LAB MANUAL

Advanced Data Structures & Algorithms Lab

(20A05301P)

R20 Regulation

II B Tech I Sem (CSE)

DEPARTMENT COMPUTER SCIENCE & ENGINEERING

ANANTHA LAKSHMI INSTITUTE OF TECHNOLOGY & SCIENCES

Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to JNTUA, Ananthapuramu

Near S.K. University, Itikalapalli (V), Anantapuramu (Dist)-515721

Andhra Pradesh

DEPARTMENT VISION AND MISSION

Department Vision	To produce technically competent computer science professionals with high quality education in cutting edge technologies and professional ethics.
Department Mission	**M1:** Impart quality technical education in design and implementation of IT applications through innovative teaching - learning practice. **M2:** Provide state-of-art computing infrastructure to enable practical learning experience that foster problem solving and technical communication skills. **M3:** Provide quality learning experiences through experiential learning forstudents and faculty to carry out multidisciplinary research projects with innovative ideas and professional ethics for sustainable development.

COURSE OUTCOMES

CO1	Student can able to **Analyse** and implement different operations on advanced data structures.
CO2	Student can able to **Develop** programs using greedy, divide and conquer approaches for the given problem.
CO3	Student can able to **Develop** programs using dynamic programming and backtracking algorithms for the real world applications.

PROGRAM OUTCOMES

PO 1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO 2	**Problem Analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO 3	**Design/Development of Solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO 4	**Conduct Investigations of Complex Problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5	**Modern Tool Usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO 6	**The Engineer and Society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7	**Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO 8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO 9	**Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO 10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO 11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO 12	**Life-long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

PSO 1	Apply the knowledge of principles of programming languages, data structures, computer networks and software engineering in modeling of computer-based application, complex data bases and network protocols.
PSO 2	Identify processes, modules, algorithms and apply standard software engineering principles to design and develop efficient web based computational applications under realistic constraints.
PSO 3	Apply theoretical principles of core and advanced computer science to solve engineering problems

PROGRAM EDUCATIONAL OBJECTIVES

PEO 1	Demonstrate proficiency in fundamental concepts and advanced technologies of computer science to succeed in their careers and/or obtain a higher degree.
PEO 2	Analyze complex computing problems in multidisciplinary area and creatively solve them with analytical decision making and programming skills
PEO 3	Recognize ethical dilemmas in work environment and apply professional code of ethics to excel as successful software professional, researcher and entrepreneur.

INDEX

.

WEEK-1

1 a .Write a Program to implementation of a binary search tree:
i.Insertion ii.Deletion iii.Display

Program:

```
class Node:

	# Constructor to create a new node
	def __init__(self, key):
		self.key = key
		self.left = None
		self.right = None

# A utility function to do inorder traversal of BST
def inorder(root):
	if root is not None:
		inorder(root.left)
		print(root.key)
		inorder(root.right)

# A utility function to insert a
# new node with given key in BST
def insert(node, key):

	# If the tree is empty, return a new node
	if node is None:
		return Node(key)

	# Otherwise recur down the tree
	if key < node.key:
		node.left = insert(node.left, key)
	else:
		node.right = insert(node.right, key)

	# return the (unchanged) node pointer
	return node

def minValueNode(node):
	current = node
	# loop down to find the leftmost leaf
	while(current.left is not None):
		current = current.left
	return current
```

```
def deleteNode(root, key):

    # Base Case
    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:

        # Node with only one child or no child
        if root.left is None:
            temp = root.right
            root = None
            return temp

        elif root.right is None:
            temp = root.left
            root = None
            return temp

        # Node with two children:
        # Get the inorder successor
        # (smallest in the right subtree)
        temp = minValueNode(root.right)

        # Copy the inorder successor's
        # content to this node
        root.key = temp.key

        # Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)

    return root

# Driver code
""" Let us create following BST
            50
        /    \
        30   70
        / \ / \
    20 40 60 80 """
```

```
root = None
list=[50,30,20,40,70,60,80]
for i in list:
    root = insert(root, i)

print("Inserted Values")
inorder(root)

print ("\tDelete 20")
root = deleteNode(root, 20)
print ("Inorder traversal of the modified tree")
inorder(root)

print ("\tDelete 30")
root = deleteNode(root, 30)
print ("Inorder traversal of the modified tree")
inorder(root)

print ("\tDelete 50")
root = deleteNode(root, 50)
print ("Inorder traversal of the modified tree")
inorder(root)
```

Output:

```
Inserted Values
20
30
40
50
60
70
80
        Delete 20
Inorder traversal of the modified tree
30
40
50
60
70
80
        Delete 30
Inorder traversal of the modified tree
40
50
60
70
80
```

```
	Delete 50
Inorder traversal of the modified tree
40
60
70
80
```

1 b. Write a program to implement search operation on Binary Search Tree

```
class Node:

	# Constructor to create a new node
	def __init__(self, key):
		self.key = key
		self.left = None
		self.right = None

# A utility function to do inorder traversal of BST
def inorder(root):
	if root is not None:
		inorder(root.left)
		print(root.key,end=" ")
		inorder(root.right)

# A utility function to insert a
# new node with given key in BST
def insert(node, key):

	# If the tree is empty, return a new node
	if node is None:
		return Node(key)

	# Otherwise recur down the tree
	if key < node.key:
		node.left = insert(node.left, key)
	else:
		node.right = insert(node.right, key)

	# return the (unchanged) node pointer
	return node
def search(root, value):
   if root.key == value:     #check if value is equal to the key val
     print("\nThe node is present")
     return
   if value < root.key:
       if root.left:
          search(root.left,value)
```

```
        else:
            print("\nThe node is empty in the tree!")
    else:
        if root.right:
            search(root.right,value)
        else:
            print("\nThe node is empty in the tree!")
root = None
list=[50,30,20,40,70,60,80]
for i in list:
    root = insert(root, i)

print("Inserted Values")
inorder(root)
print("\n\nWhich element u want to search:",end=" ")
x= int(input())
search(root, x)
```

Output:

```
Inserted Values
20 30 40 50 60 70 80

Which element u want to search: 40

The node is present
```

WEEK-2

Write a program to perform a binary search for a given set of integer values.

Program:

```
# Iterative Binary Search Function method Python Implementation
# It returns index of n in given list1 if present,
# else returns -1
def binary_search(list1, n):
    low = 0
    high = len(list1) - 1
    mid = 0
    while low <= high:
        # for get integer result
        mid = (high + low) // 2
         # Check if n is present at mid
        if list1[mid] < n:
            low = mid + 1
         # If n is greater, compare to the right of mid
        elif list1[mid] > n:
            high = mid - 1
         # If n is smaller, compared to the left of mid
        else:
            return mid
              # element was not present in the list, return -1
    return -1
    # Initial list1
list1 = [12, 24, 32, 39, 45, 50, 54]
print("\n\nWhich element u want to search:",end=" ")
n= int(input())
  # Function call
result = binary_search(list1, n)
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in list")
```

OUTPUT:

```
Which element u want to search: 32
Element is present at index 2
```

WEEK 3

1. Write a program to implement Splay trees.

Aim:

Program:

```
class TreeNode:
  def __init__(self, data):
    self.data = data
    self.parent = None
    self.left = None
    self.right = None
class SplayTree:
  def __init__(self):
    self.root = None

  def leftRotate(self, x):
    y = x.right
    x.right = y.left
    if y.left != None:
      y.left.parent = x
    y.parent = x.parent
    # x is root
    if x.parent == None:
      self.root = y
    # x is left child
    elif x == x.parent.left:
      x.parent.left = y
    # x is right child
    else:
      x.parent.right = y
    y.left = x
    x.parent = y

  def rightRotate(self, x):
    y = x.left
    x.left = y.right
    if y.right != None:
      y.right.parent = x
    y.parent = x.parent
    # x is root
    if x.parent == None:
      self.root = y
    # x is right child
    elif x == x.parent.right:
      x.parent.right = y
    # x is left child
    else:
      x.parent.left = y
    y.right = x
```

```
    x.parent = y

def splay(self, n):
    # node is not root
    while n.parent != None:
        # node is child of root, one rotation
        if n.parent == self.root:
            if n == n.parent.left:
                self.rightRotate(n.parent)
            else:
                self.leftRotate(n.parent)
        else:
            p = n.parent
            g = p.parent  # grandparent

            if n.parent.left == n and p.parent.left == p:  # both are left children
                self.rightRotate(g)
                self.rightRotate(p)

            elif n.parent.right == n and p.parent.right == p:  # both are right children
                self.leftRotate(g)
                self.leftRotate(p)

            elif n.parent.right == n and p.parent.left == p:
                self.leftRotate(p)
                self.rightRotate(g)

            elif n.parent.left == n and p.parent.right == p:
                self.rightRotate(p)
                self.leftRotate(g)

def insert(self, n):
    y = None
    temp = self.root
    while temp != None:
        y = temp
        if n.data < temp.data:
            temp = temp.left
        else:
            temp = temp.right
    n.parent = y
    if y == None:  # newly added node is root
        self.root = n
    elif n.data < y.data:
        y.left = n
    else:
        y.right = n
```

```
        self.splay(n)

    def bstSearch(self, n, x):
        if x == n.data:
            self.splay(n)
            return n
        elif x < n.data:
            return self.bstSearch(n.left, x)
        elif x > n.data:
            return self.bstSearch(n.right, x)
        else:
            return None

    def preOrder(self, n):
        if n != None:
            print(n.data)
            self.preOrder(n.left)
            self.preOrder(n.right)

if __name__ == '__main__':
    tree = SplayTree()
    a = TreeNode(10)
    b = TreeNode(20)
    c = TreeNode(30)
    d = TreeNode(100)
    e = TreeNode(90)
    f = TreeNode(40)
    g = TreeNode(50)
    tree.insert(a)
    tree.insert(b)
    tree.insert(c)
    tree.insert(d)
    tree.insert(e)
    tree.insert(f)
    tree.insert(g)
    tree.bstSearch(tree.root, 90)
    tree.preOrder(tree.root)
```

OUTPUT:

The Splay tree after searching 90 then it becomes root:
90
50
40
30
20
10
100

WEEK 4

1. Write a program to implement Merge sort for the given list of integer values.

AIM:

PROGRAM:

```
def mergeSort(myList):
    if len(myList) > 1:
        mid = len(myList) // 2
        left = myList[:mid]
        right = myList[mid:]

        # Recursive call on each half
        mergeSort(left)
        mergeSort(right)

        # Two iterators for traversing the two halves
        i = 0
        j = 0
        # Iterator for the main list
        k = 0

        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
              # The value from the left half has been used
              myList[k] = left[i]
              # Move the iterator forward
              i += 1
            else:
                myList[k] = right[j]
                j += 1
            # Move to the next slot
            k += 1

        # For all the remaining values
        while i < len(left):
            myList[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            myList[k]=right[j]
            j += 1
            k += 1
myList = [54,26,93,17,77,31,44,55,20]
print("The List before sort:",myList)
mergeSort(myList)
print("The List After Merge sort:",myList)
```

OUTPUT:

The List before sort: [54, 26, 93, 17, 77, 31, 44, 55, 20]

The List After Merge sort: [17, 20, 26, 31, 44, 54, 55, 77, 93]

WEEK 5

1. Write a program to implement Quicksort for the given list of integer values.

```
# Python program for implementation of Quicksort Sort
 # Function to find the partition position
def partition(array, low, high):

   # choose the rightmost element as pivot
   pivot = array[high]

   # pointer for greater element
   i = low - 1

   # traverse through all elements
   # compare each element with pivot
   for j in range(low, high):
      if array[j] <= pivot:

         # If element smaller than pivot is found
         # swap it with the greater element pointed by i
         i = i + 1
          # Swapping element at i with element at j
         (array[i], array[j]) = (array[j], array[i])

   # Swap the pivot element with the greater element specified by i
   (array[i + 1], array[high]) = (array[high], array[i + 1])

   # Return the position from where partition is done
   return i + 1

# function to perform quicksort

def quickSort(array, low, high):
   if low < high:

      # Find pivot element such that
      # element smaller than pivot are on the left
      # element greater than pivot are on the right
      pi = partition(array, low, high)

      # Recursive call on the left of pivot
      quickSort(array, low, pi - 1)

      # Recursive call on the right of pivot
      quickSort(array, pi + 1, high)

 data = [1, 7, 4, 1, 10, 9, -2]
```

```
print("Unsorted List")
print(data)

size = len(data)

quickSort(data, 0, size - 1)
 print('Sorted List in Ascending Order using Quick Sort:')
print(data)
```

OUTPUT:

Unsorted List

[1, 7, 4, 1, 10, 9, -2]

Sorted List in Ascending Order using Quick Sort:

[-2, 1, 1, 4, 7, 9, 10]

WEEK 6

Write a program to find the solution for the knapsack problem using the greedy method.

```
def fractional_knapsack(value, weight, capacity):
  index = list(range(len(value)))
  ratio = [v/w for v, w in zip(value, weight)]
  max_value = 0
  index.sort(key=lambda i: ratio[i], reverse=True)
  for i in index:
    if capacity!=0:
      if weight[i] <= capacity:
        max_value += value[i]
        capacity -= weight[i]
      else:
        max_value += value[i]
        capacity-= weight[i]*(capacity/weight[i])
  return max_value

profit=[10,5,15,7,6,18,3]
weight = [2,3,5,7,1,4,1]
capacity = 15
max_value = fractional_knapsack(profit, weight, capacity)
print('The maximum Profit of items that can be carried:', max_value)
```

Output:

The maximum Profit of items that can be carried: 57

WEEK 7

Write a program to find minimum cost spanning tree using Prim’s algorithm

```
INF = 9999999
# number of vertices in graph
V = 5
# create a 2d array of size 5x5
# for adjacency matrix to represent graph
G = [[0, 9, 75, 0, 0],
    [9, 0, 95, 19, 42],
    [75, 95, 0, 51, 66],
    [0, 19, 51, 0, 31],
    [0, 42, 66, 31, 0]]
selected = [0, 0, 0, 0, 0]
# set number of edge to 0
no_edge = 0
selected[0] = True
print("Edge : Weight\n")
while (no_edge < V - 1):
    minimum = INF
    x = 0
    y = 0
    for i in range(V):
        if selected[i]:
            for j in range(V):
                if ((not selected[j]) and G[i][j]):
                    # not in selected and there is an edge
                    if minimum > G[i][j]:
                        minimum = G[i][j]
                        x = i
                        y = j
    print(str(x) + " - " + str(y) + ": " + str(G[x][y]))
    selected[y] = True
    no_edge += 1
```

OUTPUT:

```
Edge : Weight
0 -1: 9
1-3: 19
3-4: 31
3-2: 51
```

WEEK 8

Write a program to find minimum cost spanning tree using Kruskal's algorithm

```
class Graph:
    def __init__(self, vertex):
        self.V = vertex
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def search(self, parent, i):
        if parent[i] == i:
            return i
        return self.search(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.search(parent, x)
        yroot = self.search(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    def kruskal(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.search(parent, u)
            y = self.search(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
```

```
        print("Edge:",u, v,end =" ")
        print("-",weight)

 g = Graph(5)
g.add_edge(0, 1, 8)
g.add_edge(0, 2, 5)
g.add_edge(1, 2, 9)
g.add_edge(1, 3, 11)
g.add_edge(2, 3, 15)
g.add_edge(2, 4, 10)
g.add_edge(3, 4, 7)
g.kruskal()
```

OUTPUT:

Edge: 0 2 - 5

Edge: 3 4 - 7

Edge: 0 1 - 8

Edge: 2 4 - 10

WEEK 9

Write a program to find a single source shortest path for a given graph.

```
class Graph():
     def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
     def printSolution(self, dist):
        print("Vertex \t Distance from Source 0 to vertex")
        for node in range(self.V):
            print(node, "\t\t", dist[node])
    def minDistance(self, dist, sptSet):
        min = 1e7
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v
         return min_index

    def dijkstra(self, src):
         dist = [1e7] * self.V
        dist[src] = 0
        sptSet = [False] * self.V
         for cout in range(self.V):
             u = self.minDistance(dist, sptSet)
            sptSet[u] = True
            for v in range(self.V):
                if (self.graph[u][v] > 0 and
                   sptSet[v] == False and
                   dist[v] > dist[u] + self.graph[u][v]):
                    dist[v] = dist[u] + self.graph[u][v]
         self.printSolution(dist)
# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
           [0, 0, 0, 9, 0, 10, 0, 0, 0],
           [0, 0, 4, 14, 10, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 2, 0, 1, 6],
           [8, 11, 0, 0, 0, 0, 1, 0, 7],
           [0, 0, 2, 0, 0, 0, 6, 7, 0]
           ]
 g.dijkstra(0)
```

Output:

Vertex	Distance from Source 0 to vertex
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

WEEK 10

Write a program to find the solution for job sequencing with deadlines problems.

```
class Job:
    def __init__(self, taskId, deadline, profit):
self.taskId = taskId
self.deadline = deadline
self.profit = profit
# Function to schedule jobs to maximize profit
def scheduleJobs(jobs, T):
    profit = 0
    slot = [-1] * T
     jobs.sort(key=lambda x: x.profit, reverse=True)
    for job in jobs:
        for j in reversed(range(job.deadline)):
            if j < T and slot[j] == -1:
                slot[j] = job.taskId
                profit += job.profit
                break
   print('The scheduled jobs are=', list(filter(lambda x: x != -1, slot)))
   print('The total profit earned is=', profit)
jobs = [
        Job(1,9,15),Job(2,2,2),Job(3,5,18),Job(4,7,1),Job(5,4,25),
        Job(6,2,20),Job(7,5,8),Job(8,7,10),Job(9,4,12),Job(10,3,5)
    ]
T = 15
scheduleJobs(jobs, T)
```

OUT PUT:

The scheduled jobs are= [7, 6, 9, 5, 3, 4, 8, 1]
The total profit earned is= 109

WEEK 11

Write a program to find the solution for a 0-1 knapsack problem using dynamic programming.

```
# a dynamic approach

# Returns the maximum value that can be stored by the bag

def knapSack(W, wt, val, n):

  K = [[0 for x in range(W + 1)] for x in range(n + 1)]

  #Table in bottom up manner

  for i in range(n + 1):

    for w in range(W + 1):

      if i == 0 or w == 0:

        K[i][w] = 0

elifwt[i-1] <= w:

        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])

      else:

        K[i][w] = K[i-1][w]

  return K[n][W]

#Main

val = [50,100,150,200]

wt = [8,16,32,40]

W = 64

n = len(val)

print("knapsack is =",knapSack(W, wt, val, n))
```

output:

knapsack is = 350

WEEK 12

Write a program to solve Sum of subsets problem for a given set of distinct numbers using backtracking.

```
def sum_of_subset(s,k,rem):
    x[k]=1
    if s+my_list[k]==target_sum:
        list1=[]
        for i in range (0,k+1):
            if x[i]==1:
                list1.append(my_list[i])
        print( list1 )
    elif s+my_list[k]+my_list[k+1]<=target_sum :
        sum_of_subset(s+my_list[k],k+1,rem-my_list[k])
    if s+rem-my_list[k]>=target_sum and s+my_list[k+1]<=target_sum :
        x[k]=0
        sum_of_subset(s,k+1,rem-my_list[k])

my_list=[]
n=int(input("Enter number of elements: "))
total=0
print("enter elements: ")
for i in range (0,n):
    ele=int(input())
    my_list.append(ele)
    total=total+ele
my_list.sort()
target_sum=int(input("Enter required Sum: "))
x=[0]*(n+1)
sum_of_subset(0,0,total)
```

output:

```
Enter number of elements: 6
enter elements:
5
10
12
13
15
18
Enter required Sum: 30
[5, 10, 15]
[5, 12, 13]
[12, 18]
```

WEEK 13

Python program to solve N Queen Problem using backtracking

```
#Number of queens
print ("Enter the number of queens")
N = int(input())

#chessboard
#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]

def is_attack(i, j):
   #checking if there is a queen in row or column
   for k in range(0,N):
      if board[i][k]==1 or board[k][j]==1:
         return True
   #checking diagonals
   for k in range(0,N):
      for l in range(0,N):
         if (k+l==i+j) or (k-l==i-j):
            if board[k][l]==1:
               return True
   return False

def N_queen(n):
   #if n is 0, solution found
   if n==0:
      return True
   for i in range(0,N):
      for j in range(0,N):
         '''checking if we can place a queen here or not
         queen will not be placed if the place is being attacked
         or already occupied'''
         if (not(is_attack(i,j))) and (board[i][j]!=1):
            board[i][j] = 1
            #recursion
            #wether we can put the next queen with this arrangment or not
            if N_queen(n-1)==True:
               return True
            board[i][j] = 0

   return False

N_queen(N)
```

```
for i in board:
    print (i)
```

Output: The 1 values indicate placements of queens

```
Enter the number of queens

4

[0, 1, 0, 0]

[0, 0, 0, 1]

[1, 0, 0, 0]

[0, 0, 1, 0]
```

ADDITIONAL PROGRAMS

1. Write a Program to implementation of a AVL tree

```
class treeNode(object):
	def __init__(self, value):
		self.value = value
		self.l = None
		self.r = None
		self.h = 1

class AVLTree(object):

	def insert(self, root, key):
		if not root:
			return treeNode(key)
		elif key < root.value:
			root.l = self.insert(root.l, key)
		else:
			root.r = self.insert(root.r, key)

		root.h = 1 + max(self.getHeight(root.l),
						self.getHeight(root.r))

		b = self.getBal(root)

		if b > 1 and key < root.l.value:
			return self.rRotate(root)

		if b < -1 and key > root.r.value:
			return self.lRotate(root)

		if b > 1 and key > root.l.value:
			root.l = self.lRotate(root.l)
			return self.rRotate(root)

		if b < -1 and key < root.r.value:
			root.r = self.rRotate(root.r)
			return self.lRotate(root)

		return root

	def lRotate(self, z):

		y = z.r
		T2 = y.l
```

```
        y.l = z
        z.r = T2

        z.h = 1 + max(self.getHeight(z.l),
                      self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),
                      self.getHeight(y.r))

        return y

    def rRotate(self, z):

        y = z.l
        T3 = y.r

        y.r = z
        z.l = T3

        z.h = 1 + max(self.getHeight(z.l),
                      self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),
                      self.getHeight(y.r))

        return y

    def getHeight(self, root):
        if not root:
            return 0

        return root.h

    def getBal(self, root):
        if not root:
            return 0

        return self.getHeight(root.l) - self.getHeight(root.r)

    def preOrder(self, root):

        if not root:
            return

        print("{0} ".format(root.value), end="")
        self.preOrder(root.l)
        self.preOrder(root.r)
```

```
Tree = AVLTree()
root = None

root = Tree.insert(root, 1)
root = Tree.insert(root, 2)
root = Tree.insert(root, 3)
root = Tree.insert(root, 4)
root = Tree.insert(root, 5)
root = Tree.insert(root, 6)

# Preorder Traversal
print("Preorder traversal of the",
      "constructed AVL tree is")
Tree.preOrder(root)
print()
```

OUTPUT:

```
Preorder traversal of the constructed AVL tree is
4 2 1 3 5 6
```

2. Write a Python program for implementation of heap Sort

```
# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap

 def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1  # left = 2*i + 1
    r = 2 * i + 2  # right = 2*i + 2

 # See if left child of root exists and is
 # greater than root

    if l < n and arr[i] < arr[l]:
        largest = l

 # See if right child of root exists and is
 # greater than root

    if r < n and arr[largest] < arr[r]:
        largest = r

 # Change root, if needed

    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i])  # swap

  # Heapify the root.

        heapify(arr, n, largest)

# The main function to sort an array of given size

def heapSort(arr):
    n = len(arr)

 # Build a maxheap.
 # Since last parent will be at ((n//2)-1) we can start at that location.

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

 # One by one extract elements

    for i in range(n - 1, 0, -1):
```

```
        (arr[i], arr[0]) = (arr[0], arr[i])  # swap
        heapify(arr, i, 0)

# Driver code to test above
 arr = [12, 11, 13, 5, 6, 7, ]
heapSort(arr)
n = len(arr)
print('Sorted array is')
for i in range(n):
    print(arr[i])
```

OUTPUT:

Sorted array is
5
6
7
11
12
13