1.1 11.12.2023 Binary Search Tree Operations

I. Insert

II. Delete

III. Search

IV. Traversal

Binary Tree Application

1.2 12.12.2023 Application in Binary Search Tree

I. Insert

II. Delete

III. Search

Binary Tree Special Operations

1.3 18.12.2023 Special Operations in BST Application – 1

I. Minimum and maximum value node and its level

II. Find the third minimum and maximum node and its level

III. Find the sum of all numerical value on the left and right subtree separately

IV. Find the distance between any two given nodes in terms of levels

1.4 26.12.2023 Special Operations in BST Application – 2

I. Delete all the leaf nodes

II. Count the total number of nodes

III. Find the height of the tree

IV. Create and display the right threads of leaf node

V. Create and display the left threads of leaf node

AVL Tree

2 08.01.2024 AVL Tree Operations

I. Insertion

II. Deletion

III. Inorder Traversal

IV. Counting total nodes

V. Searching for a key

VI. Checking if the key is empty

Heap Operations

3.1 22.01.2024 Heap Tree Operations

I. Insertion

II. Deletion

III. Display the elements in array format

IV. Print the children of given node

V. Print the index of 1st leaf and last non-leaf nodes

Application of Heap

3.2 24.01.2024 Application in Heap Tree

I. Insertion

II. Deletion

III. Display the elements in array format

AVL

rh

eh

lh

lh-rh = bf

r of l - l-subtree, r-node(right)

60 60 55

/ left / right / \

50 ==== 55 ====== 50 60

\ /

55 50

Binary search tree

class Node:

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

self.key = key

self.left = None

self.right = None

class BinarySearchTree:

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

self.root = None

def insert(self, key):

self.root = self.\_insert(self.root, key)

def \_insert(self, root, key):

if root is None:

return Node(key)

if key < root.key:

root.left = self.\_insert(root.left, key)

elif key > root.key:

root.right = self.\_insert(root.right, key)

return root

def delete(self, key):

self.root = self.\_delete(self.root, key)

def \_delete(self, root, key):

if root is None:

return root

if key < root.key:

root.left = self.\_delete(root.left, key)

elif key > root.key:

root.right = self.\_delete(root.right, key)

else:

if root.left is None:

return root.right

elif root.right is None:

return root.left

root.key = self.\_min\_value\_node(root.right).key

root.right = self.\_delete(root.right, root.key)

return root

def \_min\_value\_node(self, node):

current = node

while current.left is not None:

current = current.left

return current

def search(self, key):

return self.\_search(self.root, key)

def \_search(self, root, key):

if root is None or root.key == key:

return root

if key < root.key:

return self.\_search(root.left, key)

return self.\_search(root.right, key)

def inorder\_traversal(self, root, result):

if root:

self.inorder\_traversal(root.left, result)

result.append(root.key)

self.inorder\_traversal(root.right, result)

bst = BinarySearchTree()

n = int(input("Enter the number of nodes: "))

keys = [int(input("Enter value of node: ")) for \_ in range(n)]

for key in keys:

bst.insert(key)

print("Original BST Inorder Traversal:", bst.inorder\_traversal(bst.root, []))

delete\_key = int(input("Enter key to delete: "))

bst.delete(delete\_key)

print(f"\nBST Inorder Traversal after deleting {delete\_key}:", bst.inorder\_traversal(bst.root, []))

search\_key = int(input("Enter key to search: "))

if bst.search(search\_key):

print(f"\n{search\_key} found in the BST.")

else:

print(f"\n{search\_key} not found in the BST.")

Application of bst

import heapq

from collections import Counter

class Node:

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

self.char = char

self.freq = freq

self.left = left

self.right = right

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def build\_huffman\_tree(text):

freq\_table = Counter(text)

nodes = [Node(char, freq) for char, freq in freq\_table.items()]

heapq.heapify(nodes)

while len(nodes) > 1:

left\_node = heapq.heappop(nodes)

right\_node = heapq.heappop(nodes)

parent\_node = Node(None, left\_node.freq + right\_node.freq, left\_node, right\_node)

heapq.heappush(nodes, parent\_node)

return nodes[0]

def encode(node, code="", codes=None):

if codes is None:

codes = {}

if node is not None:

if node.char is not None:

codes[node.char] = code

encode(node.left, code + "0", codes)

encode(node.right, code + "1", codes)

return codes

def huffman\_coding(text):

root = build\_huffman\_tree(text)

codes = encode(root)

encoded\_data = "".join([codes[c] for c in text])

decoded\_data = ""

current\_code = ""

for bit in encoded\_data:

current\_code += bit

char = next((char for char, code in codes.items() if code == current\_code), None)

if char is not None:

decoded\_data += char

current\_code = ""

return encoded\_data, decoded\_data

text = input("Enter the text to be encoded: ")

encoded\_data, decoded\_data = huffman\_coding(text)

print("Original data:", text)

print("Encoded data:", encoded\_data)

print("Decoded data:", decoded\_data)

spl op 1

class BST:

def \_\_init\_\_(self, key, name=None, quantity=None, price=None):

self.key = key

self.name = name

self.quantity = quantity

self.price = price

self.lchild = None

self.rchild = None

def insert(self, product\_id, name, quantity, price):

if self.key is None:

self.key = product\_id

self.name = name

self.quantity = quantity

self.price = price

elif self.key == product\_id:

# Update data for existing node

self.name = name

self.quantity = quantity

self.price = price

elif self.key > product\_id:

if self.lchild:

self.lchild.insert(product\_id, name, quantity, price)

else:

self.lchild = BST(product\_id, name, quantity, price)

else:

if self.rchild:

self.rchild.insert(product\_id, name, quantity, price)

else:

self.rchild = BST(product\_id, name, quantity, price)

def delete(self, product\_id):

if self.key is None:

print("Tree is empty")

return self

if product\_id < self.key:

if self.lchild:

self.lchild = self.lchild.delete(product\_id)

else:

print("Product ID not present in the tree")

elif product\_id > self.key:

if self.rchild:

self.rchild = self.rchild.delete(product\_id)

else:

print("Product ID not present in the tree")

else:

if self.lchild is None:

return self.rchild

if self.rchild is None:

return self.lchild

# 2 child

inorder\_succ = self.rchild

while inorder\_succ.lchild:

inorder\_succ = inorder\_succ.lchild

self.key = inorder\_succ.key

self.name = inorder\_succ.name

self.quantity = inorder\_succ.quantity

self.price = inorder\_succ.price

self.rchild = self.rchild.delete(inorder\_succ.key)

return self

def search(self, product\_id):

if self.key == product\_id:

print(f"Product found: {self.name} (Quantity: {self.quantity}, Price: {self.price})")

return

if product\_id < self.key:

if self.lchild:

self.lchild.search(product\_id)

else:

print("Product ID not present in the tree")

else:

if self.rchild:

self.rchild.search(product\_id)

else:

print("Product ID not present in the tree")

def inorder\_traversal(self):

result = []

self.\_inorder\_traversal(result)

return result

def \_inorder\_traversal(self, result):

if self.lchild:

self.lchild.\_inorder\_traversal(result)

result.append((self.key, self.name, self.quantity, self.price))

if self.rchild:

self.rchild.\_inorder\_traversal(result)

def find\_min\_node\_and\_level(self):

return self.\_find\_min\_node\_and\_level(self, 1)

def \_find\_min\_node\_and\_level(self, node, level):

if node is None:

return None

# Traverse to the leftmost node to find the minimum value node

while node.lchild:

node = node.lchild

level += 1

return {"node": (node.key, node.name, node.quantity, node.price), "level": level}

def find\_max\_node\_and\_level(self):

return self.\_find\_max\_node\_and\_level(self, 1)

def \_find\_max\_node\_and\_level(self, node, level):

if node is None:

return None

# Traverse to the rightmost node to find the maximum value node

while node.rchild:

node = node.rchild

level += 1

return {"node": (node.key, node.name, node.quantity, node.price), "level": level}

def find\_kth\_min\_node\_and\_level(self, k):

result = {"count": 0, "result": None}

self.\_find\_kth\_min\_node\_and\_level(self, k, result)

return result["result"]

def \_find\_kth\_min\_node\_and\_level(self, node, k, result):

if node is None or result["count"] >= k:

return

# Traverse to the leftmost node

self.\_find\_kth\_min\_node\_and\_level(node.lchild, k, result)

# Increment the count for each visited node

result["count"] += 1

# If the count matches k, store the node and level

if result["count"] == k:

result["result"] = {"node": (node.key, node.name, node.quantity, node.price), "level": result["count"]}

# Continue the traversal to the right

self.\_find\_kth\_min\_node\_and\_level(node.rchild, k, result)

def find\_kth\_max\_node\_and\_level(self, k):

result = {"count": 0, "result": None}

self.\_find\_kth\_max\_node\_and\_level(self, k, result)

return result["result"]

def \_find\_kth\_max\_node\_and\_level(self, node, k, result):

if node is None or result["count"] >= k:

return

# Traverse to the rightmost node

self.\_find\_kth\_max\_node\_and\_level(node.rchild, k, result)

# Increment the count for each visited node

result["count"] += 1

# If the count matches k, store the node and level

if result["count"] == k:

result["result"] = {"node": (node.key, node.name, node.quantity, node.price), "level": result["count"]}

# Continue the traversal to the left

self.\_find\_kth\_max\_node\_and\_level(node.lchild, k, result)

def sum\_quantities\_left\_and\_right\_subtree(self):

left\_sum = self.\_sum\_quantities(self.lchild)

right\_sum = self.\_sum\_quantities(self.rchild)

return {"left\_subtree\_sum": left\_sum, "right\_subtree\_sum": right\_sum}

def \_sum\_quantities(self, node):

if node is None:

return 0

return node.quantity + self.\_sum\_quantities(node.lchild) + self.\_sum\_quantities(node.rchild)

def find\_distance\_between\_nodes(self, node1, node2):

# Find the lowest common ancestor

lca\_result = self.\_find\_lowest\_common\_ancestor(self, node1, node2)

if lca\_result and lca\_result["node"]:

lca = lca\_result["node"]

# Calculate distances from LCA to each node

distance1 = self.\_calculate\_level\_distance(lca, node1, 0)

distance2 = self.\_calculate\_level\_distance(lca, node2, 0)

return abs(distance1 - distance2)

else:

return None

def \_find\_lowest\_common\_ancestor(self, node, node1, node2):

if node is None:

return None

# If either node1 or node2 is the current node's key, return the current node

if node.key == node1 or node.key == node2:

return {"node": node, "level": 1}

# Recursively search in the left and right subtrees

left\_result = self.\_find\_lowest\_common\_ancestor(node.lchild, node1, node2)

right\_result = self.\_find\_lowest\_common\_ancestor(node.rchild, node1, node2)

# If both nodes are found in different subtrees, return the current node as the LCA

if left\_result and right\_result:

return {"node": node, "level": 1}

# If only one node is found, return the result from that subtree

return left\_result or right\_result

def \_calculate\_level\_distance(self, node, node\_key, current\_level):

if node is None:

return float('inf') # A large value indicating infinity

if node.key == node\_key:

return current\_level

# Recursively search in the left and right subtrees

left\_distance = self.\_calculate\_level\_distance(node.lchild, node\_key, current\_level + 1)

right\_distance = self.\_calculate\_level\_distance(node.rchild, node\_key, current\_level + 1)

# Return the minimum of left and right distances

return min(left\_distance, right\_distance)

# Interactive Program

inventory\_manager = BST(None)

while True:

print("\nOptions:")

print("1. Insert Product")

print("2. Delete Product")

print("3. Search Product")

print("4. Inorder Traversal")

print("5. Find Minimum Value Node and Level")

print("6. Find Maximum Value Node and Level")

print("7. Find 3rd Minimum Node and Level")

print("8. Find 3rd Maximum Node and Level")

print("9. Sum of quantity in left and right subtree")

print("10. Find Distance Between Two Nodes")

print("11. Exit")

choice = input("Enter your choice (1-11): ")

if choice == '1':

product\_id = int(input("Enter Product ID: "))

name = input("Enter Product Name: ")

quantity = int(input("Enter Product Quantity: "))

price = float(input("Enter Product Price: "))

inventory\_manager.insert(product\_id, name, quantity, price)

elif choice == '2':

product\_id = int(input("Enter Product ID to delete: "))

inventory\_manager.delete(product\_id)

elif choice == '3':

product\_id = int(input("Enter Product ID to search: "))

inventory\_manager.search(product\_id)

elif choice == '4':

print("\nInorder Traversal:")

products = inventory\_manager.inorder\_traversal()

for product in products:

print(product)

elif choice == '5':

result = inventory\_manager.find\_min\_node\_and\_level()

if result:

print(f"\nMinimum Value Node: {result['node']}")

print(f"Level: {result['level']}")

else:

print("Tree is empty")

elif choice == '6':

result = inventory\_manager.find\_max\_node\_and\_level()

if result:

print(f"\nMaximum Value Node: {result['node']}")

print(f"Level: {result['level']}")

else:

print("Tree is empty")

elif choice == '7':

k = 3 # You can modify k as needed

result = inventory\_manager.find\_kth\_min\_node\_and\_level(k)

if result and result["node"]:

print(f"\n{k}rd Minimum Value Node: {result['node']}")

print(f"Level: {result['level']}")

else:

print(f"There are less than {k} nodes in the tree.")

elif choice == '8':

k = 3 # You can modify k as needed

result = inventory\_manager.find\_kth\_max\_node\_and\_level(k)

if result and result["node"]:

print(f"\n{k}rd Maximum Value Node: {result['node']}")

print(f"Level: {result['level']}")

else:

print(f"There are less than {k} nodes in the tree.")

elif choice == '9':

result = inventory\_manager.sum\_quantities\_left\_and\_right\_subtree()

print("\nSum of Quantity:")

print(f"Left Subtree: {result['left\_subtree\_sum']}")

print(f"Right Subtree: {result['right\_subtree\_sum']}")

elif choice == '10':

node1 = int(input("Enter Product ID 1: "))

node2 = int(input("Enter Product ID 2: "))

distance = inventory\_manager.find\_distance\_between\_nodes(node1, node2)

if distance is not None:

print(f"\nDistance between Product ID {node1} and Product ID {node2}: {distance}")

else:

print("One or both of the product IDs not found in the tree.")

elif choice == '11':

print("Exiting Program.")

break

else:

print("Invalid choice. Please enter a number between 1 and 11.")

bst op 2

class Node:

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

self.data = data

self.left = None

self.right = None

self.rthread = False

self.lthread = False

class BST:

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

self.root = None

def insert(self, data):

self.root = self.ins(self.root, data)

def ins(self, root, data):

if root is None:

return Node(data)

if data < root.data:

root.left = self.ins(root.left, data)

elif data > root.data:

root.right = self.ins(root.right, data)

return root

def inorder(self):

inorder = []

self.inord(self.root, inorder)

return inorder

def inord(self, root, inorder):

if root:

self.inord(root.left, inorder)

inorder.append(root)

self.inord(root.right, inorder)

def deleteLeafNodes(self):

print("The leaf nodes are: ")

self.root = self.delete(self.root)

def delete(self, root):

if root is None:

return None

elif root.left is None and root.right is None:

print(root.data)

return None

root.left = self.delete(root.left)

root.right = self.delete(root.right)

return root

def nodeCount(self):

return self.cnt(self.root)

def cnt(self, root):

if root is None:

return 0

return 1 + self.cnt(root.left) + self.cnt(root.right)

def height(self):

return self.hgt(self.root)

def hgt(self, root):

if root is None:

return 0

return 1 + max(self.hgt(root.left), self.hgt(root.right))

def createRightThread(self):

self.rightThread(self.root)

def rightThread(self, root):

if root is not None:

self.rightThread(root.left)

if root.left is None and root.right is None:

# root.right = self.inorderSuccessor(root)

root.rthread = True

self.rightThread(root.right)

# def inorderSuccessor(self, root):

# temp = root.right

# while temp is not None and temp.left is not None:

# temp = temp.right

# return temp

def displayRightThread(self):

rightThread = [node.data for node in self.inorder() if node.rthread]

print("Right threaded nodes: ", rightThread)

inorderValues = [node.data for node in self.inorder()]

#print(inorderValues)

for i in range(len(rightThread)-1):

current\_index = inorderValues.index(rightThread[i])

next\_index = current\_index + 1

mapped\_data = inorderValues[next\_index] if next\_index < len(inorderValues) else self.root.data

print(f"{rightThread[i]} -> {mapped\_data}")

last\_node\_index = inorderValues.index(rightThread[-1])

if last\_node\_index < len(inorderValues) - 1:

mapped\_data\_last = inorderValues[last\_node\_index + 1]

else:

mapped\_data\_last = self.root.data

print(f"{rightThread[-1]} -> {mapped\_data\_last}")

def createLeftThread(self):

self.leftThread(self.root)

def leftThread(self, root):

if root is not None:

self.leftThread(root.right)

if root.left is None and root.right is None:

root.left = self.inorderPredecessor(root)

root.lthread = True

self.leftThread(root.left)

def inorderPredecessor(self, root):

temp = root.left

while temp is not None and temp.right is not None:

temp = temp.right

return temp

def displayLeftThread(self):

leftThread = [node.data for node in self.inorder() if node.lthread]

print("Left threaded nodes: ", leftThread)

inorder\_values = [node.data for node in self.inorder()]

first\_node\_index = inorder\_values.index(leftThread[0]) - 1

mapped\_data\_first = self.root.data if first\_node\_index < 0 else inorder\_values[first\_node\_index]

print(f"{leftThread[0]} -> {mapped\_data\_first}")

for i in range(1, len(leftThread)):

current\_index = inorder\_values.index(leftThread[i])

prev\_index = current\_index - 1

mapped\_data = self.root.data if prev\_index < 0 else inorder\_values[prev\_index]

print(f"{leftThread[i]} -> {mapped\_data}")

def userInput():

values = []

print("Enter a value to be inserted or enter -1 to finish: ")

while True:

j = input()

if j != '-1':

values.append(int(j))

else:

break

return values

binarySearchTree = BST()

values = userInput()

for i in values:

binarySearchTree.insert(i)

print("Inorder: ", [node.data for node in binarySearchTree.inorder()])

while True:

print("\n1. Delete all leaf nodes\n2. Count the total number of nodes\n3. Find the height of the tree\n4. Create and display the right threads of all leaf nodes\n5. Create and display the left threads of all leaf nodes\n6. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

binarySearchTree.deleteLeafNodes()

print("Inorder after deletion of all leaf nodes: ", [node.data for node in binarySearchTree.inorder()])

elif choice == 2:

count = binarySearchTree.nodeCount()

print("Total number of nodes: ", count)

elif choice == 3:

height = binarySearchTree.height()

print("Height of the tree is: ", height)

elif choice == 4:

binarySearchTree.createRightThread()

binarySearchTree.displayRightThread()

elif choice == 5:

binarySearchTree.createLeftThread()

binarySearchTree.displayLeftThread()

elif choice == 6:

break

avl

class AVLNode:

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

self.key = key

self.height = 1

self.left = None

self.right = None

class AVLTree:

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

self.root = None

def height(self, node):

if not node:

return 0

return node.height

def update\_height(self, node):

if not node:

return 0

node.height = 1 + max(self.height(node.left), self.height(node.right))

def balance\_factor(self, node):

if not node:

return 0

return self.height(node.left) - self.height(node.right)

def left\_rotate(self, z):

y = z.right

if not y:

return z

T2 = y.left

y.left = z

z.right = T2

self.update\_height(z)

self.update\_height(y)

return y

def right\_rotate(self, y):

x = y.left

if not x:

return y

T2 = x.right

x.right = y

y.left = T2

self.update\_height(y)

self.update\_height(x)

return x

def insert(self, root, key):

if not root:

return AVLNode(key)

if key < root.key:

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

else:

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

self.update\_height(root)

balance = self.balance\_factor(root)

if balance > 1:

if key < root.left.key:

# Left-Left case

print(f"Unbalanced at node {root.key}. Performing Right Rotation (Left-Left Case).")

return self.right\_rotate(root)

else:

# Left-Right case

print(f"Unbalanced at node {root.key}. Performing Left Rotation (Left-Right Case) and then Right Rotation.")

root.left = self.left\_rotate(root.left)

return self.right\_rotate(root)

if balance < -1:

if key > root.right.key:

# Right-Right case

print(f"Unbalanced at node {root.key}. Performing Left Rotation (Right-Right Case).")

return self.left\_rotate(root)

else:

# Right-Left case

print(f"Unbalanced at node {root.key}. Performing Right Rotation (Right-Left Case) and then Left Rotation.")

root.right = self.right\_rotate(root.right)

return self.left\_rotate(root)

return root

def delete(self, root, key):

if not root:

print(f"Key {key} not found in the AVL tree.")

return root

if key < root.key:

root.left = self.delete(root.left, key)

elif key > root.key:

root.right = self.delete(root.right, key)

else:

if not root.left:

return root.right

elif not root.right:

return root.left

temp = self.find\_min(root.right)

root.key = temp.key

root.right = self.delete(root.right, temp.key)

self.update\_height(root)

balance = self.balance\_factor(root)

if balance > 1:

if self.balance\_factor(root.left) >= 0:

# Left-Left case

print(f"Unbalanced at node {root.key}. Performing Right Rotation (Left-Left Case).")

return self.right\_rotate(root)

else:

# Left-Right case

print(f"Unbalanced at node {root.key}. Performing Left Rotation (Left-Right Case) and then Right Rotation.")

root.left = self.left\_rotate(root.left)

return self.right\_rotate(root)

if balance < -1:

if self.balance\_factor(root.right) <= 0:

# Right-Right case

print(f"Unbalanced at node {root.key}. Performing Left Rotation (Right-Right Case).")

return self.left\_rotate(root)

else:

# Right-Left case

print(f"Unbalanced at node {root.key}. Performing Right Rotation (Right-Left Case) and then Left Rotation.")

root.right = self.right\_rotate(root.right)

return self.left\_rotate(root)

return root

def find\_min(self, node):

while node.left:

node = node.left

return node

def inorder\_traversal(self, root):

result = []

if root:

result += self.inorder\_traversal(root.left)

result.append(root.key)

result += self.inorder\_traversal(root.right)

return result

def count\_nodes(self, root):

if not root:

return 0

return 1 + self.count\_nodes(root.left) + self.count\_nodes(root.right)

def search(self, root, key):

if not root or root.key == key:

return root

if key < root.key:

return self.search(root.left, key)

return self.search(root.right, key)

def is\_empty(self):

return not self.root

avl\_tree = AVLTree()

while True:

print("\nAVL Tree Operations:")

print("1. Insertion\n2. Deletion\n3. Inorder Traversal\n4. Count Total Nodes\n5. Search for a Key\n6. Check if the Tree is Empty\n7. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

while True:

key = int(input("Enter the key to insert (-1 to stop): "))

if key == -1:

break

avl\_tree.root = avl\_tree.insert(avl\_tree.root, key)

elif choice == 2:

key = int(input("Enter the key to delete: "))

avl\_tree.root = avl\_tree.delete(avl\_tree.root, key)

elif choice == 3:

print("Inorder Traversal:", avl\_tree.inorder\_traversal(avl\_tree.root))

elif choice == 4:

count = avl\_tree.count\_nodes(avl\_tree.root)

print(f"Total number of nodes: {count}")

elif choice == 5:

key = int(input("Enter the key to search: "))

node = avl\_tree.search(avl\_tree.root, key)

if node:

print(f"Key {key} found in the AVL tree.")

else:

print(f"Key {key} not found in the AVL tree.")

elif choice == 6:

if avl\_tree.is\_empty():

print("The AVL tree is empty.")

else:

print("The AVL tree is not empty.")

elif choice == 7:

print("Exiting...")

break

else:

print("Invalid choice. Please enter a valid option.")

heap

class maxHeap:

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

self.heap=[]

def push(self,data):

self.heap.append(data)

self.heapify\_up()

def heapify\_up(self):

index=len(self.heap)-1

while index>0:

parent\_index=(index-1)//2

if self.heap[index]>self.heap[parent\_index]:

self.heap[index],self.heap[parent\_index]=self.heap[parent\_index],self.heap[index]

index=parent\_index

else:

break

def pop(self):

if len(self.heap)==0:

return None

if len(self.heap)==1:

return self.heap.pop()

root=self.heap[0]

self.heap[0]=self.heap.pop()

self.heapify\_down()

return root

def heapify\_down(self):

index=0

while True:

left\_child\_index=2\*index+1

right\_child\_index=2\*index+2

largest=index

if(left\_child\_index<len(self.heap) and self.heap[left\_child\_index]>self.heap[largest]):

largest=left\_child\_index

if(right\_child\_index<len(self.heap) and self.heap[right\_child\_index]>self.heap[largest]):

largest=right\_child\_index

if largest!=index:

self.heap[index],self.heap[largest]=self.heap[largest],self.heap[index]

index=largest

else:

break

def getChildren(self,parent\_index):

left\_child\_index=2\*parent\_index+1

right\_child\_index=2\*parent\_index+2

left\_child=self.heap[left\_child\_index] if left\_child\_index <len(self.heap) else None

right\_child=self.heap[right\_child\_index] if right\_child\_index<len(self.heap) else None

return left\_child,right\_child

def get\_first\_leaf\_index(self):

return len(self.heap) // 2

def get\_last\_non\_leaf\_index(self):

return (len(self.heap) - 2) // 2

heap=maxHeap()

while True:

print("1. Insertion\n2. Deletion\n3. Display the elements in array format\n4. Print the children of given node\n5. Print the index of 1st leaf and last non-leaf nodes\n6. Exit\n")

ch=int(input("Enter your choice: "))

if ch==1:

while True:

key=int(input("Enter the key to insert and -1 to stop: "))

if key==-1:

break

heap.push(key)

#print(heap.heap)

elif ch==2:

heap.pop()

print("Element deleted successfully")

elif ch==3:

print("The elements are: ",heap.heap)

elif ch==4:

node\_val=int(input("Enter the value: "))

node\_index=heap.heap.index(node\_val) if node\_val in heap.heap else None

if node\_index is not None:

left\_child,right\_child=heap.getChildren(node\_index)

print(f"Children of node {node\_val}:")

print(f"Left child: {left\_child}")

print(f"Right child: {right\_child}")

else:

print(f"Node {node\_val} not found in the heap.")

elif ch==5:

first\_leaf\_index = heap.get\_first\_leaf\_index()

last\_non\_leaf\_index = heap.get\_last\_non\_leaf\_index()

print(f"Index of the first leaf node: {first\_leaf\_index}")

print(f"Index of the last non-leaf node: {last\_non\_leaf\_index}")

elif ch==6:

break

else:

print("Invalid choice")

app of heap

import heapq

from collections import defaultdict

class HuffmanNode:

def \_\_init\_\_(self, char, frequency):

self.char = char

self.frequency = frequency

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.frequency < other.frequency

def build\_huffman\_tree(data):

frequency\_map = defaultdict(int)

for char in data:

frequency\_map[char] += 1

priority\_queue = [HuffmanNode(char, freq) for char, freq in frequency\_map.items()]

heapq.heapify(priority\_queue)

while len(priority\_queue) > 1:

node1 = heapq.heappop(priority\_queue)

node2 = heapq.heappop(priority\_queue)

internal\_node = HuffmanNode(None, node1.frequency + node2.frequency)

internal\_node.left = node1

internal\_node.right = node2

heapq.heappush(priority\_queue, internal\_node)

return priority\_queue[0]

def build\_huffman\_codes(node, current\_code="", codes={}):

if node is not None:

if node.char is not None:

codes[node.char] = current\_code

build\_huffman\_codes(node.left, current\_code + "0", codes)

build\_huffman\_codes(node.right, current\_code + "1", codes)

return codes

def huffman\_encoding(data):

if not data:

return None, None

root = build\_huffman\_tree(data)

codes = build\_huffman\_codes(root)

print("Huffman Codes:")

for char, code in codes.items():

print(f"{char}: {code}")

encoded\_data = "".join(codes[char] for char in data)

return encoded\_data, root

def huffman\_decoding(encoded\_data, root):

if not encoded\_data or root is None:

return None

decoded\_data = ""

current\_node = root

for bit in encoded\_data:

if bit == "0":

current\_node = current\_node.left

else:

current\_node = current\_node.right

if current\_node.char is not None:

decoded\_data += current\_node.char

current\_node = root

return decoded\_data

data = input("Enter the word to be encoded: ")

encoded\_data, huffman\_tree = huffman\_encoding(data)

decoded\_data = huffman\_decoding(encoded\_data, huffman\_tree)

print("Original data:", data)

print("Encoded data:", encoded\_data)

print("Decoded data:", decoded\_data)