N. Chaitanya Durgesh
CWID: A20561894

CS - 430
MID - EXAM

1. Given that,

- The LuckyPick algorithm gaurantees that the selected pivot will be between the 33^{rd} & 75^{th} percentiles of the sorted values in the subarray $A[p....r]$.

- In worst case, the partioning step will divide the subarray into two parts, where one part contains has 67% & remaining has 33% of elements.

Consider, size of subarray as 'n'

Worst case time complexity of recurrence relation $T(n)$ is

$$\cancel{T(n) = T(0.67n) + T(0.33n) + O(1)}$$

Recurrance relation ~~Master~~ theorem : $T(n) = aT(n/b) + f(n)$

$b = 4/3 = 1.33$

$\Rightarrow a = 2$

$b = 1.33$

$f(n) = O(1)$

$\log_{1.33} 2 \approx 1.33$

Applying Master Theorem:

$$f(n) = \Theta(n^{\log_b a}), \text{ then } T(n) = \Theta(n^{\log_b a})$$

$$f(n) = \Theta(1) = O(n^{\log_{1.33} 2 - \epsilon}) \quad \text{for } \epsilon > 0$$

$$\cdot f(n) = \Theta(1) \text{ which is } O(n^{1.33 - \epsilon}) \quad \text{for } \epsilon > 0$$

we get, $T(n) = \Theta(n^{\log_{1.33} 2})$

$\therefore$ The worst case time complexity of alternate version of Quicksort that uses the Lucky Pick algorithm to select the pivot is

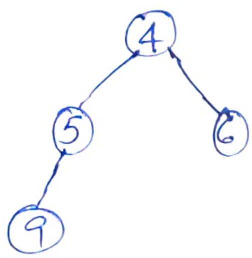
$$T(n) = \Theta(n^{\log_{1.33} 2}) = \Theta(n^{1.33})$$

2. Jason suggested that, we can use Merge (A_1, A_2) algorithm to merge two sorted arrays.

- However, this approach does not work, because it fails to preserve the ~~min-heap~~ MIN-HEAP property.

- Let's, consider an example;

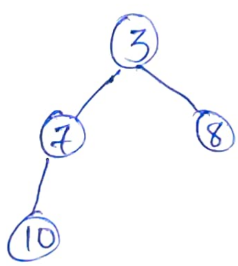
⇒ Heap A_1 :



Array representation :

4	5	6	9

⇒ Heap A_2 :



Array representation :

3	7	8	10

* Merge Algorithm :

- Using Jason's method we get,

4	5	6	9	3	7	8	10

we know that this does not represent a binary min-heap. As it violates Heap Properties.

① Heap Property:

- Element at index 1 (5) has parent at index 0 (4) $\boxed{\text{True}}$
- Element at index 2 (6) has parent at index 0 (4) $\boxed{\text{True}}$
- Element at index 3 (9) has parent at index 1 (5) $\boxed{\text{True}}$
- Element at index 4 (3) has parent at index 1 (5) $\boxed{\text{False}}$

∴ This violates the heap property since $3 < 5$.

Similarly other elements also violate the Heap property.

⟹ CORRECT APPROACH

① Combined array:

4	5	6	9	3	7	8	10

② Heapify:

- Element at index 3 (9), has no children
- Element at index 2 (6), compare with index 5 & index 6

  No swap required.

- Element at index 1 (5), compare with index 3, index 4

  * Swap 5 with 3.
  * New array

4	3	6	9	5	7	8	10

- Element at index 0 (4), compare with index 1 & index 2

  * Swap 4 with 3
  * New array

3	4	6	9	5	7	8	10

⇒ New array satisfies the MIN-HEAP property

Therefore, we can conclude that Jason's suggestion does not work.

3. a) Pseudo-code for $O(h)$ - time algorithm NextInPost (x) that outputs the next node to "visit" after x in a POSTORDER - TREE - WALK (T.root) of a BST 'T', where h is height of T.

Pseudo-code:

```
NextInPost (x):
    if x.right != NULL:
        y = x.right              # if right child exists, we visit leftmost node
                                   in right subtree
        while y.left != NULL:
            y = y.left
        return y

    else:
        y = x                    # if right child doesn't exist, we find first
        z = y.parent             # ancestor of x; whose left child is ancestor of x
        while z != NULL and y == z.right:
            y = z
            z = z.parent
        return z.
```

Therefore, the time complexity of NextInPost (x) is $O(h)$, where h is height of T.

- In worst case scenario, we need to traverse up tree from x to root.

3. b) To analyze the total running time of 1 call of TREE-MINIMUM (T. root) by successive calls of NextInPost(.) in the worst case, we will consider height of tree.

⇒ Time complexity for TREE-MINIMUM : $O(h)$

* For, n-1 successive calls of NextInPost(.), total running time is $O(nh)$.

Therefore, total running time for worst case is $O(h) + O(nh) = O(nh)$

⇒ 'n' is number of nodes in BST 'T'
⇒ 'h' is height of 'T'

Using Big-Theta notation, the total running time is $\boxed{\Theta(nh)}$.

4. a)

RB (x) = (the quantity of red nodes in subtree root x)

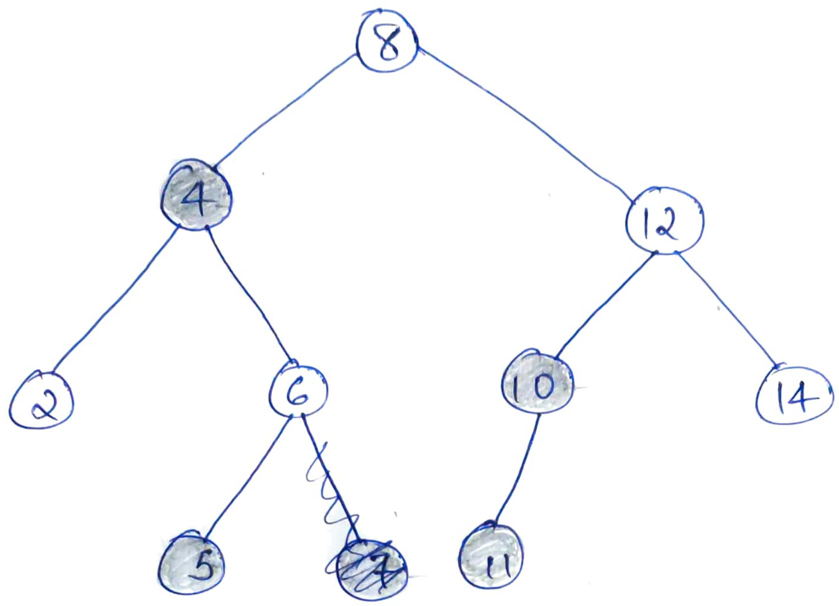
− (quantity of black nodes in subtree rooted at x)

a) Yes, we can maintain red balances in a red-black tree as node char. while maintaining $O(\log n)$, insertion & deletion efficiency.

- The red balance can be maintained in a red-black tree as a node attribute, with additional $O(1)$ time overhead, for each local operation.
- Insertion & deletion involve $O(\log n)$ for operations, so the overall time complexity remain $O(\log n)$.
  - So, it will not affect $O(\log n)$ performance of insertion & deletion.

- Maintaining red balance during deletion require updating the red balance for affected nodes. Each rotation & recoloring step requires $O(1)$ time, as the no. of steps is $O(\log n)$ in worst case, the red balance can be updated quickly.

b). Red - Black tree with maximum possible Red - balance on the root.



Here; shaded regions are red, others are black

In this tree, it has red balance of 3.

RB(x) = (quantity of red nodes in subtree root x) - (Quantity of black nodes in subtree root x).

Root : RB (8) = ~~[illegible]~~ 4 - 1 = 3

Left subtree rooted at '4', = 3 - 1 = 2

Right subtree rooted at '12', = 2 - 1 = 1

Therefore, this tree configuration maximizes the red - balance on the root '8' with '3' = RB(x).