1483. Kth Ancestor of a Tree Node

Hard · Tree · Depth-First Search · Breadth-First Search · Design · Binary Search

Leetcode Link

Problem Description

In this problem, we are given a tree structure with `n` nodes that are numbered from `0` to `n - 1`. A tree is a hierarchical structure where each node can have one parent and potentially multiple children, but no cycles exist. The tree is defined through a parent array named `parent`, where `parent[i]` represents the parent of the `i`th node. The node with number `0` is the root of the tree and it does not have a parent. Our task is to determine the `k`th ancestor of a given node, where the `k`th ancestor is the `k`th node in the path from the given node to the root. To represent this tree structure and perform ancestry queries, we need to implement a `TreeAncestor` class with two methods:

1. `TreeAncestor(int n, int[] parent)`: Constructor method that initializes the tree with the number of nodes (`n`) and the parent array (`parent`).
2. `getKthAncestor(int node, int k)`: Method that returns the `k`th ancestor of the provided node (`node`). If the ancestor does not exist (i.e., the `k`th level ancestor is beyond the root), it should return `-1`.

Intuition

Finding the `k`th ancestor naively by following the parent pointers one by one can be very inefficient, especially when `k` is large. To optimize the query time, we can precompute some data that enables us to "jump" multiple nodes at once.

This optimization relies on the concept of "binary lifting", which essentially means that we can represent the jump of `k` nodes as a sum of powers of 2, since any number can be represented as a combination of powers of 2. With this idea, we can precompute for each node what the 1st, 2nd, 4th (... 2^i-th ...) ancestor is. We store this information in a two-dimensional list `p`, where `p[i][j]` represents the 2^j-th ancestor of node `i`.

The constructor method prepares the `p` table, where the first column (`j = 0`) is just the direct parent of each node. For larger jumps (when `j > 0`), we can find the `2^j`-th ancestor by looking at the `2^(j-1)`-th ancestor of the node that is already the `2^(j-1)`-th ancestor of the current node -- effectively making a "double jump".

When we want to find the `k`th ancestor of a node, we look at the binary representation of `k`. For each bit that is set in `k`, we jump to an ancestor that is `2^i` steps up from the current node, where `i` is the position of that bit. This method drastically reduces the number of steps needed to find the `k`th ancestor.

This binary lifting technique helps in reducing a potentially large number of steps to a manageable few, bounded by the number of bits used to represent the number `k`.

Solution Approach

The solution uses a method called binary lifting to compute and query the `k`th ancestor of a node in a tree efficiently. Here is an in-depth explanation of how the algorithm works and the solution is implemented:

Data Structure Initialization

- A 2D list `self.p` of size `n x 18` is created to store the powers of two ancestors for each node.
  - Why 18? Because for the constraints generally found in such problems, `2^17` is usually enough to cover the height of any tree (since `2^17` is greater than `10^5`, which is a common maximum limit for `n`).
  - `self.p[i][j]` holds the `2^j`-th ancestor of node `i`. `-1` is used to indicate that such an ancestor does not exist.
- The `__init__` method fills up the `self.p` matrix.
  - First, it copies the `parent` array to `self.p[i][0]` for all nodes because the `0`-th power of `2` corresponds to immediate parents.
  - Then for each node `i`, it iterates over `j` from `1` to `17` (inclusive), using previously computed ancestors (from the `j-1` step) to calculate the `j`-th power of `2` ancestor.
  - If at any point it encounters a `-1`, it implies that further ancestors do not exist, and it breaks from the inner loop.

Querying Ancestors

- The `getKthAncestor` method computes the `k`th ancestor for a given node.
  - It loops through powers of two from `2^17` down to `2^0`, examining the bits of `k`.
  - For each `i` in `17` down to `0`, it checks if the `i`-th bit of `k` is set (`k >> i & 1`). If it is, it jumps to the ancestor that is `2^i` steps up by accessing `self.p[node][i]`.
  - If during this process the node becomes `-1`, the method breaks out of the loop, as it indicates that we have tried to query above the root Ancestor (above the height of the tree).
  - The method returns the current node after the loop ends, which represents the `k`-th ancestor if one exists, or `-1` if it doesn't.

This solution efficiently calculates the `k`-th ancestor for any given node by making at most `log(k)` jumps, drastically reducing the time complexity compared to a naive approach which would potentially require `k` jumps.

Example Walkthrough

Let's say we have a tree with 5 nodes where the `parent` array is [-1, 0, 0, 1, 2] and we want to find the Kth ancestor of node 4.

1. First, we initialize the `TreeAncestor` class with `n = 5` and the `parent` array. This initiates the data structure and computes all 2^i-th ancestors for each node.
2. The initialization will produce a `self.p` table that might look something like this after computing the ancestors:

Node	1st(2^0)	2nd(2^1)	4th(2^2)	...
0	-1	-1	-1	...
1	0	-1	-1	...
2	0	-1	-1	...
3	1	0	-1	...
4	2	0	-1	...

3. Now let's find the 2nd ancestor of node 4. We call `getKthAncestor(4, 2)`.
4. In binary, `k = 2` is `10`. This means we want to find the ancestor that is `2^1` (the second bit from right to left in the binary representation of `k`) steps up from our current node.
5. We start from node 4 and check the highest power of two within 2, which is `2^1`, and find the ancestor 2 steps up:
   - Access `self.p[4][1]`, which gives us node 0 as the `2^1`-th ancestor of node 4.
6. Since there is no higher power of two within 2, we have reached our result. The 2nd ancestor of node 4 is node 0 as per our `self.p` table.

This walkthrough demonstrates how the binary lifting method is used to compute the `k`th ancestor of a node by looking up precomputed ancestors using binary steps, significantly reducing the number of computations needed.

Python Solution

```python
from typing import List

class TreeAncestor:
    def __init__(self, n: int, parent: List[int]):
        # Create a list of lists to store the ancestors. The outer list has a length of n,
        # and the inner lists have fixed length of 18 (assuming a ceiling of log2(n)).
        self.ancestors = [[-1] * 18 for _ in range(n)]

        # Initialize the immediate ancestors from the parent array.
        for i, direct_parent in enumerate(parent):
            self.ancestors[i][0] = direct_parent

        # Precompute ancestors using dynamic programming.
        # Iterate up to 2^17 (which covers all binary representations for k up to n).
        for i in range(n):
            for j in range(1, 18):
                if self.ancestors[i][j - 1] == -1:
                    continue  # There's no ancestor at this level, skip to the next
                # Set the ancestor at the jth binary up-step to be the (j-1)th ancestor
                # of the (j-1)th ancestor of the current node.
                self.ancestors[i][j] = self.ancestors[self.ancestors[i][j - 1]][j - 1]

    def getKthAncestor(self, node: int, k: int) -> int:
        # To find the kth ancestor, we look at the binary representation of k.
        # We use bit manipulation to move upwards step by step.
        for i in range(17, -1, -1):
            # Check if the ith bit is set in the binary representation of k.
            if k & (1 << i):
                # Move up by 2^i ancestors.
                node = self.ancestors[node][i]
                # If there isn't an ancestor at this level, return -1.
                if node == -1:
                    break
        return node

# Usage example:
# tree_ancestor = TreeAncestor(n, parent)
# ancestor = tree_ancestor.getKthAncestor(node, k)

```

Java Solution

```java
class TreeAncestor {
    // Sparse table to keep ancestors at power of two distance
    private int[][] sparseTable;

    // Constructor to initialize the sparse table with the direct parents provided
    public TreeAncestor(int n, int[] parent) {
        // Initialize sparse table, allowing us to jump up in powers of two
        sparseTable = new int[n][18];
        for (int[] row : sparseTable) {
            Arrays.fill(row, -1); // Fill the table with -1 to indicate no ancestor
        }
        // Fill the first column of the sparse table with the given parents
        for (int i = 0; i < n; ++i) {
            sparseTable[i][0] = parent[i];
        }
        // Compute ancestors at 2^j distance for dynamic programming approach
        for (int i = 0; i < n; ++i) {
            for (int j = 1; j < 18; ++j) {
                if (sparseTable[i][j - 1] != -1) { // If there is an ancestor at 2^(j-1) distance
                    // Set the ancestor at 2^j distance by doubling the previous distance ancestor
                    sparseTable[i][j] = sparseTable[sparseTable[i][j - 1]][j - 1];
                }
            }
        }
    }

    // Returns the k-th ancestor of the node, or -1 if it does not exist
    public int getKthAncestor(int node, int k) {
        // Traverse bits of k in reverse order (start from highest bit)
        for (int i = 17; i >= 0; --i) {
            if (((k >> i) & 1) == 1) {
                // If the ith bit is set, move up by 2^i in the tree
                node = sparseTable[node][i];
                // If there's no ancestor at this power of two, exit the loop early
                if (node == -1) {
                    break;
                }
            }
        }
        // Return the final ancestor, or -1 if not found
        return node;
    }
}

/**
 * Your TreeAncestor object will be instantiated and called as such:
 * TreeAncestor obj = new TreeAncestor(n, parent);
 * int param_1 = obj.getKthAncestor(node, k);
 */

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class TreeAncestor {
public:
    // Initialize the data structure with the number of nodes `n` and their direct parent array.
    TreeAncestor(int n, vector<int>& parent) {
        ancestors = vector<vector<int>>(n, vector<int>(MAX_POWER, -1));

        // Direct parent (1st ancestor) for each node.
        for (int i = 0; i < n; ++i) {
            ancestors[i][0] = parent[i];
        }

        // Pre-compute all 2^j ancestors for each node where `j` ranges from 1 to MAX_POWER-1.
        // This uses dynamic programming and the idea that the 2^j-th ancestor is the 2^(j-1)-th ancestor
        // of the node's 2^(j-1)-th ancestor.
        for (int i = 0; i < n; ++i) {
            for (int j = 1; j < MAX_POWER; ++j) {
                if (ancestors[i][j - 1] == -1) {
                    continue; // If there is no ancestor, skip the computation.
                }
                ancestors[i][j] = ancestors[ancestors[i][j - 1]][j - 1];
            }
        }
    }

    // Returns the k-th ancestor of the given node, or -1 if it does not exist.
    int getKthAncestor(int node, int k) {
        for (int i = MAX_POWER - 1; i >= 0; --i) {
            if ((k >> i) & 1) { // Check each bit of `k`.
                node = ancestors[node][i]; // Move up the tree by 2^i steps.
                if (node == -1) {
                    break; // If an ancestor does not exist at this level, return -1.
                }
            }
        }
        return node;
    }

private:
    vector<vector<int>> ancestors; // 2D array where `ancestors[i][j]` is the 2^j-th ancestor of node `i`.
    static const int MAX_POWER = 18; // The maximum power of 2 needed (2^17 covers more than 10^5 which is the typical constraint f
};


/**
 * Your TreeAncestor object will be instantiated and called as such:
 * TreeAncestor* obj = new TreeAncestor(n, parent);
 * int param_1 = obj->getKthAncestor(node,k);
 */

```

Typescript Solution

```typescript
// Array to store preprocessed ancestor information.
let precomputedAncestors: number[][];

/**
 * Function to initialize and preprocess the ancestor data.
 * @param {number} size - The number of nodes in the tree.
 * @param {number[]} parent - The array where the index represents the node and the value represents its parent.
 */
function initialize(size: number, parent: number[]): void {
    // Initialize the precomputedAncestors array with dimensions 'size' x 18 and default value -1.
    precomputedAncestors = new Array(size).fill(0).map(() => new Array(18).fill(-1));

    // Populate the immediate parents of each node.
    for (let i = 0; i < size; ++i) {
        precomputedAncestors[i][0] = parent[i];
    }

    // Precompute ancestors for binary lifting.
    for (let i = 0; i < size; ++i) {
        for (let j = 1; j < 18; ++j) {
            if (precomputedAncestors[i][j - 1] === -1) {
                continue;
            }
            precomputedAncestors[i][j] = precomputedAncestors[precomputedAncestors[i][j - 1]][j - 1];
        }
    }
}

/**
 * Function to find the k-th ancestor of a node using binary lifting.
 * @param {number} node - The node for which the k-th ancestor is required.
 * @param {number} k - The distance 'k' to the ancestor.
 * @returns {number} - The node number of the k-th ancestor or -1 if it does not exist.
 */
function getKthAncestor(node: number, k: number): number {
    // Traverse bits of 'k' from highest to lowest.
    for (let i = 17; i >= 0; --i) {
        // Check if the i-th bit is set in 'k'.
        if (((k >> i) & 1) === 1) {
            node = precomputedAncestors[node][i];

            // If there is no ancestor at this distance, return -1.
            if (node === -1) {
                break;
            }
        }
    }
    // Return the final ancestor or -1 if not found.
    return node;
}

```

Time and Space Complexity

The given code defines a data structure for finding the k-th ancestor of a node in a tree with a pre-processing step that builds a sparse table for ancestor queries, and a query process that traverses this table to find the k-th ancestor.

Time Complexity

- **Pre-processing (`__init__` method)**: The pre-processing step iterates over each of the `n` nodes and fills the sparse table called `self.p`. This table has `18` levels because `2^17` is the highest power of 2 that is below the potential maximum of `n` which may be up to `50000` (as per usual constraints on LeetCode problems) allowing us to reach any ancestor `k < n` with at most `17` jumps. Therefore, the time complexity for the pre-processing part is `O(n * log n)` since we have a nested loop where the outer loop runs for `n` and the inner loop for up to `log n` (which is `18` in this specific implementation assuming a reasonable upper bound for `n`).

- **Query (`getKthAncestor` method)**: For a single query to find the k-th ancestor, the method uses a loop that potentially iterates `18` times (the maximum number of jumps we need to make). For each iteration, it performs an `O(1)` check to see if the k-th ancestor exists for the current power of 2. Hence, the time complexity per query is `O(log n)`.

Space Complexity

- **Pre-processing (`__init__` method)**: The space complexity is `O(n * logn)`. A sparse table (`self.p`) of size `n x 18` is built, and each entry at `self.p[i][j]` represents the `2^j`-th ancestor of node `i`. Since `18` is a constant which relates to the potential `log n` levels, the space complexity can be considered `O(n * log n)`.

In conclusion, the pre-processing step has a time complexity of `O(n * log n)` and also a space complexity of `O(n * log n)`. The query step has a time complexity of `O(log n)` per query. These complexities are assuming that bitwise operations and list indexing are `O(1)`.