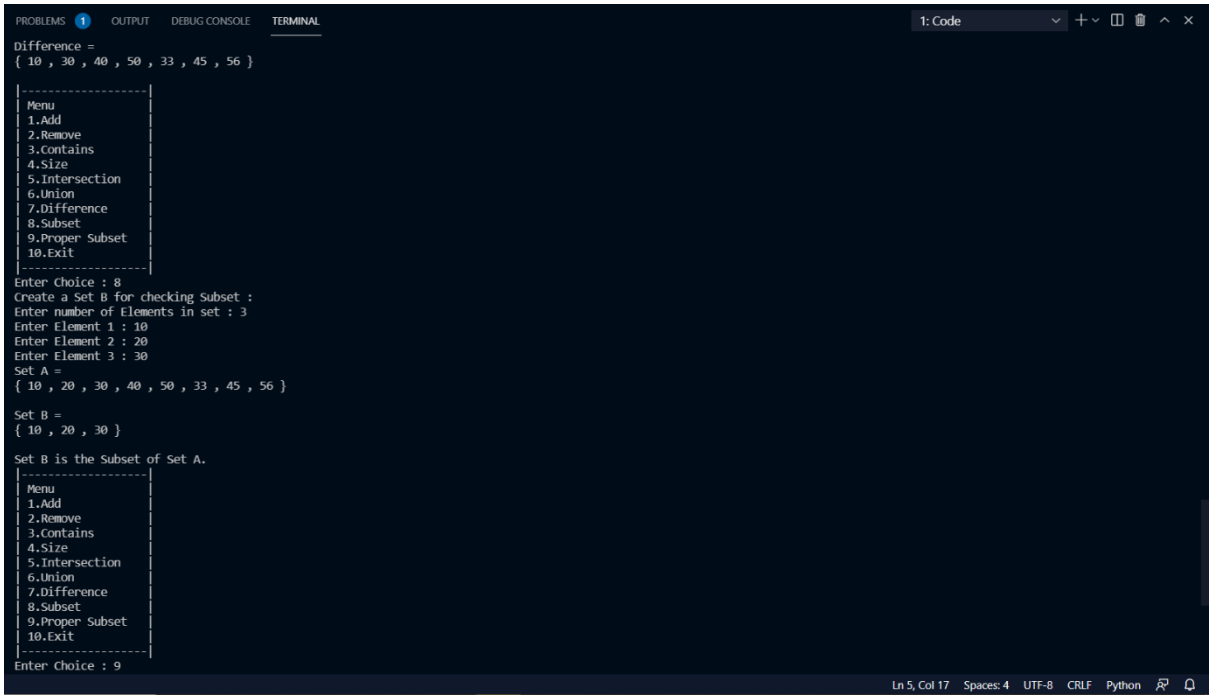

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
1: Code
| 9.Proper Subset |
| 10.Exit |
Enter Choice : 3
Enter Number to Search : 40
Number Present in Set.
{ 10 , 20 , 30 , 40 , 50 }
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
| 10.Exit |
Enter Choice : 3
Enter Number to Search : 21
Number is not Present in Set.
{ 10 , 20 , 30 , 40 , 50 }
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
| 10.Exit |
Enter Choice : 4
Set Contains 5 elements
Ln 5, Col 17 Spaces: 4 UTF-8 CRLF Python

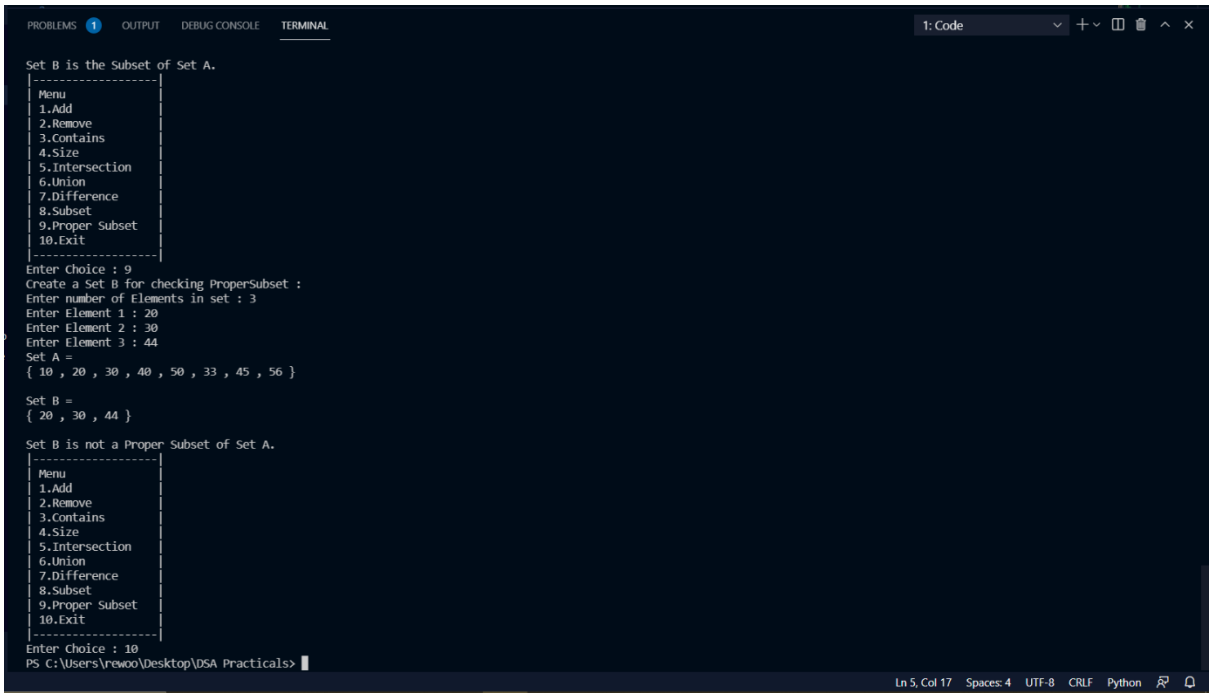

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
1: Code
{ 10 , 20 , 30 , 40 , 50 , 44 }
|------------------|
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 2
Enter Number to Remove : 44
{ 10 , 20 , 30 , 40 , 50 }
|------------------|
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 3
Enter Number to Search : 40
Number Present in Set.
{ 10 , 20 , 30 , 40 , 50 }
|------------------|
| Menu |
| 1.Add |
Ln 5, Col 17 Spaces: 4 UTF-8 CRLF Python


PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
1: Code
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 4
Set Contains 5 elements
|------------------|
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 5
Create a Set B for doing Intersection Operation :
Enter number of Elements in set : 3
Enter Element 1 : 30
Enter Element 2 : 44
Enter Element 3 : 50
Set A =
{ 10 , 20 , 30 , 40 , 50 }
Set B =
{ 30 , 44 , 50 }
Intersection =
{ 30 , 50 }
|------------------|
| Menu |
| 1.Add |
Ln 5, Col 17 Spaces: 4 UTF-8 CRLF Python


PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
1: Code
|------------------|
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 6
Create a Set B for doing Union Operation :
Enter number of Elements in set : 3
Enter Element 1 : 33
Enter Element 2 : 45
Enter Element 3 : 56
Set A =
{ 10 , 20 , 30 , 40 , 50 , 33 , 45 , 56 }
Set B =
{ 33 , 45 , 56 }
Union =
{ 10 , 20 , 30 , 40 , 50 , 33 , 45 , 56 }
|------------------|
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 7
Ln 5, Col 17 Spaces: 4 UTF-8 CRLF Python


PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
1: Code
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 7
Create a Set B for calculating Set Difference :
Enter number of Elements in set : 4
Enter Element 1 : 22
Enter Element 2 : 35
Enter Element 3 : 67
Enter Element 4 : 20
Set A =
{ 10 , 20 , 30 , 40 , 50 , 33 , 45 , 56 }
Set B =
{ 22 , 35 , 67 , 20 }
Difference =
{ 10 , 30 , 40 , 50 , 33 , 45 , 56 }
|------------------|
| Menu |
| 1.Add |
| 2.Remove |
| 3.Contains |
| 4.Size |
| 5.Intersection |
| 6.Union |
| 7.Difference |
| 8.Subset |
| 9.Proper Subset |
10.Exit
Enter Choice : 8
Create a Set B for checking Subset :
Enter number of Elements in set : 3
Enter Element 1 : 10
Ln 5, Col 17 Spaces: 4 UTF-8 CRLF Python





Conclusion:

Thus we have implemented Set operations.

Experiment No. 3

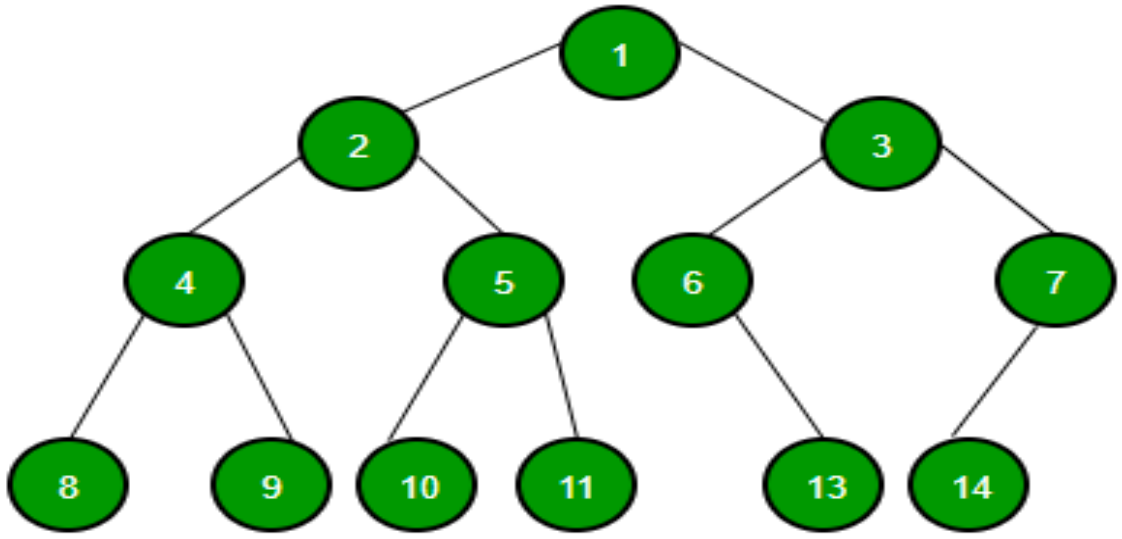
Experiment Name: Tree Creation and Display

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:

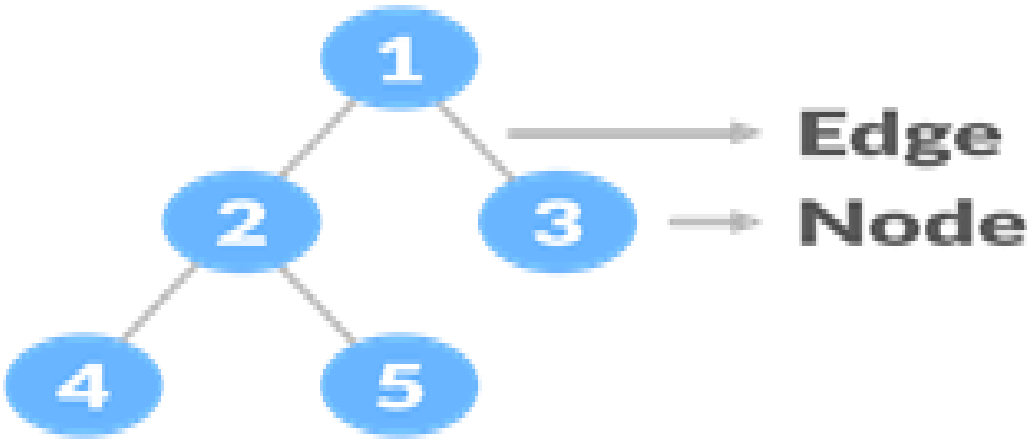
A node is an entity that contains a key or value and pointers to its child nodes.

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:

```
// 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>
using namespace std;

class Node
{
public:
    string label;
    int ch_count;
    Node *child[10];
};

class Tree
{
public:
    Node *root;
    void create_tree(Node *root);
    void display(Node *root);

    Tree()
    {
        root = NULL;
    }
};

void Tree::create_tree(Node *root)
{
    cout << "\nEnter The Name of Book: ";
    cin >> root->label;
    cout << "\nEnter The No. of Chapters in Book: ";
    cin >> root->ch_count;

    for (int i = 0; i < root->ch_count; i++)
    {
        root->child[i] = new Node;
        cout << "\nEnter The Name of Chapter " << i + 1 << ": ";
        cin >> root->child[i]->label;
        cout << "\nEnter No. of Sections in The Chapter " << root->child[i]-
>label << ": ";
        cin >> root->child[i]->ch_count;

        for (int j = 0; j < root->child[i]->ch_count; j++)
        {
            root->child[i]->child[j] = new Node;
            cout << "\nEnter The Name of Section " << j + 1 << ": ";
```

```
            cin >> root->child[i]->child[j]->label;
        }
    }
    cout << "\nTree Created Successfully!\n";
}

void Tree::display(Node *root)
{
    int i, j, k;
    if (root != NULL)
    {
        cout << "\n***** Book *****\n";
        cout << "\nTitle : " << root->label << "\n";

        for (i = 0; i < root->ch_count; i++)
        {
            cout << "\nChapter " << i + 1 << ": ";
            cout << root->child[i]->label << "\n";
            cout << "\nSections\n";
            for (j = 0; j < root->child[i]->ch_count; j++)
            {
                cout << root->child[i]->child[j]->label << "\n";
            }
        }
    }
}

int main()
{
    int userInput;
    string content("\n****** Book Tree Creation ******\n\t1.Create\n\t2.Display\n\t3.E
xit\n>>>> ");
    Tree tree;
    Node *root = NULL;

    while (true)
    {
        cout << content;
        cin >> userInput;

        switch (userInput)
        {
        case 1:
            root = new (nothrow) Node;
            if (!root)
                throw bad_alloc();
            tree.create_tree(root);
            break;
        case 2:
            if (!root)
                cout << "\nFirst Create Tree!\n";
            tree.display(root);
            break;
        case 3:
```

```
            exit(0);
        default:
            cout << "\nPlease Enter Correct Input!\n";
        }
    }
}
```

Output:

```
// Output

// ****** Book Tree Creation ******
//         1.Create
//         2.Display
//         3.Exit
// >>>> 2

// First Create Tree!

// ****** Book Tree Creation ******
//         1.Create
//         2.Display
//         3.Exit
// >>>> 1

// Enter The Name of Book: DSA

// Enter The No. of Chapters in Book: 2

// Enter The Name of Chapter 1: Hashing

// Enter No. of Sections in The Chapter Hashing: 2

// Enter The Name of Section 1: Open_Addressing

// Enter The Name of Section 2: Close_Addressing

// Enter The Name of Chapter 2: Trees

// Enter No. of Sections in The Chapter Trees: 2

// Enter The Name of Section 1: Types

// Enter The Name of Section 2: Traversals

// Tree Created Successfully!

// ****** Book Tree Creation ******
//         1.Create
//         2.Display
//         3.Exit
// >>>> 2
```

```
// ***** Book *****

// Title : DSA

// Chapter 1: Hashing

// Sections
// Open_Addressing
// Close_Addressing

// Chapter 2: Trees

// Sections
// Types
// Traversals

// ****** Book Tree Creation ******
//         1.Create
//         2.Display
//         3.Exit
// >>>> 3
```

Conclusion:

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

Experiment No. 4

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 inthe 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:

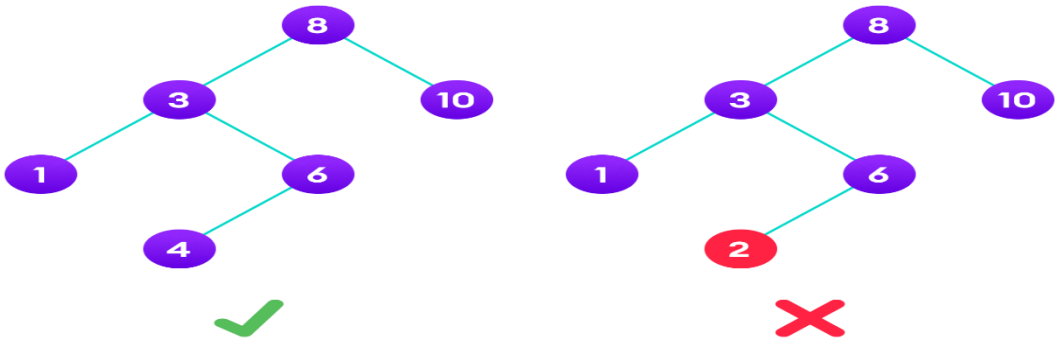
Binary Search Tree:

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
returncreateNode(data)
if (data< node->data)
    node->left  = insert(node->left, data);
elseif (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:

```cpp
// Create binary search tree. Find height of the tree and print leaf nodes.
// Find mirror image, print original and mirror image using level-wise printing.

#include <iostream>
#include <queue>
using namespace std;

class Node
{
public:
    int data;
    Node *left, *right;

    Node(int val)
    {
        data = val;
        left = NULL;
        right = NULL;
    }
};

class Tree
```

```
{
public:
    Node *root;
    Tree()
    {
        root = NULL;
    }

    int height(Node *);
    int print(Node *);
    Node *mirror(Node *);
    void create();
    void insert(Node *, Node *);
    void levelOrder();
    void preorder(Node *);
    void min(Node *);
    bool search(Node *, int);
    int printLeafNodes(Node *);
};

int Tree::height(Node *root)
{
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;
    return (max(height(root->left), height(root->right)) + 1);
}

int Tree::printLeafNodes(Node *root)
{
    if (root == NULL)
        return 0;

    if (root->left == NULL && root->right == NULL)
    {
        cout << "  " << root->data;
        return 1;
    }
    return (printLeafNodes(root->left) + printLeafNodes(root->right));
}

void Tree::create()
{
    root = NULL;
    string ch;
    do
    {
        int data;
        cout << "\nEnter Data: ";
        cin >> data;
        Node *temp = new Node(data);
        if (root == NULL)
            root = temp;
```

```
        else
        {
            insert(root, temp);
        }
        cout << "\nDo you want to continue?";
        cin >> ch;
    } while (ch == "y" || ch == "Y" || ch == "Yes" || ch == "yes");
}

void Tree::insert(Node *root, Node *temp)
{
    char ch1;

    if (temp->data < root->data)
    {
        if (root->left == NULL)
            root->left = temp;
        else
            insert(root->left, temp);
    }
    else if (temp->data > root->data)
    {
        if (root->right == NULL)
            root->right = temp;
        else
            insert(root->right, temp);
    }
}

Node *Tree::mirror(Node *root)
{
    if (root == NULL)
        return NULL;

    Node *temp;
    Node *T = root;

    temp = T->left;
    T->left = mirror(T->right);
    T->right = mirror(temp);
    return T;
}

void Tree::levelOrder()
{
    if (root == NULL)
        return;

    queue<Node *> que;
    que.push(root);
    que.push(NULL);

    while (!que.empty())
    {
```

```
        Node *node = que.front();
        que.pop();

        if (node != NULL)
        {
            cout << node->data << " ";

            if (node->left)
                que.push(node->left);

            if (node->right)
                que.push(node->right);
        }
    }
}

void Tree::preorder(Node *root)
{
    if (root != NULL)
    {
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
}

void Tree::min(Node *root)
{
    Node *ptr = root;
    while (ptr->left != NULL)
        ptr = ptr->left;
    cout << ptr->data;
}

bool Tree::search(Node *root, int val)
{
    if (root == NULL)
        return false;
    else if (val == root->data)
        return true;
    else if (val < root->data)
        return search(root->left, val);
    else
        return search(root->right, val);
}

int main()
{
    Tree *tree = NULL;
    int leafNodes, userInput, search;
    bool found;

    while (true)
    {
```

```
        cout << "\n1. Create\n2. Mirror\n3. Print Leaf Nodes\n4. Height\n5. Preorder D
isplay\n6. Minimum Value\n7. Search\n8. Exit\n>>>> ";
        cin >> userInput;

        switch (userInput)
        {
        case 1:
            tree = new (nothrow) Tree;
            if (!tree)
                throw bad_alloc();
            tree->create();
            break;

        case 2:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }

            cout << "\nLevel Order Traversal on Original Tree: ";
            tree->levelOrder();
            Tree *mirror;
            mirror->root = tree->mirror(tree->root);
            cout << "\nLevel Order Traversal on Mirror Tree: ";
            mirror->levelOrder();
            cout << "\n";
            break;

        case 3:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }
            cout << "\n";
            leafNodes = tree->printLeafNodes(tree->root);
            cout << "\nNo of Leaf Nodes = " << leafNodes << "\n";
            break;

        case 4:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }
            cout << "\nHeight = " << tree->height(tree->root) << "\n";
            break;

        case 5:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
```

```
            }
            cout << "\n";
            tree->preorder(tree->root);
            cout << "\n";
            break;

        case 6:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }
            cout << "\n";
            tree->min(tree->root);
            cout << "\n";
            break;

        case 7:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }
            cout << "\nEnter Element To Search: ";
            cin >> search;
            found = tree->search(tree->root, search);
            if (found)
                cout << "\nData Found!\n";
            else
                cout << "\nData Not Found!\n";
            break;

        case 8:
            exit(0);

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }
    return 0;
}
```

Output:

```
// Output

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
```

```
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 3

// Tree is Empty!

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 4

// Tree is Empty!

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 5

// Tree is Empty!

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 6

// Tree is Empty!

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 7

// Tree is Empty!
```

```
// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 1

// Enter Data: 1

// Do you want to continue?y

// Enter Data: 4

// Do you want to continue?y

// Enter Data: 2

// Do you want to continue?y

// Enter Data: 5

// Do you want to continue?y

// Enter Data: 6

// Do you want to continue?e

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 3

//   2  6
// No of Leaf Nodes = 2

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 4
```

```
// Height = 3

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 5

// 1 4 2 5 6

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 6

// 1

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 7

// Enter Element To Search: 5

// Data Found!

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 2

// Level Order Traversal on Original Tree: 1 4 2 5 6
// Level Order Traversal on Mirror Tree: 1 4 5 2 6

// 1. Create
```

```
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 3

//   6  2
// No of Leaf Nodes = 2

// 1. Create
// 2. Mirror
// 3. Print Leaf Nodes
// 4. Height
// 5. Preorder Display
// 6. Minimum Value
// 7. Search
// 8. Exit
// >>>> 8
```

Conclusion:

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

Experiment No. 5

Experiment Name: Expression Tree and operations on it.

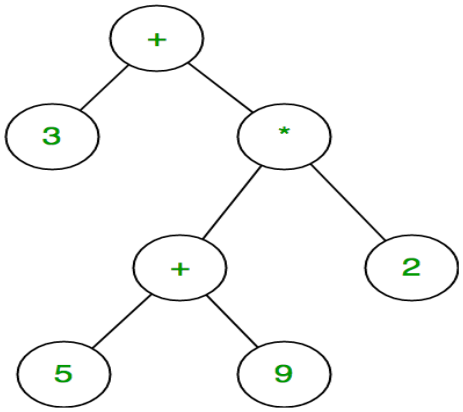
Aim: Construct an expression tree from the given prefix and traverse it using post order traversal and then delete the entire tree.

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:

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

class Node
{
public:
    char data;
    Node *left;
    Node *right;

    Node(char val)
    {
        data = val;
        left = NULL;
        right = NULL;
    }
};

class Tree
{
    string prefix;

public:
    Node *root;
    void expression(string);
    void nonRecPostorder();
    void deleteEntireTree();
};

bool isOperator(char ch)
{
```

```
    return ch == '^' || ch == '/' || ch == '*' || ch == '-' || ch == '+';
}

void Tree::expression(string prefix)
{
    stack<Node *> s;
    Node *t1, *t2;

    for (int i = prefix.length() - 1; i >= 0; i--)
    {
        if (isalpha(prefix[i]) || isdigit(prefix[i]))
        {
            root = new Node(prefix[i]);
            s.push(root);
        }
        else if (isOperator(prefix[i]))
        {
            t2 = s.top();
            s.pop();

            t1 = s.top();
            s.pop();
            root = new Node(prefix[i]);
            root->left = t2;
            root->right = t1;
            s.push(root);
        }
    }
    root = s.top();
    s.pop();
}

void Tree::nonRecPostorder()
{
    if (root == NULL)
        return;

    stack<Node *> s1, s2;
    s1.push(root);
    Node *temp;

    while (!s1.empty())
    {
        temp = s1.top();
        s1.pop();
        s2.push(temp);

        if (temp->left != NULL)
            s1.push(temp->left);
        if (temp->right != NULL)
            s1.push(temp->right);
    }

    while (!s2.empty())
```

```
    {
        temp = s2.top();
        s2.pop();
        cout << temp->data << " ";
    }
}

void Tree::deleteEntireTree()
{
    if (root == NULL)
        return;

    root = NULL;
    delete root;
}

int main()
{
    int userInput;
    string expr;
    bool isAlpha = false, isDigit = false;
    Tree *tree = NULL;

    while (true)
    {
        cout << "\n1. Enter Prefix Expression\n2. Non Recursive Postorder\n3. Delete\n
4. Exit\n>>>> ";
        cin >> userInput;

        switch (userInput)
        {
        case 1:
            tree = new (nothrow) Tree;
            if (!tree)
                throw bad_alloc();

            cout << "\nEnter Prefix Expression: ";
            cin >> expr;

            for (int i = 0; i < expr.length(); i++)
            {
                if (isalpha(expr[i]))
                    isAlpha = true;
                else if (isdigit(expr[i]))
                    isDigit = true;
            }
            if (isAlpha && isDigit)
            {
                cout << "\nPlease Enter Valid Expression Tree!\n";
                isAlpha = false;
                isDigit = false;
                break;
            }
            tree->expression(expr);
```

```
            break;

        case 2:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }
            cout << "\n";
            tree->nonRecPostorder();
            cout << "\n";
            break;

        case 3:
            if (!tree)
            {
                cout << "\nTree is Empty!\n";
                break;
            }
            tree->deleteEntireTree();
            cout << "\nTree Deleted!\n";
            tree = NULL;
            break;

        case 4:
            exit(0);

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }
}
```

Output:

```
// Output

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 2

// Tree is Empty!

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 1
```

```
// Enter Prefix Expression: +--a*bc/123

// Please Enter Valid Expression Tree!

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 1

// Enter Prefix Expression: +--a*bc/def

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 2

// a b c * - d e / - f +

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 3

// Tree Deleted!

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 2

// Tree is Empty!

// 1. Enter Prefix Expression
// 2. Non Recursive Postorder
// 3. Delete
// 4. Exit
// >>>> 4
```

Conclusion:

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

Experiment No. 6

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.

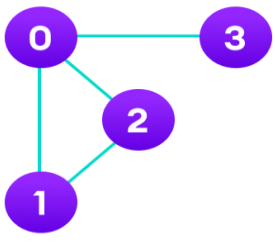
Objective/Theory:

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.



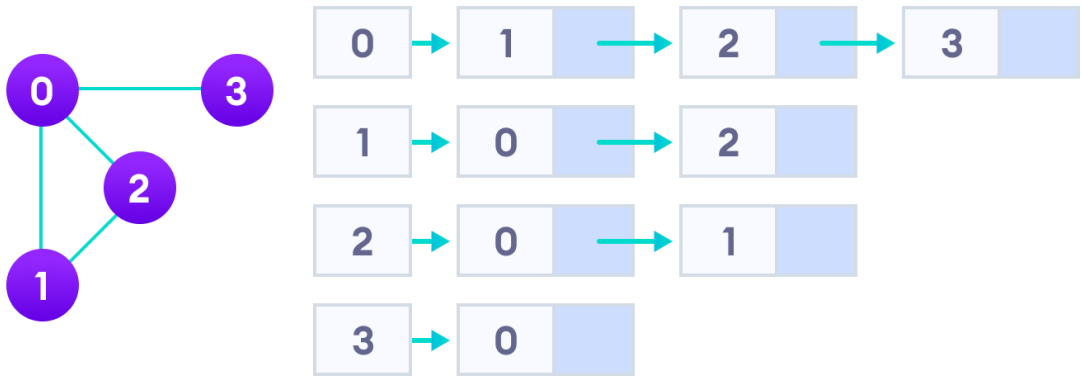
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

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:

```
#include <iostream>
#include <unordered_map>
#include <deque>
#include <stack>
#include <queue>
#include <list>
using namespace std;
```

```
class AdjacencyMatrix
{
    int vertices, edges;
    int **matrix = NULL;

public:
    AdjacencyMatrix(int vertices, int edges)
    {
        this->vertices = vertices;
        this->edges = edges;

        matrix = new (nothrow) int *[vertices];
        if (!matrix)
            throw bad_alloc();

        for (int i = 0; i < vertices; i++)
        {
            matrix[i] = new (nothrow) int[vertices];
            if (!matrix[i])
                throw bad_alloc();
        }
    }
    ~AdjacencyMatrix()
    {
        delete[] matrix;
    }
    void create();
    void display();
    void DepthFirstSearch(deque<bool> &);
};

void AdjacencyMatrix::create()
{
    for (int i = 0; i < vertices; i++)
    {
        for (int j = 0; j < vertices; j++)
        {
            matrix[i][j] = 0;
        }
    }

    int i, j;
    for (int k = 0; k < edges; k++)
    {
        cout << "\nEnter The Pair of Vertices For Edge " << k + 1 << " : ";
        cin >> i >> j;
        matrix[i][j] = 1;
        matrix[j][i] = 1;
    }
}

void AdjacencyMatrix::display()
{
    for (int i = 0; i < vertices; i++)
```

```
    {
        for (int j = 0; j < vertices; j++)
        {
            cout << " " << matrix[i][j];
        }
        cout << endl;
    }
}

void AdjacencyMatrix::DepthFirstSearch(deque<bool> &visited)
{
    stack<int> st;
    st.push(0);

    while (!st.empty())
    {
        int vertex = st.top();
        st.pop();

        if (!visited[vertex])
        {
            cout << vertex << " ";
            visited[vertex] = 1;
            for (int i = 0; i < vertices; i++)
            {
                if (matrix[vertex][i] == 1 and !visited[i])
                {
                    st.push(i);
                }
            }
        }
    }
    cout << endl;
}

class AdjacencyList
{
    int vertices, edges;
    unordered_map<int, list<int>> *adjacencylist;

public:
    AdjacencyList(int vertices, int edges)
    {
        this->vertices = vertices;
        this->edges = edges;
        adjacencylist = new (nothrow) unordered_map<int, list<int>>;
        if (!adjacencylist)
            throw bad_alloc();
    }
    ~AdjacencyList()
    {
        delete[] adjacencylist;
    }
    void create();
```

```
    void BreadthFirstSearch(deque<bool> &);
};

void AdjacencyList::create()
{
    int i, j;
    for (int k = 0; k < edges; k++)
    {
        cout << "\nEnter The Pair of Vertices For Edge " << k + 1 << " : ";
        cin >> i >> j;
        (*adjacencylist)[i].push_back(j);
    }
}

void AdjacencyList::BreadthFirstSearch(deque<bool> &visited)
{
    queue<int> que;
    que.push(0);

    while (!que.empty())
    {
        int vertex = que.front();
        cout << vertex << " ";
        visited[vertex] = true;
        que.pop();

        list<int>::iterator iter;
        for (iter = (*adjacencylist)[vertex].begin(); iter != (*adjacencylist)[vertex]
.end(); iter++)
        {
            if (!visited[*iter])
            {
                visited[*iter] = true;
                que.push(*iter);
            }
        }
    }
    cout << endl;
}

int main()
{
    int vertices, edges;
    int userInput;
    AdjacencyMatrix *adjm = NULL;
    AdjacencyList *adjl = NULL;

    while (true)
    {
        cout << "\n1. Create Graph Using Adjacency Matrix\n2. Display Adjacency Matrix
\n3. Create Graph Using Adjacency List\n4. Depth First Traversal\n5. Breadth First Tra
versal\n6. Exit\n>>>> ";
        cin >> userInput;
```

```
        switch (userInput)
        {
        case 1:
            cout << "\nEnter No. of Vertices: ";
            cin >> vertices;
            cout << "\nEnter No. of Edges: ";
            cin >> edges;
            adjm = new (nothrow) AdjacencyMatrix(vertices, edges);
            adjm->create();
            break;

        case 2:
            if (adjm == NULL)
            {
                cout << "\nGraph is Empty!\n";
                break;
            }
            cout << "\nAdjacency Matrix: "
                 << "\n";
            adjm->display();
            break;

        case 3:
            cout << "\nEnter No. of Vertices: ";
            cin >> vertices;
            cout << "\nEnter No. of Edges: ";
            cin >> edges;
            adjl = new (nothrow) AdjacencyList(vertices, edges);
            adjl->create();
            break;

        case 4:
        {
            if (adjm == NULL)
            {
                cout << "\nGraph is Empty!\n";
                break;
            }
            deque<bool> visited(vertices, false);
            cout << "\nDepth First Search Traversal: ";
            adjm->DepthFirstSearch(visited);
            break;
        }
        case 5:
        {
            if (adjl == NULL)
            {
                cout << "\nGraph is Empty!\n";
                break;
            }
            deque<bool> visited(vertices, false);
            cout << "\nBreadth First Search Traversal: ";
            adjl->BreadthFirstSearch(visited);
            break;
```

```
        }
        case 6:
            exit(0);
            break;

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }
    delete adjm;
    delete adjl;
    return 0;
}
```

Output:

```
// Output

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 2

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 4

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 5

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
```

```
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 1

// Enter No. of Vertices: 4

// Enter No. of Edges: 6

// Enter The Pair of Vertices For Edge 1 : 0 1

// Enter The Pair of Vertices For Edge 2 : 0 2

// Enter The Pair of Vertices For Edge 3 : 0 3

// Enter The Pair of Vertices For Edge 4 : 1 2

// Enter The Pair of Vertices For Edge 5 : 1 3

// Enter The Pair of Vertices For Edge 6 : 2 3

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 2

// Adjacency Matrix:
//  0 1 1 1
//  1 0 1 1
//  1 1 0 1
//  1 1 1 0

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 4

// Depth First Search Traversal: 0 3 2 1

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 5

// Graph is Empty!
```

```
// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 3

// Enter No. of Vertices: 4

// Enter No. of Edges: 6

// Enter The Pair of Vertices For Edge 1 : 0 1

// Enter The Pair of Vertices For Edge 2 : 0 2

// Enter The Pair of Vertices For Edge 3 : 0 3

// Enter The Pair of Vertices For Edge 4 : 1 2

// Enter The Pair of Vertices For Edge 5 : 1 3

// Enter The Pair of Vertices For Edge 6 : 2 3

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 5

// Breadth First Search Traversal: 0 1 2 3

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 6
```

Conclusion:

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

Experiment No. 7

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.

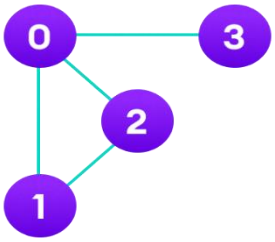
Objective/Theory:

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.

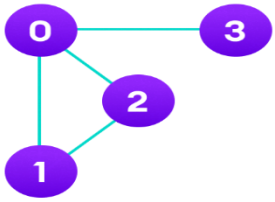
Graph Representation:

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.

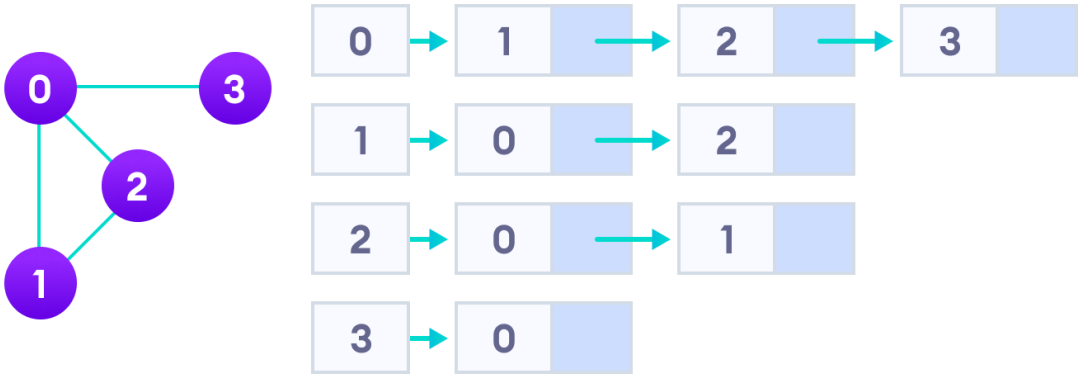


	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

•*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:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <stack>
#include <queue>
#include <list>
using namespace std;

class AdjacencyList
{
    int cities, paths;
    vector<string> listOfCities;
    unordered_map<int, list<int>> *adjacencylist;

public:
    AdjacencyList(int cities, int paths)
    {
        this->cities = cities;
        this->paths = paths;

        adjacencylist = new (nothrow) unordered_map<int, list<int>>;
        if (!adjacencylist)
            throw bad_alloc();
    }
    ~AdjacencyList()
    {
        delete[] adjacencylist;
    }
    void create();
    void DepthFirstSearch(vector<bool> &);
    void BreadthFirstSearch(vector<bool> &);
    bool isConnected(vector<bool> &);
};

int index(vector<string> list, string name)
{
    for (int i = 0; i < list.size(); i++)
    {
        if (list[i] == name)
            return i;
    }
    return -1;
}

void AdjacencyList::create()
{
    string name;
    for (int i = 0; i < cities; i++)
    {
        cout << "\nEnter The Name of City " << i + 1 << " : ";
        cin >> name;
        listOfCities.push_back(name);
```

```
    }

    string first, second;
    int idx1, idx2;
    vector<string>::iterator iter;
    for (int k = 0; k < paths; k++)
    {
        cout << "\nEnter The Name of cities For Path " << k + 1 << " : ";
        cin >> first;
        cin >> second;
        idx1 = index(listOfCities, first);
        idx2 = index(listOfCities, second);
        (*adjacencylist)[idx1].push_back(idx2);
    }
}

void AdjacencyList::DepthFirstSearch(vector<bool> &visited)
{
    stack<int> st;
    st.push(0);

    while (!st.empty())
    {
        int vertex = st.top();
        st.pop();

        cout << listOfCities[vertex] << " ";
        visited[vertex] = true;

        list<int>::iterator iter;
        for (iter = (*adjacencylist)[vertex].begin(); iter != (*adjacencylist)[vertex]
.end(); iter++)
        {
            if (!visited[*iter])
            {
                visited[*iter] = true;
                st.push(*iter);
            }
        }
    }
    cout << endl;
}

void AdjacencyList::BreadthFirstSearch(vector<bool> &visited)
{
    queue<int> que;
    que.push(0);

    while (!que.empty())
    {
        int vertex = que.front();
        cout << listOfCities[vertex] << " ";
        visited[vertex] = true;
        que.pop();
```

```
        list<int>::iterator iter;
        for (iter = (*adjacencylist)[vertex].begin(); iter != (*adjacencylist)[vertex]
.end(); iter++)
        {
            if (!visited[*iter])
            {
                visited[*iter] = true;
                que.push(*iter);
            }
        }
    }
    cout << endl;
}

bool AdjacencyList::isConnected(vector<bool> &visited)
{
    bool connected = true;
    DepthFirstSearch(visited);
    for (int i = 0; i < visited.size(); i++)
    {
        if (!visited[i])
            connected = false;
    }
    return connected;
}

int main()
{
    int cities, paths;
    int userInput;
    AdjacencyList *adjl = NULL;

    while (true)
    {
        cout << "\n1. Create Graph\n2. Depth First Traversal\n3. Breadth First Travers
al\n4. Check if Graph is Connected\n5. Exit\n>>>> ";
        cin >> userInput;

        switch (userInput)
        {
        case 1:
            cout << "\nEnter No. of Cities: ";
            cin >> cities;
            cout << "\nEnter No. of Paths: ";
            cin >> paths;
            adjl = new (nothrow) AdjacencyList(cities, paths);
            adjl->create();
            break;

        case 2:
        {
            if (adjl == NULL)
            {
```

```
                cout << "\nGraph is Empty!\n";
                break;
            }
            vector<bool> visited(cities, false);
            cout << "\nDepth First Search Traversal: ";
            adjl->DepthFirstSearch(visited);
            break;
        }

        case 3:
        {
            if (adjl == NULL)
            {
                cout << "\nGraph is Empty!\n";
                break;
            }
            vector<bool> visited(cities, false);
            cout << "\nBreadth First Search Traversal: ";
            adjl->BreadthFirstSearch(visited);
            break;
        }

        case 4:
        {
            vector<bool> visited(cities, false);
            bool connected = adjl->isConnected(visited);
            if (connected)
                cout << "\nThe Graph is Connected!\n";
            else
                cout << "\nThe Graph is Not Connected!\n";
            break;
        }

        case 5:
            exit(0);
            break;

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }
    delete adjl;
    return 0;
}
```

Output:

```
// Output

// GRAPH IS CONNECTED

// 1. Create Graph
```

```
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 2

// Graph is Empty!

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 3

// Graph is Empty!

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 1

// Enter No. of Cities: 4

// Enter No. of Paths: 6

// Enter The Name of City 1 : Mumbai

// Enter The Name of City 2 : Delhi

// Enter The Name of City 3 : Bangalore

// Enter The Name of City 4 : Pune

// Enter The Name of cities For Path 1 : Mumbai Delhi

// Enter The Name of cities For Path 2 : Mumbai Bangalore

// Enter The Name of cities For Path 3 : Mumbai Pune

// Enter The Name of cities For Path 4 : Delhi Bangalore

// Enter The Name of cities For Path 5 : Delhi Pune

// Enter The Name of cities For Path 6 : Bangalore Pune

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 2

// Depth First Search Traversal: Mumbai Pune Bangalore Delhi

// 1. Create Graph
```

```
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 3

// Breadth First Search Traversal: Mumbai Delhi Bangalore Pune

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 4

// The Graph is Connected!

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Exit
// >>>> 5

// Exit




// GRAPH IS NOT CONNECTED

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Check if Graph is Connected
// 5. Exit
// >>>> 1

// Enter No. of Cities: 4

// Enter No. of Paths: 2

// Enter The Name of City 1 : Mumbai

// Enter The Name of City 2 : Delhi

// Enter The Name of City 3 : Bangalore

// Enter The Name of City 4 : Pune

// Enter The Name of cities For Path 1 : Mumbai Pune

// Enter The Name of cities For Path 2 : Delhi Bangalore

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
```

```
// 4. Check if Graph is Connected
// 5. Exit
// >>>> 2

// Depth First Search Traversal: Mumbai Pune

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Check if Graph is Connected
// 5. Exit
// >>>> 3

// Breadth First Search Traversal: Mumbai Pune

// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Check if Graph is Connected
// 5. Exit
// >>>> 4
// Mumbai Pune

// The Graph is Not Connected!


// 1. Create Graph
// 2. Depth First Traversal
// 3. Breadth First Traversal
// 4. Check if Graph is Connected
// 5. Exit
// >>>> 5

// Exit
```

Conclusion:

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

Experiment No. 8

Experiment Name: Suggesting appropriate data structures.

Aim: You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.

Objectives:

1. Create a Graph and Display graph using adjacency matrix or adjacency list
2. Implement Prims algorithm to find MST

Perquisite: Basic Knowledge MST

Input: Read a graph from user using 2D array

Output: MST of entered graph

Theory:

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in

Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm:

- ☐ Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- ☐ Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices that are connected to growing spanning tree, into the Priority Queue.
- ☐ Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Program:

```
#include <iostream>
using namespace std;

int main()
{
    int n, i, j, k, row, col, mincost = 0, min;
    char op;
    cout << "Enter no. of vertices: ";
    cin >> n;
    int cost[n][n];
    int visit[n];
    for (i = 0; i < n; i++)
        visit[i] = 0;
    for (i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cost[i][j] = -1;
    for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
```

```
        {
            cout << "Do you want an edge between " << i + 1 << " and " << j +
1 << ": ";
            //use 'i' & 'j' if your vertices start from 0
            cin >> op;
            if (op == 'y' || op == 'Y')
            {
                cout << "Enter weight: ";
                cin >> cost[i][j];
                cost[j][i] = cost[i][j];
            }
        }
    }
    visit[0] = 1;
    for (k = 0; k < n - 1; k++)
    {
        min = 999;
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                if (visit[i] == 1 && visit[j] == 0)
                {
                    if (cost[i][j] != -1 && min > cost[i][j])
                    {
                        min = cost[i][j];
                        row = i;
                        col = j;
                    }
                }
            }
        }
        mincost += min;
        visit[col] = 1;
        cost[row][col] = cost[col][row] = -1;
        cout << row + 1 << "->" << col + 1 << endl;
        //use 'row' & 'col' if your vertices start from 0
    }
    cout << "\nMin. Cost: " << mincost;
    return 0;
}
```

Output:

```
// Output

// Enter no. of vertices: 4
// Do you want an edge between 1 and 2: y
// Enter weight: 20
// Do you want an edge between 1 and 3: y
// Enter weight: 30
// Do you want an edge between 1 and 4: y
// Enter weight: 40
// Do you want an edge between 2 and 3: n
// Do you want an edge between 2 and 4: n
// Do you want an edge between 3 and 4: y
// Enter weight: 50
// 1->2
// 1->3
// 1->4

// Min. Cost: 90
```

Conclusion: Thus we have created a set of lines that connects all your offices with a minimum total cost by suggesting appropriate data structures.

Experiment No. 9

Experiment Name: Binary Search Tree

Aim: Given sequence k = k1 <k2 < … <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?

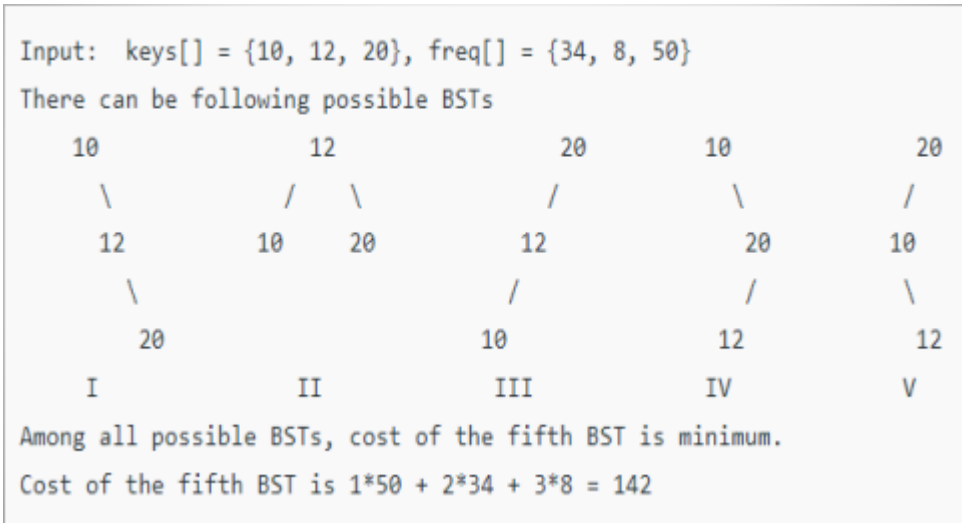
Objective:

1. Build the binary search tree with least search cost

Theory:

An optimal binary search tree, sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time that is minimum cost for a given sequence of accesses (or access probabilities)

Example:



Algorithm:

 Initial step:

w(0,0)=q0= 2 w(1,1)= q1=3

w(2,2)=q2= 1 w(3,3)= q3=1

w(4,4) =q4=1

And

c(0,0)=c(1,1)=c(2,2)= c(3,3)= c(4,4)=0

r(0,0)=r(1,1)=r(2,2)= r(3,3)= r(4,4)=0

 For second step

w(i , j) = w(i,j-1)+ pj + qj

So,w(0,1) = w(0,0) + p1 +q1

=2+3+3=8

Similarly, w(1,2) = 7 w(2,3) =3 w(3,4)=3

w(0,2)= 12 w(1,3)=9 w(2,4)=5

w(0,3)=14 w(1,4)=11

w(0,4)=16

	0	1	2	3	4
Key		10	20	30	40
Pi		3	3	1	1
qi	2	3	1	1	1

C(i , j) = min {c(i,k-1)+c(k,j)} + w(i,j)

i<k<=j

 C(0,1)=min {c(0,1-1)+c(1,1)} + w(0,1)

0<k<=1

Here K can be 1 only

Therefore

 C(0,1)={c(0,0)+c(1,1)} + w(0,1)

=0+0+8

=8

Similarly

C(1,2)=7 c(2,3)=3 c(3,4)=3 and

As k is having only 1 value equal to j

r(0,1)= 1 r(1,2)=2 r(2,3)=3 r(3,4)=4

 Similarly using same formula we can find

C(1,3)=12 r(1,3)=2

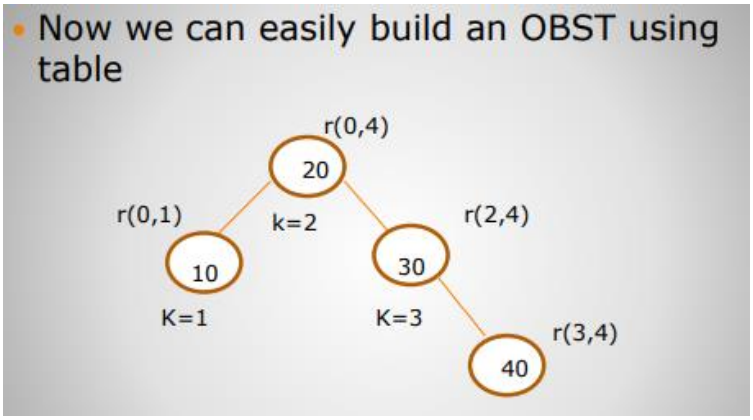
C(2,4)= 8 r(2,4)=3

C(0,3)= 25 r(0,3)=2

C(1,4)=19 r(1,4)=2

C(0,4)= 3 r(0,4)=2

r(i,j) value is the k value which gives Minimum cost



Program:

```
/* This program is to implement optimal binary search tree*/
#include<iostream>
using namespace std;
#define SIZE 10
class OBST
{
int p[SIZE]; // Probabilities with which we search for an element
int q[SIZE];//Probabilities that an element is not found
int a[SIZE];//Elements from which OBST is to be built
int w[SIZE][SIZE];//Weight ?w[i][j]? of a tree having root
//?r[i][j]?
int c[SIZE][SIZE];//Cost ?c[i][j] of a tree having root ?r[i][j]
int r[SIZE][SIZE];//represents root
int n; // number of nodes
public:
/* This function accepts the input data */
void get_data()
{
int i;
cout<<"\n Optimal Binary Search Tree \n";
cout<<"\n Enter the number of nodes";
cin>>n;
cout<<"\n Enter the data as?\n";
for(i=1;i<=n;i++)
{
cout<<"\n a["<<i<<"]";
cin>>a[i];
}
for(i=1;i<=n;i++)
{
cout<<"\n p["<<i<<"]";
cin>>p[i];
}
for(i=0;i<=n;i++)
{
cout<<"\n q["<<i<<"]";
cin>>q[i];
}
}
/* This function returns a value in the range ?r[i][j-1]? to ?r[i+1][j]?so
that the cost ?c[i][k-1]+c[k][j]?is minimum */
int Min_Value(int i,int j)
{
int m,k;
int minimum=32000;
```

```
for(m=r[i][j-1];m<=r[i+1][j];m++)
{
if((c[i][m-1]+c[m][j])<minimum)
{
 minimum=c[i][m-1]+c[m][j];
 k=m;
}
}
return k;
}
/* This function builds the table from all the given probabilities It
basically computes C,r,W values */
void build_OBST()
{
 int i,j,k,l,m;
 for(i=0;i<n;i++)
 {
//initialize
w[i][i]=q[i];
r[i][i]=c[i][i]=0;
//Optimal trees with one node
w[i][i+1]=q[i]+q[i+1]+p[i+1];
r[i][i+1]=i+1;
c[i][i+1]=q[i]+q[i+1]+p[i+1];
 }
 w[n][n]=q[n];
 r[n][n]=c[n][n]=0;
 //Find optimal trees with m nodes
 for(m=2;m<=n;m++)
 {
for(i=0;i<=n-m;i++)
{
 j=i+m;
 w[i][j]=w[i][j-1]+p[j]+q[j];
 k=Min_Value(i,j);
 c[i][j]=w[i][j]+c[i][k-1]+c[k][j];
 r[i][j]=k;
}
 }
}
/* This function builds the tree from the tables made by the OBST function */
void build_tree()
{
 int i,j,k;
 int queue[20],front=-1,rear=-1;
 cout<<"The Optimal Binary Search Tree For the Given Node Is\n";
 cout<<"\n The Root of this OBST is ::"<<r[0][n];
 cout<<"\nThe Cost of this OBST is::"<<c[0][n];
```

```
 cout<<"\n\n\t NODE \t LEFT CHILD \t RIGHT CHILD ";
 cout<<"\n";
 queue[++rear]=0;
 queue[++rear]=n;
 while(front!=rear)
 {
i=queue[++front];
j=queue[++front];
 k=r[i][j];
 cout<<"\n\t"<<k;
if(r[i][k-1]!=0)
{
cout<<"\t\t"<<r[i][k-1];
queue[++rear]=i;
queue[++rear]=k-1;
}
else
cout<<"\t\t";
if(r[k][j]!=0)
{
cout<<"\t"<<r[k][j];
queue[++rear]=k;
queue[++rear]=j;
}
else
cout<"\t";
 }//end of while
 cout<<"\n";
}
};//end of the class
/*This is the main function */
int main()
{
OBST obj;
obj.get_data();
obj.build_OBST();
obj.build_tree();
return 0;
}
```

Output:

```
/*-------------output---------------
Optimal Binary Search Tree
Enter the number of nodes 4
Enter the data as
a[1] 1
a[2] 2
```

```
a[3] 3
a[4] 4
p[1] 3
p[2]3
p[3]1
p[4]1
q[0]2
q[1]3
q[2]1
q[3]1
q[4]1
The Optimal Binary Search Tree For the Given Node Is
The Root of this OBST is ::2
The Cost of this OBST is::32
NODE LEFT CHILD RIGHT CHILD
2 1 3
1
3 4
4
--------------------*/
```

Conclusion: Thus we build the Binary Search tree that has the least search cost given the access probability for each key.

Experiment No. 10

Experiment Name: Priority Queue

Aim: Implement priority queue as ADT using single linked list for servicing patients in an hospital with priorities as

i) Serious (top priority)
ii) medium illness (medium priority)
iii) General (Least priority)

Objective:

1) To understand the concept of priority Queue.
2) How data structures Queue is represented as an ADT.

Theory:

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position.

This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data,

objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer. Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists. Queue is a data structure that maintains "First in First Out" (FIFO) order. And can be viewed as people queuing up to buy a ticket. In programming, queue is usually used as a data structure for BFS (Breadth First Search).

Operations on queue:

1. enqueue - insert item at the back of queue Q

2. dequeue - return (and virtually remove) the front item from queue Q

3. displayfront - return (without deleting) the front item from queue Q

4. displayrear - return (without deleting) the reart item from queue Q

Theoretically, one characteristic of a queue is that it does not have a specific capacity.

Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again. A practical implementation of a queue, e.g. with pointers, of course does have some capacity limit, that depends on the concrete situation it is used in. For a data structure the executing computer will eventually run out of memory, thus limiting the queue size. Queue overflow results

from trying to add an element onto a full queue and queue underflow happens when trying to remove an element from an empty queue. A bounded queue is a queue limited to a fixed number of items. priority queue is an abstract data type in computer programming that supports the following three operations:

 insertWithPriority: add an element to the queue with an associated priority

 getNext: remove the element from the queue that has the highest priority, and

return it (also known as "PopElement(Off)", or "GetMinimum")

 peekAtNext (optional): look at the element with highest priority without removing it.

The rule that determines who goes next is called a queuing discipline. The simplest queuing discipline is called FIFO, for "first-in-first-out." The most general queuing discipline is priority queuing, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queuing disciplines are "fair," but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: a Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queuing policy. As with most ADTs, there are a number of ways to implement queues

Since a queue is a collection of items, we can use any of the basic mechanisms for storing

Collections: arrays, lists, or vectors. Our choice among them will be based in part on their Performance--- how long it takes to perform the operations we want to perform--- and partly on ease of implementation.

Algorithm:

Define structure for Queue (Priority, Patient Info, Next Pointer).

Empty Queue:

Return true if Queue is Empty else False.

isEmpty(Front)

Front is pointer of structure,which is first element of Queue.

Step 1: If Front = = NULL

Step 2: Return 1

Step 3: Return 0

Insert Function:

Insert Patient in Queue with respect to the Priority.

Front is pointer variable of type Queue,which is 1st node of Queue.

Patient is a pointer variable of type Queue, which hold the information about new patient.

Insert(Front, Queue)

Step 1: If Front = = NULL //Queue Empty

Then Front = Patient;

Step 2: Else if Patient->Priority > Front->Priority

Then i) Patient->Next = Front;

ii) Front=Patient;

Step 3: Else A) Temp = Front;

B) Do Steps a while Temp != NULL And

Patient->Priority <= Temp->Next->Priority

a) Temp=Temp->Next;

c) Patient->Next = Temp->Next;

Temp->Next = Patient;

Step 4: Stop.

Delete Patient details from Queue after patient get treatment:

Front is pointer variable of type Queue, which is 1st node of Queue.

Delete Node from Front.

Delete(Front)

Step 1: Temp = Front;

Step 2: Front = Front->Next;

Step 3: return Temp

Display Queue Front:

Front is pointer variable of type Queue,which is 1st node of Queue.

Display(Front)

Step 1: Temp = Front;

Step 2: Do Steps while Temp != NULL

a) Display Temp Data

b) If Priority 1 Then —General Checkupǁ;

Else If Priority 2 Then Display — Non-serious";

Else If Priority 3 Then Display "Serious"

Else Display "Unknown";

c) Temp = Temp->Next;

Step 3: Stop.

Display Queue rear:

Front is pointer variable of type Queue,which is 1st node of Queue.

Display(Rear)

Step 1: Temp = Rear;

Step 2: Do Steps while Temp != NULL

a) Display Temp Data

b) If Priority 1 Then —General Checkupǁ;

Else If Priority 2 Then Display — Non-serious";

Else If Priority 3 Then Display "Serious"

Else Display "Unknown";

c) Temp = Temp->Next;

Step 3: Stop.

INPUT:

Test Case O/P

Queue Empty Display message —Queue Emptyǁ

Queue Full Display message —Queue Fullǁ

Name of patients & category of patient like

a) Serious (top priority), b) non-serious (medium priority), c) General Checkup (Least

priority).

E.g. Enter patient arrival in the hospital with following priorities 1, 3, 2, 2, 1, 3

OUTPUT:

Priority queue cater services to the patients based on priorities.

Patient should be given service in the following order 3, 3, 2, 2, 1, 1

Program:

```
#include<iostream>
#include<string>

#define N 20

#define SERIOUS 10
#define NONSERIOUS 5
#define CHECKUP 1

using namespace std;
string Q[N];
int Pr[N];
int r = -1,f = -1;
void enqueue(string data,int p)//Enqueue function to insert data and its prior
ity in queue
{
    int i;
    if((f==0)&&(r==N-1)) //Check if Queue is full
        cout<<"Queue is full";
    else {
        if(f==-1) { //if Queue is empty
            f = r = 0;
            Q[r] = data;
```

```
            Pr[r] = p;

        }
        else if(r == N-1) { //if there there is some elemets in Queue
            for(i=f;i<=r;i++) {
                                Q[i-f] = Q[i];
                                Pr[i-f] = Pr[i];
                                r = r-f;
                                f = 0;
                                for(i = r;i>f;i--) {
                    if(p>Pr[i]) {
                        Q[i+1] = Q[i];
                        Pr[i+1] = Pr[i];
                    }
                    else break;

                    Q[i+1] = data;
                    Pr[i+1] = p;
                    r++;
                }
            }
        }
        else {
            for(i = r;i>=f;i--) {
                if(p>Pr[i]) {
                    Q[i+1] = Q[i];
                    Pr[i+1] = Pr[i];
                }
                else break;
            }
            Q[i+1] = data;
            Pr[i+1] = p;
            r++;
        }
    }

}
void print() { //print the data of Queue
    int i;
    for(i=f;i<=r;i++) {
        cout << "Patient's Name - "<<Q[i];
        switch(Pr[i]) {
            case 1:
                cout << " Priority - 'Checkup' " << endl;
            break;
            case 5:
                cout << " Priority - 'Non-serious' " << endl;
            break;
```

```
            case 10:
                cout << " Priority - 'Serious' " << endl;
            break;
            default:
                cout << "Priority not found" << endl;
        }
    }
}

void dequeue() { //remove the data from front
    if(f == -1) {
        cout<<"Queue is Empty";
    }
    else {
    cout<<"deleted Element ="<<Q[f]<<endl;
    cout<<"Its Priority = "<<Pr[f]<<endl;
        if(f==r) f = r = -1;
        else f++;
    }
}

int main() {
    string data;
    int opt,n,i,p;
    cout<<"Enter Your Choice:-"<<endl;
    do {
    cout << "1 for Insert the Data in Queue" << endl << "2 for show the Data i
n Queue " << endl << "3 for Delete the data from the Queue"
        << endl << "0 for Exit"<< endl;
    cin >> opt;
        switch(opt) {
            case 1:
                cout << "Enter the number of patinent" << endl;
                cin >> n;
                i = 0;
                while(i < n) {
                    cout << "Enter your name of the patient : ";
                    cin >> data;
                    ifnotdoagain:
                        cout << "Enter your Prioritys (0: serious, 1: non-
serious, 2: genral checkup) : ";
                        cin >> p;
                        switch(p) {
                            case 0:
                                enqueue(data,SERIOUS);
                            break;
                            case 1:
                                enqueue(data,NONSERIOUS);
```

```
                            break;
                            case 2:
                                enqueue(data,CHECKUP);
                            break;
                            default:
                                goto ifnotdoagain;
                        }

                    i++;
                }
            break;
            case 2:
                print();
            break;
            case 3:
                 dequeue();
            break;
            case 0:
                cout << "Bye Bye !" << endl;
            break;
            default:
            cout<<"Incorrect Choice"<<endl;

        }
    }while(opt!=0);
        return 0;
}
```

Output:

Enter Your Choice:-

1 for Insert the Data in Queue

2 for show the Data in Queue

3 for Delete the data from the Queue

0 for Exit

1

Enter the number of patient

10

Enter your name of the patient : akash

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 2

Enter your name of the patient : akash

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 1

Enter your name of the patient : ashish

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 0

Enter your name of the patient : harsh

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 2

Enter your name of the patient : shreeya

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 2

Enter your name of the patient : siddhi

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 1

Enter your name of the patient : kartik

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 0

Enter your name of the patient : varun

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 2

Enter your name of the patient : karan

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup) : 1

Enter your name of the patient : vikas

Enter your Priorities (0: serious, 1: non-serious, 2: general checkup)
: 2

1 for Insert the Data in Queue

2 for show the Data in Queue

3 for Delete the data from the Queue

0 for Exit

2

Patient's Name - ashish Priority - 'Serious'

Patient's Name - kartik Priority - 'Serious'

Patient's Name - akash Priority - 'Non-serious'

Patient's Name - siddhi Priority - 'Non-serious'

Patient's Name - karan Priority - 'Non-serious'

Patient's Name - akash Priority - 'Checkup'

Patient's Name - harsh Priority - 'Checkup'

Patient's Name - shreeya Priority - 'Checkup'

Patient's Name - varun Priority - 'Checkup'

Patient's Name - vikas Priority - 'Checkup'

1 for Insert the Data in Queue

2 for show the Data in Queue

3 for Delete the data from the Queue

0 for Exit

3

deleted Element =ashish

Its Priority = 10

1 for Insert the Data in Queue

2 for show the Data in Queue

3 for Delete the data from the Queue

0 for Exit

3

deleted Element =kartik

Its Priority = 10

1 for Insert the Data in Queue

2 for show the Data in Queue

3 for Delete the data from the Queue

0 for Exit

0

Bye Bye !

Conclusion: Thus we have implemented the priority queue to cater services to patients

Experiment No. 11

Experiment Name: Use sequential file to maintain data.

Aim: 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.

Objectives:

The local and physical organization of files.

To understand the concept of sequential files.

To know about File Operations

To understand the use of sequential files.

Sequential file handling methods.

Creating, reading, writing and deleting records from a variety of file structures.

Creating code to carry out the above operations.

Mapping of assignments

Analyse the algorithmic solutions for resource requirement and optimisation.

Use appropriate modern tools to understand and analyse the functionalise confine to the secondary storage.

Outcome

CO1: Understand the ADT/libraries, hash tables and dictionary to design algorithms for a

specific problem.

CO2: Choose most appropriate data structures and apply algorithms for graphical solutions of

the problems.

CO3: Apply and analyze non linear data structures to solve real world complex problems.

CO4: Apply and analyze algorithm design techniques for indexing, sorting, multi-way searching, file organization and compression.

CO5: Analyze the efficiency of most appropriate data structure for creating efficient solutionsfor engineering design situations.

Theory:

A sequential file contains records organized by the order in which they were entered. The order of the records is fixed.

Records in sequential files can be read or written only sequentially.

After you place a record into a sequential file, you cannot shorten, lengthen, or delete the record. However, you can update (`REWRITE`) a record if the length does not change. New records are added at the end of the file.

If the order in which you keep records in a file is not important, sequential organization is a good choice whether there are many records or only a few. Sequential output is also useful for printing reports

Types of File Organization-

Sequential access file organization

Indexed sequential access file organization

Direct access file organization

Sequential access file organization

Storing and sorting in contiguous block within files on tape or disk is called as sequential accessfile organization.

In sequential access file organization, all records are stored in a sequential order. The records arearranged in the ascending or descending order of a key field.

Sequential file search starts from the beginning of the file and the records can be added at the endof the file.

In sequential file, it is not possible to add a record in the middle of the file without rewriting thefile.

Primitive Operations on Sequential files

Open—This opens the file and sets the file pointer to immediately before the first record

Read-next—This returns the next record to the user. If no record is present, then EOFcondition will be set.

Close—This closes the file and terminates the access to the file.

Write-next—File pointers are set to next of last record and write the record to the file.

EOF—If EOF condition occurs, it returns true, otherwise it returns false.

Search—Search for the record with a given key.

Update—Current record is written at the same position with updated values.

Program:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <unordered_set>
using namespace std;

class Student
{
    int rollNumber;
    string name;
    string division;
    string address;

public:
    Student()
    {
        rollNumber = 0;
        name = "";
        division = "";
        address = "";
    }

    bool isEmpty(ifstream &);
    bool addStudent(int, string, string, string);
    bool deleteStudent(int);
    bool displayStudent(int);
    void displayStudents();
    friend ostream &operator<<(ostream &, const Student &); // Insertion Opera
tor Overloading
    friend istream &operator>>(istream &, Student &);       // Extraction Oper
ator Overloading
};

bool Student::isEmpty(ifstream &file)
{
    return file.peek() == ifstream::traits_type::eof();
}

bool Student::addStudent(int rollNumber, string name, string division, string
address)
{
    ofstream file;
    file.open("studentDetails.txt", ios::app);

    if (file.is_open())
    {
        this->rollNumber = rollNumber;
```

```
        this->name = name;
        this->division = division;
        this->address = address;
        file << *this;
        file.close();
        return true;
    }

    file.close();
    return false;
}

bool Student::deleteStudent(int rollNumber)
{
    bool flag = false;
    vector<Student> vect;
    ifstream file;
    Student student;
    file.open("studentDetails.txt", ios::in);

    if (file.is_open())
    {
        while (file >> student)
        {
            if (student.rollNumber == rollNumber)
            {
                flag = true;
                continue;
            }
            vect.push_back(student);
        }
    }

    file.close();

    if (flag)
    {
        ofstream file("studentDetails.txt");
        if (file.is_open())
        {
            for (int i = 0; i < vect.size(); i++)
            {
                file << vect[i];
            }
            file.close();
            return flag;
        }
        else
```

```
            cout << "Error! While Opening File.";

        flag = false;
        file.close();
    }
    return flag;
}

bool Student::displayStudent(int rollNumber)
{
    ifstream file;
    Student student;
    file.open("studentDetails.txt", ios::in);

    if (file.is_open())
    {
        while (file >> student)
        {
            if (student.rollNumber == rollNumber)
            {
                cout << "\nRoll Number: " << student.rollNumber << "\n";
                cout << "Name: " << student.name << "\n";
                cout << "Division: " << student.division << "\n";
                cout << "Address: " << student.address << "\n";
                file.close();
                return true;
            }
        }
    }

    else
        cout << "Error! While Opening File.";

    file.close();
    return false;
}

void Student::displayStudents()
{
    ifstream file;
    Student student;
    file.open("studentDetails.txt");

    if (file.is_open())
    {
        while (file >> student)
        {
            cout << "\nRoll Number: " << student.rollNumber << "\n";
```

```
            cout << "Name: " << student.name << "\n";
            cout << "Division: " << student.division << "\n";
            cout << "Address: " << student.address << "\n";
        }
    }

    else
        cout << "Error! While Opening File.";

    file.close();
}

ostream &operator<<(ostream &file, const Student &obj)
{
    file << obj.rollNumber << "\n"
         << obj.name << "\n"
         << obj.division << "\n"
         << obj.address << "\n";
    return file;
}

istream &operator>>(istream &file, Student &obj)
{
    file >> obj.rollNumber;
    file >> obj.name;
    file >> obj.division;
    file >> obj.address;
    return file;
}

int main()
{
    Student *student = new (nothrow) Student();
    if (!student)
        throw bad_alloc();

    int userInput, rollNumber;
    string name, division, address;
    bool flag;

    while (true)
    {
        cout << "\n1. Press 1 To Add Student\n2. Press 2 To Delete Student\n3.
 Press 3 To Display Student\n4. Press 4 To Display All Students\n5. Press 5 To
 Exit\n>>>> ";
        cin >> userInput;

        switch (userInput)
```

```
        {
        case 1:
            cout << "\nEnter Name of Student: ";
            cin.ignore();
            getline(cin, name);
            cout << "\nEnter Roll Number of " << name << ": ";
            cin >> rollNumber;
            cout << "\nEnter Division of " << name << ": ";
            cin >> division;
            cout << "\nEnter Address of " << name << ": ";
            cin >> address;
            flag = student->addStudent(rollNumber, name, division, address);
            if (flag)
                cout << "\nStudent Added Successfully!\n";
            else
                cout << "\nNot Able To Add Student!\n";
            break;
        case 2:
        {
            ifstream file("studentDetails.txt", ios::in);
            if (student->isEmpty(file))
            {
                cout << "\nFile is Empty!\n";
                break;
            }
            cout << "\nEnter Roll Number of Student: ";
            cin >> rollNumber;
            flag = student->deleteStudent(rollNumber);
            if (flag)
                cout << "\nStudent Deleted Successfully!\n";
            else
                cout << "\nStudent Data is Not Available!\n";
            break;
            file.close();
        }
        case 3:
        {
            ifstream file("studentDetails.txt", ios::in);
            if (student->isEmpty(file))
            {
                cout << "\nFile is Empty!\n";
                break;
            }
            cout << "\nEnter Roll Number of Student: ";
            cin >> rollNumber;
            flag = student->displayStudent(rollNumber);
            if (!flag)
                cout << "\nStudent Data is Not Available!\n";
```

```
            break;
            file.close();
        }
        case 4:
        {
            ifstream file("studentDetails.txt", ios::in);
            if (student->isEmpty(file))
            {
                cout << "\nFile is Empty!\n";
                break;
            }
            student->displayStudents();
            break;
        }
        case 5:
            exit(0);

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }

    return 0;
}
```

Output:

```
// Output

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 2

// File is Empty!

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 3

// File is Empty!
```

```
// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 4

// File is Empty!

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 1

// Enter Name of Student: Ashish

// Enter Roll Number of Ashish: 51

// Enter Division of Ashish: A

// Enter Address of Ashish: Shahada

// Student Added Successfully!

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 3

// Enter Roll Number of Student: 51

// Roll Number: 51
// Name: Ashish
// Division: A
// Address: Shahada

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 4
```

```
// Roll Number: 51
// Name: Ashish
// Division: A
// Address: Shahada

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 1

// Enter Name of Student: Mayur

// Enter Roll Number of Mayur: 52

// Enter Division of Mayur: A

// Enter Address of Mayur: Dhule

// Student Added Successfully!

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 3

// Enter Roll Number of Student: 52

// Roll Number: 52
// Name: Mayur
// Division: A
// Address: Dhule

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 4

// Roll Number: 51
// Name: Ashish
// Division: A
// Address: Shahada
```

```
// Roll Number: 52
// Name: Mayur
// Division: A
// Address: Dhule

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 2

// Enter Roll Number of Student: 52

// Student Deleted Successfully!

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 4

// Roll Number: 51
// Name: Ashish
// Division: A
// Address: Shahada

// 1. Press 1 To Add Student
// 2. Press 2 To Delete Student
// 3. Press 3 To Display Student
// 4. Press 4 To Display All Students
// 5. Press 5 To Exit
// >>>> 5
```

Conclusion:

Thus we have used sequential file to store student data.

Experiment No. 12

Experiment Name: Use sequential file to maintain data.

Aim: Company maintains a employee information. The file contains employeeID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If record of employeedoes not exists an appropriate message is displayed. If it is, then the system displays the employeedetails. Use sequential file to main the data.

Objectives:

The local and physical organization of files.

To understand the concept of sequential files.

To know about File Operations

To understand the use of sequential files.

Sequential file handling methods.

Creating, reading, writing and deleting records from a variety of file structures.

Creating code to carry out the above operations.

Mapping of assignments

Analyse the algorithmic solutions for resource requirement and optimisation.

Use appropriate modern tools to understand and analyse the functionalise confine to the secondary storage.

Outcome

CO1: Understand the ADT/libraries, hash tables and dictionary to design algorithms for a

specific problem.

CO2: Choose most appropriate data structures and apply algorithms for graphical solutions of

the problems.

CO3: Apply and analyzenon linear data structures to solve real world complex problems.

CO4:Applyandanalyzealgorithmdesigntechniquesforindexing,sorting ,multi-way searching, file organization and compression.

CO5: Analyze the efficiency of most appropriate data structure for creating efficient solutionsfor engineering design situations.

Theory:

A sequential file contains records organized by the order in which they were entered. The order of the records is fixed.

Records in sequential files can be read or written only sequentially.

After you place a record into a sequential file, you cannot shorten, lengthen, or delete the record. However, you can update (`REWRITE`) a record if the length does not change. New records are added at the end of the file.

If the order in which you keep records in a file is not important, sequential organization is a good choice whether there are many records or only a few. Sequential output is also useful for printing reports

Types of File Organization-

Sequential access file organization

Indexed sequential access file organization

Direct access file organization

Sequential access file organization

Storing and sorting in contiguous block within files on tape or disk is called as sequential accessfile organization.

In sequential access file organization, all records are stored in a sequential order. The records arearranged in the ascending or descending order of a key field.

Sequential file search starts from the beginning of the file and the records can be added at the endof the file.

In sequential file, it is not possible to add a record in the middle of the file without rewriting thefile.

Primitive Operations on Sequential files

Open—This opens the file and sets the file pointer to immediately before the first record

Read-next—This returns the next record to the user. If no record is present, then EOFcondition will be set.

Close—This closes the file and terminates the access to the file.

Write-next—File pointers are set to next of last record and write the record to the file.

EOF—If EOF condition occurs, it returns true, otherwise it returns false.

Search—Search for the record with a given key.

Update—Current record is written at the same position with updated values

Program:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <unordered_set>
using namespace std;

class Employee
{
    int empId;
    string name;
    string designation;
    string salary;

public:
    Employee()
    {
        empId = 0;
        name = "";
        designation = "";
        salary = "";
    }

    bool isEmpty(ifstream &);
    bool addEmployee(int, string, string, string);
    bool deleteEmployee(int);
    bool displayEmployee(int);
    void displayEmployees();
    friend ostream &operator<<(ostream &, const Employee &); // Insertion Oper
ator Overloading
    friend istream &operator>>(istream &, Employee &);       // Extraction Ope
rator Overloading
};

bool Employee::isEmpty(ifstream &file)
{
    return file.peek() == ifstream::traits_type::eof();
}

bool Employee::addEmployee(int empId, string name, string designation, string
salary)
{
    ofstream file;
    file.open("employeeDetails.txt", ios::app);

    if (file.is_open())
    {
        this->empId = empId;
```

```
        this->name = name;
        this->designation = designation;
        this->salary = salary;
        file << *this;
        file.close();
        return true;
    }

    file.close();
    return false;
}

bool Employee::deleteEmployee(int empId)
{
    bool flag = false;
    vector<Employee> vect;
    ifstream file;
    Employee Employee;
    file.open("employeeDetails.txt", ios::in);

    if (file.is_open())
    {
        while (file >> Employee)
        {
            if (Employee.empId == empId)
            {
                flag = true;
                continue;
            }
            vect.push_back(Employee);
        }
    }

    file.close();

    if (flag)
    {
        ofstream file("employeeDetails.txt");
        if (file.is_open())
        {
            for (int i = 0; i < vect.size(); i++)
            {
                file << vect[i];
            }
            file.close();
            return flag;
        }
        else
```

```
            cout << "Error! While Opening File.";

        flag = false;
        file.close();
    }
    return flag;
}

bool Employee::displayEmployee(int empId)
{
    ifstream file;
    Employee employee;
    file.open("employeeDetails.txt", ios::in);

    if (file.is_open())
    {
        while (file >> employee)
        {
            if (employee.empId == empId)
            {
                cout << "\nEmployee Id: " << employee.empId << "\n";
                cout << "Name: " << employee.name << "\n";
                cout << "Designation: " << employee.designation << "\n";
                cout << "Salary: " << employee.salary << "\n";
                file.close();
                return true;
            }
        }
    }

    else
        cout << "Error! While Opening File.";

    file.close();
    return false;
}

void Employee::displayEmployees()
{
    ifstream file;
    Employee employee;
    file.open("employeeDetails.txt");

    if (file.is_open())
    {
        while (file >> employee)
        {
            cout << "\nEmployee Id: " << employee.empId << "\n";
```

```
            cout << "Name: " << employee.name << "\n";
            cout << "Designation: " << employee.designation << "\n";
            cout << "Salary: " << employee.salary << "\n";
        }
    }

    else
        cout << "Error! While Opening File.";

    file.close();
}

ostream &operator<<(ostream &file, const Employee &obj)
{
    file << obj.empId << "\n"
         << obj.name << "\n"
         << obj.designation << "\n"
         << obj.salary << "\n";
    return file;
}

istream &operator>>(istream &file, Employee &obj)
{
    file >> obj.empId;
    file >> obj.name;
    file >> obj.designation;
    file >> obj.salary;
    return file;
}

int main()
{
    Employee *employee = new (nothrow) Employee();
    if (!employee)
        throw bad_alloc();

    int userInput, empId;
    string name, designation, salary;
    bool flag;

    while (true)
    {
        cout << "\n1. Press 1 To Add Employee\n2. Press 2 To Delete Employee\n
3. Press 3 To Display Employee\n4. Press 4 To Display All Employees\n5. Press
5 To Exit\n>>>> ";
        cin >> userInput;

        switch (userInput)
```

```
        {
        case 1:
            cout << "\nEnter Name of Employee: ";
            cin>>name;
            cout << "\nEnter Employee Id of " << name << ": ";
            cin >> empId;
            cout << "\nEnter Designation of " << name << ": ";
            cin>>designation;
            cout << "\nEnter Salary of " << name << ": ";
            cin >> salary;
            flag = employee->addEmployee(empId, name, designation, salary);
            if (flag)
                cout << "\nEmployee Added Successfully!\n";
            else
                cout << "\nNot Able To Add Employee!\n";
            break;
        case 2:
        {
            ifstream file("employeeDetails.txt", ios::in);
            if (employee->isEmpty(file))
            {
                cout << "\nFile is Empty!\n";
                break;
            }
            cout << "\nEnter Employee Id of Employee: ";
            cin >> empId;
            flag = employee->deleteEmployee(empId);
            if (flag)
                cout << "\nEmployee Deleted Successfully!\n";
            else
                cout << "\nEmployee Data is Not Available!\n";
            break;
            file.close();
        }
        case 3:
        {
            ifstream file("employeeDetails.txt", ios::in);
            if (employee->isEmpty(file))
            {
                cout << "\nFile is Empty!\n";
                break;
            }
            cout << "\nEnter Employee Id of Employee: ";
            cin >> empId;
            flag = employee->displayEmployee(empId);
            if (!flag)
                cout << "\nEmployee Data is Not Available!\n";
```

```
            break;
            file.close();
        }
        case 4:
        {
            ifstream file("employeeDetails.txt", ios::in);
            if (employee->isEmpty(file))
            {
                cout << "\nFile is Empty!\n";
                break;
            }
            employee->displayEmployees();
            break;
        }
        case 5:
            exit(0);

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }

    return 0;
}
```

Output:

```
// Output

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 2

// File is Empty!

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 3

// File is Empty!
```

```
// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 4

// File is Empty!

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 1

// Enter Name of Employee: Ashish

// Enter Employee Id of Ashish: 51

// Enter Designation of Ashish: Python_Developer

// Enter Salary of Ashish: 1000000/-

// Employee Added Successfully!

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 3

// Enter Employee Id of Employee: 51

// Employee Id: 51
// Name: Ashish
// Designation: Python
// Salary: Developer

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 4
```

```
// Employee Id: 51
// Name: Ashish
// Designation: Python_Developer
// Salary: 1000000/-

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 1

// Enter Name of Employee: Mayur

// Enter Employee Id of Mayur: 52

// Enter Designation of Mayur: Cpp_Developer

// Enter Salary of Mayur: 8000000/-

// Employee Added Successfully!

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 3

// Enter Employee Id of Employee: 52

// Employee Id: 52
// Name: Mayur
// Designation: Cpp_Developer
// Salary: 8000000/-

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 2

// Enter Employee Id of Employee: 52

// Employee Deleted Successfully!

// 1. Press 1 To Add Employee
```

```
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 4

// Employee Id: 51
// Name: Ashish
// Designation: Python_Developer
// Salary: 1000000/-

// 1. Press 1 To Add Employee
// 2. Press 2 To Delete Employee
// 3. Press 3 To Display Employee
// 4. Press 4 To Display All Employees
// 5. Press 5 To Exit
// >>>> 5

// Exit
```

Conclusion:

Thus we have used sequential file to store employee data.