1. ii)

Let's go through each operation step by step:

### Initial State:

- Sequence \( a = [1, 5, 7, 8, 9, 2] \)

- Stack \( S \): Empty

- Queue \( Q \): Empty

### Step-by-Step Execution:

1. \*\*Operation I\*\*: Push elements of \( a \) from \( a\_0 \) to \( a\_5 \) in order into \( S \).

- \( S \) after pushes: [1, 5, 7, 8, 9, 2] (top is 2)

2. \*\*Operation II\*\*: Enqueue elements of \( a \) from \( a\_0 \) to \( a\_5 \) in order into \( Q \).

- \( Q \) after enqueues: [1, 5, 7, 8, 9, 2] (front is 1)

3. \*\*Operation III\*\*: Pop an element from \( S \).

- Popped: 2

- \( S \): [1, 5, 7, 8, 9] (top is 9)

4. \*\*Operation IV\*\*: Dequeue an element from \( Q \).

- Dequeued: 1

- \( Q \): [5, 7, 8, 9, 2] (front is 5)

5. \*\*Operation V\*\*: Pop an element from \( S \).

- Popped: 9

- \( S \): [1, 5, 7, 8] (top is 8)

6. \*\*Operation VI\*\*: Dequeue an element from \( Q \).

- Dequeued: 5

- \( Q \): [7, 8, 9, 2] (front is 7)

7. \*\*Operation VII\*\*: Dequeue an element from \( Q \) and push it into \( S \).

- Dequeued: 7

- \( Q \): [8, 9, 2]

- Pushed into \( S \): 7

- \( S \): [1, 5, 7, 8, 7] (top is 7)

8. \*\*Operation VIII\*\* (first repeat): Dequeue an element from \( Q \) and push it into \( S \).

- Dequeued: 8

- \( Q \): [9, 2]

- Pushed into \( S \): 8

- \( S \): [1, 5, 7, 8, 7, 8] (top is 8)

9. \*\*Operation VIII\*\* (second repeat): Dequeue an element from \( Q \) and push it into \( S \).

- Dequeued: 9

- \( Q \): [2]

- Pushed into \( S \): 9

- \( S \): [1, 5, 7, 8, 7, 8, 9] (top is 9)

10. \*\*Operation VIII\*\* (third repeat): Dequeue an element from \( Q \) and push it into \( S \).

- Dequeued: 2

- \( Q \): []

- Pushed into \( S \): 2

- \( S \): [1, 5, 7, 8, 7, 8, 9, 2] (top is 2)

11. \*\*Operation IX\*\*: Pop an element from \( S \).

- Popped: 2

- \( S \): [1, 5, 7, 8, 7, 8, 9] (top is 9)

12. \*\*Operation X\*\*: Pop an element from \( S \).

- Popped: 9

- \( S \): [1, 5, 7, 8, 7, 8] (top is 8)

### Final Answer:

The top element of \( S \) after all operations is \*\*8\*\*.

1. ii)

**Initial State**

* Stack S1S1S1: [100, 200, 300, 400] (400 is at the top)
* Stack S2S2S2: [] (empty)

**Available Operations**

1. **PushToS2**: Pop the top element from S1S1S1 and push it onto S2S2S2.
2. **PushToS1**: Pop the top element from S2S2S2 and push it onto S1S1S1.
3. **GenerateOutput**: Pop the top element from S1S1S1 and output it.

Since S1S1S1 is initially full, we can start by directly using the **GenerateOutput** operation on S1S1S1 to output elements. However, if we want a different order or need to free up space in S1S1S1 for reordering, we can use **PushToS2** and **PushToS1**.

**Step-by-Step Execution**

**Goal: Generate all possible outputs using the operations.**

1. **GenerateOutput**: Pop 400 from S1S1S1.
   * Output: 400
   * S1S1S1: [100, 200, 300]
2. **GenerateOutput**: Pop 300 from S1S1S1.
   * Output: 300
   * S1S1S1: [100, 200]
3. **PushToS2**: Pop 200 from S1S1S1 and push it to S2S2S2.
   * S1S1S1: [100]
   * S2S2S2: [200]
4. **GenerateOutput**: Pop 100 from S1S1S1.
   * Output: 100
   * S1S1S1: []
5. **PushToS1**: Pop 200 from S2S2S2 and push it to S1S1S1.
   * S1S1S1: [200]
   * S2S2S2: []
6. **GenerateOutput**: Pop 200 from S1S1S1.
   * Output: 200
   * S1S1S1: []

**Final Output Sequence**

The output sequence generated is: **400, 300, 100, 200**

1. ii)

**Full Condition**

The stacks are considered "full" when:

top1+1=top2\text{top1} + 1 = \text{top2}top1+1=top2

**Explanation**

* If top1+1=top2 there is no more space between the two stacks in the array to accommodate any new element in either stack.
* This condition ensures efficient use of space because both stacks can grow until they meet in the middle, utilizing all available array slots.

So, the **condition for the array to be full** when implementing two stacks in this way is:

Top1+1=top2

1. ii)

**Stack Operations**

The stack operations are:

1. **Push(54)**: Stack becomes [54].
2. **Push(55)**: Stack becomes [54, 55].
3. **Pop()**: Pops the top element, 55. Stack becomes [54].
4. **Push(62)**: Stack becomes [54, 62].
5. **S = pop()**: Pops the top element, 62, and assigns it to SSS.

After these operations, S=62S = 62S=62.

**Queue Operations**

The queue operations are:

1. **Enqueue(21)**: Queue becomes [21].
2. **Enqueue(24)**: Queue becomes [21, 24].
3. **Dequeue()**: Removes the front element, 21. Queue becomes [24].
4. **Enqueue(28)**: Queue becomes [24, 28].
5. **Enqueue(32)**: Queue becomes [24, 28, 32].
6. **Q = dequeue()**: Removes the front element, 24, and assigns it to QQQ.

After these operations, Q=24Q = 24Q=24.

**Final Calculation**

The value of S+Q=62+24=86S + Q = 62 + 24 = 86S+Q=62+24=86.

1. ii)

**Step-by-Step Evaluation**

1. **Push 8**: Stack = [8]
2. **Push 2**: Stack = [8, 2]
3. **Push 3**: Stack = [8, 2, 3]
4. **Exponentiation (^)**: Pop 2 and 3, compute 23=82^3 = 823=8, and push the result.
   * Stack = [8, 8]
5. **Division (/)**: Pop 8 and 8, compute 8/8=18 / 8 = 18/8=1, and push the result.
   * Stack = [1]
6. **Push 2**: Stack = [1, 2]
7. **Push 3**: Stack = [1, 2, 3]
8. **Multiplication (\*)**: Pop 2 and 3, compute 2∗3=62 \* 3 = 62∗3=6, and push the result.
   * Stack = [1, 6]

**Final Answer**

The top two elements of the stack after the first \* is evaluated are **6** and **1**.

1. ii)

**Step 1: Push Items onto the Stack**

The items are pushed onto the stack in the following order: **P, Q, R, S, T**. After pushing all items, the stack looks like this:

css

Copy code

Stack (top to bottom): T, S, R, Q, P

**Step 2: Pop Four Items from the Stack and Insert Them into the Queue**

We pop the top four items from the stack and insert them into the queue. The sequence of popping is **T, S, R, Q**.

* After popping **T**: Queue becomes [T]
* After popping **S**: Queue becomes [T, S]
* After popping **R**: Queue becomes [T, S, R]
* After popping **Q**: Queue becomes [T, S, R, Q]

Now, the stack only contains **P**:

vbnet

Copy code

Stack: [P]

Queue (front to back): T, S, R, Q

**Step 3: Dequeue Two Elements from the Queue and Push Them Back onto the Stack**

We dequeue the first two elements from the queue (i.e., **T** and **S**) and push them back onto the stack.

* Dequeue **T** and push it onto the stack: Stack becomes [P, T]
* Dequeue **S** and push it onto the stack: Stack becomes [P, T, S]

Now we have:

css

Copy code

Stack (top to bottom): S, T, P

Queue (front to back): R, Q

**Step 4: Pop One Item from the Stack**

**Final Answer**

The popped item is **S**.

1. ii)

The incorrect statement is: (ii) The Queue can be used to implement the least recently used (LRU) page fault algorithm and the Quicksort algorithm.

Explanation: Queues are not used for implementing Quicksort, which relies on a recursive or stack-based approach. Queues are, however, suitable for implementing the LRU page replacement algorithm.

1. ii)

The **3rd largest element** of the tree is stored at **index 249** in the array representation.

9)ii) The function fun **reverses the order of the elements in the queue** Q.

10)ii) **set Difference A−BA - BA−B**

The set difference A−BA - BA−B consists of the elements that are in AAA but not in BBB.

* A={1,2,3}A = \{1, 2, 3\}A={1,2,3}
* B={1,2,4}B = \{1, 2, 4\}B={1,2,4}

The elements in AAA that are not in BBB are:

A−B={3}A - B = \{3\}A−B={3}

**Cardinality of A−BA - BA−B**

The cardinality of A−BA - BA−B, denoted ∣A−B∣|A - B|∣A−B∣, is the number of elements in the set A−BA - BA−B, which is:

∣A−B∣=1|A - B| = 1∣A−B∣=1

**Conclusion**

The value of ∣A−B∣|A - B|∣A−B∣ is **1**.

11)ii) The post-order traversal of the binary search tree is:

11, 12, 10, 16, 19, 18, 20, 154

12)ii) 1

/ \

2 3

/ \

4 5

/ \

6 7

/ \

8 9

**Step 4: Calculate the Height of the Tree**

To calculate the height of the tree, we look for the longest path from the root to any leaf:

* The height of a node is the length of the longest path from that node to a leaf.
* For node **1**, the left subtree has a height of 4 (from node 1 → 2 → 4 → 6 → 8) and the right subtree has a height of 1 (from node 1 → 3).
* Therefore, the height of the tree is the maximum of the heights of the left and right subtrees plus 1 for the root node:  
  Height of tree=1+max⁡(4,1)=5\text{Height of tree} = 1 + \max(4, 1) = 5Height of tree=1+max(4,1)=5.

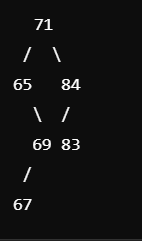
**Conclusion**

The height of the binary tree is **5**.

13)ii) **Use the formula I = 2L - 1 to find the number of internal nodes with two children.**

I=2×20−1=40−1=39

14)ii)



Ans: 67

15)ii)

The value of a+10b is **11**.

16)ii)

The first number to be inserted in the tree must be **n−p** in order for the right subtree of the root to contain exactly p nodes.

17)ii)

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

18)ii)

The time complexity of the strangeOrder(t) function for a binary search tree with **n** nodes is **O(n)**.

19)ii)

The number of possible values for k(the index of the maximum element) is **53**.

20)ii) **Steps:**

1. **Initialize Min-Heap**: Create an empty Min-Heap of size kkk.
2. **Process each element**:
   * For each element in the dataset:
     + If the heap contains fewer than kkk elements, insert the current element into the heap.
     + If the heap is full (contains kkk elements) and the current element is larger than the root of the heap (the smallest element in the heap):
       - Replace the root with the current element (this ensures the heap always holds the kkk largest elements).
3. **Return the heap**: Once all elements are processed, the Min-Heap will contain the top kkk largest elements.

21)ii)

function minMeetingRooms(meetings):

if meetings is empty:

return 0

// Step 1: Sort meetings by start time

sort(meetings, by\_start\_time)

// Step 2: Initialize a Min-Heap

MinHeap heap = new MinHeap()

// Step 3: Process each meeting

for each meeting in meetings:

if heap is not empty and heap.root() <= meeting.start:

// Reuse a room: Pop the earliest ending meeting

heap.pop()

// Add the current meeting's end time to the heap

heap.push(meeting.end)

// The size of the heap is the number of rooms required

return heap.size()

22)ii) 23

/ \

14 10

/ \ / \

6 13 12 1

/ \

5 9

/

8

**Step 2: Verify the Min-Max Heap Properties**

1. **Level 0 (Root)**: The root must hold the minimum element. Here, the root is **23**, but it should be the minimum element of the heap. So, the root violates the min-max heap property.
2. **Level 1 (Odd level)**: These should contain **maximum** elements. For the left child **14** and right child **10**:
   * **14** should be greater than both its children **6** and **13**, which it is.
   * **10** should be greater than both its children **12** and **1**, which it is not, as **1** is less than **10**.
3. **Level 2 (Even level)**: These should contain **minimum** elements. For each node:
   * **6** should be smaller than both its children **5** and **9**, but **6** is greater than **5**. This violates the min-max heap property.
   * **13** should be smaller than its only child **8**, which is true.
   * **12** should be smaller than its only child **8**, which is true.
   * **1** is correctly placed as a minimum element.

**Step 3: Rearrange the Array to Satisfy Min-Max Heap Property**

To fix the violations, let's reorder the elements to maintain the min-max heap properties.

* The minimum element should go to the root.
* Then, we need to ensure that every even-numbered level holds the minimum elements and every odd-numbered level holds the maximum elements.

**Step 4: Rearranged Min-Max Heap**

After rearranging, the correct min-max heap will look like this:

markdown

Copy code

1

/ \

6 10

/ \ / \

5 9 12 23

/ \

8 13

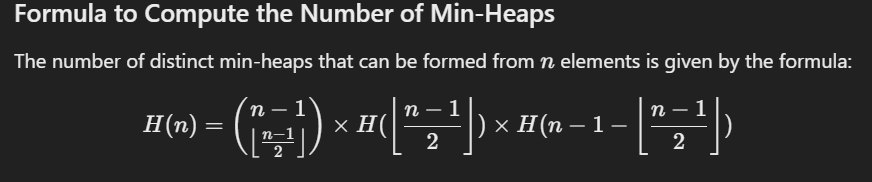
The new array representation is:

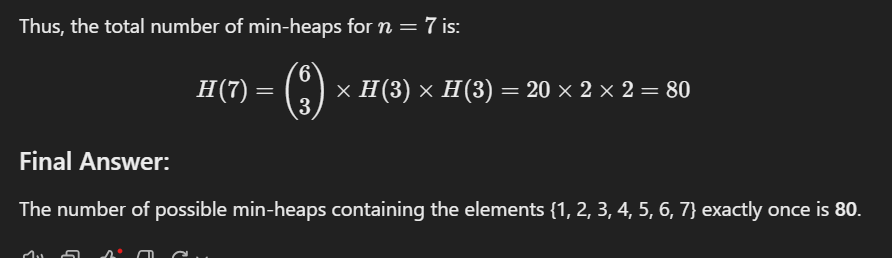
1, 6, 10, 5, 9, 12, 23, 8, 13

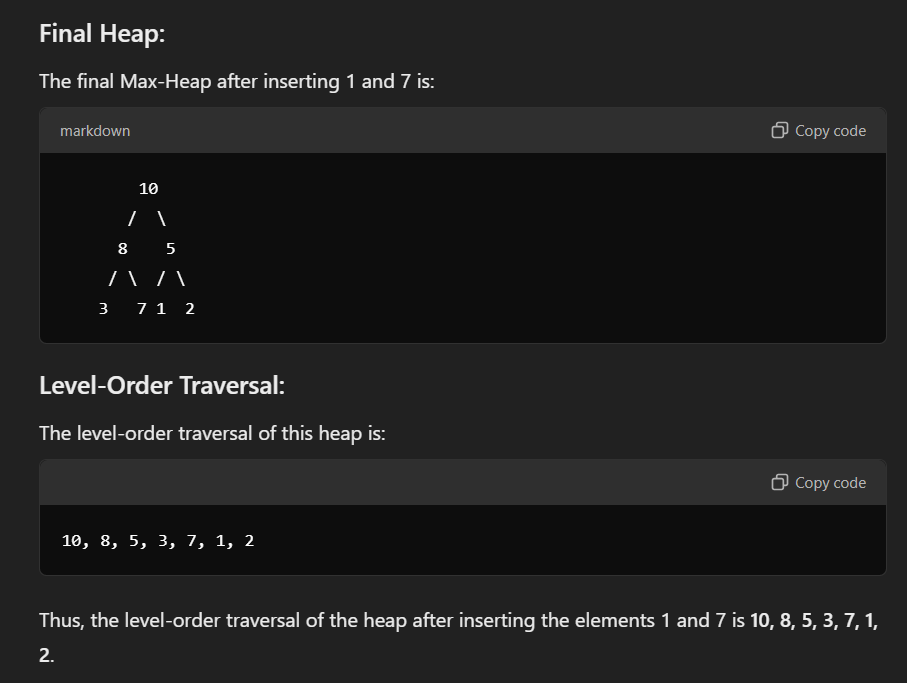
**Explanation of Rearranged Structure:**

* The root **1** is the minimum.
* The second level has **6** and **10**, both of which satisfy the maximum condition.
* The third level has **5**, **9**, **12**, and **23**, which satisfy the minimum condition for level 2, and the odd level (level 1) holds the maximum values **10**, **23**.

23)ii)





24)ii) 

25)ii)

function findKthSmallest(stream, k):

# Initialize an empty max-heap

MaxHeap heap

for each number in stream:

if heap.size() < k:

# Insert the number into the heap if the heap size is less than k

heap.insert(number)

else:

# If the heap already has k elements, check if the new number is smaller than the root

if number < heap.root():

# Replace the root (largest of the smallest k elements) with the new number

heap.pop() # Remove the largest element in the heap

heap.insert(number) # Insert the new number

# After processing all numbers in the stream, the root of the heap contains the k-th smallest element

return heap.root()

26)ii)

**Why an AVL Tree is Suitable for a Real-Time Leaderboard:**

1. **Balanced Structure**: AVL trees are self-balancing, ensuring **O(log n)** time complexity for insertion, deletion, and search operations, making frequent score updates efficient.
2. **Efficient Score Updates**: Insert and update operations (when a player’s score changes) are performed in **O(log n)** time, keeping the tree balanced.
3. **Quick Rank Retrieval**: The tree maintains sorted order, allowing efficient **rank queries** and **player search** in **O(log n)** time.
4. **Sorted Order**: The AVL tree ensures players are always sorted by score, with ties broken by player ID, ensuring the leaderboard remains up-to-date.
5. **Scalable**: Handles a large number of players and frequent updates efficiently due to logarithmic time complexity.

In essence, the AVL tree provides a **fast and balanced structure** for real-time leaderboard updates and queries.

27)ii)

he number of nodes that become unbalanced depends on the specific structure of the tree before the insertion. However, typically:

* **At least node "g" will become unbalanced**.
* **Possibly other ancestors of "g"** could become unbalanced, depending on the heights of their subtrees and whether they need to be rebalanced as the insertion propagates upwards.

Therefore, **at least 1 node** (the node "g") will become unbalanced, and possibly more if the balance factors of its ancestors are also affected.

28)ii)

**Step 1: Inserting 10**

1. **Insert 10**: Since the tree is empty, the first node inserted is 10. No imbalance occurs, and the tree remains balanced.
   * Tree after insertion:

Copy code

10

* + No rotations are required.

**Step 2: Inserting 20**

1. **Insert 20**: After inserting 20, the tree looks like this:

markdown

10

\

20

* + The balance factor of node 10 is now -1 (because the left subtree is empty, and the right subtree has one node). A balance factor of -1 is within the acceptable range for an AVL tree, so no rotations are needed.
  + Tree after insertion:

10

\

20

* + No rotations are required.

**Step 3: Inserting 30**

1. **Insert 30**: Now, we insert 30. The tree becomes:

markdown

10

\

20

\

30

* + The balance factor of node 10 is -2 (since the left subtree has height 0, and the right subtree has height 2). This is an imbalance, so we need to perform a rotation to restore balance.
  + Since the imbalance is on the right subtree of the right child (node 20), this is a **right-right (RR) case**, which requires a **left rotation** at node 10.

**Performing the Left Rotation**

Performing a **left rotation** at node 10 will make node 20 the new root, and 10 will become the left child of 20, while 30 remains the right child of 20. The tree becomes:

markdown

20

/ \

10 30

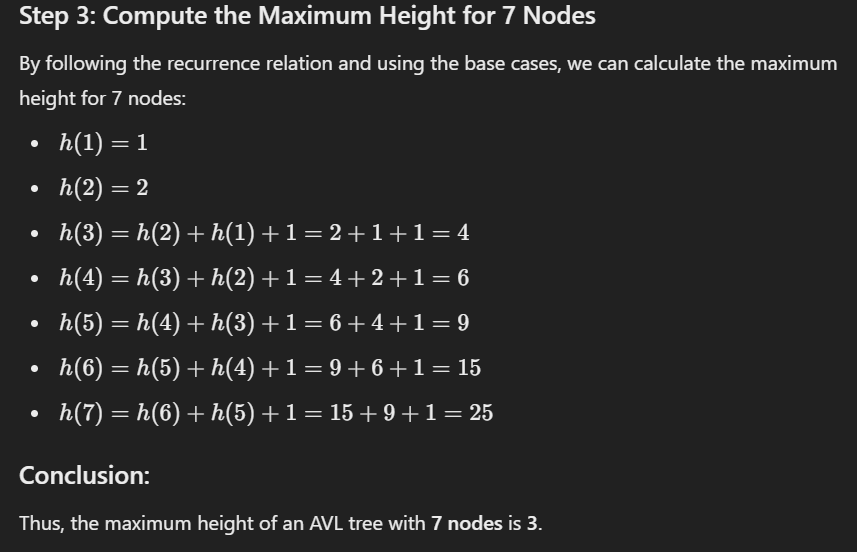
* After the left rotation, the tree is balanced, and no further rotations are required.

**Conclusion**

* **Total number of rotations**: 1 (a single left rotation at node 10).

Thus, **1 rotation** is required to balance the tree after inserting 10, 20, and 30.

29)ii)



30)ii)

**Initial Red-Black Tree:**

mathematica

Copy code

40 (Black)

/ \

20 (Black) 50 (Black)

/ \ \

10 (Red) 30 (Red) 60 (Red)

**Step 1: Insert node 25**

We insert node 25. Since 25 is greater than 20 but less than 30, it is inserted as the left child of node 30. The tree looks like this after the insertion:

40 (Black)

/ \

20 (Black) 50 (Black)

/ \ \

10 (Red) 30 (Red) 60 (Red)

/

25 (Red)

**Step 2: Violation of Red-Black Properties**

The insertion of node 25 creates a **red-red** violation, because both node 25 and its parent node 30 are red. According to red-black tree rules, no two adjacent red nodes are allowed. We need to fix this violation.

**Step 3: Fixing the Violation**

1. **Parent is Red, Uncle is Red**:
   * The parent of node 25 is node 30, and the uncle of node 25 is node 10 (the left child of node 20).
   * Both node 30 and node 10 are red, so we **perform a color flip** at node 20.
2. **Color Flip**:
   * Change node 20's color to red.
   * Change node 30's color to black.
   * Change node 10's color to black.

After the color flip, the tree looks like this:

40 (Black)

/ \

20 (Red) 50 (Black)

/ \ \

10 (Black) 30 (Black) 60 (Red)

/

25 (Red)

**Step 4: Check for Further Violations**

* Now, node 20 is red and its parent, node 40, is black.
* Since there are no more adjacent red nodes, the tree is balanced, and no further rotations or changes are necessary.

**Final Red-Black Tree:**

mathematica

40 (Black)

/ \

20 (Red) 50 (Black)

/ \ \

10 (Black) 30 (Black) 60 (Red)

/

25 (Red)

**Summary of Operations:**

1. **Insert node 25**: Inserted as the left child of node 30.
2. **Color Flip**: Performed at node 20 to fix the red-red violation.
   * Node 20 turned red.
   * Node 30 and node 10 turned black.
3. **Tree is balanced**: No further rotations were needed.

Thus, the sequence of operations involved one **color flip** at node 20, and the tree is now balanced. No rotations were required.

31)ii)

**Initial Red-Black Tree:**

30 (Black)

/ \

20 (Red) 40 (Black)

/ \ \

10 (Black) 25 (Black) 50 (Red)

**Step 1: Insert node 15**

We need to insert node 15 into the tree. Since 15 is less than 30, we move to the left child (node 20). Then, since 15 is less than 20, we move to the left child of node 20 (node 10). Finally, since 15 is greater than 10, we insert node 15 as the right child of node 10.

The tree after the insertion will look like this:

30 (Black)

/ \

20 (Red) 40 (Black)

/ \ \

10 (Black) 25 (Black) 50 (Red)

\

15 (Red)

**Step 2: Violation of Red-Black Properties**

Now, we have a red-red violation: both node 15 and its parent node 10 are red. According to red-black tree rules, no two adjacent red nodes are allowed. We need to fix this violation.

**Step 3: Fixing the Violation**

1. **Parent is Red, Uncle is Black**:
   * The parent of node 15 is node 10, and the uncle of node 15 is node 25.
   * Node 25 is black, so we need to **perform a rotation** to fix the red-red violation.
2. **Right-Left (RL) Case**:
   * Since node 15 is the **right child** of node 10, and node 10 is the **left child** of node 20, this creates a **Right-Left (RL) case**, which requires a **right rotation** at node 10 followed by a **left rotation** at node 20.

**Step 4: Perform the Right Rotation at Node 10**

Perform a right rotation at node 10 to handle the RL case. This will make node 15 the new parent of node 10.

Before the rotation:

10 (Black)

\

15 (Red)

After the right rotation:

15 (Red)

/

10 (Black)

**Step 5: Perform the Left Rotation at Node 20**

Now, we perform a left rotation at node 20 to complete the RL case fix. This will make node 15 the new parent of node 20.

Before the rotation:

20 (Red)

/ \

15 (Red) 25 (Black)

/

10 (Black)

After the left rotation:

15 (Red)

/ \

10 (Black) 20 (Black)

\

25 (Black)

**Step 6: Fix Color Violations**

After the rotations, we still need to fix the color violations:

1. The new root (node 15) must be **black** because the root of a red-black tree is always black.
2. We change the color of node 15 to black and the color of node 20 to red.

Final tree after the color fix:

15 (Black)

/ \

10 (Black) 20 (Red)

\

25 (Black)

**Final Tree:**

30 (Black)

/ \

15 (Black) 40 (Black)

/ \ \

10 (Black) 20 (Red) 50 (Red)

\

25 (Black)

**Summary of Operations:**

1. **Insertion**: Insert node 15 as the right child of node 10.
2. **Red-Red Violation**: A red-red violation occurred between node 15 and node 10.
3. **Rotation**: Perform a **right rotation** at node 10 and a **left rotation** at node 20 to fix the violation.
4. **Color Fix**: Change node 15 to black and node 20 to red.

After these operations, the tree is balanced and satisfies the red-black tree properties.

32)ii)

**Initial Red-Black Tree:**

18 (Black)

/ \

5 (Red) 20 (Black)

/ \ \

3 (Black) 8 (Black) 25 (Red)

We need to delete the node 20 and rebalance the tree.

**Step 1: Delete node 20**

We start by deleting node 20. When deleting a node in a Red-Black tree, we need to ensure that we maintain the tree's properties. In this case, node 20 has one child (node 25), so we can simply replace node 20 with node 25.

After this replacement, the tree looks like:

18 (Black)

/ \

5 (Red) 25 (Red)

/ \

3 (Black) 8 (Black)

**Step 2: Check for Red-Black Violations**

Now that we've replaced node 20 with node 25, we need to check if the Red-Black tree properties are violated. Specifically, we need to check if there are two consecutive red nodes or if the black height is unbalanced.

* Node 25 is red and its parent, node 18, is black, so there is no red-red violation.
* The black height is still consistent (each path from the root to a leaf has the same number of black nodes).

**Step 3: Balancing the Tree**

At this point, the tree does not violate any Red-Black properties, and no rotations or further color changes are required. Therefore, the tree remains balanced, and we have completed the deletion and rebalancing.

**Final Tree:**

18 (Black)

/ \

5 (Red) 25 (Red)

/ \

3 (Black) 8 (Black)

**Summary of Steps:**

1. **Delete node 20**: Replaced with node 25.
2. **Check for violations**: No violations were found, so no further rotations or color changes were needed.

Thus, the tree is now balanced and satisfies all Red-Black properties after the deletion of node 20.

33)ii)

**Initial Red-Black Tree:**

The tree starts empty.

**Insert 7:**

1. The tree is empty, so 7 becomes the root and must be **black**.

Tree after insertion:

7 (Black)

**Insert 3:**

1. Insert 3. Since 3 is less than 7, it becomes the left child of 7 and is **red** (all newly inserted nodes are red).

Tree after insertion:

7 (Black)

/

3 (Red)

**Insert 18:**

1. Insert 18. Since 18 is greater than 7, it becomes the right child of 7 and is **red**.

At this point, there is no red-red violation, because the parent (node 7) is black.

Tree after insertion:

7 (Black)

/ \

3 (Red) 18 (Red)

**Insert 10:**

1. Insert 10. Since 10 is less than 18 but greater than 7, it becomes the left child of 18 and is **red**.

Now, we have a **red-red** violation between nodes 10 and 18, where both nodes are red. To fix this:

* Perform a **left rotation** on node 18 to move 10 up and balance the tree.
* After the rotation, we change the color of 10 to black and 18 to red.

Tree after rotation and color fix:

7 (Black)

/ \

3 (Red) 10 (Black)

/ \

18 (Red) (Empty)

**Insert 22:**

1. Insert 22. Since 22 is greater than 10 and 18, it becomes the right child of 18 and is **red**.

There is no violation, and the tree remains balanced.

Tree after insertion:

7 (Black)

/ \

3 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

**Insert 8:**

1. Insert 8. Since 8 is greater than 3 but less than 10, it becomes the right child of 3 and is **red**.

Now, we have a **red-red** violation between nodes 3 and 8, where both nodes are red. To fix this, we need to perform rotations and color changes.

* First, perform a **right rotation** at node 7 to move node 3 up and make node 7 the right child of node 3.
* After the rotation, we need to adjust the colors:
  + The new root node (node 3) will be black.
  + Node 7 will be red, and node 10 will be black.

Tree after rotation and color fix:

3 (Black)

/ \

2 (Red) 7 (Red)

/ \

6 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

**Insert 11:**

1. Insert 11. Since 11 is greater than 10 but less than 18, it becomes the left child of 18 and is **red**.

Tree after insertion:

3 (Black)

/ \

2 (Red) 7 (Red)

/ \

6 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

/

11 (Red)

**Insert 26:**

1. Insert 26. Since 26 is greater than 22, it becomes the right child of 22 and is **red**.

Tree after insertion:

3 (Black)

/ \

2 (Red) 7 (Red)

/ \

6 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

/ \

11 (Red) 26 (Red)

**Insert 2:**

1. Insert 2. Since 2 is less than 3, it becomes the left child of 3 and is **red**.

Tree after insertion:

3 (Black)

/ \

2 (Red) 7 (Red)

/ \

6 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

/ \

11 (Red) 26 (Red)

**Insert 6:**

1. Insert 6. Since 6 is greater than 3 but less than 7, it becomes the right child of 2 and is **red**.

There is no red-red violation, and the tree remains balanced.

Tree after insertion:

3 (Black)

/ \

2 (Red) 7 (Red)

\ / \

6 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

/ \

11 (Red) 26 (Red)

**Insert 13:**

1. Insert 13. Since 13 is greater than 10 but less than 18, it becomes the left child of 18 and is **red**.

After this insertion, we have a red-red violation between nodes 13 and 18. To fix this, we perform a **color flip** at node 18. Node 13 becomes black, and node 18 becomes red.

Final tree after insertion:

3 (Black)

/ \

2 (Red) 7 (Red)

\ / \

6 (Red) 10 (Black)

/ \

18 (Red) 22 (Red)

/ \

11 (Red) 26 (Red)

\

13 (Black)

**Summary:**

* After each insertion, we checked the Red-Black tree properties and performed rotations (right or left) and color flips as needed.
* The steps for inserting 18 and 8 involved rotations and color changes to fix red-red violations and ensure the tree remained balanced and adhered to Red-Black tree properties.

34)ii)

**Initial Structure of the Splay Tree:**

We begin with a balanced binary search tree where the root is 40, and the nodes are arranged as follows:

40

/ \

20 60

/ \ / \

10 30 50 70

**Splay Operation for Element 20:**

The goal is to perform a search for the element 20, bringing it to the root using splay operations. Splay operations involve moving the accessed node (20) to the root by performing a series of tree rotations (zig, zig-zig, or zig-zag), depending on the node’s position and its parent’s position.

**Step-by-Step Splay Process:**

1. **Initial Tree**:

40

/ \

20 60

/ \ / \

10 30 50 70

1. **Step 1: Zig-Zig (since 20 is the left child of 40, and 20 is the left child of 40’s left child)**:
   * Perform a **right rotation** on 40. This brings 20 closer to the root.
   * The tree after this rotation:

20

/ \

10 40

/ \

30 60

/ \

50 70

1. **Step 2: Zig (since 20 is the left child of 40)**:
   * Perform another **right rotation** on 40, making 20 the root.
   * The tree after this rotation:

20

/ \

10 40

/ \

30 60

/ \

50 70

At this point, 20 is the root of the tree, and the search operation is complete.

**What becomes the right child of 20?**

After the splay operation, the right child of 20 is 40.

**Final Answer:**

The element that becomes the right child of 20 after the splay operation is **40**.

35)ii)

**Step 1: Inserting 15**

Since the tree is empty, 15 becomes the root.

15

**Step 2: Inserting 10**

Since 10 is less than 15, it becomes the left child of 15.

15

/

10

**Step 3: Inserting 20**

Since 20 is greater than 15, it becomes the right child of 15.

15

/ \

10 20

**Step 4: Inserting 25**

Since 25 is greater than 20, it becomes the right child of 20.

15

/ \

10 20

\

25

**Step 5: Inserting 5** Since 5 is less than 15 and less than 10, it becomes the left child of 10.

15

/ \

10 20

/ \

5 25

**Step 6: Inserting 1**

Since 1 is less than 15, less than 10, and less than 5, it becomes the left child of 5.

15

/ \

10 20

/ \

5 25

/

1

**Step 7: Inserting 8**

Since 8 is less than 15, greater than 5, it becomes the right child of 5.

15

/ \

10 20

/ \

5 25

/ \

1 8

**Final Tree Structure Before Splay Operation**

The tree after all the insertions is:

15

/ \

10 20

/ \

5 25

/ \

1 8

**Step 8: Splay Operation for 1**

Now, we perform the splay operation to bring 1 to the root. We will use rotations to move 1 up to the root.

1. **Initial Tree**:

15

/ \

10 20

/ \

5 25

/ \

1 8

1. **Step 1: Zig-Zig Rotation** (since 1 is the left child of 5, and 5 is the left child of 10):
   * Perform a **right rotation** on 5 to move 1 up:

15

/ \

10 20

/ \

1 25

\

5

\

8

1. **Step 2: Zig-Zig Rotation** (since 1 is still the left child of 10):
   * Perform another **right rotation** on 10 to move 1 up:

15

/ \

1 20

\ / \

5 10 25

\ \

8 8

1. **Step 3: Final Rotation**:
   * Perform a **right rotation** on 15 to move 1 to the root:

1

/ \

5 15

/ \ / \

8 10 20 25

**Final Tree Structure After Splay Operation**

After the splay operation for the value 1, the final tree structure is:

1

/ \

5 15

/ \ / \

8 10 20 25

**Summary of Steps:**

1. Inserted values: 15, 10, 20, 25, 5, 1, 8.
2. The tree structure after all insertions.
3. Performed a splay operation for 1, bringing it to the root using a series of **right rotations**.

This is the final tree after performing the splay operation for the value 1.

36)ii)

**Initial Splay Tree TTT:**

The tree TTT starts with the following nodes:

10

/ \

5 15

/ \ \

2 8 20

/ \

6 9

Now, we'll perform the two operations one by one.

**1. Search for the item with key 9**

To splay the tree, we need to perform rotations to bring the item 9 to the root. We perform a sequence of rotations based on the tree structure and the key we are searching for.

**Step 1: Search for 9**

* 9 is the right child of 8 and is greater than 8.
* 9 is the right child of 6 and is greater than 6.
* 9 is the left child of 10.

The sequence of operations is:

* **Zig-Zig rotation** (Right-Right): First, we perform a right rotation on 8 to move 9 up, and then a right rotation on 6 to bring 9 to the root.

After these rotations, the tree becomes:

9

/ \

8 10

/ / \

6 5 15

/ / \

2 20 20

So, after searching for the key 9, the resulting splay tree is:

9

/ \

8 10

/ / \

6 5 15

/ / \

2 20 20

**2. Remove the item with key 6**

After splaying 9, now we need to remove the node with key 6.

**Step 1: Remove 6**

* The node with key 6 is a leaf node.
* We can simply remove it.

After removal, the tree becomes:

9

/ \

8 10

/ / \

2 5 15

\

20

Thus, after removing the node with key 6, the resulting splay tree is:

9

/ \

8 10

/ / \

2 5 15

\

20

**Summary of Steps:**

1. **Search for 9**: The tree was modified with a **Zig-Zig rotation** to bring 9 to the root.
2. **Remove 6**: The node 6 was removed, leaving the tree as shown above.

**37)ii)**

**Initial Splay Tree:**

5

/ \

3 6

/ \ \

1 4 7

/ \ \ \

0 2 8 9

**Step 1: Access Node 2**

We perform a **splay operation** for the node 2, which will bring it to the root.

1. 2 is the right child of 1, and 1 is the left child of 3. This is a **Zig-Zig** case, so we first perform a **right rotation** on 1 to move 2 up, then another **right rotation** on 3 to move 2 to the root.

After these rotations, the tree becomes:

2

/ \

1 5

/ \ / \

0 3 4 6

\

7

\

8

**Step 2: Access Node 4**

Now we perform a **splay operation** for the node 4, which will bring it to the root.

1. 4 is the right child of 3, and 3 is the left child of 5. This is a **Zig-Zig** case, so we first perform a **right rotation** on 3 to move 4 up, then another **right rotation** on 5 to bring 4 to the root.

After these rotations, the tree becomes:

4

/ \

2 5

/ \ / \

1 3 6 7

/ \

0 8

**Step 3: Access Node 0**

Next, we perform a **splay operation** for the node 0.

1. 0 is the left child of 1, and 1 is the left child of 2. This is a **Zig-Zig** case, so we first perform a **right rotation** on 1 to move 0 up, then another **right rotation** on 2 to move 0 to the root.

After these rotations, the tree becomes:

0

/ \

2 4

/ \ / \

1 3 5 6

\

7

\

8

**Step 4: Access Node 6**

Now we perform a **splay operation** for the node 6.

1. 6 is the right child of 5, and 5 is the right child of 4. This is a **Zig-Zig** case, so we first perform a **left rotation** on 5 to move 6 up, then another **left rotation** on 4 to bring 6 to the root.

After these rotations, the tree becomes:

6

/ \

4 7

/ \ \

2 5 8

/ \

1 3

/

0

**Step 5: Access Node 1**

Finally, we perform a **splay operation** for the node 1.

1. 1 is the left child of 2, and 2 is the left child of 4. This is a **Zig-Zig** case, so we first perform a **right rotation** on 2 to move 1 up, then another **right rotation** on 4 to bring 1 to the root.

After these rotations, the tree becomes:

1

/ \

0 2

/ \

4 6

/ \ / \

3 5 7 8

**Final Result:**

After performing all the splay operations in the order 2, 4, 0, 6, and 1, the final splay tree is:

1

/ \

0 2

/ \

4 6

/ \ / \

3 5 7 8

This is the final structure of the splay tree after all the operations.

38)ii)

In a Leftist Heap, the **rank** of a node is the length of the shortest path to a leaf. This structure keeps the left subtree heavier than the right, resulting in a **right-skewed** shape for efficient merging.

Given a minimum of 6 and a maximum of 20, the root will contain 6. Due to the heap's right-skewed nature, the **rank of the root** is likely **2 or 3**, as the heap keeps right paths short to optimize merges.

**Justification**:

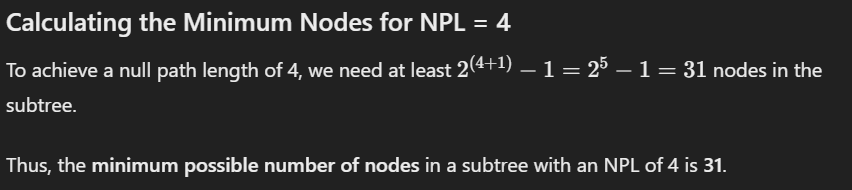
1. The right path in Leftist Heaps is designed to be shorter, so the rank of the root is often kept low.
2. Since the heap has a limited number of elements, with only values ranging from 6 to 20, the maximum depth of the tree is also limited.
3. Therefore, the rank of the root in this Leftist Heap should be around **2 or 3**, depending on the exact configuration of the tree and the specific values in the left and right subtrees.

**Influence of Rank on Heap Balance**

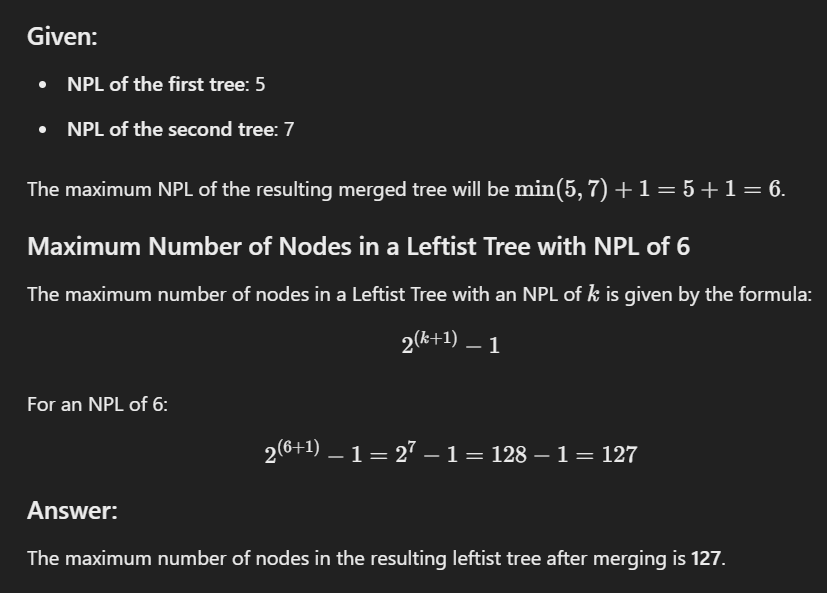
The rank influences the shape and balance of the heap:

* Nodes on the **right path** from the root to the minimum-ranked leaf are likely to have a rank of **1** or **2** because the path is shorter.
* Nodes on the **left path** will generally have higher ranks, as the heap’s leftist nature keeps the left subtree deeper than the right.

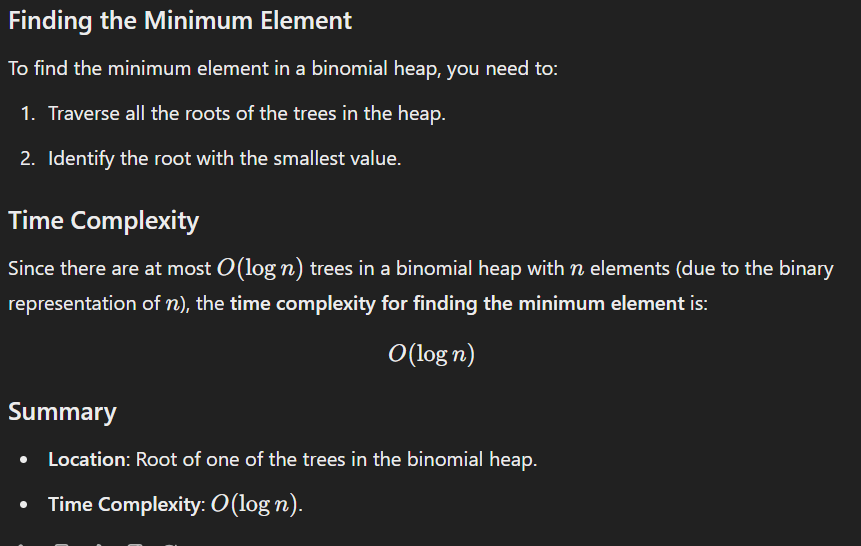
39)ii)



40)ii)



41)ii)



42)ii)

steps for merging two binomial heaps:

1. **Align Trees by Order**: Arrange trees in both heaps by their order.
2. **Merge Trees of the Same Order**:
   * Starting from the smallest order, combine trees of the same order as in binary addition.
   * If only one tree of a certain order exists, add it directly to the result.
   * If two trees of the same order exist, merge them to form a tree of the next higher order, carrying it forward.
3. **Update the Resulting Heap**: Continue this process for all orders, creating a single binomial heap with unique tree orders.

**Example Outline**

Suppose we have two binomial heaps, Heap A and Heap B, with trees of orders 0,1,0, 1,0,1, and 333 in Heap A, and trees of orders 1 and 2 in Heap B:

* Begin with the lowest order (order 000), add Heap A’s tree of order 000 to the resulting heap.
* For order 1, both heaps have a tree, so we merge them to form a tree of order 2 and carry it.
* For order 2, Heap B has a tree, and we also have a carry from order 1. We merge these to form a tree of order 3.
* Finally, for order 3, we merge the resulting order 3 trees from Heap A and the carry to get an order 4 tree.

The final heap is balanced with trees in increasing order by size, maintaining the binomial heap properties.

**Summary**

Merging binomial heaps is efficient because:

* It involves tree-by-tree merging in increasing order.
* Trees of the same order are combined as in binary addition, ensuring there’s only one tree of each order in the final heap.
* The operation is O(logn), making it well-suited for systems needing efficient queue merges, like a high-load server system.

43)ii)

In a complete binary search tree represented as a binary heap array, the elements are stored in an in-order traversal format by increasing values. This means the largest elements will be located toward the end of the array.

For a complete binary search tree with **1000 elements**:

1. The largest element is at index **999** (since array indexing starts at 0).
2. The second-largest element is at index **998**.
3. The third-largest element is at index **997**.

**Answer:**

The **3rd largest element** is stored at **index 997**.