



**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Aim of Practical** | **Practical**  **Date** | **Submission Date** | **Remark** |
| **1** | **Write a program to implement singly linked list with insertion ,deletion , traversal operations.** |  |  |  |
| **2** | **Write a program to implement doubly linked list with insertion ,deletion , traversal operations.** |  |  |  |
| **3** | **Write a program to implement Queue with insertion ,deletion , traversal operations** |  |  |  |
| **4** | **Write a program to implement priority Queue with insertion ,deletion , traversal operations.** |  |  |  |
| **5** | **Write a program to implement stack with insertion ,deletion , traversal operations.** |  |  |  |
| **6** | **Write a program to implement Binary tree with insertion ,deletion , traversal operations.** |  |  |  |
| **7** | **Write a program to implement Huffman coding.** |  |  |  |
| **8** | **Write a program to implement Travelling salesman problem.** |  |  |  |
| **9** | **Write a program to create basic hash table for insertion, deletion,traversal operation(assume that there is no collisions)** |  |  |  |
| **10** | **Write a program to create hash table to handle collision using overflow chaining.** |  |  |  |

**Practical 1**

**Aim: Write a program to implement singly linked list with insertion ,deletion , traversal operations.**

# Linked list operations in Python

# Create a node

class Node:

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

self.data = data

self.next = None

class LinkedList:

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

self.head = None

# Insert at the beginning

def insertAtBeginning(self, new\_data):

new\_node = Node(new\_data)

new\_node.next = self.head

self.head = new\_node

# Insert after a node

def insertAfter(self, prev\_node, new\_data):

if prev\_node is None:

print("The given previous node must inLinkedList.")

return

new\_node = Node(new\_data)

new\_node.next = prev\_node.next

prev\_node.next = new\_node

# Insert at the end

def insertAtEnd(self, new\_data):

new\_node = Node(new\_data)

if self.head is None:

self.head = new\_node

return

last = self.head

while (last.next):

last = last.next

last.next = new\_node

# Deleting a node

def deleteNode(self, position):

if self.head is None:

return

temp = self.head

if position == 0:

self.head = temp.next

temp = None

return

# Find the key to be deleted

for i in range(position - 1):

temp = temp.next

if temp is None:

break

# If the key is not present

if temp is None:

return

if temp.next is None:

return

next = temp.next.next

temp.next = None

temp.next = next

# Search an element

def search(self, key):

current = self.head

while current is not None:

if current.data == key:

return True

current = current.next

return False

# Sort the linked list

def sortLinkedList(self, head):

current = head

index = Node(None)

if head is None:

return

else:

while current is not None:

# index points to the node next to current

index = current.next

while index is not None:

if current.data > index.data:

current.data, index.data = index.data, current.data

index = index.next

current = current.next

# Print the linked list

def printList(self):

temp = self.head

while (temp):

print(str(temp.data) + " ", end="")

temp = temp.next

if \_\_name\_\_ == '\_\_main\_\_':

llist = LinkedList()

llist.insertAtEnd(1)

llist.insertAtBeginning(2)

llist.insertAtBeginning(3)

llist.insertAtEnd(4)

llist.insertAfter(llist.head.next, 5)

print('linked list:')

llist.printList()

print("\nAfter deleting an element:")

llist.deleteNode(3)

llist.printList()

print()

item\_to\_find = 3

if llist.search(item\_to\_find):

print(str(item\_to\_find) + " is found")

else:

print(str(item\_to\_find) + " is not found")

llist.sortLinkedList(llist.head)

print("Sorted List: ")

llist.printList()

output:

linked list:

3 2 5 1 4

After deleting an element:

3 2 5 4

3 is found

Sorted List:

2 3 4 5 l

# Practical 2

**Aim: Write a program to implement doubly linked list with insertion ,deletion , traversal operations.**

class Node:

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

self.data = data;

self.previous = None;

self.next = None;

class DoublyLinkedList:

#Represent the head and tail of the doubly linked list

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

self.head = None;

self.tail = None;

#addNode() will add a node to the list

def addNode(self, data):

newNode = Node(data);

#If list is empty

if(self.head == None):

#Both head and tail will point to newNode

self.head = self.tail = newNode;

#head's previous will point to None

self.head.previous = None;

#tail's next will point to None, as it is the last node of the list

self.tail.next = None;

else:

#newNode will be added after tail such that tail's next will point to newNode

self.tail.next = newNode;

#newNode's previous will point to tail

newNode.previous = self.tail;

#newNode will become new tail

self.tail = newNode;

#As it is last node, tail's next will point to None

self.tail.next = None;

#display() will print out the nodes of the list

def display(self):

#Node current will point to head

current = self.head;

if(self.head == None):

print("List is empty");

return;

print("Nodes of doubly linked list: ");

while(current != None):

#Prints each node by incrementing pointer.

print(current.data);

current = current.next;

dList = DoublyLinkedList();

#Add nodes to the list

dList.addNode(1);

dList.addNode(2);

dList.addNode(3);

dList.addNode(4);

dList.addNode(5);

#Displays the nodes present in the list

dList.display();

output:

Nodes of doubly linked list:

1

2

3

4

5

# Practical 3

**Aim: Write a program to implement Queue with insertion ,deletion , traversal operations.**

# Queue implementation in Python

class Queue:

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

self.queue = []

# Add an element

def enqueue(self, item):

self.queue.append(item)

# Remove an element

def dequeue(self):

if len(self.queue) < 1:

return None

return self.queue.pop(0)

# Display the queue

def display(self):

print(self.queue)

def size(self):

return len(self.queue)

q = Queue()

q.enqueue(1)

q.enqueue(2)

q.enqueue(3)

q.enqueue(4)

q.enqueue(5)

q.display()

q.dequeue()

print("After removing an element")

q.display()

output:

[1, 2, 3, 4, 5]

After removing an element

[2, 3, 4, 5]

# Practical 4

**Aim: Write a program to implement priority Queue with insertion ,deletion , traversal operations.**

# Priority Queue implementation in Python

# Function to heapify the tree

def heapify(arr, n, i):

# Find the largest among root, left child and right child

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

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

largest = l

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

largest = r

# Swap and continue heapifying if root is not largest

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

heapify(arr, n, largest)

# Function to insert an element into the tree

def insert(array, newNum):

size = len(array)

if size == 0:

array.append(newNum)

else:

array.append(newNum)

for i in range((size // 2) - 1, -1, -1):

heapify(array, size, i)

# Function to delete an element from the tree

def deleteNode(array, num):

size = len(array)

i = 0

for i in range(0, size):

if num == array[i]:

break

array[i], array[size - 1] = array[size - 1], array[i]

array.remove(size - 1)

for i in range((len(array) // 2) - 1, -1, -1):

heapify(array, len(array), i)

arr = []

insert(arr, 3)

insert(arr, 4)

insert(arr, 9)

insert(arr, 5)

insert(arr, 2)

print ("Max-Heap array: " + str(arr))

deleteNode(arr, 4)

print("After deleting an element: " + str(arr))

output:

Max-Heap array: [9, 5, 4, 3, 2]

After deleting an element: [9, 5, 2, 3]

# Practical 5

**Aim: Write a program to implement stack with insertion ,deletion , traversal operations.**

# Creating a stack

def create\_stack():

stack = []

return stack

# Creating an empty stack

def check\_empty(stack):

return len(stack) == 0

# Adding items into the stack

def push(stack, item):

stack.append(item)

print("pushed item: " + item)

# Removing an element from the stack

def pop(stack):

if (check\_empty(stack)):

return "stack is empty"

return stack.pop()

stack = create\_stack()

push(stack, str(1))

push(stack, str(2))

push(stack, str(3))

push(stack, str(4))

print("popped item: " + pop(stack))

print("stack after popping an element: " + str(stack))

output:

pushed item: 1

pushed item: 2

pushed item: 3

pushed item: 4

popped item: 4

stack after popping an element: ['1', '2', '3']

# Practical 6

**Aim: Write a program to implement Binary tree with insertion ,deletion , traversal operations.**

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()

**output:**

Pre order Traversal: 1 2 4 3

In order Traversal: 2 1

Post order Traversal: 1

# Practical 7

**Aim: Write a program to implement Huffman coding.**

# Huffman Coding in python

string = 'BCAADDDCCACACAC'

# Creating tree nodes

class NodeTree(object):

def \_\_init\_\_(self, left=None, right=None):

self.left = left

self.right = right

def children(self):

return (self.left, self.right)

def nodes(self):

return (self.left, self.right)

def \_\_str\_\_(self):

return '%s\_%s' % (self.left, self.right)

# Main function implementing huffman coding

def huffman\_code\_tree(node, left=True, binString=''):

if type(node) is str:

return {node: binString}

(l, r) = node.children()

d = dict()

d.update(huffman\_code\_tree(l, True, binString + '0'))

d.update(huffman\_code\_tree(r, False, binString + '1'))

return d

# Calculating frequency

freq = {}

for c in string:

if c in freq:

freq[c] += 1

else:

freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

nodes = freq

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman\_code\_tree(nodes[0][0])

print(' Char | Huffman code ')

print('----------------------')

for (char, frequency) in freq:

print(' %-4r |%12s' % (char, huffmanCode[char]))

**output:**

Char | Huffman code

----------------------

'C' | 0

'A' | 11

'D' | 101

'B' | 100

# Practical 8

**Aim: Write a program to implement Travelling salesman problem.**

# problem using naive approach.

from sys import maxsize

from itertools import permutations

V = 4

# implementation of traveling Salesman Problem

def travellingSalesmanProblem(graph, s):

# store all vertex apart from source vertex

vertex = []

for i in range(V):

if i != s:

vertex.append(i)

# store minimum weight Hamiltonian Cycle

min\_path = maxsize

next\_permutation=permutations(vertex)

for i in next\_permutation:

# store current Path weight(cost)

current\_pathweight = 0

# compute current path weight

k = s

for j in i:

current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][s]

# update minimum

min\_path = min(min\_path, current\_pathweight)

return min\_path

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

# matrix representation of graph

graph = [[0, 10, 15, 20], [10, 0, 35, 25],

[15, 35, 0, 30], [20, 25, 30, 0]]

s = 0

print(travellingSalesmanProblem(graph, s))

**output:**

80

# Practical 9

**Aim: Write a program to create basic hash table for insertion, deletion,traversal operation(assume that there is no collisions)**

hashTable = [[],] \* 10

def checkPrime(n):

if n == 1 or n == 0:

return 0

for i in range(2, n//2):

if n % i == 0:

return 0

return 1

def getPrime(n):

if n % 2 == 0:

n = n + 1

while not checkPrime(n):

n += 2

return n

def hashFunction(key):

capacity = getPrime(10)

return key % capacity

def insertData(key, data):

index = hashFunction(key)

hashTable[index] = [key, data]

def removeData(key):

index = hashFunction(key)

hashTable[index] = 0

insertData(123, "apple")

insertData(432, "mango")

insertData(213, "banana")

insertData(654, "guava")

print(hashTable)

removeData(123)

print(hashTable)

**output:**

[[], [], [123, 'apple'], [432, 'mango'], [213, 'banana'], [654, 'guava'], [], [], [], []] [[], [], 0, [432, 'mango'], [213, 'banana'], [654, 'guava'], [], [], [], []]

# Practical 10

**Aim: Write a program to create hash table to handle collision using overflow chaining.**

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, 'Allahabad') insert(HashTable, 25, 'Mumbai') insert(HashTable, 20, 'Mathura') insert(HashTable, 9, 'Delhi') insert(HashTable, 21, 'Punjab')

insert(HashTable, 21, 'Noida')

display\_hash (HashTable)

**output:**

0 --> Allahabad --> Mathura

1 --> Punjab --> Noida

2

3

4

5 --> Mumbai

6

7

8

9 --> Delhi