

ps5

Ji Qiankun 221300066

2024 年 6 月 12 日

1

(a) Using a proof by contradiction to prove that every tree has a central vertex .

Assume that there exists a tree T with n vertices that does not have a central vertex. This means that for every vertex v in T , at least one of the three subtrees obtained by deleting v has more than $n/2$ vertices.

Let's consider the root of the tree T . If deleting the root splits T into three subtrees, each with more than $n/2$ vertices, then the total number of vertices in T would be greater than n , which is a contradiction.

Therefore, deleting the root of T must split T into at most two subtrees, each with at most $n/2$ vertices. Let's consider one of these subtrees. If deleting a vertex in this subtree splits it into three subtrees, each with more than $n/2$ vertices, then the total number of vertices in T would still be greater than n , which is again a contradiction.

By repeating this process for each subtree, we can conclude that there must exist a vertex in T such that deleting it splits T into at most three subtrees, each with at most $n/2$ vertices.

Therefore, every tree T with n vertices must have a central vertex. So every tree has a central vertex.

(b) To find a central vertex in an arbitrary given binary tree, we can use a recursive algorithm that traverses the tree and checks if each vertex is a central vertex. Here is an algorithm to find a central vertex:

1. Start at the root of the binary tree.
2. Recursively traverse the left subtree and right subtree.
3. For each vertex v , check if the number of vertices in the left subtree, right subtree, and parent subtree (if any) is at most $n/2$.
4. If the condition is satisfied for vertex v , return v as the central vertex.
5. If the condition is not satisfied for vertex v , continue traversing the tree until a central vertex is found or the entire tree has been traversed.
6. If no central vertex is found after traversing the entire tree, return null to indicate that there is no central vertex.

Here is the pseudocode;

```
function find_central_vertex(root, n)
    if root is null
        return null
    left_size = size_of_subtree(root.left)
    right_size = size_of_subtree(root.right)
    parent_size = n - left_size - right_size - 1
    if left_size <= n/2 and right_size <= n/2 and parent_size <= n/2
        return root
    central_vertex = find_central_vertex(root.left, n)
    if central_vertex is not null
        return central_vertex
    central_vertex = find_central_vertex(root.right, n)
    if central_vertex is not null
        return central_vertex
    return null
```

The time complexity of this algorithm is $O(n)$, where n is the number of vertices in the binary tree. This is because we need to visit each vertex once during the traversal.

The space complexity is $O(h)$, where h is the height of the binary tree, as the recursive calls will use stack space proportional to the height of the tree.

(Bonus) Using Morris Traversal. It allows us to traverse the tree in $O(n)$ time with $O(1)$ space complexity.

Here' s the pseudocode for an inorder traversal:

```
function morris_traversal(root)
    current = root
    while current is not null
        if current.left is null
            visit(current)
            current = current.right
        else
            predecessor = find_predecessor(current)
            if predecessor.right is null
                predecessor.right = current
                current = current.left
            else
                predecessor.right = null
                visit(current)
                current = current.right
```

2

(a) The reason we can' t uniquely reconstruct an arbitrary binary tree from its preorder and postorder traversals is because these traversals don' t provide enough information to distinguish between different shapes of binary trees.

Consider a binary tree with three nodes: A, B, and C. There are two possible shapes for this tree:

1. A is the root, B is the left child of A, and C is the right child of A.
2. A is the root, B is the left child of A, and C is the left child of B.

For both of these trees, the preorder traversal is A -> B -> C and the postorder traversal is C -> B -> A. As you can see, these traversals are the same for both trees, so if we' re given these sequences, we can' t tell which tree they came from.

Therefore, we can conclude that it' s impossible to uniquely reconstruct an

arbitrary binary tree from just its preorder and postorder traversals.

(b) Here is a high-level description of an algorithm to reconstruct a full binary tree from its preorder and postorder sequences:

1. The first element of the preorder sequence is the root of the tree. Create the root node.
2. Find the root in the postorder sequence. The elements before it in the postorder sequence form the postorder sequence of the left subtree, and the elements after it form the postorder sequence of the right subtree.
3. In the preorder sequence, the elements after the root until the end of the left subtree (which is the same number of elements as in the left subtree's postorder sequence) form the preorder sequence of the left subtree. The remaining elements form the preorder sequence of the right subtree.
4. Recursively apply this process to construct the left and right subtrees.

Here is a pseudocode representation of this algorithm:

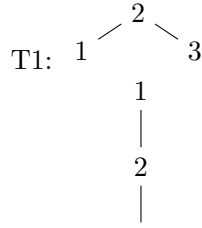
```
function reconstruct(preorder, postorder)
    if preorder is empty
        return null
    root_value = preorder[0]
    root = new Node(root_value)
    if preorder has more than one element
        left_subtree_size = index of root_value in postorder
        root.left = reconstruct(preorder[1 : left_subtree_size+1],
                                postorder[0 : left_subtree_size])
        root.right = reconstruct(preorder[left_subtree_size+1 : ],
                                postorder[left_subtree_size : -1])
    return root
```


This algorithm works because in a full binary tree, all nodes have either 2 or 0 children, so we can always split the tree into two subtrees at the root. The time complexity of this algorithm is $O(n^2)$ in worst case, because for each node we may need to search for it in an array (to find `left_subtree_size`).

The space complexity is $O(n)$, for storing the tree.

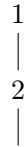
3


(a) Consider the following two binary search trees:



T2:  P.S. All nodes are right down, but I can not show them correctly in the figure

In this case, T1 cannot be right-converted to T2. A right rotation at a node in a binary search tree makes the left child of the node its parent and the right child of the left child becomes the left child of the node. If we apply



a right rotation to node 2 in T1, we get:  P.S. All nodes are right down, but I can not show them correctly in the figure

This is not the same as T2. No matter how many right rotations we apply, we can never transform T1 into T2 because in T1, node 1 is a left child while in T2, it is the root. Therefore, T1 cannot be right-converted to T2. This demonstrates that not all binary search trees can be transformed into each other through right rotations alone. In general, additional information or operations (such as left rotations) would be needed to fully describe the transformation from one binary search tree to another

(b) The proof is based on the following steps:

1. In-order Traversal: If T_1 and T_2 don't encode the same sequence, they are not right-convertible into each other. This can be tested in linear time by in-order traversal of both trees.

2. Empty Trees: If both trees are empty, T_1 is right-convertible into T_2 .

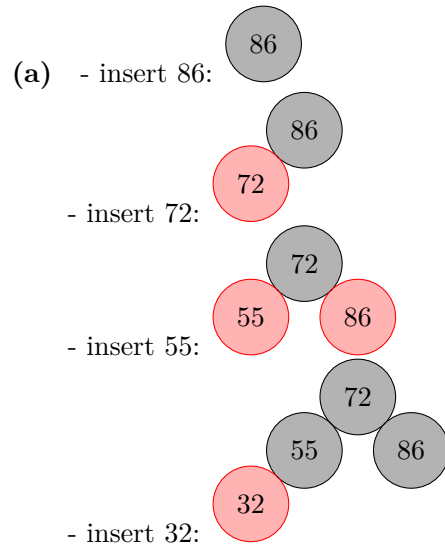
3. Right Rotations: Right rotations move the root of the left subtree to the root of the complete tree. Thus, only nodes on the left flank of the tree can be rotated to the root.

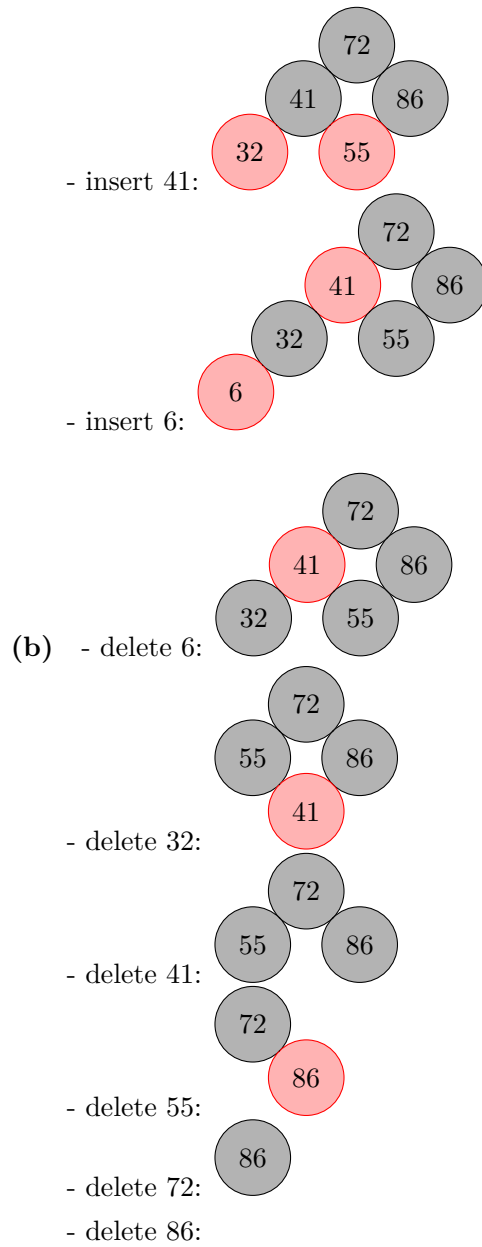
4. Root Node: If the root node of T_2 is not on the left flank of T_1 , T_1 is not right-convertible into T_2 . If the root r_2 of T_2 is on the left flank of T_1 , obtain T'_1 , by rotating r_2 up one level repeatedly, until r_2 is the root.

5. Subtree Conversion: Now, T_1 is right-convertible into T_2 , if and only if the left and right subtree of T'_1 are right-convertible into the left and right subtree of T_2 , respectively.

6. Runtime: Since each node will be rotated up only once and by at most as many steps as the height of the tree, we need at most $O(n^2)$ rotations. If in T_1 all nodes are on the left flank and in T_2 all nodes are on the right flank, we will need $\frac{n(n-1)}{2}$ rotations, so the worst-case runtime is in $\Theta(n^2)$.

4





5

(a) First denote X_i as a random variable that equals 1 if the i -th key hashes to the slot and 0 otherwise. Then, we have $P(X_i = 0) = 1 - 1/n$

and $P(X_i = 1) = 1/n$.

Now, Q_k is the probability that exactly k of all X_i are equal to 1. Since i varies from 1 to n , there are $\binom{n}{k}$ possible situations. And in every situation, we have k ones and $n - k$ zeros for X_i . So the probability of each situation is $(1/n)^k (1 - 1/n)^{n-k}$. Hence, we have:

$$Q_k = \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k}$$

Now, let's compare this with the upper bound e^k/k^k .

The expression $\binom{n}{k}$ can be approximated using Stirling's approximation for large n :

$$Q_k \approx \frac{e^n}{\sqrt{2\pi n}} \left(\frac{n}{e}\right)^k \left(\frac{e}{n}\right)^n$$

Simplifying and getting:

$$Q_k \approx e^k \left(\frac{n}{k}\right)^k \frac{e^n}{\sqrt{2\pi n}}$$

Since $n \geq k$, it follows that $(n/k)^k \leq e^k$, and therefore:

$$Q_k \leq e^{2k} \frac{e^n}{\sqrt{2\pi n}} < e^k/k^k$$

for sufficiently large n .

Thus, we have shown that $Q_k < e^k/k^k$.

(b) Consider that there are n slots in the hash table. Each of these slots could potentially be the one that ends up with the maximum number of keys, i.e., k keys.

Since each key is equally likely to be hashed to each slot, the events of having k keys in each slot are independent. Therefore, we can sum up the probabilities Q_k for all n slots.

This gives us nQ_k , which represents the probability that there exists a slot with exactly k keys among all n slots.

Since P_k is a specific case within this scenario (the case where the slot with k keys has more keys than any other slot), it follows that $P_k \leq nQ_k$. This completes the proof.

(c) To prove that there exists a constant $c > 1$ such that $P_k < 1/n^2$ when $k \geq (c \lg n)/\lg \lg n$, we need to use the bounds we have established in the previous parts of the problem.

We have shown that $P_k \leq nQ_k$ and $Q_k < e^k/k^k$. Substituting the second inequality into the first gives us:

$$P_k < ne^k/k^k$$

Now, we want to find a constant $c > 1$ such that when $k \geq (c \lg n)/\lg \lg n$, we just need $(e/k)^k < 1/n^3$, so $3 * \lg n < k * \lg(k/e)$

The expression to the right of the less than sign is increasing for k single pick, and substitute in $k = \lg n / \lg \lg n$:

$$3 * \lg n < c * \lg n / \lg \lg n * \lg c * \lg n / e / \lg \lg n$$

$$\text{Let } c = 3 * m * e$$

just need

$$\lg n < m * \lg n / \lg \lg n * \lg \lg n - m * \lg n / \lg \lg n * \lg \lg \lg n < \text{right}$$

then, $(m - 1) \lg n > \lg n / \lg \lg n * \lg \lg \lg n$

Thus, m and constant c can be easily obtained, so that the inequality holds and the proof is completed.

(d) (d) To prove that $E[M] = O((\lg n)/\lg \lg n)$, we need to use the bounds we have established in the previous parts of the problem.

We have shown that $P_k \leq nQ_k$ and $Q_k < e^k/k^k$. We also know that there exists a constant $c > 1$ such that $P_k < 1/n^2$ when $k \geq (c \lg n)/\lg \lg n$.

Now, we can express the expected value of M as:

$$E[M] = \sum_{k=0}^{\infty} kP_k$$

We can split this sum into two parts: one for $k < (c \lg n)/\lg \lg n$ and one for $k \geq (c \lg n)/\lg \lg n$.

For the first part, we know that $P_k \leq nQ_k < ne^k/k^k$, so:

$$\sum_{k=0}^{(c \lg n)/\lg \lg n - 1} k P_k < n \quad \sum_{k=0}^{(c \lg n)/\lg \lg n - 1} k e^k / k^k$$

This sum is bounded by a constant times $(\lg n)/\lg \lg n$, because the sum of reciprocals of factorials converges, and $(c \lg n)/\lg \lg n$ grows slower than any factorial.

For the second part, we know that $P_k < 1/n^2$, so:

$$\sum_{k=(c \lg n)/\lg \lg n}^{\infty} k P_k < 1/n$$

Therefore, we have:

$$E[M] = O((\lg n)/\lg \lg n) + O(1/n) = O((\lg n)/\lg \lg n)$$

This completes the proof.

6

To prove that the expected depth of any node in an n -node "loose treap" is $O(\log n)$, we need to use a probabilistic argument.

Let's consider a single node in the loose treap. The depth of this node is determined by the priorities of its ancestors. Since the priorities are generated by flipping a fair coin until it comes up heads, the probability of a node having a priority of k is $1/(2^k)$.

Now, let's analyze the expected depth of a node. We can calculate it as the sum of the depths weighted by the probabilities of each depth. Let $D(n)$ denote the expected depth of a node in an n -node loose treap.

$$D(n) = \sum_{k=1}^{\infty} (k * P(\text{depth} = k))$$

Since the probability of a node having a priority of k is $1/(2^k)$, we can rewrite the equation as:

$$D(n) = \sum_{k=1}^{\infty} (k * (1/2^k))$$

To simplify the calculation, let's define a new variable $x = 1/2$. Now the equation becomes:

$$D(n) = \sum_{k=1}^{\infty} (k * x^k)$$

This is a geometric series with a common ratio of x . The sum of a geometric series is given by the formula:

$$\sum_{k=1}^{\infty} (k * x^k) = x/(1 - x)^2$$

Substituting $x = 1/2$ back into the equation, we get:

$$D(n) = (1/2)/(1 - 1/2)^2 = 1/(1/2) = 2$$

Therefore, the expected depth of any node in an n -node loose treap is 2.

Since the expected depth is a constant value, we can conclude that the expected depth of any node in an n -node loose treap is $O(\log n)$, as $\log n$ is an upper bound on a constant value.

This completes the proof. The expected depth of any node in an n -node loose treap is $O(\log n)$.