

BLACKBOXAI

Generated on: 7/16/2025 10:13:48 PM

User

// Optimize the below code and explain the difference between the provided code and optimized code with time complexity analysis with detailed explanation.

```
public int getUltimateParent (int node) { if (parent[node] != node) return getUltimateParent(parent[node]); return node; }
```

BLACKBOXAI

The provided code is a recursive implementation of a function that finds the ultimate parent (or root) of a node in a disjoint-set (or union-find) data structure. The function checks if the current node is its own parent; if not, it recursively calls itself with the parent of the current node until it finds the ultimate parent.

Provided Code Analysis

```
java
public int getUltimateParent(int node) {
    if (parent[node] != node)
        return getUltimateParent(parent[node]);
    return node;
}
```

Time Complexity

- The time complexity of this recursive approach in the worst case is $O(h)$, where (h) is the height of the tree representing the disjoint-set. In the worst case, this can be linear, $O(n)$, if the tree is unbalanced (i.e., a chain of nodes).

Optimized Code

Generated by BlackBox AI

To optimize this code, we can use path compression. This technique flattens the structure of the tree whenever we find the ultimate parent, making future queries faster.

Here's the optimized version of the code:

```
java
public int getUltimateParent(int node) {
    if (parent[node] != node) {
        parent[node] = getUltimateParent(parent[node]); // Path compression
    }
    return parent[node];
}
```

Explanation of the Optimized Code

- The key difference is the addition of path compression. When we find the ultimate parent of a node, we also update the parent of that node to point directly to the ultimate parent. This reduces the height of the tree, leading to faster future queries.

Time Complexity

- With path compression, the time complexity of the `getUltimateParent` function becomes nearly constant, specifically $O(\alpha(n))$, where α is the inverse Ackermann function. This function grows very slowly, meaning that for all practical purposes, the time complexity can be considered almost constant.

Summary of Differences

- 1. Recursion vs. Path Compression:

- The original code uses simple recursion without any optimization, leading to potentially deep recursion and higher time complexity. - The optimized code uses path compression to flatten the tree structure, significantly improving the efficiency of future queries.

- 2. Time Complexity:

- Original Code: ($O(h)$) (up to ($O(n)$) in the worst case). - Optimized Code: ($O(\alpha(n))$), which is nearly constant for all practical input sizes.

By implementing path compression, we ensure that the disjoint-set operations become much more efficient, especially when dealing with a large number of nodes and queries.