X-Fast Trie, Dynamic Ordering Scheme Transpose, OBST and Splay Tree

Data Structures - Assignment 3

Winter 2022

1. Question

[15 marks] Draw the x-fast trie that maintains the following set S = {0, 2, 8, 11, 14} in universe {0,1,2,...,15}. Thus, n = 5 and u = 16.

- Your solution should include the tree structure, the descendant links, values stored in the leaves, and pointers between leaves.
- To show the dynamic perfect hash table constructed, simply give all the keys stored in the hash table.

Solution:

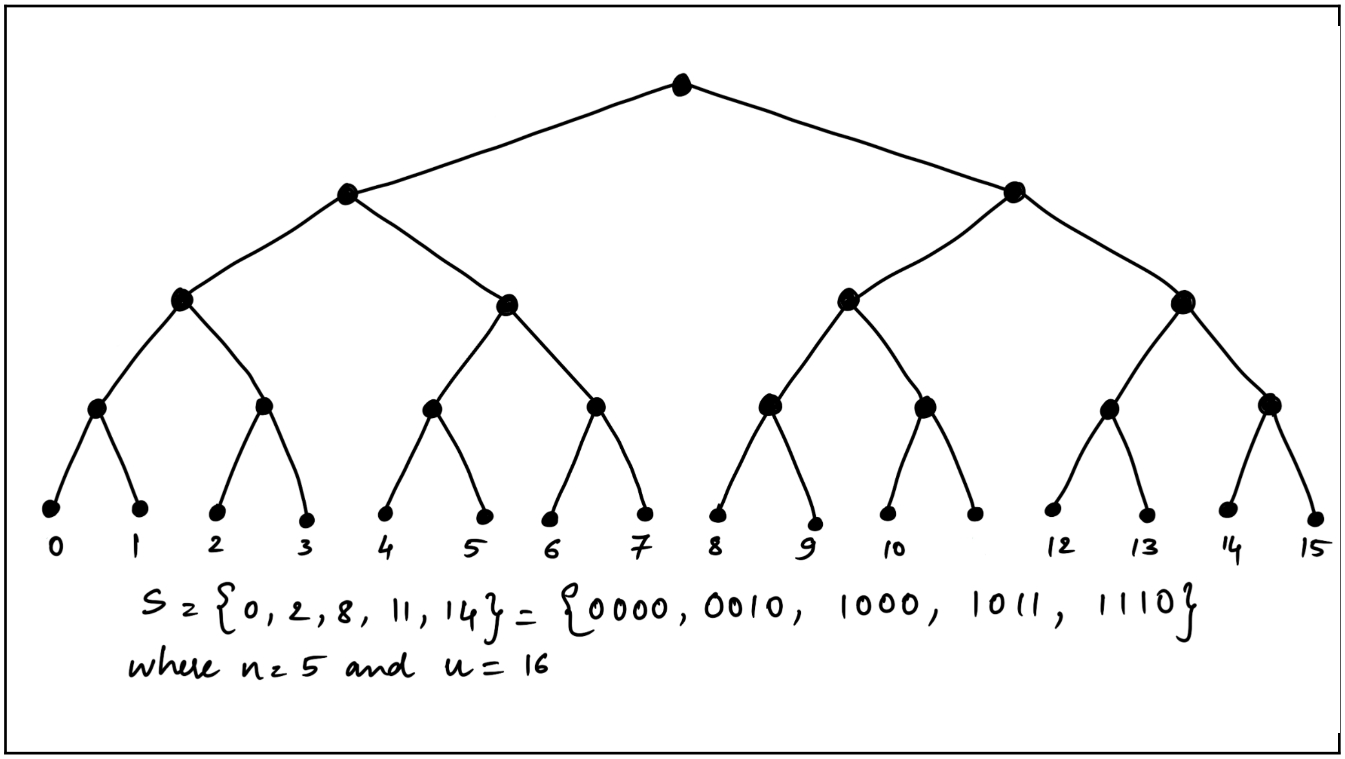
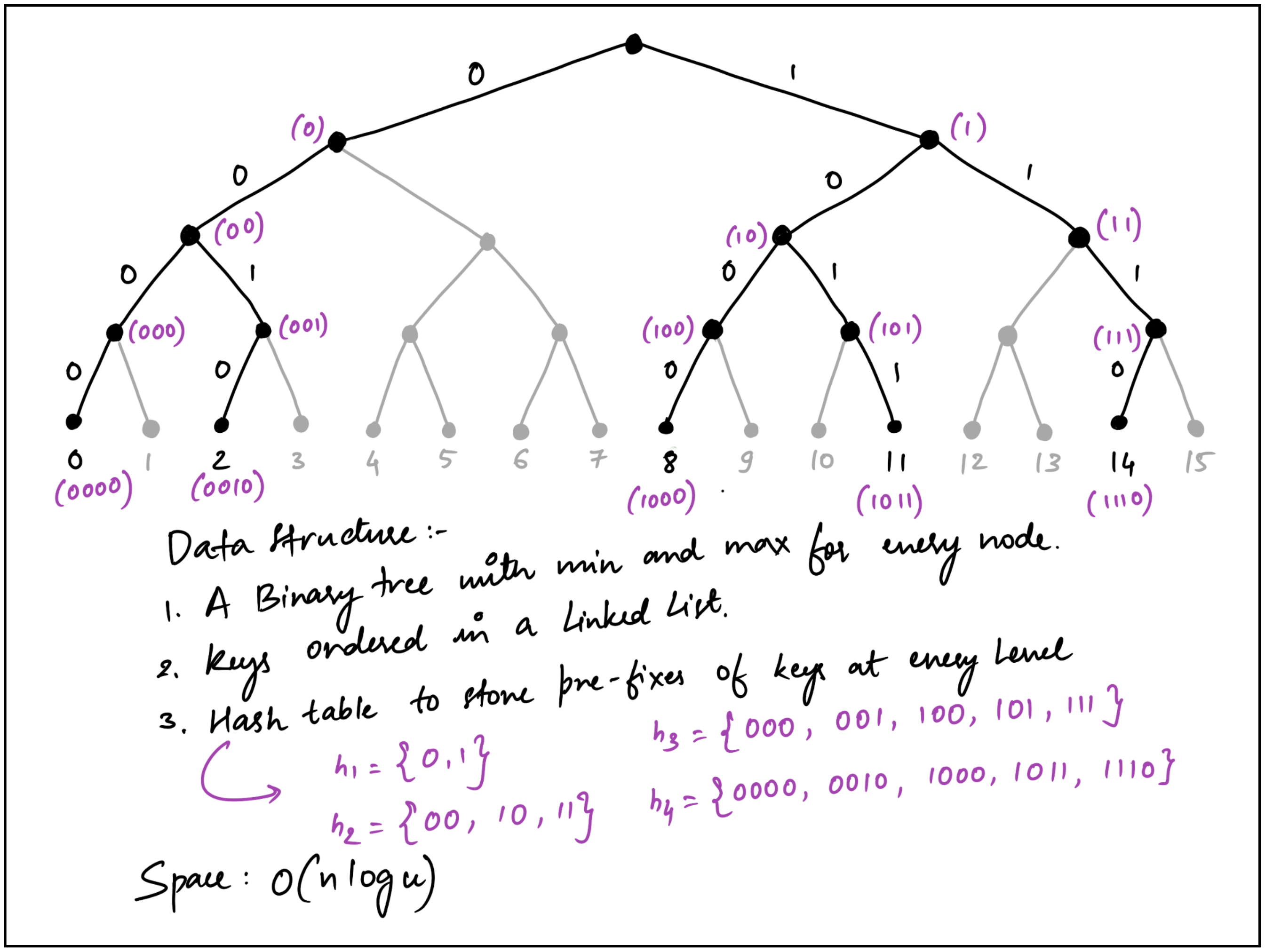


Figure 1: Given Data



Data Structure :-

1. A Binary tree with min and max for every node.
2. Keys ordered in a Linked List.
3. Hash table to store pre-fixes of keys at every Level

$h_1 = \{0, 1\}$

$h_2 = \{00, 10, 11\}$

$h_3 = \{000, 001, 100, 101, 111\}$

$h_4 = \{0000, 0010, 1000, 1011, 1110\}$

Space : $O(n \log u)$

Figure 2: X Fast Trie - Data Structure

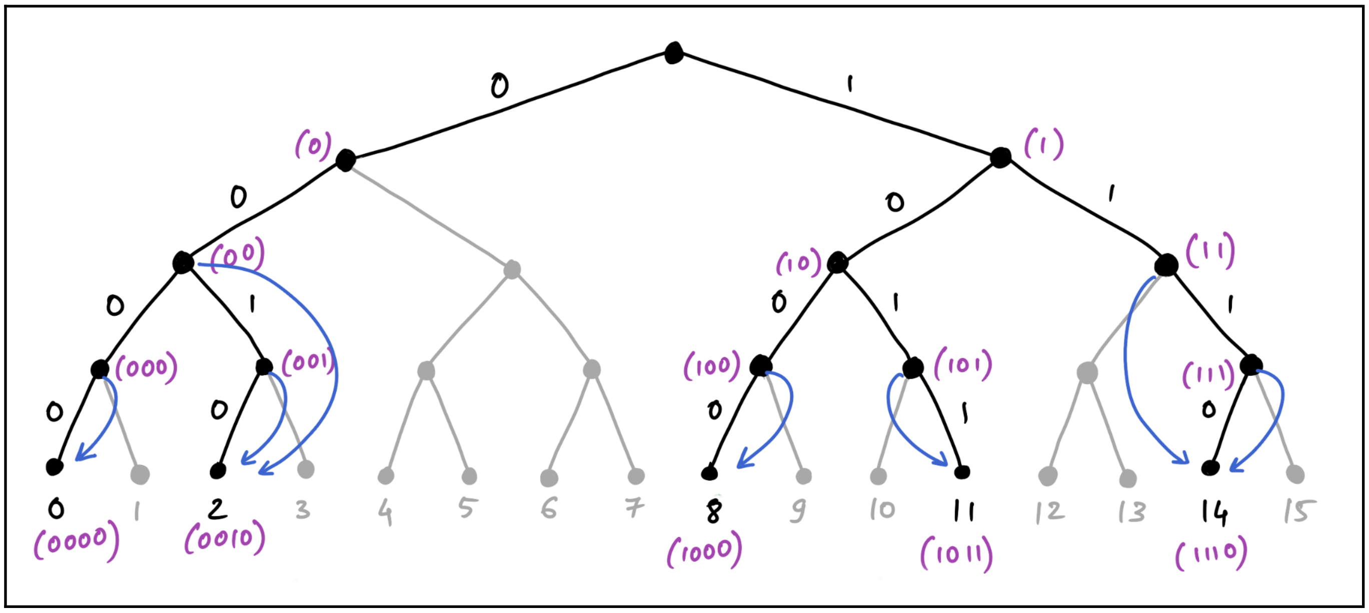


Figure 3: X Fast Trie - Descendant Links

Figure 2 shows the tree structure, values stored in the leaves, pointers between leaves, and keys stored in the hash table.

Figure 3 shows the descendant links: marked in blue.

---

2. Question

[15 marks] This question asks you to perform competitive analysis of transpose (TR).

- [9 marks] Suppose that you are maintaining a list of n elements under access operation only. The cost of access to the i-th element in the list is i. Let S be a request sequence of m access operations over this list. For any sufficiently large m, construct a request sequence S such that for this request sequence, the total cost of TR divided by the total cost of Sopt is ω(1).

- [6 marks] Use the result of (i) to argue that TR is not competitive.

Solution:

Scenario 1:
For the sake of better explanation, consider an input list, A = (1, 2, 3, 4, 5,..., n), the request sequence, S = 1, 2, 1, 2, 1, 2,... and m is the length of the request sequence S (number of accesses). The transpose will pay a cost of ~m, so is the cost of static optimal.

The competitive ratio, Cost(TR)/Cost(Static Opt), will be a constant, where both are individually in order of m; hence the ratio will be a constant or 1, by which we can say that the approximate lower bound is ω(1).

Scenario 2:
For the sake of better explanation, consider an input list, A = (3, 4, 5,..., n, 1, 2), the request sequence, S = 1, 2, 1, 2, 1, 2,... and m is the length of the request sequence S (number of accesses).

Below is the explanation considering the average cost and the total cost:

The cost of Transpose is very bad under this scheme

This strategy will pay : mn

$\text{Cost}(TR) = \sum_{i=1}^{n} i + (m-n)n \sim mn \longrightarrow (1)$

Because the elements 1, 2 never make it out of the back of the list.

On the other hand, $\text{Cost}(S_{opt}) = \sum_{i=1}^{n} i + 1.5m \sim 1.5m \longrightarrow (2)$

Because only 2 items are requested.

(Considering a claim - for the static optimal ordering, 1.5 comparisons would suffice [1])

$$\therefore \frac{\text{Cost}(TR)}{\text{Cost}(S_{opt})} = \frac{mn}{1.5m} = \frac{n}{1.5} = \frac{2n}{3} \longrightarrow (3)$$

To prove $\frac{\text{Cost}(TR)}{\text{Cost}(S_{opt})}$ is $\omega(1)$; First, Let's prove $\frac{\text{Cost}(TR)}{\text{Cost}(S_{opt})} \in \omega(1)$

$f(n) = \frac{2n}{3}$ and $g(n) = 1$

$f(n) > c \cdot g(n)$ for all $n \geq k$

$\frac{2n}{3} > c \cdot 1$ for all $n \geq k$

$n > \frac{3c}{2}$ for all $n \geq k$

choose $k > \frac{3c}{2}$, say: $k = \frac{3c}{2} + 1$

Choose $c = 2$

$f(n) > c \cdot g(n)$ for all $n \geq k$

$\frac{2n}{3} > 2 \cdot 1$; $n \geq \frac{3 \cdot 2}{2} + 1$

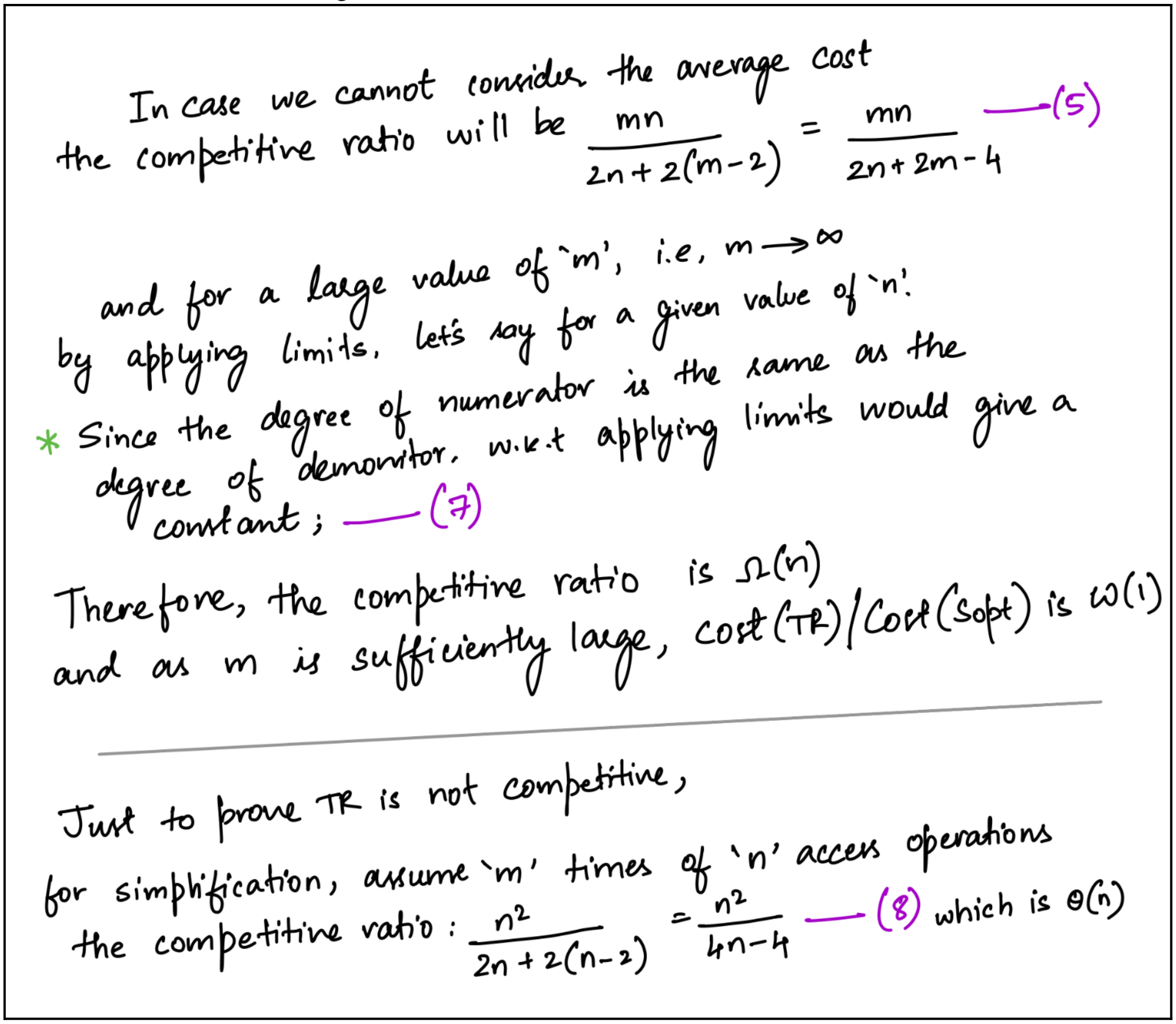
$\frac{2n}{3} > 2$; $n \geq 4$

$n > 3$ for all $n \geq 4$

which is True

Therefore, $f(n) \in \omega(g(n)) \rightarrow \frac{2n}{3} \in \omega(1) \longrightarrow (4)$

Continuation considering the total cost:



In case we cannot consider the average cost the competitive ratio will be $\frac{mn}{2n+2(m-2)} = \frac{mn}{2n+2m-4}$ $\longrightarrow$ (5)

and for a large value of 'm', i.e, $m \rightarrow \infty$ and for a given value of 'n'. by applying limits, let's say for a given value of 'n'.

* Since the degree of numerator is the same as the degree of demonitor, w.r.t applying limits would give a constant ; $\longrightarrow$ (7)

Therefore, the competitive ratio is $\Omega(n)$ and as m is sufficiently large, cost(TR)/Cost(Sopt) is $\omega(1)$

Just to prove TR is not competitive,

for simplification, assume 'm' times of 'n' access operations the competitive ratio : $\frac{n^2}{2n+2(n-2)} = \frac{n^2}{4n-4}$ $\longrightarrow$ (8) which is $\Theta(n)$

TR is not competitive

To prove that TR is not competitive, we can simply prove that the competitive ratio is order of n and not a constant. In order to do so, we will have to find the total Cost of Transpose and the Cost of Static Optimal. If the order of the Cost(TR)/Cost(Static Opt) is in order of n, our claim will stand as expected.

From equation (3), or even equation (4) or (8). It's clear that the competitive ratio is in order of n, which is Ω(n).

Since we have well demonstrated that the ratio of cost of transpose to that of the optimal static ordering cannot be bounded by any constant, we can conclude that TR is not competitive.

---

3. Question

Now prove the following statement: If pn = qn = 0, then an optimal binary search tree storing A1,A2,...,An with probabilities p1,...,pn, q0,..., qn can be obtained by making the following change to any optimal binary search tree for A1, A2,..., An−1 with probabilities p1,...,pn−1,q0,...,qn−1: simply replace the dummy node labeled n − 1 with the structure shown in Figure 2.

Solution:

Given that for a binary search tree, the probability of accessing elements in the tree is p1,...,p(n−1), and the probability of accessing dummy nodes is q0,...,q(n−1)

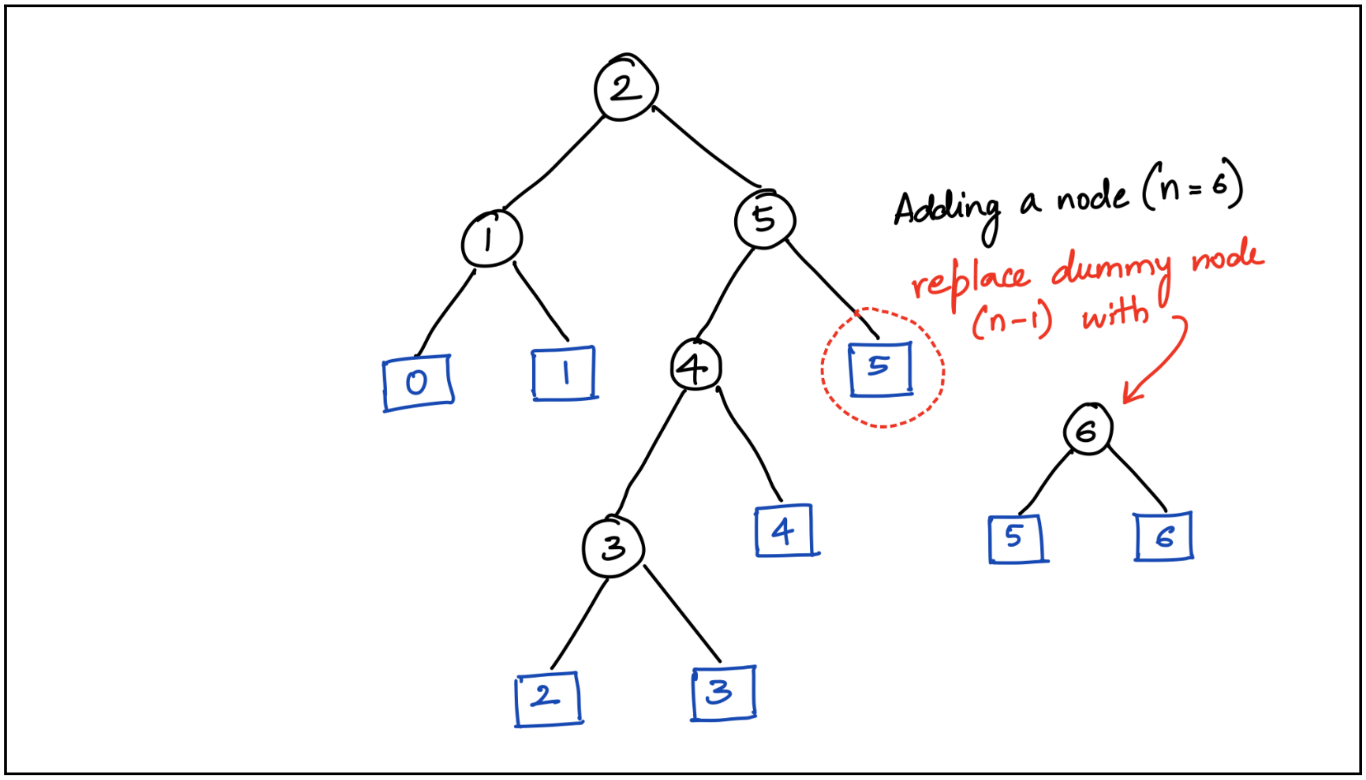


Figure 4: Re-drawn binary search tree

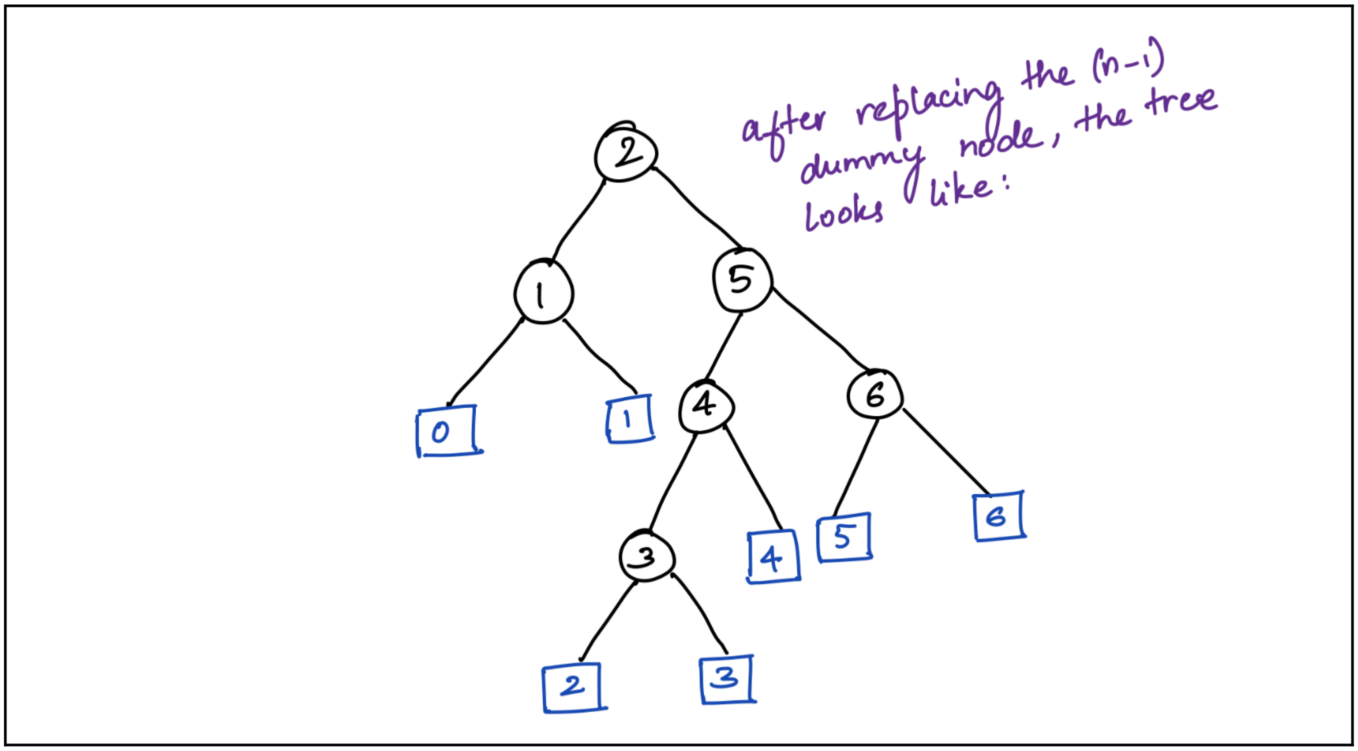


Figure 5: Replaced the dummy node (n − 1) with given structure

Figure 4 and *Figure 5* shows the expected tree after replacing the (n-1) dummy node with the given structure.

Considering the summations of probabilities to prove that replacing the (n-1) dummy node with the given structure is the same as the tree with n nodes when qn = pn = 0.

Proof:

w.r.t $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$

for the given tree: 1 to 5 ; n-1 = 5
which is 1 to (n-1) ; i=1 & j = n-1

$$w(1, n-1) = \sum_{l=1}^{n-1} p_l + \sum_{l=0}^{n-1} q_l \quad \text{—(1)}$$

Now for $w(i, j+1) = \sum_{l=1}^{j+1} p_l + \sum_{l=i-1}^{j+1} q_l$

$$w(i, j+1) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l + p_{j+1} + q_{j+1}$$

$$w(1, n) = \underbrace{\sum_{l=1}^{n-1} p_l + \sum_{l=0}^{n-1} q_l}_{\text{from (1)}} + \underset{\substack{\downarrow \\ \text{given } p_n = q_n = 0}}{p_n + q_n} \quad \text{—(2)}$$

$$= w(1, n-1) + 0$$

although taking the example, generalized to w(i, j) ; when $p_n = q_n = 0$

$\hookrightarrow w(i, j+1) = w(i, j)$

Substituting equation (1) in (2) and the given: pn = qn = 0; we have the proof.

4. Question

[10 marks] Let T be an arbitrary splay tree storing n elements A1, A2,...,An, where A1 ≤ A2 ≤ ... ≤ An. We perform n search operations in T, and the ith search operation looks for element Ai. That is, we search for items A1, A2,...,An one by one.

- [5 marks] What will T look like after all these n operations are performed? For example, what will the shape of the tree be like? Which node stores A1, which node stores A2, etc.?

- [5 marks] Prove the answer you gave for (i) formally. Your proof should work no matter what the shape of T was like before these operations.

Solution:

We know that the given splay tree is arbitrary, which means that the initial shape or structure of the splay tree is random/unknown.
Therefore, trying the n search operations on the arbitrary splay tree for different initial structures, as shown in *Figures 1, 2, and 3.*

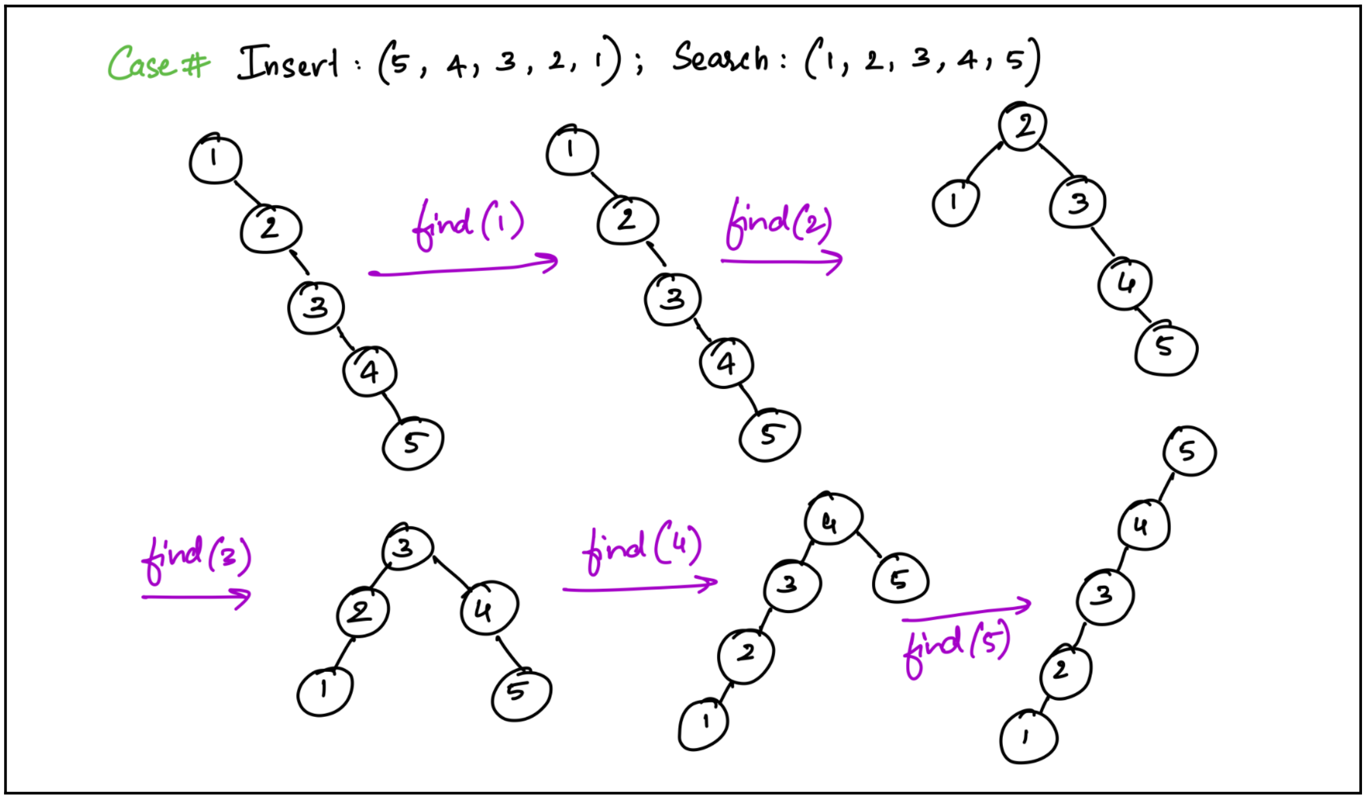


Figure 6: Case #1 for n search operations on an arbitrary splay tree

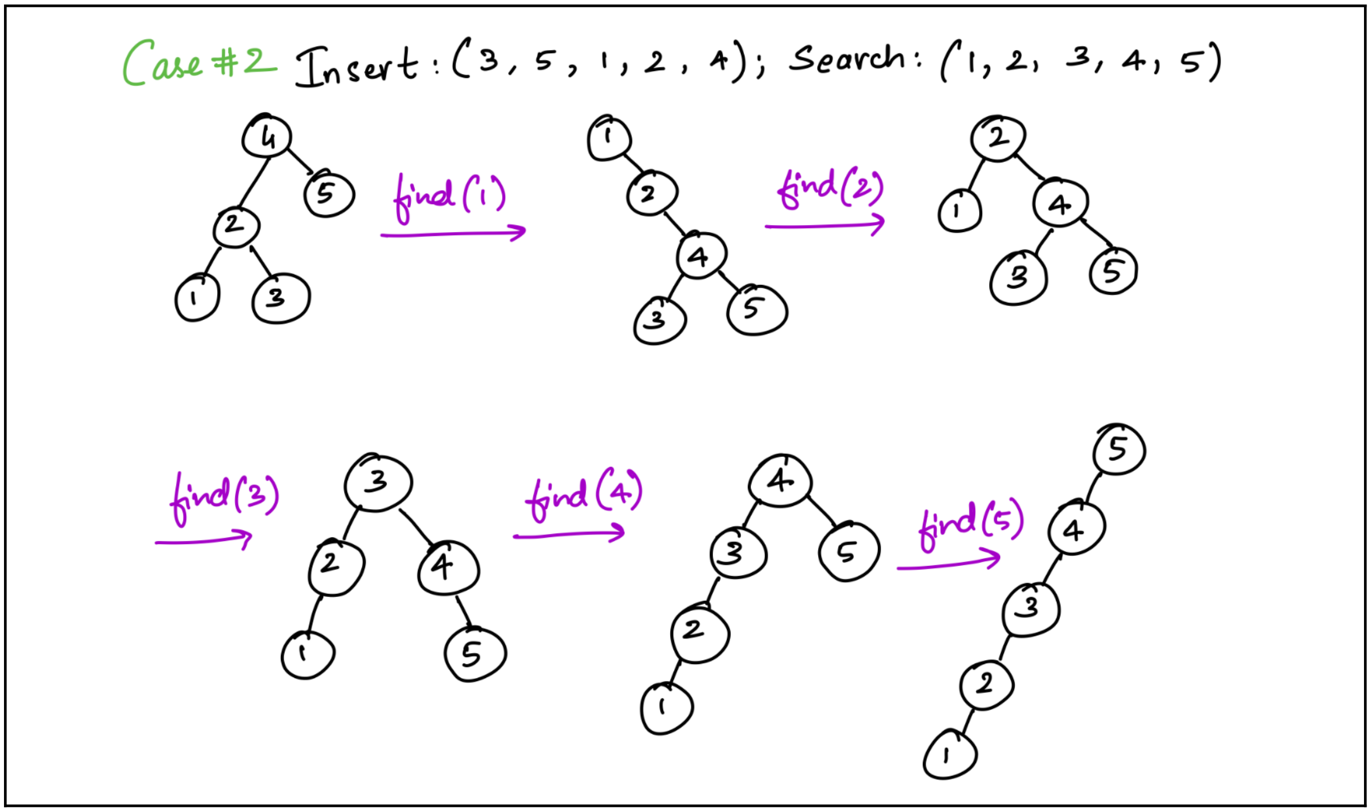


Figure 7: Case #2 for n search operations on an arbitrary splay tree

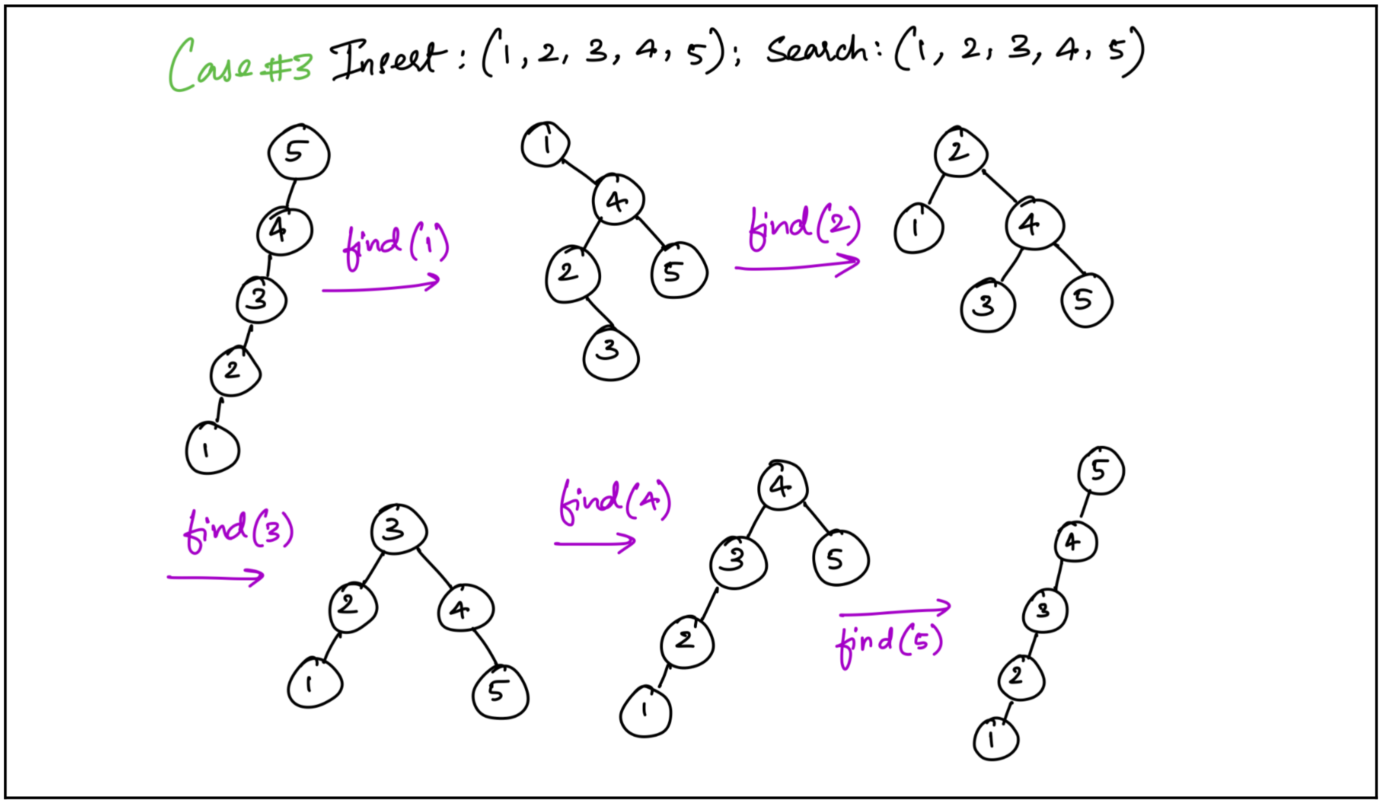


Figure 8: Case #3 for n search operations on an arbitrary splay tree

Incase the input contains duplicates; Example:

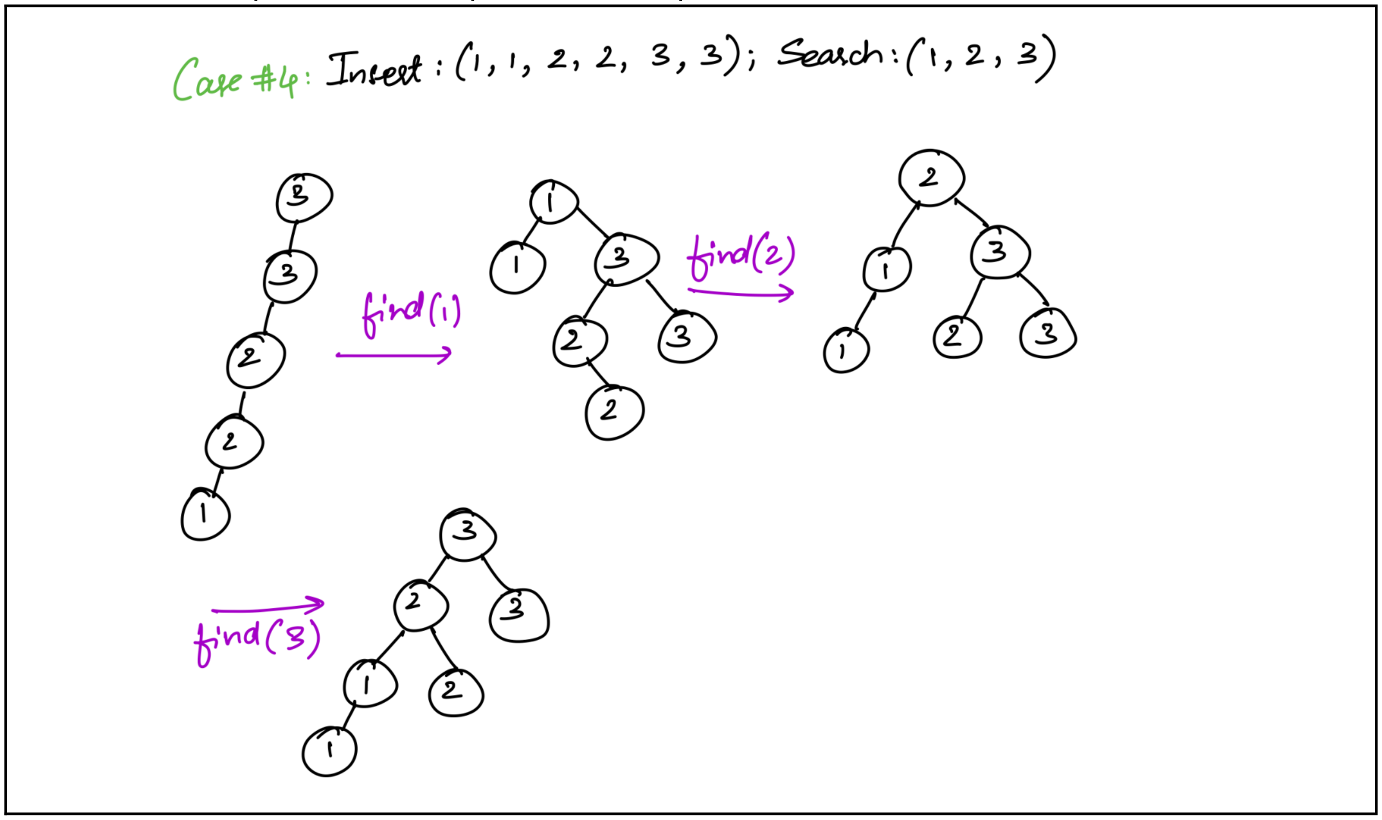


Figure 9: Case #4 for n search operations on an arbitrary splay tree with duplicates

For the searching operation in the sequence, A1, A2,...,An, such that: A1 ≤ A2 ≤ ... ≤ An, we can conclude that:

For distinct elements:

1. Every node in the resultant tree has only a left subtree (except the leaf node).
2. The smallest element: A1 is a leaf node of the left sub-tree.
3. The largest element: An is the root node.
4. A(n-1) is the left child of An, An-2 is the left child of An-1, and so on; in general, A(n) is the left child of A(n+1).

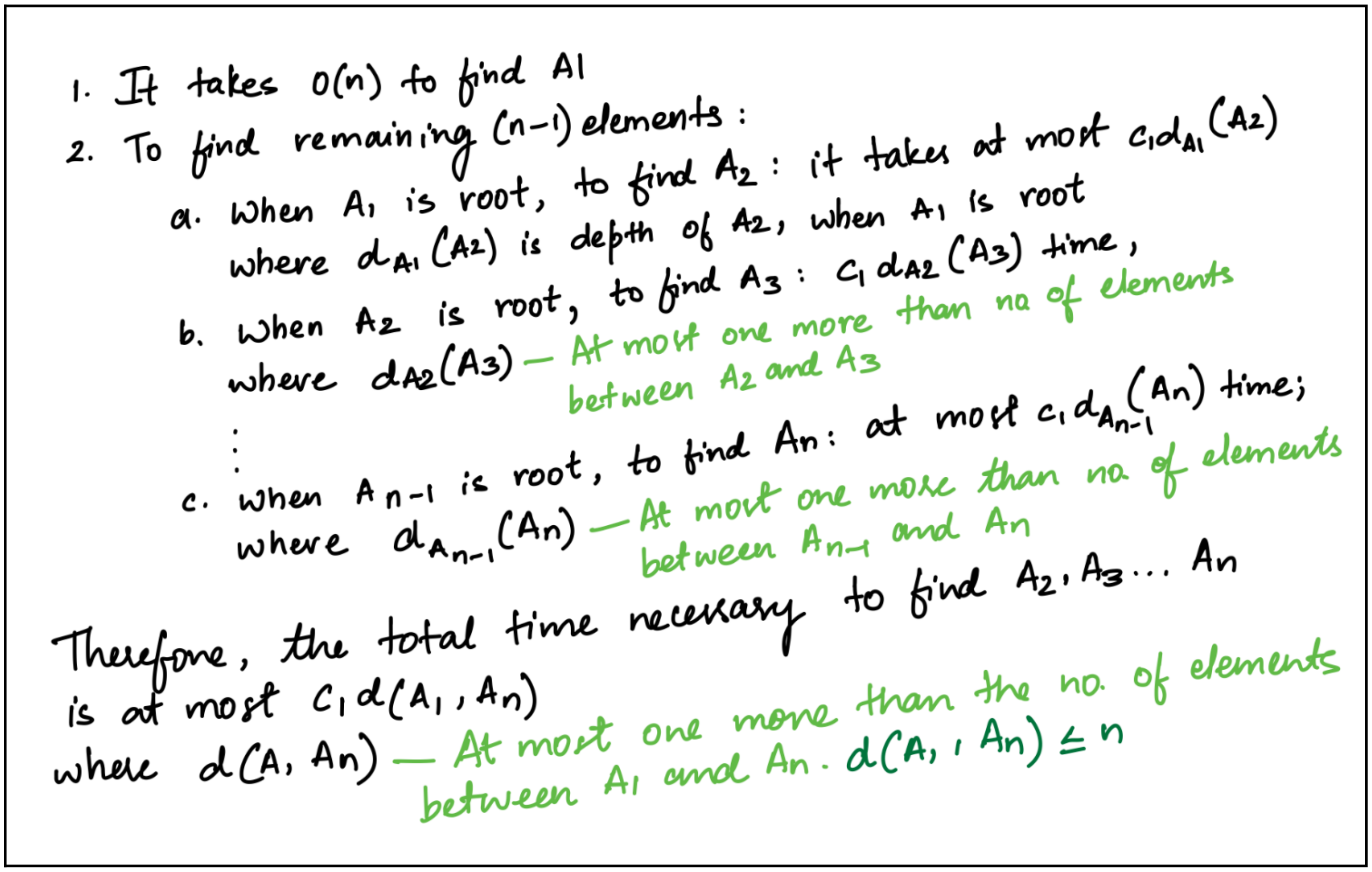
Irrespective of distinct or not:
More importantly, as observed, when searching for elements in the increasing order, it takes a total of O(n) time.

Consider an arbitrary splay tree T with "n" nodes; Let A1, A2,···, An (elements in increasing order).

Let c(A) be the time taken to find an element A. Then, the constant c1 exists independent of the splay tree, such that: c(A) <= c1(d(A)); d(A) is the depth of A.

- To search an element A starting from the root node, it takes c2(d(A)) for some constant c2
- Now, to splay A to the root node, it takes each of the splay operation(s) a constant to move by 1 or 2 rotations. So let's say, for a constant c3, it takes c3(d(A)) to move A to root.
- And let c1 = c2 + c3

Before we prove for any shape of the splay tree, for the shape of the tree after the n search operations in increasing order:



1. It takes $O(n)$ to find A1
2. To find remaining $(n-1)$ elements :
    a. When A_1 is root, to find A_2 : it takes at most $c_1 d_{A_1}(A_2)$ where $d_{A_1}(A_2)$ is depth of A_2, when A_1 is root
    b. When A_2 is root, to find A_3 : $c_1 d_{A_2}(A_3)$ time, where $d_{A_2}(A_3)$ — At most one more than no of elements between A_2 and A_3

    ⋮

    c. When A_{n-1} is root, to find A_n: at most $c_1 d_{A_{n-1}}(A_n)$ time; where $d_{A_{n-1}}(A_n)$ — At most one more than no. of elements between A_{n-1} and A_n

Therefore, the total time necessary to find $A_2, A_3 \ldots A_n$ is at most $c_1 d(A_1, A_n)$ where $d(A, A_n)$ — At most one more than the no. of elements between A_1 and A_n. $d(A_1, A_n) \leq n$

Similarly, for any splay tree, the nth step is always a constant; the remaining the (n-1) elements depend on the depth between An and A(n-1) as explained earlier, with the total time required to search the sequence in increasing order is at most O(n).

References:

[1] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. Comm. ACM 28:404–411, 1985.

[2] F. R. K. Chung, D. J. Hajela, P. D. Seymour: Self-organizing sequential search and Hilbert's inequalities. J. Comp. Systems Sc. 36(2):148–157, 1988.

[3] J. R. Bitner. Heuristics that Dynamically Organize Data Structures. SIAM J. Comput. 8(1):82–110, Feb. 1979.

[4] G. H. Gonnet, J. I. Munro and H. Suwanda. Exegesis of Self-Orgainzing Linear Search. SIAM J. Comput. 10(3):613–637, Aug. 1981.