DAA-Assignment 3

Md Adnan Jakati, M Sai Nishanth, Praneeth Kumar T

mj3184 - sm11326 - pt2427

Answer 1 a)

Binary Min-Heap Insertions:

1. Insert 60:
2. Insert 93:
3. Insert 25:
4. Insert 29 (After 4th Insertion):
5. Insert 58:
6. Insert 70:
7. Insert 62:
8. Insert 43(After 8th Insertion):
9. Insert 45:
10. Insert 17:
11. Insert 89:
12. Insert 83(After 12th Insertion):
13. Insert 44:
14. Insert 30:
15. Insert 15(After Final Insertion):

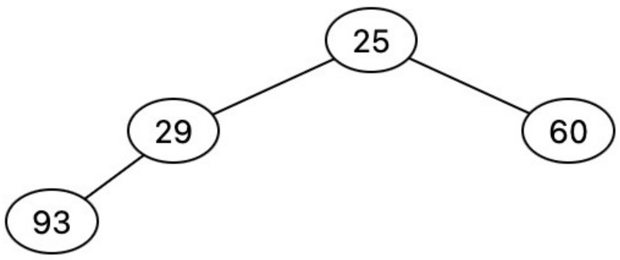
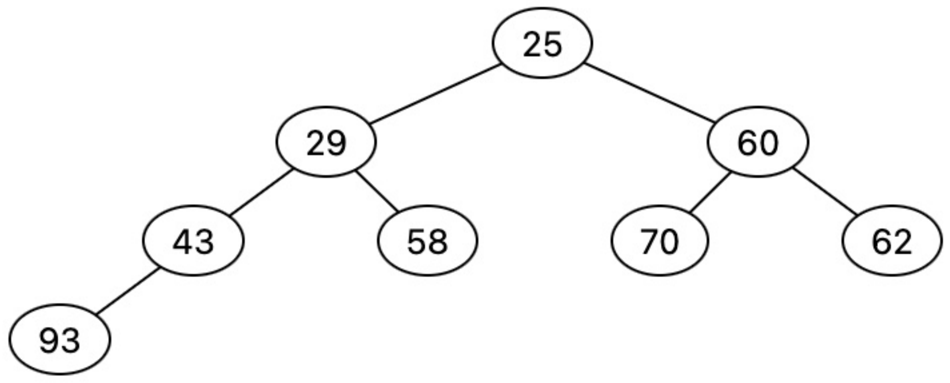
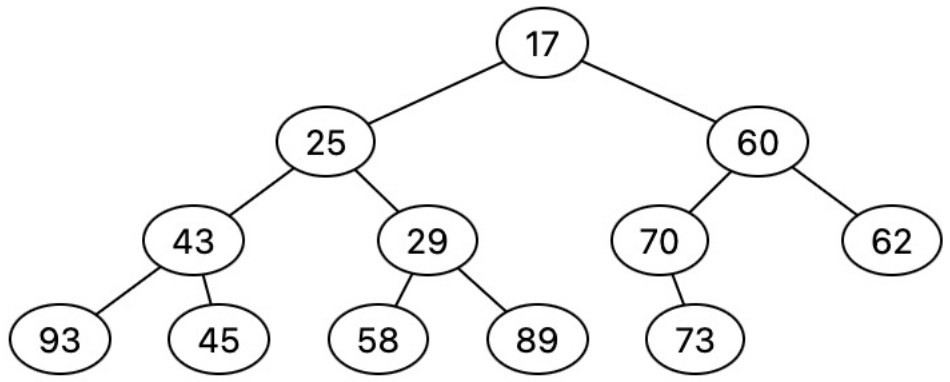


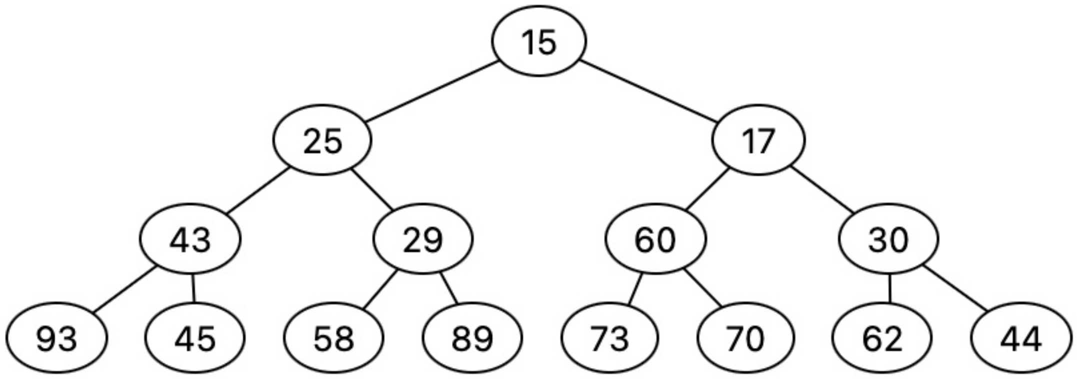
Figure 1: After 4 insertions



After 8 Insertions.



After 12 Insertions



After 15 Insertions – Final Min Heap.

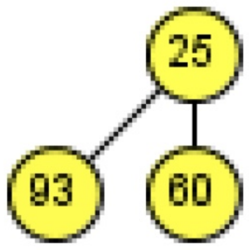


Figure 1: After 3 insertions

Answer 1 b)

Ternary Min-Heap Insertions:

1. Insert 60:
2. Insert 93:
3. Insert 25: (3rd Insertion)
4. Insert 29:
5. Insert 58:
6. Insert 70: (6th Insertion)
7. Insert 62:
8. Insert 43:
9. Insert 45: (9th Insertion)
10. Insert 17:
11. Insert 89:
12. Insert 83: (12th Insertion)
13. Insert 44:
14. Insert 30:
15. Insert 15(15th Insertion)

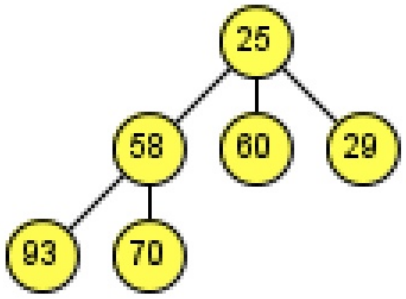


Figure 2: After 6 Insertions

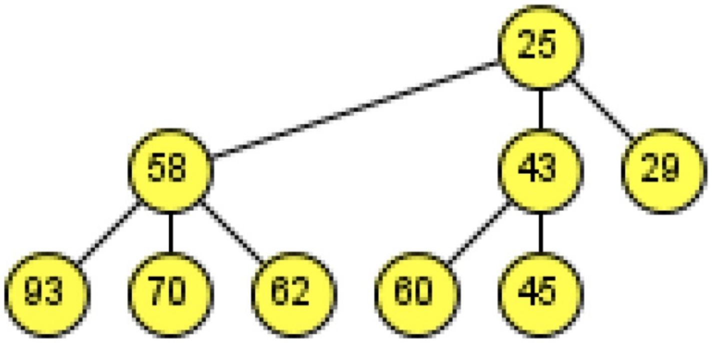


Figure 3: After 9 Insertions

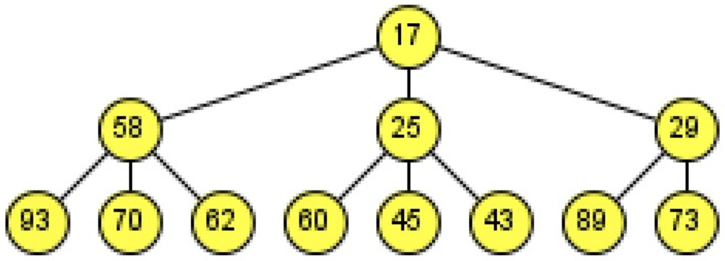


Figure 4: After 12 Insertions

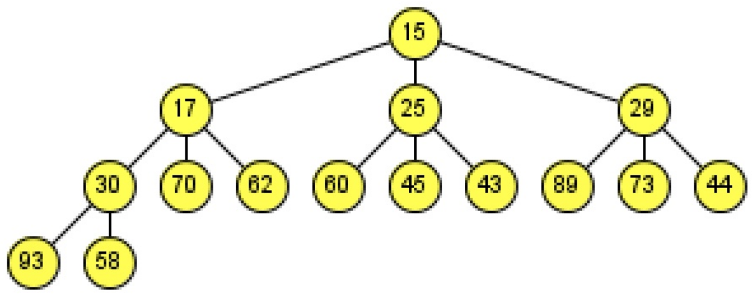
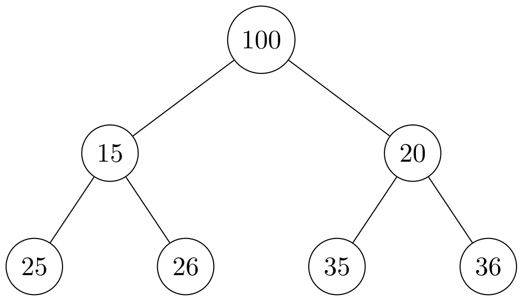


Figure 5: After 15 Insertions

Solution 2

(i)"7-Element" Example of a Confused Heap



(ii)In this Confused Heap - The element with the largest key (100) is at level 0 (the root), so it takes $O(1)$ time to find it. - The element with the smallest key (10) is at the maximum depth of the heap (level 2), which takes $O(\log n)$ time to find in the worst case.

(iii)Confused-Heap-Insert(*heap*, *element*):

1. Insert the *element* as the last element of the heap.
2. Check the level of the newly inserted *element*:
   - If it's at an even level, compare its key with its parent's key. If its key is smaller than its parent's key, swap the element with its parent and repeat this process until the ordering condition is satisfied.
   - If it's at an odd level, compare its key with its parent's key. If its key is larger than its parent's key, swap the element with its parent and repeat this process until the ordering condition is satisfied.

The worst-case time complexity for the Confused-Heap-Insert operation is $O(\log n)$, where 'n' is the number of elements in the heap. This is because the insertion process may require traversing the height of the heap, and the height of a binary heap is **O(logn)**

Algorithm 1 Confused Heap Insertion

```
Input: Confused heap $H$, Element $x$ to insert
Output: Confused heap $H$ with $x$ inserted
 1: Insert the new element $x$ as the last element in the heap.
 2: Initialize a pointer *current* to $x$.
 3: while *current* has a parent and violates the confused heap property: do
 4:   if *current* is at an even level (level 0, 2, 4, ...): then
 5:     Compare the key of *current* with the key of its parent.
 6:     if The key of *current* is smaller than the key of its parent: then
 7:       Swap *current* with its parent.
 8:       Update *current* to be the parent.
 9:     else
10:       Stop.
11:     end if
12:   else
13:     If *current* is at an odd level (level 1, 3, 5, ...):
14:     Compare the key of *current* with the key of its parent.
15:     if The key of *current* is larger than the key of its parent: then
16:       Swap *current* with its parent.
17:       Update *current* to be the parent.
18:     else
19:       Stop.
20:     end if
21:   end if
22: end while=0
```

Solution 3

Algorithm 2 Finding the kth Smallest Element in a Binary Min-Heap

```
0: Initialize $count$ to 0
0: Create a priority queue (min-heap) $Q$
0: Insert (root_value, 1) into $Q$, where root_value is the value of the root, and 1 is its index
0: while $count < k - 1$ do
0:   Extract the minimum element (min_value, min_index) from $Q$
0:   Increment $count$
0:   if min_index has a left child then
0:     Insert (left_child_value, left_child_index) into $Q$, where left_child_value is the value of the left child, and left_child_index is its index
0:   if min_index has a right child then
0:     Insert (right_child_value, right_child_index) into $Q$, where right_child_value is the value of the right child, and right_child_index is its index
0: The $k$th smallest value is min_value =0
```

In an implicit binary min-heap, the root contains the smallest value, and the values increase as you move down the heap. We want to find the kth smallest value without modifying the heap.

In an implicit binary min-heap, the kth smallest value can be located anywhere in the heap, depending on the distribution of values in the heap. However, we can make some general observations:

If k is 1, the kth smallest value is at the root of the heap since the root contains the smallest value.

If k is 2, the kth smallest value can be in one of the two child nodes of the root, depending on their values. The smaller of the two children will contain the kth smallest value.

As k increases, the kth smallest value can be found at deeper levels of the heap. It can be in the left or right subtrees of the root or in any of their children's subtrees.

The exact location of the kth smallest value within the heap can vary widely depending on the specific values in the heap. The algorithm you described finds the kth smallest value by exploring the heap in a way that ensures it is found correctly, regardless of its specific location.

Algorithm Description

1. Initialize a variable *count* to 0 and a priority queue (min-heap) Q. The priority queue will hold pairs (value, index), where value is the element's

value, and index is the index of the element in the implicit heap. Start by inserting the pair (root_value, 1) into the priority queue, where root_value is the value of the root element, and 1 is its index (assuming the root has index 1).

2. While *count* $< k - 1$, repeat the following steps:
   (a) Extract the minimum element (min_value, min_index) from the priority queue.
   (b) Increment *count*.
   (c) If the element at index min_index has a left child, insert the pair (left_child_value, left_child_index) into the priority queue, where left_child_value is the value of the left child, and left_child_index is its index.
   (d) If the element at index min_index has a right child, insert the pair (right_child_value, right_child_index) into the priority queue, where right_child_value is the value of the right child, and right_child_index is its index.
3. Once *count* reaches $k - 1$, the priority queue will contain the kth smallest value at the front of the queue.

Algorithm Complexity

This algorithm is efficient because it only explores the first $k - 1$ levels of the implicit binary heap, and it doesn't depend on the total number of elements n. Therefore, its worst-case running time is **O(k*logk)** because the priority queue operations (insertion and extraction of minimum) are logarithmic in the size of the priority queue, which has at most k elements in this case.

Solution 4

Recurrences using the "Baby" Master's Theorem

(a) $T(n) = 9T(n/3) + n$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 9$, $b = 3$, and $d = 1$. Since $a(9)$ is greater than $b^d(3^1)$, it falls under the case where $a > b^d$. Therefore, the solution is $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$.

(b) $T(n) = 3T(n/9) + 1$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 3$, $b = 9$, and $d = 0$. Since $a(3)$ is greater than $b^d(9^0)$, it falls under the case where $a > b^d$. Therefore, the solution is $T(n) = \Theta(n^{\log_9 3}) = \Theta(n^{0.5})$.

(c) $T(n) = 5T(n/5) + n^2$

This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 5$, $b = 5$, and $d = 2$. Since $a(5)$ is less than $b^d(5^2)$, it falls under the case where $a < b^d$. Therefore, the solution is $T(n) = \Theta(n^2) = \Theta(n^2)$.

(d) $T(n) = 25T(n/5) + n^3$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 25$, $b = 5$, and $d = 3$. Since $a(25)$ is lesser than $b^d(5^3)$, it falls under the case where $a < b^d$. Therefore, the solution is $T(n) = \Theta(n^3)$.

(e) $T(n) = 7T(n/7) + n$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 7$, $b = 7$, and $d = 1$. Since $a(7)$ is equal to $b^d(7^1)$, it falls under the case where $a = b^d$. Therefore, the solution is $T(n) = \Theta(n^d \log n) = \Theta(n \log n)$.

(f) $T(n) = 7T(n/49) + n$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 7$, $b = 49$, and $d = 1$. Since $a(7)$ is less than $b^d(49^1)$, it falls under the case where $a < b^d$. Therefore, the solution is $T(n) = \Theta(n^d) = \Theta(n)$.

(g) $T(n) = 121T(n/11) + n^2$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 121$, $b = 11$, and $d = 2$. Since $a(121)$ is equal to $b^d(11^2)$, it falls under the case where $a = b^d$. Therefore, the solution is $T(n) = \Theta(n^2 \log n))$.

(h) $T(n) = 9T(n/3) + 1$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 9$, $b = 3$, and $d = 0$. Since $a(9)$ is greater than $b^d(3^0)$, it falls under the case where $a > b^d$. Therefore, the solution is $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$.

(i) $T(n) = 3T(n/9) + \sqrt{n}$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 3$, $b = 9$, and $d = 0.5$. Since $a(3)$ is equal to $b^d(9^{0.5})$, it falls under the case where $a = b^d$. Therefore, the solution is $T(n) = \Theta(n^d \log n) = \Theta(\sqrt{n} \log n)$.

(j) $T(n) = 13T(n/13) + 1$
This recurrence is in the form $T(n) = aT(n/b) + \Theta(n^d)$ with $a = 13$, $b = 13$, and $d = 0$. Since $a(13)$ is greater than $b^d(13^0)$, it falls under the case where $a > b^d$. Therefore, the solution is $T(n) = \Theta(n^{\log_1 313}) = \Theta(n)$.

Solution 5

1. Define two classes: `TreeNode` to represent the binary tree nodes and `SubtreeInfo` to store information about a subtree.

   ```
   class TreeNode:
       def __init__(self, value):
           self.value = value
           self.left = None
           self.right = None

   class SubtreeInfo:
       def __init__(self, root=None, height=0, is_complete=False):
           self.root = root  # Root of the largest complete subtree
           self.height = height  # Height of the largest complete subtree
           self.is_complete = is_complete  # Whether the subtree is complete
   ```

2. Create a recursive function `largestCompleteSubtree(root)` that takes the root of the binary tree as input.

3. In the `largestCompleteSubtree` function:

   (a) Check if the root is `None`, and if so, return an empty `SubtreeInfo` object.

   (b) Recursively compute information for the left and right subtrees by calling `largestCompleteSubtree(root.left)` and `largestCompleteSubtree(root.right)`.

   (c) Check if the current subtree rooted at `root` is complete by verifying that both left and right subtrees are complete and have the same height.

   (d) Calculate the height of the current subtree as the maximum of the heights of the left and right subtrees, plus one.

   (e) Determine the root of the largest complete subtree:

   - If the current subtree is complete, set it as the largest complete subtree.
   - Otherwise, choose the largest complete subtree from either the left or right subtrees based on their heights.

   (f) Return a `SubtreeInfo` object containing the root of the largest complete subtree, its height, and whether it is complete.

4. Finally, you can use the `largestCompleteSubtree` function to find the largest complete subtree of the given binary tree.

The time complexity of this algorithm is $O(n)$, where n is the number of nodes in the binary tree, as it visits each node exactly once in a bottom-up manner.

Algorithm 3 Compute the Largest Complete Subtree

```
Require: Root of the binary tree: $root$
Ensure: Root and height of the largest complete subtree
0: procedure LargestCompleteSubtree($root$)
0:   if $root$ is null then
0:     return SubtreeInfo(null, 0, False) {Empty subtree}
0:   $leftInfo \leftarrow$ LargestCompleteSubtree($root.left$)
0:   $rightInfo \leftarrow$ LargestCompleteSubtree($root.right$)
0:   $isComplete \leftarrow leftInfo.isComplete$ and $rightInfo.isComplete$ and
   $leftInfo.height = rightInfo.height$
0:   $height \leftarrow \max(leftInfo.height, rightInfo.height) + 1$
0:   if $isComplete$ then
0:     return SubtreeInfo($root$, $height$, True)
0:   else if $leftInfo.height > rightInfo.height$ then
0:     return $leftInfo$
0:   else
0:     return $rightInfo$
   =0
```