Lab 1

1.

Pasul similar este ca in amandoua algoritmele gasim elementul minim il extragem si il punem la inceput doar ca la heapsort il luam pe cel mai mare prin maxHeap.

2.

Prima oara se face un maxHeap prin heapify se extrage radacina intr un vector si se apeleaza din nou heapify care face un maxHeap pana cand este sortat vectorul.

Lab 2

1. Stergerea capului unei liste circulare simplu inlantuite

def delete(self, key):

if self.head is None:

return

current = self.head

prev = None

while True:

if current.key == key:

if prev:

prev.next = current.next

else:

if current.next == self.head:

self.head = None

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

temp.next = current.next

self.head = current.next

return

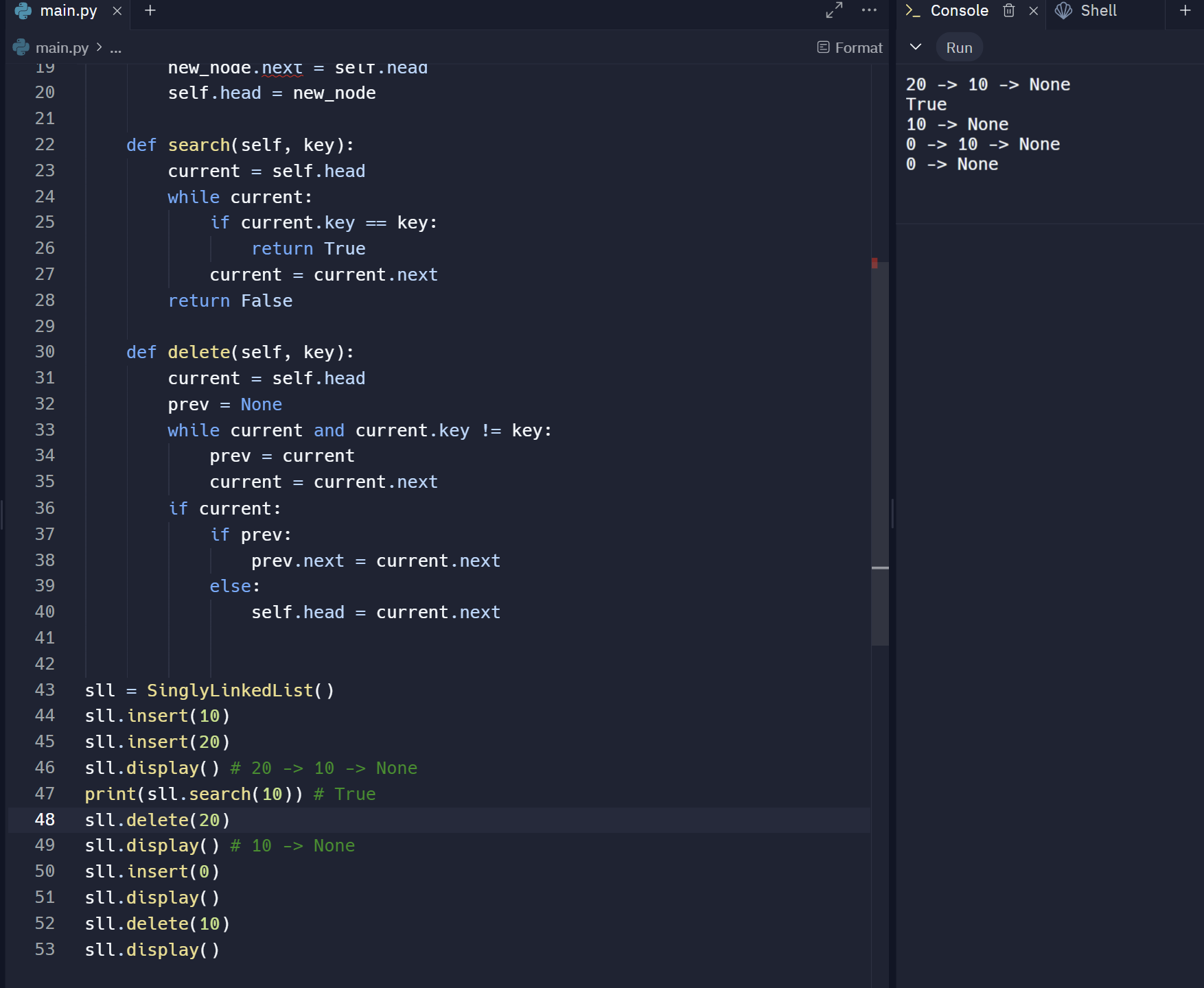
prev = current

current = current.next

if current == self.head:

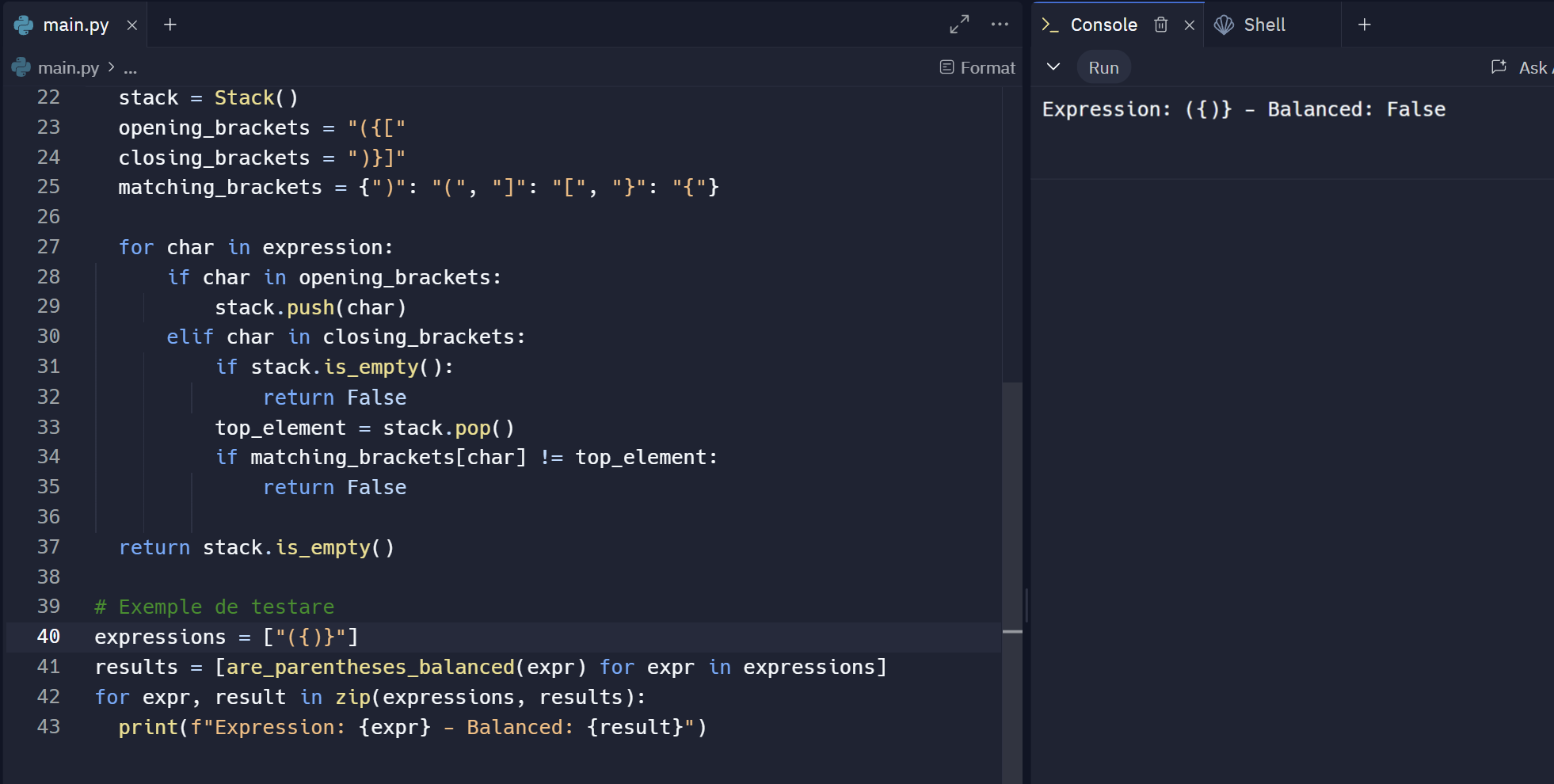
Break

2.



Lab 3

1.



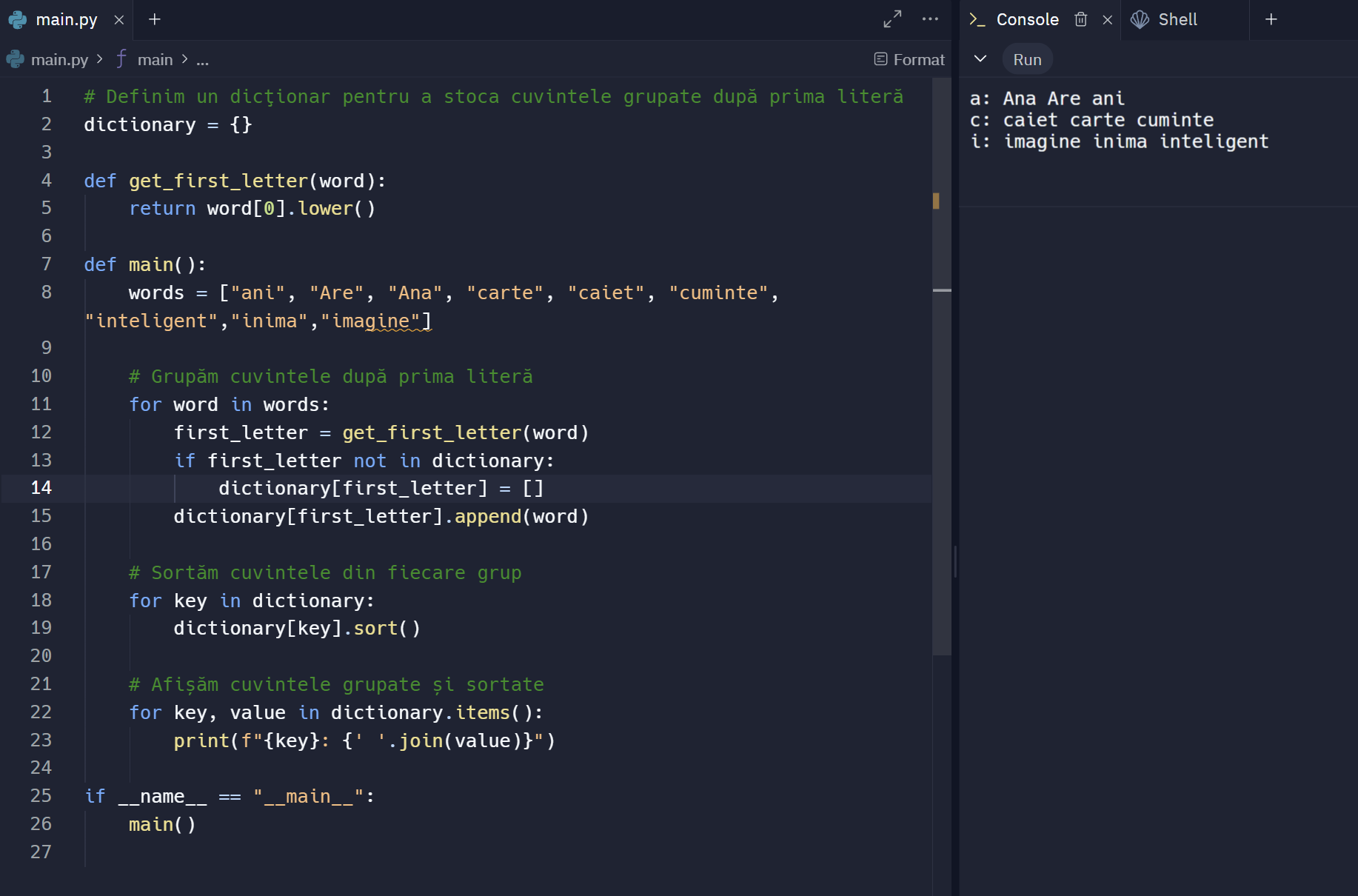
2.

Pentru fiecare element din vector, vom compara elementele din stiva cu elementul curent. Daca elementul curent este mai mare decat elementul din varful stivei, am gasit un "next greater element" pentru elementul din stiva.

Daca nu am gasit un "next greater element" pentru elementul curent, adaugam elementul curent si pozitia sa in stiva.

Perechile de pozitii (indexul elementului si indexul "next greater element") vor fi stocate intr-o coada.

Lab 4



Lab 5

1.

Elementul median al vectorului sortat devine radacina arborelui/subarborelui.

Repeta procesul pentru partea stanga si partea dreapta a vectorului sortat pentru a construi subarborii stanga si dreapta.

Creeaza noduri si le leaga in mod corespunzator pentru a forma arborele.

2.

Obtinem o lista sortata a elementelor fiecarui arbore folosind traversarea inordine.

Interclasam cele doua liste sortate pentru a obtine o singura lista sortata.

Construim un BST echilibrat din lista sortata rezultata.

Lab 6

1.

Kruskal adauga muchii la MST, lucrand cu muchiile graf-ului si folosind o structura de date Union-Find pentru a gestiona ciclurile.

Prim adauga noduri la MST, pornind de la un nod arbitrar si extinzand MST-ul prin adaugarea de noduri conectate cu cea mai mica muchie posibila.

2.

Alege nodul de pornire si initializeaza costurile.

Adauga nodul de pornire in min-heap.

Repeta pana cand min-heap-ul este gol:

Extrage nodul cu costul minim.

Actualizeaza costurile nodurilor adiacente daca se gaseste o muchie mai ieftina.

Foloseste array-ul parintilor pentru a construi si afisa arborele MST.

1.

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_idx = i

for j in range(i+1, n):

if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

return arr

# Exemplu de utilizare

arr = [64, 25, 12, 22, 11]

sorted\_arr = selection\_sort(arr)

print("Sorted array using Selection Sort:", sorted\_arr)

def heapify(arr, n, i):

largest = i

left = 2 \* i + 1

right = 2 \* i + 2

if left < n and arr[left] > arr[largest]:

largest = left

if right < n and arr[right] > arr[largest]:

largest = right

if largest != i:

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

heapify(arr, n, largest)

def heapsort(arr):

n = len(arr)

# Construcția heap-ului (max heap)

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

heapify(arr, n, i)

# Extracția elementelor din heap

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

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

return arr

# Exemplu de utilizare

arr = [64, 25, 12, 22, 11]

sorted\_arr = heapsort(arr)

print("Sorted array using Heapsort:", sorted\_arr)

2.

class Node:

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

self.key = key

self.next = None

class SinglyLinkedList:

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

self.head = None

def display(self):

current = self.head

while current:

print(current.key, end=" -> ")

current = current.next

print("None")

def insert(self, key):

new\_node = Node(key)

new\_node.next = self.head

self.head = new\_node

def search(self, key):

current = self.head

while current:

if current.key == key:

return True

current = current.next

return False

def delete(self, key):

current = self.head

prev = None

while current and current.key != key:

prev = current

current = current.next

if current:

if prev:

prev.next = current.next

else:

self.head = current.next

class DoublyNode:

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

self.key = key

self.next = None

self.prev = None

class DoublyLinkedList:

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

self.head = None

def display(self):

current = self.head

while current:

print(current.key, end=" <-> ")

current = current.next

print("None")

def insert(self, key):

new\_node = DoublyNode(key)

new\_node.next = self.head

if self.head:

self.head.prev = new\_node

self.head = new\_node

def search(self, key):

current = self.head

while current:

if current.key == key:

return True

current = current.next

return False

def delete(self, key):

current = self.head

while current and current.key != key:

current = current.next

if current:

if current.prev:

current.prev.next = current.next

if current.next:

current.next.prev = current.prev

if current == self.head:

self.head = current.next

class CircularNode:

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

self.key = key

self.next = None

class CircularLinkedList:

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

self.head = None

def display(self):

if self.head is None:

print("List is empty")

return

current = self.head

while True:

print(current.key, end=" -> ")

current = current.next

if current == self.head:

break

print("HEAD")

def insert(self, key):

new\_node = CircularNode(key)

if self.head is None:

self.head = new\_node

new\_node.next = self.head

else:

current = self.head

while current.next != self.head:

current = current.next

current.next = new\_node

new\_node.next = self.head

def search(self, key):

if self.head is None:

return False

current = self.head

while True:

if current.key == key:

return True

current = current.next

if current == self.head:

break

return False

def delete(self, key):

if self.head is None:

return

current = self.head

prev = None

while True:

if current.key == key:

if prev:

prev.next = current.next

else:

if current.next == self.head:

self.head = None

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

temp.next = current.next

self.head = current.next

return

prev = current

current = current.next

if current == self.head:

break

sll = SinglyLinkedList()

sll.insert(10)

sll.insert(20)

sll.display() # 20 -> 10 -> None

print(sll.search(10)) # True

sll.delete(20)

sll.display() # 10 -> None

dll = DoublyLinkedList()

dll.insert(10)

dll.insert(20)

dll.display() # 20 <-> 10 <-> None

print(dll.search(10)) # True

dll.delete(20)

dll.display() # 10 <-> None

cll = CircularLinkedList()

cll.insert(10)

cll.insert(20)

cll.display() # 10 -> 20 -> HEAD

print(cll.search(10)) # True

cll.delete(20)

cll.display() # 10 -> HEAD

3.1

class Stack:

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

self.items = []

def is\_empty(self):

return len(self.items) == 0

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

return None

def peek(self):

if not self.is\_empty():

return self.items[-1]

return None

def are\_parentheses\_balanced(expression):

stack = Stack()

opening\_brackets = "({["

closing\_brackets = ")}]"

matching\_brackets = {")": "(", "]": "[", "}": "{"}

for char in expression:

if char in opening\_brackets:

stack.push(char)

elif char in closing\_brackets:

if stack.is\_empty():

return False

top\_element = stack.pop()

if matching\_brackets[char] != top\_element:

return False

return stack.is\_empty()

# Exemple de testare

expressions = ["{[()()]}", "{[(])}", "((()))", "[{}]", "{[()]}[]"]

results = [are\_parentheses\_balanced(expr) for expr in expressions]

for expr, result in zip(expressions, results):

print(f"Expression: {expr} - Balanced: {result}")

3.2

class Node:

def \_\_init\_\_(self, data=None):

self.data = data

self.next = None

class Stack:

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

self.top = None

def push(self, data):

new\_node = Node(data)

new\_node.next = self.top

self.top = new\_node

def pop(self):

if self.top is None:

return None

popped\_node = self.top

self.top = self.top.next

return popped\_node.data

def peek(self):

if self.top is None:

return None

return self.top.data

def is\_empty(self):

return self.top is None

class QueueNode:

def \_\_init\_\_(self, data=None):

self.data = data

self.next = None

class Queue:

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

self.front = None

self.rear = None

def enqueue(self, data):

new\_node = QueueNode(data)

if self.rear is None:

self.front = self.rear = new\_node

return

self.rear.next = new\_node

self.rear = new\_node

def dequeue(self):

if self.front is None:

return None

temp = self.front

self.front = self.front.next

if self.front is None:

self.rear = None

return temp.data

def is\_empty(self):

return self.front is None

def next\_greater\_elements(arr):

stack = Stack()

result\_queue = Queue()

n = len(arr)

for i in range(n):

current = arr[i]

while not stack.is\_empty() and current > stack.peek()[0]:

value, index = stack.pop()

result\_queue.enqueue((index, i))

stack.push((current, i))

while not stack.is\_empty():

value, index = stack.pop()

result\_queue.enqueue((index, -1))

return result\_queue

# Funcție pentru afișarea elementelor din coadă

def print\_queue(queue):

while not queue.is\_empty():

print(queue.dequeue())

# Exemplu de utilizare

arr = [4, 5, 2, 10, 8]

result\_queue = next\_greater\_elements(arr)

# Afișare rezultate

print("Rezultatele perechilor (index element, index next greater element):")

print\_queue(result\_queue)

4.

# Definim un dicționar pentru a stoca cuvintele grupate după prima literă

dictionary = {}

def get\_first\_letter(word):

return word[0].lower()

def main():

words = ["ana", "si", "bogdan", "au", "multe", "mere", "albastre"]

# Grupăm cuvintele după prima literă

for word in words:

first\_letter = get\_first\_letter(word)

if first\_letter not in dictionary:

dictionary[first\_letter] = []

dictionary[first\_letter].append(word)

# Sortăm cuvintele din fiecare grup

for key in dictionary:

dictionary[key].sort()

# Afișăm cuvintele grupate și sortate

for key, value in dictionary.items():

print(f"{key}: {' '.join(value)}")

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

main()

5.1

class TreeNode:

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

self.key = key

self.left = None

self.right = None

def sorted\_array\_to\_bst(nums):

if not nums:

return None

mid = len(nums) // 2

root = TreeNode(nums[mid])

root.left = sorted\_array\_to\_bst(nums[:mid])

root.right = sorted\_array\_to\_bst(nums[mid+1:])

return root

def inorder\_traversal(root):

if root is None:

return []

return inorder\_traversal(root.left) + [root.key] + inorder\_traversal(root.right)

# Exemplu de utilizare

sorted\_array = [1, 2, 3, 4, 5, 6, 7]

bst\_root = sorted\_array\_to\_bst(sorted\_array)

# Afișare traversare inordine (care ar trebui să fie identică cu vectorul sortat original)

print("Inorder Traversal of BST:", inorder\_traversal(bst\_root))

5.2

class TreeNode:

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

self.key = key

self.left = None

self.right = None

def inorder\_traversal(root):

if root is None:

return []

return inorder\_traversal(root.left) + [root.key] + inorder\_traversal(root.right)

def merge\_sorted\_lists(list1, list2):

merged\_list = []

i = 0

j = 0

while i < len(list1) and j < len(list2):

if list1[i] < list2[j]:

merged\_list.append(list1[i])

i += 1

else:

merged\_list.append(list2[j])

j += 1

while i < len(list1):

merged\_list.append(list1[i])

i += 1

while j < len(list2):

merged\_list.append(list2[j])

j += 1

return merged\_list

def sorted\_array\_to\_bst(nums):

if not nums:

return None

mid = len(nums) // 2

root = TreeNode(nums[mid])

root.left = sorted\_array\_to\_bst(nums[:mid])

root.right = sorted\_array\_to\_bst(nums[mid+1:])

return root

def merge\_two\_bsts(root1, root2):

list1 = inorder\_traversal(root1)

list2 = inorder\_traversal(root2)

merged\_list = merge\_sorted\_lists(list1, list2)

return sorted\_array\_to\_bst(merged\_list)

# Funcție pentru a adăuga noduri într-un BST (folosit pentru a construi arbori de test)

def insert\_into\_bst(root, key):

if root is None:

return TreeNode(key)

if key < root.key:

root.left = insert\_into\_bst(root.left, key)

else:

root.right = insert\_into\_bst(root.right, key)

return root

# Construirea celor doi arbori binari de căutare

root1 = None

root1 = insert\_into\_bst(root1, 1)

root1 = insert\_into\_bst(root1, 3)

root1 = insert\_into\_bst(root1, 5)

root2 = None

root2 = insert\_into\_bst(root2, 2)

root2 = insert\_into\_bst(root2, 4)

root2 = insert\_into\_bst(root2, 6)

# Interclasarea celor doi BST-uri

merged\_root = merge\_two\_bsts(root1, root2)

# Afișarea BST-ului rezultat

print("Inorder Traversal of Merged BST:", inorder\_traversal(merged\_root))

6.

class DisjointSet:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

def kruskal(n, edges):

mst = []

ds = DisjointSet(n)

edges.sort(key=lambda edge: edge[2])

for u, v, weight in edges:

if ds.find(u) != ds.find(v):

ds.union(u, v)

mst.append((u, v, weight))

return mst

# Exemplu de utilizare

edges = [

(0, 1, 10),

(0, 2, 6),

(0, 3, 5),

(1, 3, 15),

(2, 3, 4)

]

n = 4 # Numărul de noduri

mst = kruskal(n, edges)

print("Arborele de Cost Minim (Kruskal):", mst)

import heapq

def prim(n, edges):

mst = []

visited = [False] \* n

min\_heap = [(0, 0)] # (cost, node)

total\_cost = 0

adjacency\_list = [[] for \_ in range(n)]

for u, v, weight in edges:

adjacency\_list[u].append((weight, v))

adjacency\_list[v].append((weight, u))

while min\_heap:

cost, u = heapq.heappop(min\_heap)

if visited[u]:

continue

visited[u] = True

total\_cost += cost

if cost != 0:

mst.append((u, parent, cost))

for weight, v in adjacency\_list[u]:

if not visited[v]:

heapq.heappush(min\_heap, (weight, v))

parent = u

return mst, total\_cost

# Exemplu de utilizare

edges = [

(0, 1, 10),

(0, 2, 6),

(0, 3, 5),

(1, 3, 15),

(2, 3, 4)

]

n = 4 # Numărul de noduri

mst, total\_cost = prim(n, edges)

print("Arborele de Cost Minim (Prim):", mst)

print("Costul Total:", total\_cost)