GROUP A

1) Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client's telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers.

Program 1: double hashing

```
from Record import Record

class doubleHashTable:
    # initialize hash Table
    def __init__(self):
        self.size = int(input("Enter the Size of the hash table : "))

        # initialize table with all elements 0
        self.table = list(None for i in range(self.size))
        self.elementCount = 0
        self.comparisons = 0


    # method that checks if the hash table is full or not
    def isFull(self):
        if self.elementCount == self.size:
            return True
        else:
            return False


    # First hash function
    def h1(self, element):
        return element % self.size

    # Second hash function
    def h2(self, element):
        return 5-(element % 5)


    # method to resolve collision by double hashing method
    def doubleHashing(self, record):
        posFound = False
```

```
        # limit variable is used to restrict the function from going into
infinite loop
        # limit is useful when the table is 80% full
        limit = self.size
        i = 1
        # start a loop to find the position
        while i <= limit:
            # calculate new position by quadratic probing
            newPosition = (self.h1(record.get_number()) +
i*self.h2(record.get_number())) % self.size
            # if newPosition is empty then break out of loop and return new
Position
            if self.table[newPosition] == None:
                posFound = True
                break
            else:
                # as the position is not empty increase i
                i += 1
        return posFound, newPosition


    # method that inserts element inside the hash table
    def insert(self, record):
        # checking if the table is full
        if self.isFull():
            print("Hash Table Full")
            return False

        posFound = False

        position = self.h1(record.get_number())

        # checking if the position is empty
        if self.table[position] == None:
            # empty position found , store the element and print the message
            self.table[position] = record
            print("Phone number of " + record.get_name() + " is at position " +
str(position))
            isStored = True
            self.elementCount += 1

        # If collision occured
        else:
            print("Collision has occured for " + record.get_name() + "'s phone
number at position " + str(position) + " finding new Position.")
```

```
            while not posFound:
                posFound, position = self.doubleHashing(record)
                if posFound:
                    self.table[position] = record
                    #print(self.table[position])
                    self.elementCount += 1
                    #print(position)
                    #print(posFound)
                    print("Phone number of " + record.get_name() + " is at
position " + str(position))

        return posFound


    # searches for an element in the table and returns position of element if
found else returns False
    def search(self, record):
        found = False
        position = self.h1(record.get_number())
        self.comparisons += 1

        if(self.table[position] != None):
            if(self.table[position].get_name() == record.get_name()):
                print("Phone number found at position {}".format(position) + "
and total comparisons are " + str(1))
                return position

            # if element is not found at position returned hash function
            # then we search element using double hashing
            else:
                limit = self.size
                i = 1

                newPosition = position
                # start a loop to find the position
                while i <= limit:
                    # calculate new position by double Hashing
                    position = (self.h1(record.get_number()) +
i*self.h2(record.get_number())) % self.size
                    self.comparisons += 1
                    # if element at newPosition is equal to the required element

                    if(self.table[position] != None):
                        if self.table[position].get_name() == record.get_name():
```

```
                            found = True
                            break

                        elif self.table[position].get_name() == None:
                            found = False
                            break

                        else:
                            # as the position is not empty increase i
                            i += 1


            if found:
                print("Phone number found at position {}".format(position) + "
and total comparisons are " + str(i+1))
                #return position
            else:
                print("Record not Found")
                return found


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

Program 2: linear probing

```
# Program to implement Hashing with Linear Probing

from Record import Record

class hashTable:
    # initialize hash Table
    def __init__(self):
        self.size = int(input("Enter the Size of the hash table : "))
        # initialize table with all elements 0
        self.table = list(None for i in range(self.size))
        self.elementCount = 0
        self.comparisons = 0
```

```
    # method that checks if the hash table is full or not
    def isFull(self):
        if self.elementCount == self.size:
            return True
        else:
            return False


    # method that returns position for a given element
    def hashFunction(self, element):
        return element % self.size


    # method that inserts element into the hash table
    def insert(self, record):
        # checking if the table is full
        if self.isFull():
            print("Hash Table Full")
            return False

        isStored = False

        position = self.hashFunction(record.get_number())

        # checking if the position is empty
        if self.table[position] == None:
            self.table[position] = record
            print("Phone number of " + record.get_name() + " is at position " +
str(position))
            isStored = True
            self.elementCount += 1

        # collision occured hence we do linear probing
        else:
            print("Collision has occured for " + record.get_name() + "'s phone
number at position " + str(position) + " finding new Position.")
            while self.table[position] != None:
                position += 1
                if position >= self.size:
                    position = 0

            self.table[position] = record
```

```
            print("Phone number of " + record.get_name() + " is at position " +
str(position))
            isStored = True
            self.elementCount += 1
        return isStored


    # method that searches for an element in the table
    # returns position of element if found
    # else returns False
    def search(self, record):
        found = False

        position = self.hashFunction(record.get_number())
        self.comparisons += 1

        if(self.table[position] != None):
            if(self.table[position].get_name() == record.get_name() and
self.table[position].get_number() == record.get_number()):
                isFound = True
                print("Phone number found at position {} ".format(position) + "
and total comparisons are " + str(1))
                return position



        # if element is not found at position returned hash function

            else:
                position += 1
                if position >= self.size-1:
                    position = 0
                while self.table[position] != None or self.comparisons <=
self.size:

                    if(self.table[position].get_name() == record.get_name() and
self.table[position].get_number() == record.get_number()):
                        isFound = True
                        #i=0
                        i = self.comparisons + 1
                        print("Phone number found at position {}
".format(position) + " and total comparisons are " + str(i) )
                        return position

                    position += 1
```

```
                    #print(position)
                    if position >= self.size-1:
                        position = 0

                    #print(position)
                    self.comparisons += 1
                    #print(self.comparisons)

                if isFound == False:
                    print("Record not found")
                    return false



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

Program 3: main

```
from LinearProbing import hashTable
from Record import Record
from DoubleHashing import doubleHashTable

def input_record():
    record = Record()
    name = input("Enter Name:")
    number = int(input("Enter Number:"))
    record.set_name(name)
    record.set_number(number)
    return record


choice1 = 0
while(choice1 != 3):
    print("************************")
    print("1. Linear Probing      *")
    print("2. Double Hashing      *")
    print("3. Exit                *")
    print("************************")
```

```
choice1 = int(input("Enter Choice"))
if choice1>3:
    print("Please Enter Valid Choice")

if choice1 == 1:
    h1 = hashTable()
    choice2 = 0
    while(choice2 != 4):
        print("************************")
        print("1. Insert               *")
        print("2. Search               *")
        print("3. Display              *")
        print("4. Back                 *")
        print("************************")

        choice2 = int(input("Enter Choice"))
        if choice2>4:
            print("Please Enter Valid Choice")

        if(choice2==1):
            record = input_record()
            h1.insert(record)

        elif(choice2 == 2):
            record = input_record()
            position = h1.search(record)

        elif(choice2 == 3):
            h1.display()



elif choice1 == 2:
    h2 = doubleHashTable()
    choice2 = 0
    while(choice2 != 4):
        print("************************")
        print("1. Insert               *")
        print("2. Search               *")
        print("3. Display              *")
        print("4. Back                 *")
        print("************************")

        choice2 = int(input("Enter Choice"))
```

```
            if choice2>4:
                print("Please Enter Valid Choice")

            if(choice2==1):
                record = input_record()
                h2.insert(record)

            elif(choice2 == 2):
                record = input_record()
                position = h2.search(record)

            elif(choice2 == 3):
                h2.display()
```

Program 4: record

```
class Record:
    def __init__(self):
        self._name = None
        self._number = None

    def get_name(self):
        return self._name

    def get_number(self):
        return self._number

    def set_name(self,name):
```

```
        self._name = name

    def set_number(self,number):
        self._number = number

    def __str__(self):
        record = "Name: "+str(self.get_name())+"\t"+"\tNumber:
"+str(self.get_number())
        return record
```

2) To create 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 Iterator () Return an iterator used to loop over collection, e. Intersection of two sets , f. Union of two sets, g. Difference between two sets, h. Subset

Program 1: menu

```
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 or not")
        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 or not")
        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")
```

Program 2: set operations

```
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)
```

GROUP B

3) 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.

Program 1:

```
# include <iostream>
# include <cstdlib>
# include <string.h>
using namespace std;
/*
 * Node Declaration
 */
struct node
{
   char label[10];
   int ch_count;
   struct node *child[10];
}*root;

/*
 * Class Declaration
 */
class GT
{
```

```
   public:
       void create_tree();
 void display(node * r1);

       GT()
       {
          root = NULL;
       }
};

void GT::create_tree()
{
 int tbooks,tchapters,i,j,k;
 root = new node;
 cout<<"Enter name of book";
 cin>>root->label;
 cout<<"Enter no. of chapters in book";
 cin>>tchapters;
 root->ch_count = tchapters;
 for(i=0;i<tchapters;i++)
 {
  root->child[i] = new node;
  cout<<"Enter Chapter name\n";
  cin>>root->child[i]->label;
  cout<<"Enter no. of sections in  Chapter: "<<root->child[i]->label;
```

```
 cin>>root->child[i]->ch_count;
 for(j=0;j<root->child[i]->ch_count;j++)
 {
  root->child[i]->child[j] = new node;
  cout<<"Enter Section "<<j+1<<"name\n";
  cin>>root->child[i]->child[j]->label;
  //cout<<"Enter no. of subsections in "<<r1->child[i]->child[j]->label;
  //cin>>r1->child[i]->ch_count;
 }

}

}


void GT::display(node * r1)
{
 int i,j,k,tchapters;
 if(r1 != NULL)
 {
  cout<<"\n-----Book Hierarchy---";

  cout<<"\n Book title : "<<r1->label;
  tchapters = r1->ch_count;
  for(i=0;i<tchapters;i++)
```

```
  {

   cout<<"\n  Chapter "<<i+1;
   cout<<" "<<r1->child[i]->label;
    cout<<"\n Sections";
   for(j=0;j<r1->child[i]->ch_count;j++)
   {
    //cin>>r1->child[i]->child[j]->label;
    cout<<"\n  "<<r1->child[i]->child[j]->label;
   }

  }
 }
}



/*
 * Main Contains Menu
 */
int main()
{
    int choice;
   GT gt;
    while (1)
    {
```

```
        cout<<"-----------------"<<endl;
        cout<<"Book Tree Creation"<<endl;
        cout<<"-----------------"<<endl;
        cout<<"1.Create"<<endl;
        cout<<"2.Display"<<endl;
        cout<<"3.Quit"<<endl;
        cout<<"Enter your choice : ";
        cin>>choice;
        switch(choice)
        {
        case 1:
             gt.create_tree();
        case 2:
             gt.display(root);
           break;
        case 3:
           exit(1);
        default:
           cout<<"Wrong choice"<<endl;
        }
    }
}
```

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

Program 1:

```
/*************************************************************************

Reference:--https://www.educative.io/edpresso/how-to-find-the-height-of-a-binary-tree

*************************************************************************/


#include <iostream>


using namespace std;


/*
Experiment No. 2 : 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.
*/
struct node
{    int data;
  node *left,*right;
};


class tree
 {public:
        node *root,*temp;
        int  height1(node *T);//recursive counterpart of height()
        int  print0(node *T);//recursive counterpart of count_leaf_nodes()
        node * mirror1(node *T);//recursive counterpart of mirror()
```

```
        tree() { root=NULL;}
        void create();
        void insert(node *,node *);
int height(){return(height1(root));}
        int print_leaf_nodes(){return(print0(root));}
        void level_wise();//level wise traversal
void preorder(node *);
void min(node *);
int count(node *);
void search(node *,int);
 };

class Q
{
        node *data[30];
        int R,F;
        public:
                Q(){ R=F=-1; }
                void init()
                  {
                        R=F=-1;
                  }

   int empty()
     {
                if(R==-1)
                    return 1;
                return 0;
     }
```

```
    void insert(node *p)
     {
                    if(empty())
                            R=F=0;
                    else
                            R=R+1;
                    data[R]=p;
     }

    node *Delete()
     {
          node *p=data[F];
          if(R==F)
                    R=F=-1;
          else
                    F=F+1;
          return(p);
     }
};

int  tree::height1(node *T)
{
          if(T==NULL)
                    return(0);
          if(T->left==NULL && T->right==NULL)
                    return(0);
          return(max(height1(T->left),height1(T->right))+1);
}
```

```
int  tree::count(node *T)
 {
          if(T==NULL)
                    return(0);
          if(T->left==NULL && T->right==NULL)
                    return(1);
          return(max(count(T->left),count(T->right))+1);
 }

int tree::print0(node *T)
 {
          if(T==NULL)
                    return(0);
          if(T->left==NULL && T->right==NULL)
            {
                    cout<<"  "<<T->data;
                    return(1);
            }
          return(print0(T->left)+print0(T->right));
 }
void tree::create()
{
root=NULL;
   char ch;
   do{
   temp=new node;
   cout<<" enter data";
   cin>>temp->data;
   temp->left=NULL;
```

```
    temp->right=NULL;
    if(root==NULL)
    root=temp;
    else
    {
       insert(root,temp);
    }
    cout<<"do u want to continue";
    cin>>ch;
    }while(ch=='y');
}
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::mirror1(node *T)
   {
          node *temp;
          if(T==NULL)
                    return NULL;
          else
            {
                    temp=T->left;
                    T->left=mirror1(T->right);
                    T->right=mirror1(temp);
                    return T;
            }
    }

void tree::level_wise()
  {
          Q q1,q2;
          node *p1,*p2;
          node *T=root;

          if(T==NULL)
                    return;
          q1.insert(T);
          cout<<"\n   "<<T->data;
          while(!q1.empty())
```

```
        { /*Replace all nodes of the queue 'q1' with the nodes at the
           next level.Store nodes of next level in 'q2' */
                cout<<"\n";
                q2.init();
                while(!q1.empty())
                  {
                        p1=q1.Delete();
                        if(p1->left !=NULL)
                          {
                                q2.insert(p1->left);
                                cout<<"  "<<p1->left->data;
                          }
                        if(p1->right !=NULL)
                          {
                                q2.insert(p1->right);
                                cout<<"  "<<p1->right->data;
                          }
                  }
                q1=q2;
        }
}
void tree::preorder(node *root)
{
   if(root!=NULL)
   {
      cout<<root->data;
      preorder(root->left);
       preorder(root->right);
   }
```

```
 }
void tree::min(node *root)
{
   while(root->left!=NULL)
   root=root->left;
   cout<<root->data;
}
void tree::search(node * root,int x)
{
   int flag=0;
   while(root!=NULL)
   {
      if(x<root->data)
      {
         root=root->left;
      }
      else if(x>root->data)
      {
         root=root->right;
      }
      else if(x==root->data)
      {
      flag=1;
      break;
      }
   }
   if(flag==1)
   cout<<"data found";
   else
```

```
   cout<<"not found";
}

int main()
{
        tree  t1;
        int xx,op,x,c;
        do
         {
                cout<<"\n\n1)Create\n2)Mirror\n3)Print leaf nodes";
                cout<<"\n4)Height\n5)preorder display\n 6.minimum value\n 7 count\n8.Search";
                cout <<"\nEnter Your Choice :";
                cin>>op;
                switch(op)
                  {
                        case 1:  t1.create();break;
                        case 2:  cout<<"\n level Wise traversal on original tree \n";
                                 t1.level_wise();
                                 t1.root=t1.mirror1(t1.root);
                                 cout<<"\n level Wise traversal on mirror tree \n";
                                 t1.level_wise();
                                 break;
                        case 3:   xx=t1.print_leaf_nodes();
                                 cout<<"\nNo of leaf nodes= "<<xx;break;
                        case 4:  cout<<"\nHeight = "<<t1.height();break;

                    case 5:
                        t1.preorder(t1.root);
                        break;
```

```
                case 6: t1.min(t1.root);
                        break;
                case 7: c=t1.count(t1.root);
                          cout<<"no of leaf nodes"<<c;
                          break;
                case 8:
                      cout<<"enter element to search";
                          cin>>x;
                          t1.search(t1.root,x);
                          break;
                }
        }while(op!=9);
        return 0;
}
```

5) Construct an expression tree from the given prefix expression eg. +--a*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree.

Program 1:

```
#include <iostream>
using namespace std;
#include<string.h>
struct node
        {
        char data;
        node *left;
         node *right;
         };

class tree
{       char prefix[20];
        public: node *top;
                void expression(char []);
                void display(node *);
                 void non_rec_postorder(node *);
                 void del(node *);

};
class stack1
  {
        node *data[30];
        int top;
```

```
        public:
        stack1()
        {
                top=-1;
        }
                int empty()
                  {
                        if(top==-1)
                                return 1;
                        return 0;
                  }
        void push(node *p)
                  {
                        data[++top]=p;
                  }
        node *pop()
          {
                  return(data[top--]);
          }
};
 void tree::expression(char prefix[])
{char c;
stack1 s;
node *t1,*t2;
int len,i;
 len=strlen(prefix);

        for(i=len-1;i>=0;i--)
        {top=new node;
```

```
                top->left=NULL;
                top->right=NULL;
            if(isalpha(prefix[i]))
            {
                        top->data=prefix[i];
                s.push(top);

            }
            else if(prefix[i]=='+'||prefix[i]=='*'||prefix[i]=='-'||prefix[i]=='/')
            {
            t2=s.pop();

            t1=s.pop();
                top->data=prefix[i];
                top->left=t2;
                top->right=t1;
                s.push(top);

        }

                        }
          top=s.pop();

}

void tree::display(node * root)
{
         if(root!=NULL)
         {              cout<<root->data;
```

```
                display(root->left);
                display(root->right);
        }


}
 void tree::non_rec_postorder(node *top)
  {     stack1 s1,s2;    /*stack s1 is being used for flag . A NULL data
                          implies that the right subtree has not been visited */
        node *T=top;
        cout<<"\n";
        s1.push(T);
while(!s1.empty())
{
T=s1.pop();
s2.push(T);
if(T->left!=NULL)
s1.push(T->left);
if(T->right!=NULL)
s1.push(T->right);
  }
while(!s2.empty())
{
top=s2.pop();
cout<<top->data;
}}

  void tree::del(node* node)
{
```

```
   if (node == NULL) return;
     /* first delete both subtrees */
   del(node->left);
   del(node->right);
      /* then delete the node */
   cout<<" Deleting node:"<<node->data;
   free(node);
}

 int main()
 {
            char expr[20];
            tree t;



            cout<<"Enter prefix  Expression: ";
            cin>>expr;
            cout<<expr;



            t.expression(expr);
//t.display(t.top);
//cout<<endl;
 t.non_rec_postorder(t.top);
// t.del(t.top);
// t.display(t.top);

 }
```

GROUP C

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

Program 1:

Adjacency Matrix:

```
//using adj matrix -BFS(Que)
#include<iostream>
#include<stdlib.h>
using namespace std;
int cost[10][10],i,j,k,n,qu[10],front,rear,v,visit[10],visited[10];
int stk[10],top,visit1[10],visited1[10];
main()
{
int m;
cout <<"enterno of vertices";
cin >> n;
cout <<"enter no of edges";
cin >> m;
cout <<"\nEDGES \n";
for(k=1;k<=m;k++)
{
cin >>i>>j;
cost[i][j]=1;
cost[j][i]=1;
}
```

```
//display function
cout<<"The adjacency matrix of the graph is:"<<endl;
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
cout<<" "<<cost[i][j];
}
cout<<endl;
}
cout <<"Enter initial vertex";
cin >>v;
cout <<"The BFS of the Graph is\n";
cout << v;
visited[v]=1;
k=1;
while(k<n)
{
for(j=1;j<=n;j++)
if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
{
visit[j]=1;
qu[rear++]=j;
}
v=qu[front++];
cout<<v << " ";
k++;
visit[v]=0; visited[v]=1;
}
```

```
cout <<"Enter initial vertex";
cin >>v;
cout <<"The DFS of the Graph is\n";
cout << v;
visited[v]=1;
k=1;
while(k<n)
{
for(j=n;j>=1;j--)
if(cost[v][j]!=0 && visited1[j]!=1 && visit1[j]!=1)
{
visit1[j]=1;
stk[top]=j;
top++;
}
v=stk[--top];
cout<<v << " ";
k++;
visit1[v]=0; visited1[v]=1;
}
}
```

(Adjecency List)

```
#include<iostream>
using namespace std;
#define MAX 10
#define TRUE 1
#define FALSE 0
// declaring an adjacency list for storing the graph
class lgra
```

```
{
private:
struct node1
{
int vertex;
struct node1 *next;
};
node1 *head[MAX];
int visited[MAX];
public:
//static int nodecount;
lgra();
void create();
void dfs(int);
};
//constructor
lgra::lgra()
{
int v1;
for(v1=0;v1<MAX;v1++)
visited[v1]=FALSE;
for(v1=0;v1<MAX;v1++)
head[v1]=NULL;
}
void lgra::create()
{
int v1,v2;
char ans;
node1 *N,*first;
```

```
cout<<"Enter the vertices no. beginning with 0";
do
{
cout<<"\nEnter the Edge of a graph\n";
cin>>v1>>v2;
if(v1>=MAX || v2>=MAX)
cout<<"Invalid Vertex Value\n";
else
{
//creating link from v1 to v2
N = new node1;
if (N==NULL)
cout<<"Insufficient Memory\n";
N->vertex=v2;
N->next=NULL;
first=head[v1];
if (first==NULL)
head[v1]=N;
else
{ while(first->next!=NULL)
first=first->next;
first->next=N;
}
//creating link from v2 to v1
N=new node1;
if (N==NULL)
cout<<"Insufficient Memory\n";
N->vertex=v1;
N->next=NULL;
```

```
first=head[v2];
if (first==NULL)
head[v2]=N;
else
{
while(first->next!=NULL)
first=first->next;
first->next=N;
}
}
cout<<"\n Want to add more edges?(y/n)";
cin>>ans;
}while(ans=='y');
}
//dfs function
void lgra::dfs(int v1)
{
node1 *first;
cout<<endl<<v1;
visited[v1]=TRUE;
first=head[v1];
while(first!=NULL)
if (visited[first->vertex]==FALSE)
dfs(first->vertex);
else
first=first->next;
}
int main()
{
```

```
int v1;
lgra g;
g.create();
cout<<endl<<"Enter the vertex from where you want to traverse:";
cin>>v1;
if(v1>=MAX)
cout<<"Invalid Vertex\n";
else
{
cout<<"The Dfs of the graph:";
g.dfs(v1);
}
}
```

7) There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.

Program 1:

```
#include <iostream>
#include <queue>
using namespace std;
int adj_mat[50][50] = {0,0};
int visited[50] = {0};
void dfs(int s, int n, string arr[])
{
        visited[s] = 1;
        cout<<arr[s]<<" ";
        for(int i=0; i<n; i++)
        {
                if(adj_mat[s][i] && !visited[i])
                        dfs(i,n,arr);
        }
}
void bfs(int s, int n, string arr[])
{
        bool visited[n];
        for(int i=0; i<n; i++)
         visited[i] = false;
        int v;
    queue<int> bfsq;
    if(!visited[s])
    {
         cout<<arr[s]<<" ";
        bfsq.push(s);
        visited[s] = true;
        while(!bfsq.empty())
        {
                v = bfsq.front();
                for(int i=0; i<n; i++)
```

```
                {
                        if(adj_mat[v][i] && !visited[i])
                        {
                                cout<<arr[i]<<" ";
                                visited[i] = true;
                                bfsq.push(i);
                        }
                }
                bfsq.pop();
        }
    }
}
int main
()
{
        cout<<"Enter no. of cities: ";
        int n, u;
        cin>>n;
        string cities[n];
        for(int i=0; i<n; i++)
        {
                cout<<"Enter city #"<<i<<" (Airport Code): ";
                cin>>cities[i];
        }
        cout<<"\nYour cities are: "<<endl;
        for(int i=0; i<n; i++)
                cout<<"city #"<<i<<": "<<cities[i]<<endl;
        for(int i=0; i<n; i++)
        {
                for(int j=i+1; j<n; j++)
                {
                        cout<<"Enter distance between "<<cities[i]<<" and
"<<cities[j]<<": ";
                        cin>>adj_mat[i][j];
                        adj_mat[j][i] = adj_mat[i][j];
                }
        }
        cout<<endl;
        for(int i=0; i<n; i++)
                cout<<"\t"<<cities[i]<<"\t";
        for(int i=0; i<n; i++)
        {
                cout<<"\n"<<cities[i];
```

```
                for(int j=0; j<n; j++)
                        cout<<"\t"<<adj_mat[i][j]<<"\t";
                cout<<endl;
        }
        cout<<"Enter Starting Vertex: ";
        cin>>u;
        cout<<"DFS: "; dfs(u,n,cities);
        cout<<endl;
        cout<<"BFS: "; bfs(u,n,cities);
        return 0;
}
```

GROUP D

8) 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?

Program 1:

```
/* 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----------
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

----------*/

9) A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

Program 1:

```
#include<iostream>
#include<string>
using namespace std;
class dictionary;
class avlnode
{
        string keyword;
        string meaning;
        avlnode *left,*right;
        int bf;
        public:
        avlnode()
        {
                keyword='\0';
                meaning='\0';
                left=right=NULL;
                bf=0;
        }
        avlnode(string k,string m)
```

```
        {
                keyword=k;
                meaning=m;
                left=right=NULL;
                bf=0;
        }
friend class dictionary;
};

class dictionary
{
        avlnode *par,*loc;
        public:
        avlnode *root;
        dictionary()
        {
                root=NULL;
                par=loc=NULL;
        }
        void accept();
        void insert(string key,string mean);
        void LLrotation(avlnode*,avlnode*);

        void RRrotation(avlnode*,avlnode*);
```

```
        void inorder(avlnode *root);
        void deletekey(string key);
        void descending(avlnode *);
        void search(string);
        void update(string,string);

};
void dictionary::descending(avlnode *root)
{
        if(root)
        {
                descending(root->right);
                cout<<root->keyword<<" "<<root->meaning<<endl;
                descending(root->left);
        }
}

void dictionary::accept()
{
        string key,mean;
        cout<<"Enter keyword "<<endl;
        cin>>key;
        cout<<"Enter meaning "<<endl;
        cin>>mean;
        insert(key,mean);
```

```
}
void dictionary::LLrotation(avlnode *a,avlnode *b)
{
        cout<<"LL rotation"<<endl;
        a->left=b->right;
        b->right=a;
        a->bf=b->bf=0;
}

void dictionary::RRrotation(avlnode *a,avlnode *b)
{
        cout<<"RR rotation"<<endl;
        a->right=b->left;
        b->left=a;
        a->bf=b->bf=0;
}

void dictionary::insert(string key,string mean)
{
```

```
        //cout<<"IN Insert \n";
        if(!root)
        {
                //create new root
                root=new avlnode(key,mean);
                cout<<"ROOT CREATED \n";
                return;
        }
//      else
//      {
                avlnode *a,*pa,*p,*pp;
                //a=NULL;
                pa=NULL;
                p=a=root;
                pp=NULL;

                while(p)
                {
                        cout<<"In first while \n";
                        if(p->bf)
                        {
                        a=p;
                        pa=pp;
                        }
                        if(key<p->keyword){pp=p;p=p->left;}   //takes the left branch
```

```
        else if(key>p->keyword){pp=p;p=p->right;} //right branch
        else
        {
                //p->meaning=mean;
                cout<<"Already exist \n";
                return;
        }
}
cout<<"Outside while \n";
avlnode *y=new avlnode(key,mean);
if(key<pp->keyword)
{
        pp->left=y;
}
else
        pp->right=y;
cout<<"KEY INSERTED \n";

int d;
avlnode *b,*c;
//a=pp;
b=c=NULL;
if(key>a->keyword)
{
        cout<<"KEY >A->KEYWORD \n";
```

```
        b=p=a->right;
        d=-1;
        cout<<" RIGHT HEAVY \n";
}
else
{
        cout<<"KEY < A->KEYWORD \n";
        b=p=a->left;
        d=1;
        cout<<" LEFT HEAVY \n";
}

while(p!=y)
{
        if(key>p->keyword)
        {
                p->bf=-1;
                p=p->right;

        }
        else
        {
                p->bf=1;
                p=p->left;
        }
```

```
}
cout<<" DONE ADJUSTING INTERMEDIATE NODES \n";
if(!(a->bf)||!(a->bf+d))
{
        a->bf+=d;
        return;
}
//else
//{
if(d==1)
{
        //left heavy
        if(b->bf==1)
        {
                LLrotation(a,b);
                /*a->left=b->right;
                b->right=a;
                a->bf=0;
                b->bf=0;*/
        }
        else //if(b->bf==-1)
        {

                cout<<"LR rotation"<<endl;
```

```
c=b->right;
b->right=c->left;
a->left=c->right;
c->left=b;
c->right=a;
switch(c->bf)
{
        case 1:
        {
                a->bf=-1;
                b->bf=0;
                break;
        }
        case -1:
        {
                a->bf=0;
                b->bf=1;
                break;
        }

        case 0:
        {
                a->bf=0;
                b->bf=0;
                break;
```

```
                        }


                }
                c->bf=0;
                b=c;   //b is new root



        }
        //else
        //        cout<<"Balanced \n";

}


if(d==-1)
{
        if(b->bf==-1)
        {
//              cout<<"RR rotation"<<endl;
                /*a->right=b->left;
                b->left=a;
                a->bf=b->bf=0;*/
                RRrotation(a,b);
        }
        else// if(b->bf==1)
        {
```

```
                c=b->left;
//              cout<<"RL rotation"<<endl;
                a->right=c->left;
                b->left=c->right;
                c->left=a;
                c->right=b;
                switch(c->bf)
                {
                        case 1:
                        {
                                a->bf=0;
                                b->bf=-1;
                                break;
                        }
                        case -1:
                        {
                                a->bf=1;
                                b->bf=0;
                                break;
                        }

                        case 0:
                        {
                                a->bf=0;
                                b->bf=0;
```

```
                                        break;
                                }


                        }
                        c->bf=0;
                        b=c;  //b is new root


                }


                //else
                        //cout<<"Balanced \n";
        }
        //}
        if(!pa)
                root=b;
        else if(a==pa->left)
                pa->left=b;
        else
                pa->right=b;
        cout<<"AVL tree created!! \n";
        //cout<<"AVL \n";
        //inorder(root);
```

```
}
void dictionary::search(string key)
{
	cout<<"ENTER SEARCH \n";
	loc=NULL;
	par=NULL;
	if(root==NULL)
	{
		cout<<"Tree not created  "<<endl;
		//	root=key;
		loc=NULL;
		par=NULL;
	}

	//par=NULL;loc=NULL;
	avlnode *ptr;
	ptr=root;
	while(ptr!=NULL)
	{
		if(ptr->keyword==key)
		{

			//flag=1;
			loc=ptr;
			break;				//imp for delete1 else it doesnt exit while loop
```

```
                }
                else if(key<ptr->keyword)
                {
                        par=ptr;
                        ptr=ptr->left;
                }

                else
                {
                        par=ptr;                        //edit this in previous code
                        ptr=ptr->right;

                }
        }

        if(loc==NULL)
        {
                cout<<"Not found "<<endl;
        }

}

void dictionary::update(string oldkey,string newmean)
{
        search(oldkey);
```

```
        loc->meaning=newmean;
        cout<<"UPDATE SUCCESSFUL \n";
}
void dictionary::deletekey(string key)
{

}
void dictionary::inorder(avlnode *root)
{
        if(root)
        {
                inorder(root->left);
                cout<<root->keyword<<" "<<root->meaning<<endl;
                inorder(root->right);
        }
}

int main()
{
        string k,m;
        dictionary d;
        int ch;
        string key,mean;

        do
```

```
{
cout<<"1.Insert \n2.Update \n3.Ascending \n4.Descending \n5.Display \n6.Quit \n";
cin>>ch;
switch(ch)
{
        case 1:
        {
                d.accept();
                break;
        }
        case 2:
        {
                cout<<"Enter key whose meaning to update \n";
                cin>>key;
                cout<<"Enter new meaning\n";
                cin>>mean;
                d.update(key,mean);
                break;
        }
        case 3:
                d.inorder(d.root);
                break;

        case 4:
                cout<<"Descending \n";
```

```
                d.descending(d.root);
                break;

        case 5:
                d.inorder(d.root);
                break;
        default:
                break;
    }
    }while(ch!=6); /*cout<<"Enter word and its meaning"<<endl;
        cin>>k>>m;
        d.insert(k,m);*/
    //  d.accept();
        //cout<<"Enter another word and its meaning \n";
    //  cin>>k>>m;
    //  d.insert(k,m);


        //cout<<"MAIN \n";



return 0;
}
```

GROUP E

10) Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure. Analyze the algorithm.

Program 1:

```
#include<iostream>
using namespace std;
class hp
{
  int heap[20],heap1[20],x,n1,i;
  public:
  hp()
  { heap[0]=0;  heap1[0]=0;
  }
  void getdata();
  void insert1(int heap[],int);
  void upadjust1(int heap[],int);
  void insert2(int heap1[],int);
  void upadjust2(int heap1[],int);
  void minmax();
};
void hp::getdata()
{
  cout<<"\n enter the no. of students";
  cin>>n1;
```

```
  cout<<"\n enter the marks";
  for(i=0;i<n1;i++)
  {   cin>>x;
      insert1(heap,x);
      insert2(heap1,x);
  }
}
void hp::insert1(int heap[20],int x)
{
  int n;
  n=heap[0];
  heap[n+1]=x;
  heap[0]=n+1;
  upadjust1(heap,n+1);
}
void hp::upadjust1(int heap[20],int i)
{
   int temp;
   while(i>1&&heap[i]>heap[i/2])
   {
      temp=heap[i];
      heap[i]=heap[i/2];
      heap[i/2]=temp;
      i=i/2;
   }
}
void hp::insert2(int heap1[20],int x)
```

```
{
  int n;
  n=heap1[0];
  heap1[n+1]=x;
  heap1[0]=n+1;

  upadjust2(heap1,n+1);
}
void hp::upadjust2(int heap1[20],int i)
{
   int temp1;
   while(i>1&&heap1[i]<heap1[i/2])
   {
     temp1=heap1[i];
     heap1[i]=heap1[i/2];
     heap1[i/2]=temp1;
     i=i/2;
   }
}
void hp::minmax()
{
  cout<<"\n max marks"<<heap[1];

  cout<<"\n min marks"<<heap1[1];

}
int main()
```

```
{
  hp h;
  h.getdata();
  h.minmax();
  return 0;
}
```

……

//Output

……

```
enter the no. of students10
enter the marks1
2
3
4
5
6
7
8
9
10
11
max marks11
min marks1
```

GROUP F

11) 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.

Program 1:

```
#include<iostream>
#include<fstream>
#include<cstring>
using namespace std;
class tel
 {

 public:
 int rollNo,roll1;
 char name[10];
 char div;
 char address[20];
 void accept()
 {
  cout<<"\n\tEnter Roll Number : ";
```

```
 cin>>rollNo;
 cout<<"\n\tEnter the Name : ";
 cin>>name;
 cout<<"\n\tEnter the Division:";
 cin>>div;
 cout<<"\n\tEnter the Address:";
 cin>>address;
}
      void accept2()
      {
            cout<<"\n\tEnter the Roll No. to modify : ";
            cin>>rollNo;
      }
      void accept3()
      {
           cout<<"\n\tEnter the name to modify : ";
           cin>>name;
      }
      int getRollNo()
```

```
        {
         return rollNo;
        }
  void show()
  {


cout<<"\n\t"<<rollNo<<"\t\t"<<name<<"\t\t"<<div<<"\t\
t"<<address;
   }
};
int main()
{
 int
i,n,ch,ch1,rec,start,count,add,n1,add2,start2,n2,y,a,b,on,
oname,add3,start3,n3,y1,add4,start4,n4;
 char name[20],name2[20];
 tel t1;
 count=0;
 fstream g,f;
```

```
 do
 {

cout<<"\n>>>>>>>>>>>>>>>>>>>>>>MENU<<<<<<<<<<<<<<<<<<<<<";
  cout<<"\n1.Insert and overwrite\n2.Show\n3.Search & Edit(number)\n4.Search & Edit(name)\n5.Search & Edit(onlynumber)\n6.Search & edit(only name)\n 7.Delete a Student Record\n 8.Exit\n\tEnter the Choice\t:";
  cin>>ch;
  switch(ch)
  {
  case 1:
   f.open("StuRecord.txt",ios::out);
   x:t1.accept();
   f.write((char*) &t1,(sizeof(t1)));
   cout<<"\nDo you want to enter more records?\n1.Yes\n2.No";
   cin>>ch1;
```

```
    if(ch1==1)
     goto x;
    else
    {
     f.close();
     break;
    }

  case 2:
   f.open("StuRecord.txt",ios::in);
   f.read((char*) &t1,(sizeof(t1)));
   //cout<<"\n\tRoll No.\t\tName \t\t Division \t\t
Address";
   while(f)
   {
    t1.show();
    f.read((char*) &t1,(sizeof(t1)));
   }
   f.close();
```

```
 break;
case 3:
 cout<<"\nEnter the roll number you want to find";
 cin>>rec;
 f.open("StuRecord.txt",ios::in|ios::out);
 f.read((char*)&t1,(sizeof(t1)));
 while(f)
 {
  if(rec==t1.rollNo)
  {
   cout<<"\nRecord found";
   add=f.tellg();
   f.seekg(0,ios::beg);
        start=f.tellg();
   n1=(add-start)/(sizeof(t1));
   f.seekp((n1-1)*sizeof(t1),ios::beg);
   t1.accept();
   f.write((char*) &t1,(sizeof(t1)));
   f.close();
```

```
  count++;
  break;
 }
 f.read((char*)&t1,(sizeof(t1)));
    }
if(count==0)
      cout<<"\nRecord not found";
f.close();
break;

case 4:
 cout<<"\nEnter the name you want to find and edit";
 cin>>name;
f.open("StuRecord.txt",ios::in|ios::out);
f.read((char*)&t1,(sizeof(t1)));
while(f)
{
 y=(strcmp(name,t1.name));
 if(y==0)
```

```
    {
     cout<<"\nName found";
     add2=f.tellg();
     f.seekg(0,ios::beg);
     start2=f.tellg();
     n2=(add2-start2)/(sizeof(t1));
     f.seekp((n2-1)*sizeof(t1),ios::beg);
     t1.accept();
     f.write((char*) &t1,(sizeof(t1)));
     f.close();
     break;
    }
         f.read((char*)&t1,(sizeof(t1)));
   }
   break;
     case 5:
          cout<<"\n\tEnter the roll number you want to
modify";
          cin>>on;
```

```
f.open("StuRecord.txt",ios::in|ios::out);
f.read((char*) &t1,(sizeof(t1)));
while(f)
{
  if(on==t1.rollNo)
  {
    cout<<"\n\tNumber found";
    add3=f.tellg();
    f.seekg(0,ios::beg);
    start3=f.tellg();
    n3=(add3-start3)/(sizeof(t1));
    f.seekp((n3-1)*(sizeof(t1)),ios::beg);
    t1.accept2();
    f.write((char*)&t1,(sizeof(t1)));
    f.close();
    break;
  }
  f.read((char*)&t1,(sizeof(t1)));
}
```

```
        break;
     case 6:
         cout<<"\nEnter the name you want to find and
edit";
    cin>>name2;
   f.open("StuRecord.txt",ios::in|ios::out);
   f.read((char*)&t1,(sizeof(t1)));
   while(f)
   {
    y1=(strcmp(name2,t1.name));
    if(y1==0)
    {
     cout<<"\nName found";
     add4=f.tellg();
     f.seekg(0,ios::beg);
     start4=f.tellg();
     n4=(add4-start4)/(sizeof(t1));
     f.seekp((n4-1)*sizeof(t1),ios::beg);
     t1.accept3();
```

```
    f.write((char*) &t1,(sizeof(t1)));
    f.close();
    break;
   }
        f.read((char*)&t1,(sizeof(t1)));
  }
  break;
   case 7:
     int roll;
     cout<<"Please Enter the Roll No. of Student Whose
Info You Want to Delete: ";
    cin>>roll;
    f.open("StuRecord.txt",ios::in);
    g.open("temp.txt",ios::out);
    f.read((char *)&t1,sizeof(t1));
    while(!f.eof())
    {
       if (t1.getRollNo() != roll)
          g.write((char *)&t1,sizeof(t1));
```

```
        f.read((char *)&t1,sizeof(t1));
    }
   cout << "The record with the roll no. " << roll << " has
been deleted " << endl;
    f.close();
    g.close();
    remove("StuRecord.txt");
    rename("temp.txt","StuRecord.txt");
     break;
   case 8:
      cout<<"\n\tThank you";
      break;



       }
  }while(ch!=8);
}
```

12) Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.

Program 1:

```
max = 20


# Structure of Employee
class employee:
        def __init__(self):
                self.name = ''
                self.code = 0
                self.designation = ''
                self.exp = 0
                self.age = 0


num = 0
emp = [employee() for i in range(max)]
tempemp = [employee() for i in range(max)]
sortemp = [employee() for i in range(max)]
sortemp1 = [employee() for i in range(max)]
```

```
# Function to build the given datatype
def build():
    global num, emp

    print("Build The Table")
    print("Maximum Entries can be", max)

    num = int(input("Enter the number of Entries required: "))

    if num > max:
        print("Maximum number of Entries are 20")
        num = 20

    print("Enter the following data:")
    for i in range(num):
        emp[i].name = input("Name: ")
        emp[i].code = int(input("Employee ID: "))
        emp[i].designation = input("Designation: ")
        emp[i].exp = int(input("Experience: "))
        emp[i].age = int(input("Age: "))
```

```
    showMenu()


# Function to insert the data into
# given data type
def insert():
    global num, emp

    if num < max:
        i = num
        num += 1

        print("Enter the information of the Employee:")
        emp[i].name = input("Name: ")
        emp[i].code = int(input("Employee ID: "))
        emp[i].designation = input("Designation: ")
        emp[i].exp = int(input("Experience: "))
        emp[i].age = int(input("Age: "))
    else:
        print("Employee Table Full")
```

```
    showMenu()

# Function to delete record at index i
def deleteIndex(i):
    global num, emp

    for j in range(i, num - 1):
        emp[j].name = emp[j + 1].name
        emp[j].code = emp[j + 1].code
        emp[j].designation = emp[j + 1].designation
        emp[j].exp = emp[j + 1].exp
        emp[j].age = emp[j + 1].age

# Function to delete record
def deleteRecord():
    global num, emp

    code = int(input("Enter the Employee ID to Delete Record: "))
```

```
    for i in range(num):
        if emp[i].code == code:
            deleteIndex(i)
            num -= 1
            break

    showMenu()

def searchRecord():
    global num, emp

    code = int(input("Enter the Employee ID to Search Record: "))

    for i in range(num):
        # If the data is found
        if emp[i].code == code:
            print("Name:", emp[i].name)
            print("Employee ID:", emp[i].code)
            print("Designation:", emp[i].designation)
            print("Experience:", emp[i].exp)
```

```
                print("Age:", emp[i].age)
                break

    showMenu()

# Function to show menu
def showMenu():
    print("------------------------GeeksforGeeks Employee Management System------------------------\n")
    print("Available Options:\n")
    print("Build Table              (1)")
    print("Insert New Entry (2)")
    print("Delete Entry      (3)")
    print("Search a Record         (4)")
    print("Exit                     (5)")

    # Input Options
    option = int(input())

    # Call
    if option == 1:
```

```
            build()
        elif option == 2 :
            insert()
        elif option == 3 :
            deleteRecord()
        elif option == 4 :
            searchRecord()
        elif option == 5 :
            return
        else:
            print("Expected Options")
            print("are 1/2/3/4/5")
            showMenu()
# Driver code
showMenu()
```