**Assignment: 1**

****

**Afshan Azhar**

**BS (SE)-2022**

**Section “A”**

**DATA STRUCTURE AND ALGORITHM [CSC 2051]**

**Submitted to: Ms. Anum Ilyas**

**Date: 14-june-2023**

**Computer Science and Software Engineering**

**Jinnah University for women**

**Theory Assignment**

**QUESTION** **1**:

Consider a directed graph representing a transportation network, where each node represents a city and theedges represent flights between cities. You are tasked with finding the shortest path from City A to City Busing Dijkstra's algorithm.

The graph contains cities labeled A, B, C, D, E, F, G, and H, with the following connections and distances:

* City A has a direct flight to City B with a distance of 200 km, to City C with a distance of 150 km, and to City D with a distance of 300 km.
* City B has a direct flight to City E with a distance of 100 km.
* City C has a direct flight to City D with a distance of 200 km and to City F with a distance of 250km.
* City D has a direct flight to City G with a distance of 150 km.
* City E has a direct flight to City H with a distance of 180 km.
* City F has a direct flight to City H with a distance of 120 km.
* City G has a direct flight to City H with a distance of 90 km.

**CODE:**

import heapq  
  
def dijkstra(graph, start, end):  
 distances = {city: float('inf') for city in graph}  
 distances[start] = 0  
 heap = [(0, start)]  
 visited = set()  
  
 while heap:  
 current\_distance, current\_city = heapq.heappop(heap)  
  
 if current\_city == end:  
 break  
  
 if current\_city in visited:  
 continue  
  
 visited.add(current\_city)  
  
 for neighbor, distance in graph[current\_city].items():  
 new\_distance = current\_distance + distance  
 if new\_distance < distances[neighbor]:  
 distances[neighbor] = new\_distance  
 heapq.heappush(heap, (new\_distance, neighbor))  
  
 return distances[end]  
  
*# Define the transportation network graph*graph = {  
 'A': {'B': 200, 'C': 150, 'D': 300},  
 'B': {'E': 100},  
 'C': {'D': 200, 'F': 250},  
 'D': {'G': 150},  
 'E': {'H': 180},  
 'F': {'H': 120},  
 'G': {'H': 90},  
 'H': {}  
}  
  
*# Find the shortest distance from City A to City B*start\_city = 'A'  
end\_city = 'B'  
shortest\_distance = dijkstra(graph, start\_city, end\_city)  
  
print(f"The shortest distance from {start\_city} to {end\_city} is {shortest\_distance} km.")

**OUTPUT:**



**QUESTION** **2:**

A user logs into the social media platform and accesses their news feed. The news feed contains a stream of posts from the user's friends and followed accounts. Currently, the posts are displayed in chronological order, with the most recent posts appearing at the top. To enhance the user experience, the company wants to implement a sorting feature that allows users to view the posts based on the number of likes they have received. This will enable users to see the most popular posts and trending topics in their network. Your task as a software engineer is to implement a suitable sorting algorithm, such as Merge Sort, Quick Sort, or Heap Sort, to sort the posts based on the number of likes. The algorithm should rearrange the posts, displaying those with the highest number of likes at the top of the news feed. Additionally, the sorting algorithm should handle cases where multiple posts have the same number of likes. In such cases, the algorithm should ensure that the posts are further sorted chronologically, with the most recent posts appearing first. Also justify the reason for selection of algorithm. By implementing this sorting algorithm, you will provide users with a valuable feature that allows them to customize their news feed based on popularity, enabling them to stay up-to-date with the most engaging and relevant content in their social network.

**CODE**:

def merge\_sort(posts):

if len(posts) <= 1:

return posts

mid = len(posts) // 2

left = posts[:mid]

right = posts[mid:]

left = merge\_sort(left)

right = merge\_sort(right)

return merge(left, right)

def merge(left, right):

merged = []

i = 0

j = 0

while i < len(left) and j < len(right):

if left[i]['likes'] > right[j]['likes']:

merged.append(left[i])

i += 1

elif left[i]['likes'] < right[j]['likes']:

merged.append(right[j])

j += 1

else:

if left[i]['timestamp'] > right[j]['timestamp']:

merged.append(left[i])

i += 1

else:

merged.append(right[j])

j += 1

while i < len(left):

merged.append(left[i])

i += 1

while j < len(right):

merged.append(right[j])

j += 1

return merged

# Example usage

posts = [

{'id': 1, 'likes': 20, 'timestamp': '2022-01-15 10:30:00'},

{'id': 2, 'likes': 30, 'timestamp': '2022-01-15 12:45:00'},

{'id': 3, 'likes': 10, 'timestamp': '2022-01-15 11:15:00'},

{'id': 4, 'likes': 30, 'timestamp': '2022-01-15 09:00:00'},

{'id': 5, 'likes': 20, 'timestamp': '2022-01-15 14:20:00'}

]

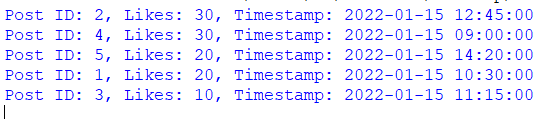
sorted\_posts = merge\_sort(posts)

# Display the sorted posts

for post in sorted\_posts:

print(f"Post ID: {post['id']}, Likes: {post['likes']}, Timestamp: {post['timestamp']}")

**OUTPUT:**



**QUESTION** **3:**

a) What do you understand by the term 'complexity'? Find the complexity of the following algorithm using Big O notation (suppose 'Module X' requires T units of time to be executed, where T is a constant and

'n' is the input size).

1. Set K := N;

2. Repeat for I = 1 to N:

3. Repeat steps 4 and 5 while K > 1

4. Module X

5. Set K := K/2.

[End of Step 3 loop]

[End of Step 2 loop]

6. Exit.

**ANSWER**:

the term "complexity" refers to the analysis of the resources required by an algorithm to solve a problem as a function of the input size. The most commonly analyzed resources are time complexity and space complexity.

Time complexity is a measure of the amount of time an algorithm takes to run, expressed as a function of the input size. It gives an estimation of how the running time of an algorithm grows as the input size increases. Big O notation is used to describe the upper bound or worst-case time complexity of an algorithm.

Now let's analyze the given algorithm and determine its time complexity using Big O notation:

Set K := N; - Constant time O(1)

Repeat for I = 1 to N: - O(N) (loop runs N times)

Repeat steps 4 and 5 while K > 1 - O(log N) (loop runs log N times)

Module X - Constant time O(1)

Set K := K/2. - Constant time O(1)

[End of Step 3 loop]

[End of Step 2 loop]

Exit. - Constant time O(1)

To find the overall time complexity, we need to consider the nested loops. The outer loop (Step 2) runs N times, and the inner loop (Step 3) runs log N times. Hence, the time complexity of the algorithm can be approximated as O(N log N).

Note: The time complexity analysis assumes that the time required to execute "Module X" is constant and does not depend on the input size.

b) You are given a dataset called "TEST" that represents the marks of 5 tests scored by 40 students in a

class. The dataset is structured as a 40 x 5 array. Your goal is to formulate an algorithm to find the

average marks of each student. Devise an algorithm to calculate the average marks of each student

based on the given dataset?

**CODE:**

dataset = [

[80, 75, 90, 85, 95],

[70, 85, 80, 75, 90],

# ... remaining rows

]

# Step 1: Initialize list for average marks

average\_marks = []

# Step 2-6: Calculate average marks for each student

for row in dataset:

# Step 3: Calculate sum of marks

total\_marks = sum(row)

# Step 4: Calculate average marks

average = total\_marks / len(row)

# Step 5: Append average marks to the list

average\_marks.append(average)

# Step 7: Display average marks

for i, average in enumerate(average\_marks):

student\_id = i + 1

print(f"Average marks for Student {student\_id}: {average}")

**OUTPUT:**

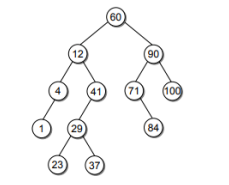


**QUESTION** **4:**

a) Consider the binary search tree below, show the resulting tree (with all necessary steps) after:

i. Inserting each of the following keys: 65, 95 and 37.

ii. Deleting each of the following keys: 41, 90 and 60.



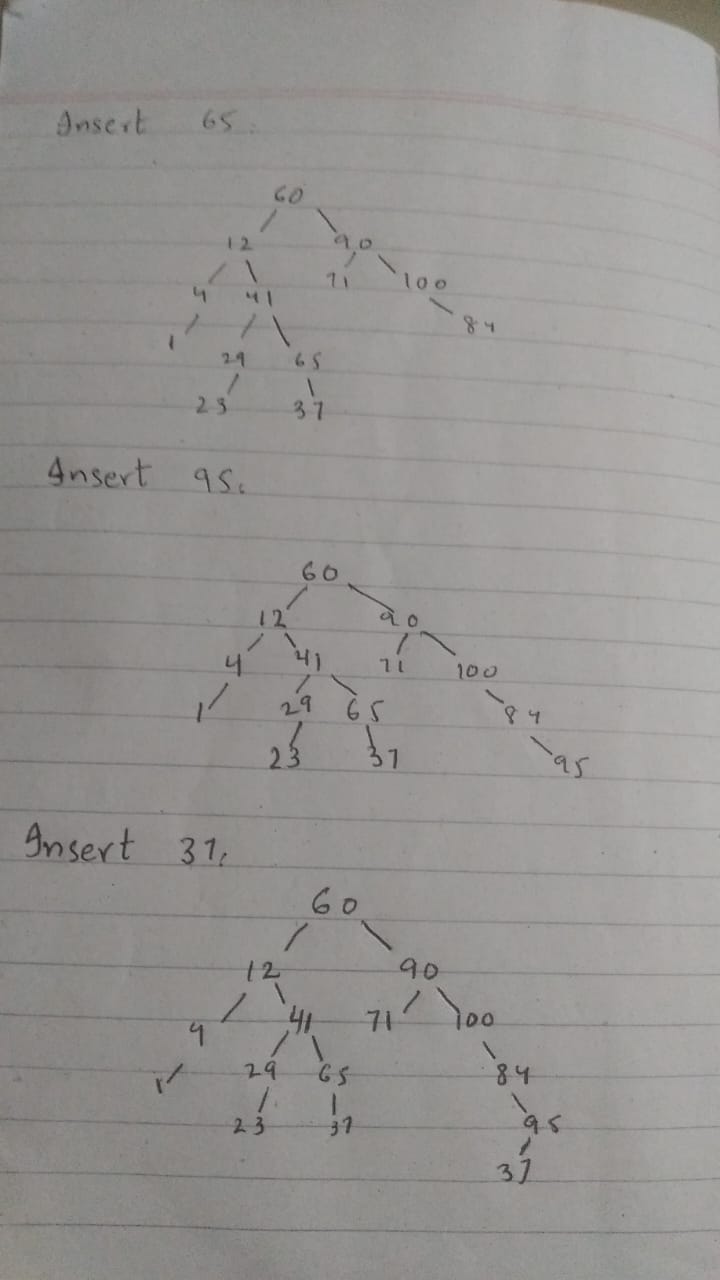
b) Show that the binary tree corresponding to an optimal binary prefix code must be full. A full binary

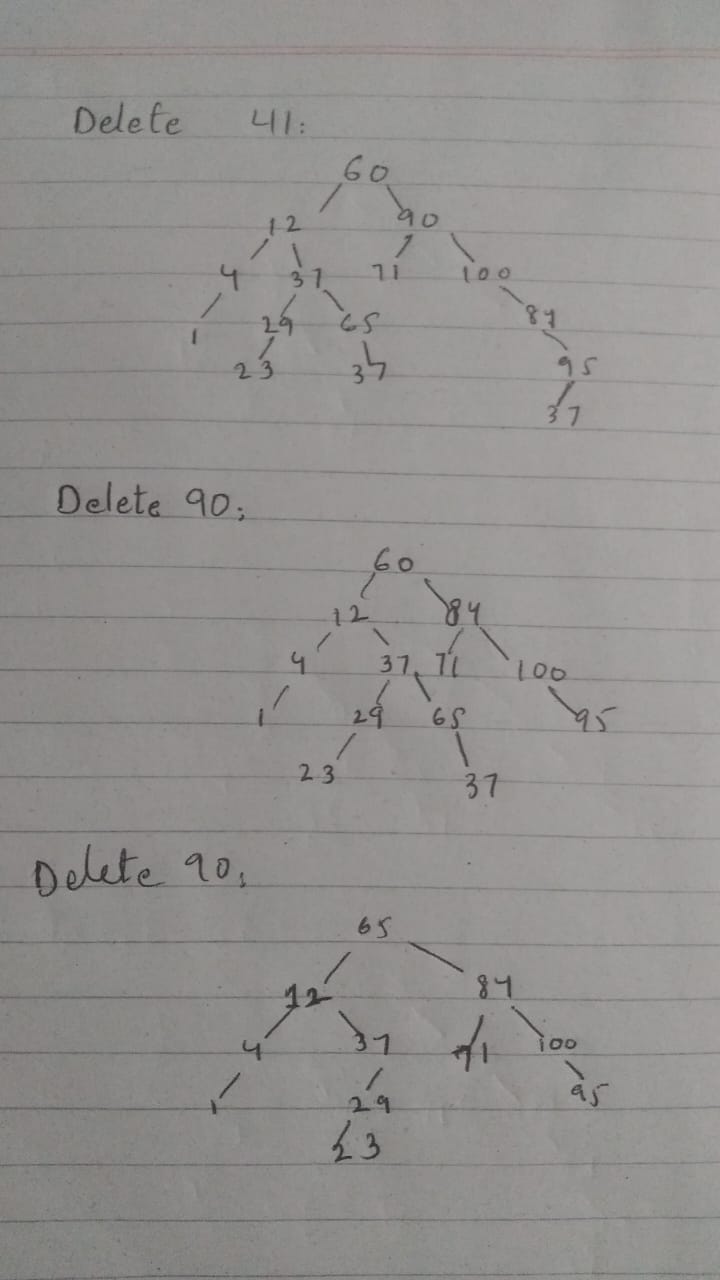
tree is a binary tree such that each node is either a leaf or it has two children.

c) Write an algorithm to check if the given binary tree is a heap.

**ANSWER**:

1. Binary Search Tree Operations:





b) To show that a binary tree corresponding to an optimal binary prefix code must be full, let's assume that there exists an optimal binary prefix code with a non-full binary tree. In a non-full binary tree, there exists a node with either no children or only one child.

However, in an optimal binary prefix code, each character is represented by a leaf node in the binary tree. If there is a node with no children or only one child, it means that there is at least one character that does not have a unique binary code. This contradicts the definition of an optimal binary prefix code, where each character has a unique binary representation.

Hence, we can conclude that a binary tree corresponding to an optimal binary prefix code must be full, where each node is either a leaf or has two children.

c) Algorithm to Check if a Binary Tree is a Heap:

To check if a given binary tree is a heap, we need to ensure that it satisfies the heap property, which states that for every node, the value of the node is greater than or equal to the values of its children (for a max heap). Here's an algorithm to check if a binary tree is a heap:

Start at the root of the binary tree.

Recursively check each node and its children:

a. If the node's value is smaller than any of its children, return False (not a heap).

b. Recursively check if the left and right subtrees are heaps.

If all nodes pass the above conditions, return True (a heap).

**QUESTION** **5**:

You are playing a game where you need to shoot a specific set of targets on a shooting range. The shooting range initially has n targets, and you have a predefined list of k targets that you need to shoot in a specific order.

Case I: For every 2 targets you shoot, one new target is added to the shooting range. In this scenario, you shoot targets based on a predetermined order and simultaneously add new targets to the shooting range.

Case II: Once you shoot a target, it is removed from the range, and you proceed to the next target in the list. In this scenario, you shoot targets based on a predetermined order and don't add any new targets to the shooting range.The question is: How many targets do you shoot to complete the entire list? Provide the value of n (initial number of targets on the shooting range) to determine the appropriate Big Oh notation for this scenario.

**ANSWER:**

In Case I, for every 2 targets you shoot, one new target is added to the shooting range. Let's analyze the scenario to determine the number of targets you need to shoot to complete the entire list.

Initially, you have n targets on the shooting range. For every 2 targets you shoot, one new target is added. This means that after shooting k targets (the predetermined list), you would have added (k/2) new targets to the shooting range.

So, to complete the entire list, you need to shoot k targets and add (k/2) new targets. Therefore, the total number of targets you shoot would be k + (k/2).

In terms of Big O notation, we can simplify the expression by ignoring the constant factor. Thus, the number of targets you shoot can be represented as O(k).

As for the value of n (initial number of targets on the shooting range), it does not directly affect the number of targets you shoot in this scenario. The focus is on the predetermined list of targets (k) and the addition of new targets during shooting.

In conclusion, for Case I, the number of targets you shoot to complete the entire list can be represented as O(k), where k is the number of predetermined targets.

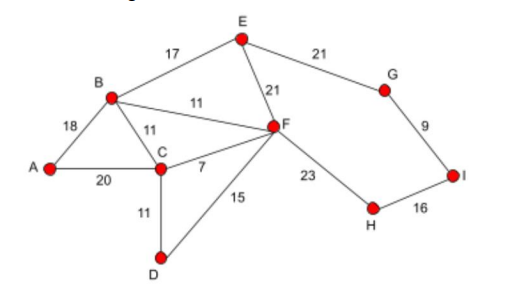
**Question 6:**

a) Does the Kruskal algorithm start by choosing an edge or a node?

b) In the given diagram, we have a network of paths in a park, where each edge represents a path with a

specific length in meters. Find a minimal spanning tree for the network in the diagram using Kruskal's

approach, and then determine its length.



c) When do you prefer using Kruskal's algorithm over any other algorithm?

**ANSWER:**

a) The Kruskal's algorithm starts by choosing an edge.

b) The length of the minimal spanning tree can be calculated by summing the lengths of all the edges in the tree. In this case, the length of the minimal spanning tree is:

length = 7+11+11+18+15+7+20+11=100 meters

c) Kruskal's algorithm is preferred over other algorithms for finding the minimum spanning tree in the following scenarios:

When the graph is sparse (has fewer edges compared to the total possible edges). Kruskal's algorithm has a time complexity of O(E log E), where E is the number of edges. Since it does not depend on the number of vertices, it performs well on sparse graphs.

When the graph is unweighted or has equal edge weights. Kruskal's algorithm does not consider the weights of the edges during the sorting process, making it efficient for unweighted or equal-weighted graphs.

When the graph is undirected. Kruskal's algorithm can be easily applied to undirected graphs, as it does not require traversing in a specific direction.

However, it's important to note that the choice of algorithm depends on the specific characteristics of the problem and the graph. Other algorithms like Prim's algorithm or Boruvka's algorithm may be more suitable in certain cases, especially for dense graphs or when edge weights are significant factors in determining the minimum spanning tree.