**Data Structures Lab**

**Labsheet 1**

**Linear search**

#include<stdio.h>

int main()

{

    int i,key=10;

    int a[] = { 2, 3, 4, 10, 40 };

    int size = sizeof(a) / sizeof(a[0]);

    for(i=0;i<size;++i)

        if(a[i]==key)

            break;

    if(i<size)

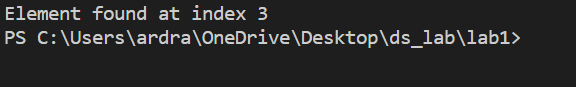
        printf("Element found at index %d",i);

    else

        printf("Element not found");

    return 0;

}



**Binary search**

#include <stdio.h>

int main()

{

int  low, high, mid;

    int i,key=10;

    int array[] = { 2, 3, 4, 10, 40 };

    int n = sizeof(array) / sizeof(array[0]);

low = 0;

high = n - 1;

mid = (low+high)/2;

while (low <= high) {

if(array[mid] < key){

low = mid + 1;

mid = (low+high)/2;

}

else if (array[mid] == key) {

printf("%d found at index %d", key, mid);

break;

}

else{

high = mid - 1;

mid = (low + high)/2;

}

}

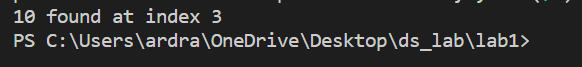
if(low > high){

printf("Not found! %d isn't present in the list", key);

}

return 0;

}



**Stack**

#include<stdio.h>

#define stackCapacity 5

int stack[stackCapacity], top=-1;

void push(int);

int pop(void);

int isFull(void);

int isEmpty(void);

void traverse(void);

void atTop(void);

void main(void)

{

    int choice, stackItem;

    while(1){

        printf("\n Stack Operations");

        printf("\n 1.  Push");

        printf("\n 2.  Pop");

        printf("\n 3.  Return SackTop");

        printf("\n 4.  Traverse");

        printf("\n 5.  Quit");

        printf("\n Enter your choice : ");

        scanf("%d",&choice);

        switch(choice){

            case 1:

                    printf("Enter a integer value to push it on to stack : ");

                    scanf("%d",&stackItem);

                    push(stackItem);

                    break;

            case 2:

                    stackItem = pop();

                    if(stackItem == 0){

                        printf("Your stack is underflow");

                    }else{

                        printf("Last popped item : %dn", stackItem);

                    }

                    break;

            case 3:

                    atTop();

                    break;

            case 4:

                    traverse();

                    break;

            case 5:

                    return;

                    break;

            default: printf("Please enter correct choice : ");

        }

    }

}

void push(int stackElement)

{

    if(isFull()){

        printf("Stack is full.It can't be overflowed.");

    }else{

        top++;

        stack[top] = stackElement;

        printf("%d has been pushed", stackElement);

    }

}

int isFull(){

    if(top == stackCapacity-1){

        return 1;

    }else{

        return 0;

    }

}

int isEmpty(){

    if(top == -1){

        return 1;

    }else{

        return 0;

    }

}

int pop(){

    if(isEmpty()){

        return 0;

    }else{

        return stack[top--];

    }

}

void atTop()

{

    if(isEmpty())

    {

        printf("Your Stack is empty");

    }else{

        printf("Element at top is : %d", stack[top]);

    }

}

void traverse(){

    if(isEmpty())

    {

        printf("Stack is empty");

    }else{

        int i;

        printf("\n Stack Elements are : ");

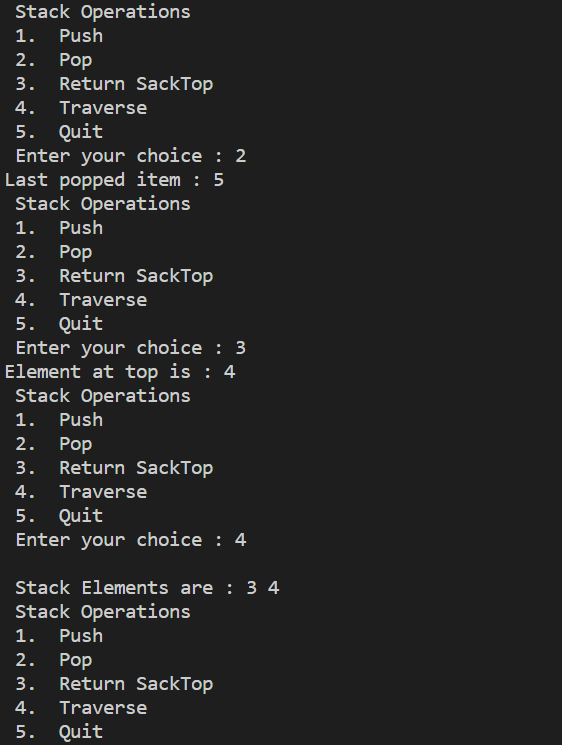
        for(i=0; i <= top; i++){

            printf("%d ", stack[i]);

        }

    }

}



**Queue**

#include <stdio.h>

# define SIZE 100

void enqueue();

void dequeue();

void show();

int arr[SIZE];

int Rear = - 1;

int Front = - 1;

main()

{

    int ch;

    while (1)

    {

        printf("1.Enqueue\n");

        printf("2.Dequeue\n");

        printf("3.Display\n");

        printf("4.Exit\n");

        printf("Enter your choice of operations : ");

        scanf("%d", &ch);

        switch (ch)

        {

            case 1:

            enqueue();

            break;

            case 2:

            dequeue();

            break;

            case 3:

            show();

            break;

            case 4:

            return;

            break;

            default: printf("Please enter correct choice : ");

        }

    }

}

void enqueue()

{

    int insert\_item;

    if (Rear == SIZE - 1)

       printf("Overflow \n");

    else

    {

        if (Front == - 1)

        Front = 0;

        printf("Element to be inserted in the Queue\n : ");

        scanf("%d", &insert\_item);

        Rear = Rear + 1;

        arr[Rear] = insert\_item;

    }

}

void dequeue()

{

    if (Front == - 1 || Front > Rear)

    {

        printf("Underflow \n");

        return ;

    }

    else

    {

        printf("Element deleted from the Queue: %d\n", arr[Front]);

        Front = Front + 1;

    }

}

void show()

{

    if (Front == - 1)

        printf("Empty Queue \n");

    else

    {

        printf("Queue: \n");

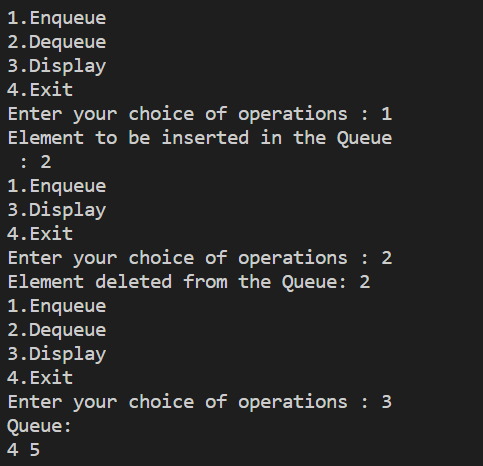
        for (int i = Front; i <= Rear; i++)

            printf("%d ", arr[i]);

        printf("\n");

    }

}

****

**Given base n, print numbers in ascending Order**

def conversion(n,j):

    if j==0:

        list.append(0)

        return list

    else:

        for b in range (j):

            if j==0:

                break

            else:

                a =int(j%n)

                list.append(a)

                j = int(j/n)

        return list

final = []

n = int(input("the base value"))

k = int(input("Enter the value"))

for i in range(0,k):

    list =[]

    result = conversion(n,i)

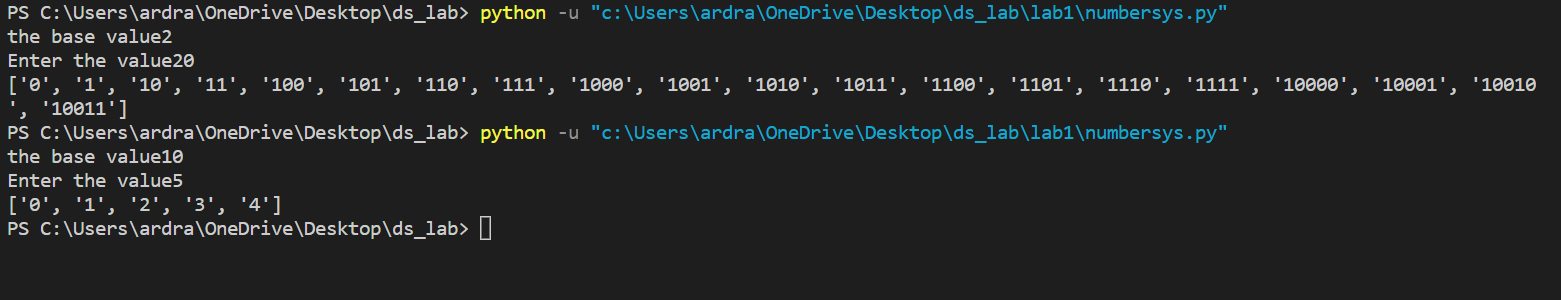
    testlist = list.copy()

    testlist.reverse()

    newlist =''.join((str(p) for p in testlist))

    final.append(newlist)

print(final)

****

**Labsheet 2**

**Insertion Sort**

def insertionsort(a):

    n=len(a)

    for i in range(1,n):

        val=a[i]

        pos=i

        while pos>0 and a[pos-1]>val:

            a[pos]=a[pos-1]

            pos=pos-1

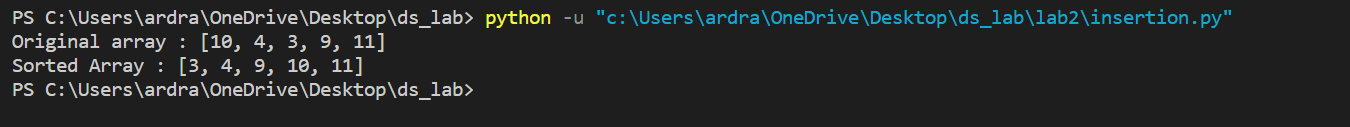
            a[pos]=val

a = [10,4,3,9,11]

print('Original array :', a)

insertionsort(a)

print('Sorted Array :',a)

****

**Quick Sort**

def partition(array, low, high):

  pivot = array[high]

  i = low - 1

  for j in range(low, high):

    if array[j] <= pivot:

      i = i + 1

      (array[i], array[j]) = (array[j], array[i])

  (array[i + 1], array[high]) = (array[high], array[i + 1])

  return i + 1

def quickSort(array, low, high):

  if low < high:

    pi = partition(array, low, high)

    quickSort(array, low, pi - 1)

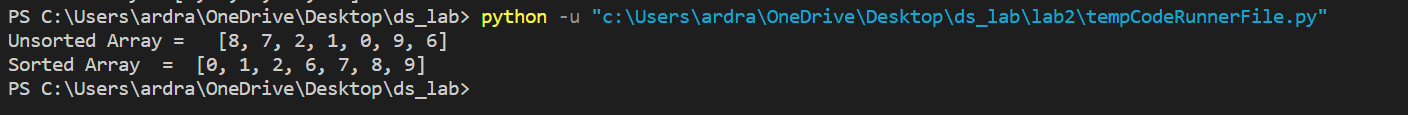
    quickSort(array, pi + 1, high)

data = [8, 7, 2, 1, 0, 9, 6]

print("Unsorted Array =  " , data)

quickSort(data, 0, len(data) - 1)

print('Sorted Array  = ', data)

****

**Labsheet 3**

**Binary Tree**

# Binary Tree in Python

class Node:

    def \_\_init\_\_(self, key):

        self.left = None

        self.right = None

        self.val = key

    # Traverse preorder

    def traversePreOrder(self):

        print(self.val, end=' ')

        if self.left:

            self.left.traversePreOrder()

        if self.right:

            self.right.traversePreOrder()

    # Traverse inorder

    def traverseInOrder(self):

        if self.left:

            self.left.traverseInOrder()

        print(self.val, end=' ')

        if self.right:

            self.right.traverseInOrder()

    # Traverse postorder

    def traversePostOrder(self):

        if self.left:

            self.left.traversePostOrder()

        if self.right:

            self.right.traversePostOrder()

        print(self.val, end=' ')

root = Node(1)

root.left = Node(2)

root.right = Node(3)

root.left.left = Node(4)

print("Pre order Traversal: ", end="")

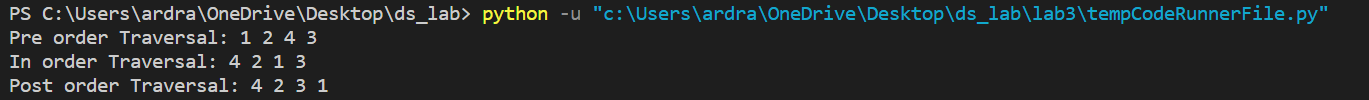
root.traversePreOrder()

print("\nIn order Traversal: ", end="")

root.traverseInOrder()

print("\nPost order Traversal: ", end="")

root.traversePostOrder()

****

**Binary Search Tree**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

def inorder(root):

    if root is not None:

        inorder(root.left)

        print(str(root.key) + "->", end=' ')

        inorder(root.right)

def insert(node, key):

    if node is None

        return Node(key)

    if key < node.key:

        node.left = insert(node.left, key)

    else:

        node.right = insert(node.right, key)

    return node

def minValueNode(node):

    current = node

    while(current.left is not None):

        current = current.left

    return current

def deleteNode(root, key):

    if root is None:

        return root

    if key < root.key:

        root.left = deleteNode(root.left, key)

    elif(key > root.key):

        root.right = deleteNode(root.right, key)

    else:

        if root.left is None:

            temp = root.right

            root = None

            return temp

        elif root.right is None:

            temp = root.left

            root = None

            return temp

        temp = minValueNode(root.right)

        root.key = temp.key

        root.right = deleteNode(root.right, temp.key)

    return root

root = None

root = insert(root, 8)

root = insert(root, 3)

root = insert(root, 1)

root = insert(root, 6)

root = insert(root, 7)

root = insert(root, 10)

root = insert(root, 14)

root = insert(root, 4)

print("Inorder traversal: ", end=' ')

inorder(root)

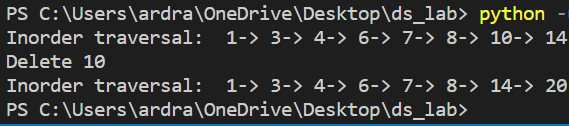
print("\nDelete 10")

root = deleteNode(root, 10)

root = insert(root, 20)

print("Inorder traversal: ", end=' ')

inorder(root)

****

**Labsheet 4**

**Graph**

def add\_node(v):

    global node\_count

    if v in nodes :

        print(v,"is already in graph")

    else:

        node\_count = node\_count + 1

        nodes.append(v)

        for n in graph:

            n.append(0)

        temp = []

        for i in range(node\_count):

            temp.append(0)

        graph.append(temp)

def add\_edge(v1,v2,cost):

    if v1 not in nodes:

        print(v1,"is not present in graph")

    elif v2 not in nodes:

        print(v2,"is not present in graph")

    else:

        index1 = nodes.index(v1)         #all vertices of graph is store in node list

        index2 = nodes.index(v2)

        graph[index1][index2] = cost

        #graph[index2][index1] = cost     #undirected,weighted graph

def delete\_node(v):

    global node\_count

    if v not in nodes:

        print(v,"is not present in graph")

    else:

        index1 = nodes.index(v)    #store node which want to deleted

        node\_count = node\_count-1  #decrement node count

        nodes.remove(v)

        graph.pop(index1)

        for i in graph :

            i.pop(index1)

def delete\_edge(v1,v2):

    if v1 not in nodes:

        print(v1,"is not present in graph")

    elif v2 not in nodes:

        print(v2,"is not present in graph")

    else:

        index1 = nodes.index(v1)

        index2 = nodes.index(v2)

        graph[index1][index2] = 0

        graph[index2][index1] = 0

#as matrix

def print\_graph():

    for i in range(node\_count):

        for j in range(node\_count):

            print(format(graph[i][j],"<3"),end="")          #format function - to print adj matrix on shape

        print()

nodes = []   #store all nodes

graph = []   #store adj matrix

node\_count = 0

# print("Before adding nodes")

# print(nodes)

# print(graph)

add\_node("A")

add\_node("B")

add\_node("D")

add\_edge("A","B",7)

add\_edge("A","D",5)

delete\_node("D")

# print\_graph()

delete\_edge("A","B")

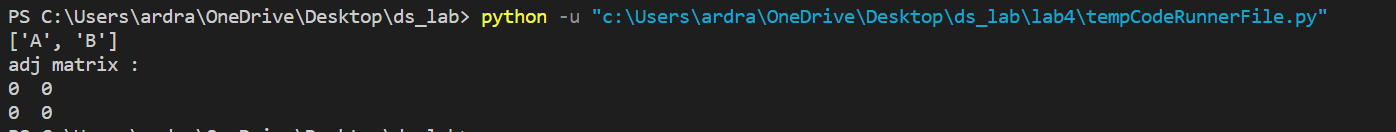
# print("After adding nodes")

print(nodes)

#print(graph)

print("adj matrix :")

print\_graph()

****