**Single Linked List**

prev = None

while temp and temp.data != key:

prev = temp

temp = temp.next

if temp is None:

print("The key is not present in the list.")

return

prev.next = temp.next

temp = None

def search(self, key):

current = self.head

while current:

if current.data == key:

return True

current = current.next

return False

def traverse(self):

elements = []

current = self.head

while current:

elements.append(current.data)

current = current.next

return elements

# Example usage

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

sll = SinglyLinkedList()

sll.insert\_at\_end(1)

sll.insert\_at\_end(2)

sll.insert\_at\_beginning(0)

sll.insert\_after(1, 1.5)

print("List after insertions:", sll.traverse())

sll.delete\_node(1.5)

print("List after deleting 1.5:", sll.traverse())

print("Searching for 2:", sll.search(2))

print("Searching for 3:", sll.search(3))

class Node:

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

self.data = data

self.next = None

class SinglyLinkedList:

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

self.head = None

def insert\_at\_beginning(self, data):

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

def insert\_at\_end(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

return

last = self.head

while last.next:

last = last.next

last.next = new\_node

def insert\_after(self, prev\_node\_data, data):

new\_node = Node(data)

current = self.head

while current and current.data != prev\_node\_data:

current = current.next

if current is None:

print("The previous node is not in the list.")

return

new\_node.next = current.next

current.next = new\_node

def delete\_node(self, key):

temp = self.head

if temp and temp.data == key:

self.head = temp.next

temp = None

return

**Double Linked List**

return

if temp.prev:

temp.prev.next = temp.next

if temp.next:

temp.next.prev = temp.prev

if temp == self.head:

self.head = temp.next

temp = None

def search(self, key):

current = self.head

while current:

if current.data == key:

return True

current = current.next

return False

def traverse\_forward(self):

elements = []

current = self.head

while current:

elements.append(current.data)

current = current.next

return elements

def traverse\_backward(self):

elements = []

current = self.head

if current is None:

return elements

while current.next:

current = current.next

while current:

elements.append(current.data)

current = current.prev

return elements

# Example usage

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

dll = DoublyLinkedList()

dll.insert\_at\_end(1)

dll.insert\_at\_end(2)

dll.insert\_at\_beginning(0)

dll.insert\_after(1, 1.5)

print("List after insertions:", dll.traverse\_forward())

dll.delete\_node(1.5)

print("List after deleting 1.5:", dll.traverse\_forward())

print("Searching for 2:", dll.search(2))

print("Searching for 3:", dll.search(3))

print("Traverse backward:", dll.traverse\_backward())

class Node:

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

self.data = data

self.next = None

self.prev = None

class DoublyLinkedList:

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

self.head = None

def insert\_at\_beginning(self, data):

new\_node = Node(data)

new\_node.next = self.head

if self.head is not None:

self.head.prev = new\_node

self.head = new\_node

def insert\_at\_end(self, data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

return

last = self.head

while last.next:

last = last.next

last.next = new\_node

new\_node.prev = last

def insert\_after(self, prev\_node\_data, data):

new\_node = Node(data)

current = self.head

while current and current.data != prev\_node\_data:

current = current.next

if current is None:

print("The previous node is not in the list.")

return

new\_node.next = current.next

new\_node.prev = current

if current.next:

current.next.prev = new\_node

current.next = new\_node

def delete\_node(self, key):

temp = self.head

while temp and temp.data != key:

temp = temp.next

if temp is None:

print("The key is not present in the list.")

**Circular Linked List**

else:

prev = None

curr = self.head

while curr.next != self.head:

prev = curr

curr = curr.next

if curr.data == key:

prev.next = curr.next

curr = curr.next

def traverse(self):

if self.head:

temp = self.head

while True:

print(temp.data, end=" -> ")

temp = temp.next

if temp == self.head:

break

print()

def search(self, key):

if self.head:

temp = self.head

while True:

if temp.data == key:

return True

temp = temp.next

if temp == self.head:

break

return False

# Example usage:

cll = CircularLinkedList()

cll.append(1)

cll.append(2)

cll.append(3)

cll.prepend(0)

cll.traverse() # Output: 0 -> 1 -> 2 -> 3 ->

print(cll.search(2)) # Output: True

print(cll.search(4)) # Output: False

cll.delete(2)

cll.traverse() # Output: 0 -> 1 -> 3 ->

cll.delete(0)

cll.traverse() # Output: 1 -> 3 ->

class Node:

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

self.data = data

self.next = None

class CircularLinkedList:

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

self.head = None

def append(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

new\_node.next = self.head

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

temp.next = new\_node

new\_node.next = self.head

def prepend(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

new\_node.next = self.head

else:

new\_node.next = self.head

temp = self.head

while temp.next != self.head:

temp = temp.next

temp.next = new\_node

self.head = new\_node

def delete(self, key):

if self.head:

if self.head.data == key:

if self.head.next == self.head: # Only one element

self.head = None

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

temp.next = self.head.next

self.head = self.head.next

**0\_1 Knapsack**

def knapsack(values, weights, capacity):

n = len(values)

dp = [[0 for x in range(capacity + 1)] for x in range(n + 1)]

for i in range(n + 1):

for w in range(capacity + 1):

if i == 0 or w == 0:

dp[i][w] = 0

elif weights[i - 1] <= w:

dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])

else:

dp[i][w] = dp[i - 1][w]

return dp[n][capacity]

def main():

# Get user input for the number of items

num\_items = int(input("Enter the number of items: "))

# Get user input for the weights and values of each item

values = []

weights = []

for i in range(num\_items):

value = int(input(f"Enter the value of item {i + 1}: "))

weight = int(input(f"Enter the weight of item {i + 1}: "))

values.append(value)

weights.append(weight)

# Get user input for the capacity of the knapsack

capacity = int(input("Enter the capacity of the knapsack: "))

# Solve the knapsack problem

max\_value = knapsack(values, weights, capacity)

# Output the result

print(f"\nMaximum value in the knapsack: {max\_value}")

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

main()

**Fractional Knapsack**

def knapsack(n, weight, profit, capacity):

x = [0.0] \* 20

tp = 0.0

u = capacity

for i in range(n):

x[i] = 0.0

for i in range(n):

if weight[i] > u:

break

else:

x[i] = 1.0

tp += profit[i]

u -= weight[i]

if i < n:

x[i] = u / weight[i]

tp += x[i] \* profit[i]

print("\nThe result vector is:")

for i in range(n):

print(f"{x[i]:.6f}\t", end="")

print(f"\nMaximum profit is: {tp:.6f}")

def main():

weight = [0.0] \* 20

profit = [0.0] \* 20

ratio = [0.0] \* 20

num = int(input("Enter the number of items: "))

print("Enter the weights and profits of each item:")

for i in range(num):

weight[i], profit[i] = map(float, input().split())

capacity = float(input("Enter the capacity of the knapsack: "))

for i in range(num):

ratio[i] = profit[i] / weight[i]

for i in range(num):

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

if ratio[i] < ratio[j]:

ratio[i], ratio[j] = ratio[j], ratio[i]

weight[i], weight[j] = weight[j], weight[i]

profit[i], profit[j] = profit[j], profit[i]

knapsack(num, weight, profit, capacity)

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

main()

**Graph Adjacency Matrix**

class Graph:

def \_\_init\_\_(self, num\_vertices):

self.num\_vertices = num\_vertices

self.adj\_matrix = [[0] \* num\_vertices for \_ in range(num\_vertices)]

self.vertices = {}

def add\_vertex(self, vertex):

if len(self.vertices) < self.num\_vertices:

self.vertices[vertex] = len(self.vertices)

else:

print("Graph: Cannot add more vertices than the initialized number.")

def add\_edge(self, vertex1, vertex2):

if vertex1 in self.vertices and vertex2 in self.vertices:

index1 = self.vertices[vertex1]

index2 = self.vertices[vertex2]

self.adj\_matrix[index1][index2] = 1

self.adj\_matrix[index2][index1] = 1 # Fixing the typo for undirected graph

else:

print("Graph: One or both vertices not found in graph.")

def remove\_edge(self, vertex1, vertex2):

if vertex1 in self.vertices and vertex2 in self.vertices:

index1 = self.vertices[vertex1]

index2 = self.vertices[vertex2]

self.adj\_matrix[index1][index2] = 0

self.adj\_matrix[index2][index1] = 0

else:

print("Graph: One or both vertices not found in graph.")

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

result = "Vertices: " + str(list(self.vertices.keys())) + "\n"

result += "Adjacency Matrix:\n"

for row in self.adj\_matrix:

result += ' '.join(map(str, row)) + "\n"

return result

# Example usage

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

num\_vertices = 5

graph = Graph(num\_vertices)

vertices = ['A', 'B', 'C', 'D', 'E']

for vertex in vertices:

graph.add\_vertex(vertex)

edges = [('A', 'B'), ('A', 'C'), ('B', 'C'), ('C', 'D')]

for edge in edges:

graph.add\_edge(edge[0], edge[1])

print(graph)

**Graph Adjacency List**

class Graph:

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

self.adj\_list = {}

def add\_vertex(self, vertex):

if vertex not in self.adj\_list:

self.adj\_list[vertex] = []

else:

print(f"Vertex '{vertex}' already exists in the graph.")

def add\_edge(self, vertex1, vertex2):

if vertex1 in self.adj\_list and vertex2 in self.adj\_list:

self.adj\_list[vertex1].append(vertex2)

self.adj\_list[vertex2].append(vertex1)

else:

print("One or both vertices not found in graph.")

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

result = ""

for vertex in self.adj\_list:

result += f"{vertex}: {self.adj\_list[vertex]}\n"

return result

# Example usage

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

graph = Graph()

vertices = ['A', 'B', 'C', 'D', 'E']

for vertex in vertices:

graph.add\_vertex(vertex)

edges = [('A', 'B'), ('A', 'C'), ('B', 'C'), ('C', 'D')]

for edge in edges:

graph.add\_edge(edge[0], edge[1])

print(graph)