**Name: Ahmed Mustafa**

**Seat No: EB20102009**

**Class: BSCS SEC A**

**Shift: Evening**

**Teacher Name: Ms. Farzeen Ashfaq**

**Title: Data Structure Lab File**

**Lab No 01**

Implement BST and perform in order, post order, preorder, level order

**Source Code :**

class BST:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.key = key

def insert(root, key):

if root is None:

return BST(key)

else:

if root.key == key:

return root

elif root.key < key:

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

else:

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

return root

def inOrder(root):

if root:

inOrder(root.left)

print(root.key)

inOrder(root.right)

def preOrder(root):

if root:

print(root.key)

preOrder(root.left)

preOrder(root.right)

def postOrder(root):

if root:

postOrder(root.left)

postOrder(root.right)

print(root.key)

root = BST(30)

root = insert(root, 20)

root = insert(root, 10)

root = insert(root, 15)

root = insert(root, 25)

root = insert(root, 23)

root = insert(root, 39)

root = insert(root, 35)

root = insert(root, 42)

print("Start of InOrder Traversal ")

inOrder(root)

print("End of InOrder Traversal \n")

print("Start of PostOrder Traversal")

postOrder(root)

print("End of PostOrder Traversal \n")

print("Start of PreOrder Traversal")

preOrder(root)

print("End of PreOrder Traversal")

**Lab No 02**

Implementation of AVL Tree

**Source Code :**

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)

def inOrder(self, root):

if not root:

return

self.inOrder(root.l)

print("{0} ".format(root.value), end="")

self.inOrder(root.r)

def postOrder(self, root):

if not root:

return

self.postOrder(root.l)

self.postOrder(root.r)

print("{0} ".format(root.value), end="")

Tree = AVLTree()

root = None

root = Tree.insert(root, 10)

root = Tree.insert(root, 20)

root = Tree.insert(root, 15)

root = Tree.insert(root, 25)

root = Tree.insert(root, 30)

root = Tree.insert(root, 16)

root = Tree.insert(root, 18)

root = Tree.insert(root, 19)

print("Preorder traversal of the",

"constructed AVL tree is")

Tree.preOrder(root)

print()

print("inorder traversal of the",

"constructed AVL tree is")

Tree.inOrder(root)

print()

print("postorder traversal of the",

"constructed AVL tree is")

Tree.postOrder(root)

print()

**Lab No 03**

Implement Hash Table, linear probing, quadratic probing, chaining

**Linear Probing:**

**Source Code:**

import numpy as np

class Hash\_table() :

def \_\_init\_\_(self,size) :

self.size = size

self.hashtable = np.array([None]\*self.size)

def hash(self,key) :

# Hash function h(x) = x%10

index = key%10

if self.hashtable[index] == None :

return index

else :

# Implementing linear probing

while self.hashtable[index] != None :

index = (index+1)%10

return index

def insert(self,key) :

index = self.hash(key)

self.hashtable[index] = key

def search(self,key) :

index = key%10

if self.hashtable[index] != key :

while self.hashtable[index] != key and self.hashtable[index] != None :

index = (index+1)%10

if self.hashtable[index] == key :

return index

else :

return None

def print\_hashtable(self) :

print("Hash table is :-\n\nindex \t value")

for x in range(len(self.hashtable)) :

print(x,"\t",self.hashtable[x])

# Initializing hash table of size 10

HT = Hash\_table(10)

# Inserting only 5 values to make λ <= 0.5

HT.insert(50)

HT.insert(30)

HT.insert(16)

HT.insert(11)

HT.insert(33)

HT.print\_hashtable()

index = HT.search(35)

if index!= None :

print("\nGiven key is present at index",index)

else :

print("\nGiven key is not present in the hash table")

**Quadratic Probing:**

**Source Code:**

import numpy as np

class Hash\_table() :

def \_\_init\_\_(self,size) :

self.size = size

self.hashtable = np.array([None]\*self.size)

def hash(self,key) :

# Hash function h(x) = x%10

index = key%10

if self.hashtable[index] == None :

return index

else :

i = 1

# Implementing Quadratic probing

while self.hashtable[(index+(i\*i))%10] != None :

i += 1

return (index+(i\*i))%10

def insert(self,key) :

index = self.hash(key)

self.hashtable[index] = key

def search(self,key) :

index = key%10

i = 0

if self.hashtable[index] != key :

while self.hashtable[(index+(i\*i))%10] != key and self.hashtable[(index+(i\*i))%10] != None :

i += 1

if self.hashtable[(index+(i\*i))%10] == key :

return (index+(i\*i))%10

else :

return None

def print\_hashtable(self) :

print("Hash table is :-\n\nindex \t value")

for x in range(len(self.hashtable)) :

print(x,"\t",self.hashtable[x])

# Initializing hash table of size 10

HT = Hash\_table(10)

# Inserting only 4 values to make λ <= 0.5

HT.insert(7)

HT.insert(90)

HT.insert(11)

HT.insert(44)

HT.print\_hashtable()

index = HT.search(11)

if index!= None :

print("\nGiven key is present at index",index)

else :

print("\nGiven key is not present in the hash table")

**Chaining Probing:**

**Source code:**

# Function to display hashtable

def display\_hash(hashTable):

for i in range(len(hashTable)):

print(i, end = " ")

for j in hashTable[i]:

print("-->", end = " ")

print(j, end = " ")

print()

# Creating Hashtable as

# a nested list.

HashTable = [[] for \_ in range(10)]

# Hashing Function to return

# key for every value.

def Hashing(keyvalue):

return keyvalue % len(HashTable)

# Insert Function to add

# values to the hash table

def insert(Hashtable, keyvalue, value):

hash\_key = Hashing(keyvalue)

Hashtable[hash\_key].append(value)

# Driver Code

insert(HashTable, 10, 'karachi')

insert(HashTable, 25, 'hyderabad')

insert(HashTable, 20, 'lahore')

insert(HashTable, 9, 'multan')

insert(HashTable, 21, 'peshawar')

insert(HashTable, 21, 'multan')

display\_hash (HashTable)

**Lab No 04**

Implement DFS and BFS traversals

**BFS Code:**

**Source Code:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

print("Following is the Breadth-First Search(BFS)")

bfs(visited, graph, '5')

**DFS Code :**

**Source Code:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set()

def dfs(visited, graph, node):

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

print("Following is the Depth-First Search(DFS)")

dfs(visited, graph, '5')