Name: Shivam Kumar Singh / (1:11)

(Q1) Ans: Primary reason for chosen the right data structure :-

1) Efficiency → the choice of data structure directly impact the efficiency of algorithm. A well-chosen data structure can significantly reduce time and space complexity. For example, using a hash table for quick lookup or a priority queue for efficient sorting.

2) memory usage → different data structure having varying memory foot print. Efficient memory usage for crucial for large data set or resources-constrained environments.

3) Algorithm design → the choice of data structure often dictates the design and logic of the algorithms. certain Algorithm are inherently suited to specific data structure.

(1-6)

4) Ease of Implementation → Some data structure are simpler to Implement than other. choosing a simpler data structure can lead to faster development time and easier debugging.

5) Problem solving sutiability → Different data structure are different suited for different Problem domain. For example graph are Ideal for representing relation and networks, while trees. are useful for hierchal data.

* categorization of data structure :

  * Int : Represents Integers values.
  * Float : Represent Floating point number.
  * char : Represent single characters.
  * bool : Represent Boolen value (true and false).

* Linear Secondary data structure

  * stack : Follows a LIFO (Last-In-First-out)
  * queue : Follow a FIFO (First-In-First-out)

(1) Ans

1-C

Non-linear secondary data structure

tree = Represent hierchical relationship between nodes.

graph = Represent a network of nodes connect else.

Key point

→ Primary data structre are the fundamental building Block.

→ Linear secondary data structe have sequential arrangent

→ Non-linear data structe have complex relationship between element ☺

(a2) Ans

(2a)

An AVL tree (name after its Inventors, Adelson-Velskii and Landis) Is a self balancing binary search tree where the height of two subtree of any node differ by at most one. this property ensures that the tree maintains Algorithmic height, leading to effiyent search, Insertion deletion, operations.

* construct an AVL tree For the sequence :-

20, 30, 10, 50, 40, 70, 80, 60

1) Insertion of 20

→ create a root node with value 20.

2) Insertion of 30.

→ 30 Is greater than 20. so it become right child of the root

3) Insertion of 10:

→ 10 Is less than 20. so. it become left child of the root.

(2-b)

4) Insertion of 50:

- 50 Is greater than 30. So it become the right child of 30.

5) Insertion of 40:

- 40 Is greater than 30 but less than 50 so it become left chid of 50.

6) Insertion of 70:

- 70 Is greater.

7) Insertion of 80:

- 80 Is greater than 70. so it become the right child of 70.

8) Insertion of 60:

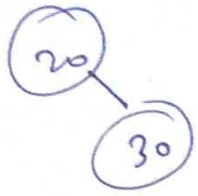
→ 60 Is greater than 50 but less then 70. so it become the left chid of 70.

(1) Her explain by using diagram - AVL tree

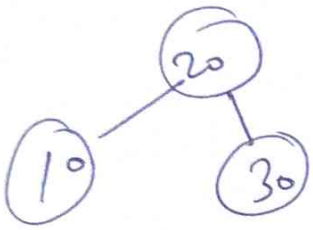
2-C

1)

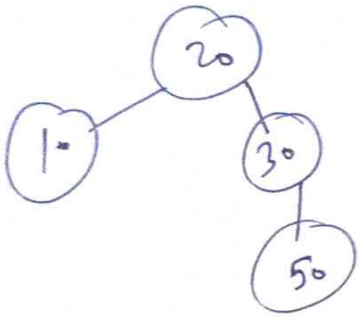
2)



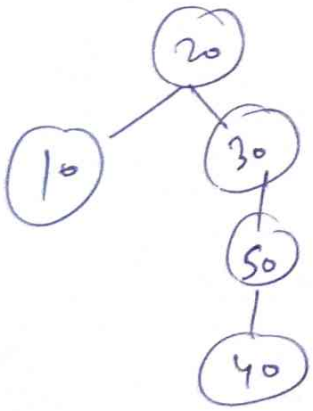
3)



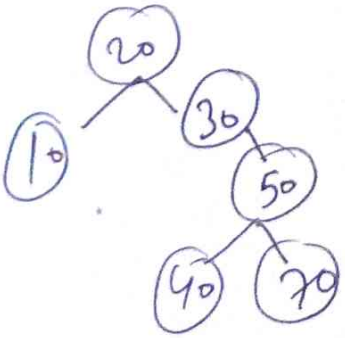
4)



5)

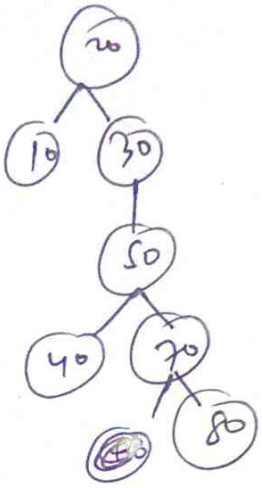


6)

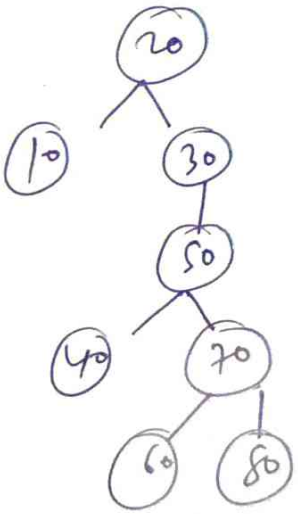


Final AVL tree

7)



8) Final AVL tree



(3) (a) Ans

Ans Insertion Sort Algorithm

(3-a)

→ Insertion sort is a simple sorting algorithm that work by iteratively building a sorted array one element at a time. It start with the second element and compares it with the element before it. shifting them to the right full until the correct position is found for entires for each element. the process continues for the current element. this process continues for each element in the array gradully building the sorted portion.

→ Time complexity

* Best case : o(n) when the array is already sorted.
* Averge case : o(n^2) . when the element are in random order
* worst case : o(n^2), . when the array is sorted in reverse order.

(*) Efficiency → Insertion sort is most efficient (3-b) for small array or when the array is partially sorted. It is also a good choice when memory write expensive as it minimizes the number of write writes.

(*) sorting the given sequence using insert sort.

step 1

[47, 12, 1, 68, 3]

step 2

[12, 47, 1, 68, 3] (12 is smaller than 47, so it is shifted to the left.)

step 3

[1, 12, 47, 68, 3] (1 is smaller than 12 and 47. So it is shifted to the left most position.

step 4

[1, 12, 47, 68, 3] (68 is already in the correct position).

3-c

step 5

[1, 3, 12, 47, 68] (3 is smaller than 12, 47 and 68 so it's shifted to the correct position.

* Final sorted array

[1, 3, 12, 47, 68]

(3-c)

step 5

[1, 3, 12, 47, 68] (3 is smaller than 12, 47 and 68 so it's shifted to the correct position.

* Final sorted array

[1, 3, 12, 47, 68]

Ans (5) (4) Ans

(4-a)

→ Recursion Involves a function calling itself directly or Indirectly To understand how recursion works under the hood. it's helpful to visualize it using a stack data structure.

*→ The following is recursive function to calculate the factorial of a number.

Python

```
def factorial (n):
    If n == 0:
    Return 1
    else :
    Return n * factorial (n-1)
```

(* ↓ (It's from the execution of factorial (3):)

(*) step (1)

factorial (3) Is called the function pushes the current value of n (3) and the return Address onto the stack.

If the call factorial (2)

step2

factorial 2 Is called the function pushes the current value of n(2) and the Return the Address of the onto the stack.

If the call factorial (1).

step3

factorial (1) Is called :
the function pushes the current value of n(1) and the Return Address on to the stacks.

If the factorial (0)

step (4)

factorial (0) Is called.
the base con Is reach. and the function Return 1.

step (5)

factorial (1) pops the Return Address and 1 from the stack.
if multiple 1 by 1 and Return 1

(4-E)

step 6

factorial(2) pops the return Address and 1 from the stack.
it multiple 2 by 1 and return 2 ÷

step 7

factorial(3) pops the return Address and 2 from the stack.
it multiplies 3 by 2 and return 6.
Visual Representation ÷

stack (@ top to Bottom)

3, return Address.
2, return Address.
1, return Addres.
0, return Addres.

1
1
2
6

(5) =

step (1) Initialize all distance by ∞ except node.

distance ⇒ [0, ∞, ∞, ∞, ∞, ∞]

step (2) Initialize visited array

(*) Initialize all element to false -

visited = [false, false, false, false, false, false].

step (3)

Find the node with minimum distance.

→ In the first iteration, the source node has smallest distance. (0)

step 4

visited[0] = true.

step 5

for node A

* Distance B through A = distance[0] + 6 = 6
* " C " A = distance[0] + 5 = 5
* update distance[1] to 6 and distance[2] to 5

Step 6 Repeat steps 3-5 until all nodes are visited.

Iteration 2

* Node C has minimum distance (5).
* marks C as visited
* update distance B and D through C.

Iteration 3

* Node B has minimum distance 6.
* mark B as visited
* Update distance D and E through B.

Iteration 4

Node D has the minimum distance (7).
mark D has visited.
update D and E through B.

Iteration 5

* Node E has the minimum distance (9).
* mark F as visited

Final distance array = [0, 6, 5, 7, 8, 5]

(5) Shortest path

S to A -> A (Distance 6)

S to B -> C -> B (Distance)

s to c -> S -> c (Distance 5)