1948. Delete Duplicate Folders in System

Problem Description

In this problem, we have a file system with many duplicate folders due to a bug. We are given a 2D array `paths`, where `paths[i]` is an array representing an absolute path to the `i`th folder in the file system.

Two folders (not necessarily on the same level) are identical if they contain the same non-empty set of identical subfolders and underlying subfolder structure. The folders do not need to be at the root level to be identical. If two or more folders are identical, then we need to mark the folders as well as all their subfolders.

The file system will delete all the marked folders and their subfolders once and then return the 2D array `ans` containing the remaining paths after the deletion. The paths can be returned in any order.

Example:

Let's walk through an example to better understand the problem.

Given the input `paths`:

```
[
  ["a"],
  ["c"],
  ["a", "x"],
  ["c", "x"],
  ["b"],
  ["c", "x", "y"],
  ["a", "x", "y"]
]
```

The file system looks like:

```
- a
  - x
    - y
- b
- c
  - x
    - y
```

As we can see, folders "a" and "c" have the same structure and same subfolders. Hence, we need to mark and delete them along with their subfolders.

After deleting, the remaining file system looks like:

```
- b
```

So, the output `ans` should be: `[["b"]]`.

Solution Explanation

To solve this problem, we can utilize a Trie data structure to represent the folder structure. Each Trie node will have a `children` map and a `deleted` flag. The key in the `children` map will be the folder name (a string) and the value will be the child Trie node representing the subfolder.

We can follow these steps to remove the duplicate folders:

1. Create and populate the Trie based on the given input `paths`.
2. Traverse the Trie recursively and build a unique representation of the subtree rooted at each Trie node. We can use the subtree string representation to maintain a map pointing to the Trie nodes with the same subtree strings.
3. Check the map created in step 2. If any subtree string has multiple Trie nodes, those nodes (and their subfolders) are duplicates, so we mark them as `deleted`.
4. Starting from the root, traverse the Trie again and construct the remaining paths by ignoring the marked Trie nodes.

Let's discuss the solution with an example.

```
paths = [
  ["a"],
  ["c"],
  ["a", "x"],
  ["c", "x"],
  ["b"],
  ["c", "x", "y"],
  ["a", "x", "y"]
]
```

First, we populate the Trie based on the paths (step 1):

```
- root
  - a
    - x
      - y
  - b
  - c
    - x
      - y
```

The traversal of the Trie to build subtree string representations (step 2) results in:

```
root => "((a(x(y)))(b)(c(x(y))))"
a    => "(x(y))"
x    => "(y)"
y    => "()"
b    => "()"
c    => "(x(y))"
```

The subtree string to Trie nodes map (`subtreeToNodes` in the code) will have:

```
{
  "()": [y, b],
  "(y)": [x],
  "(x(y))": [a, c],
  "((a(x(y)))(b)(c(x(y))))": [root]
}
```

As we can see, the subtree string "(x(y))" has two Trie nodes, so they are duplicates (step 3). Mark the nodes "a" and "c" as deleted:

```
- root (not deleted)
  - a (deleted)
    - x (deleted)
      - y (deleted)
  - b (not deleted)
  - c (deleted)
    - x (deleted)
      - y (deleted)
```

Now, traverse the Trie again to construct the remaining folder paths (step 4):

```
["b"]
```

Here is the final C++ implementation of the solution provided:

```
cpp
struct TrieNode {
  unordered_map<string, shared_ptr<TrieNode>> children;
  bool deleted = false;
};

class Solution {
 public:
  vector<vector<string>> deleteDuplicateFolder(vector<vector<string>>& paths) {
    vector<vector<string>> ans;
    vector<string> path;
    unordered_map<string, vector<shared_ptr<TrieNode>>> subtreeToNodes;

    sort(begin(paths), end(paths));

    for (const vector<string>& path : paths) {
      shared_ptr<TrieNode> node = root;
      for (const string& s : path) {
        if (!node->children.count(s))
          node->children[s] = make_shared<TrieNode>();
        node = node->children[s];
      }
    }

    buildSubtreeToRoots(root, subtreeToNodes);

    for (const auto& [_, nodes] : subtreeToNodes)
      if (nodes.size() > 1)
        for (shared_ptr<TrieNode> node : nodes)
          node->deleted = true;

    constructPath(root, path, ans);
    return ans;
  }

 private:
  shared_ptr<TrieNode> root = make_shared<TrieNode>();

  string buildSubtreeToRoots(
      shared_ptr<TrieNode> node,
      unordered_map<string, vector<shared_ptr<TrieNode>>>& subtreeToNodes) {
    string subtree = "(";
    for (const auto& [s, child] : node->children)
      subtree += s + buildSubtreeToRoots(child, subtreeToNodes);
    subtree += ")";
    if (subtree != "()")
      subtreeToNodes[subtree].push_back(node);
    return subtree;
  }

  void constructPath(shared_ptr<TrieNode> node, vector<string>& path,
                     vector<vector<string>>& ans) {
    for (const auto& [s, child] : node->children)
      if (!child->deleted) {
        path.push_back(s);
        constructPath(child, path, ans);
        path.pop_back();
      }
    if (!path.empty())
      ans.push_back(path);
  }
};
```

This solution's time complexity will be O(N * L) where N is the number of paths and L is the length of the strings involved. The space complexity will be O(N * L) as well due to the Trie data structure.## Python Solution

Now, let's implement the same solution in Python:

```
python
from collections import defaultdict

class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.deleted = False

class Solution:
    def deleteDuplicateFolder(self, paths: List[List[str]]) -> List[List[str]]:
        ans = []
        path = []
        subtree_to_nodes = defaultdict(list)

        paths.sort()

        root = TrieNode()
        for p in paths:
            node = root
            for s in p:
                node = node.children[s]

        self.build_subtree_to_nodes(root, subtree_to_nodes)

        for nodes in subtree_to_nodes.values():
            if len(nodes) > 1:
                for node in nodes:
                    node.deleted = True

        self.construct_path(root, path, ans)
        return ans

    def build_subtree_to_nodes(self, node: TrieNode, subtree_to_nodes: dict) -> str:
        subtree = "("
        for s, child in node.children.items():
            subtree += s + self.build_subtree_to_nodes(child, subtree_to_nodes)
        subtree += ")"
        if subtree != "()":
            subtree_to_nodes[subtree].append(node)
        return subtree

    def construct_path(self, node: TrieNode, path: List[str], ans: List[List[str]]):
        for s, child in node.children.items():
            if not child.deleted:
                path.append(s)
                self.construct_path(child, path, ans)
                path.pop()
        if path:
            ans.append(path[:])
```

This Python solution has the same time complexity O(N * L) and space complexity O(N * L).

JavaScript Solution

Finally, let's implement the solution in JavaScript:

```
javascript
class TrieNode {
    constructor() {
        this.children = new Map();
        this.deleted = false;
    }
}

class Solution {
    deleteDuplicateFolder(paths) {
        const ans = [];
        const path = [];
        const subtreeToNodes = new Map();

        paths.sort();

        const root = new TrieNode();
        for (const p of paths) {
            let node = root;
            for (const s of p) {
                if (!node.children.has(s))
                    node.children.set(s, new TrieNode());
                node = node.children.get(s);
            }
        }

        this.buildSubtreeToNodes(root, subtreeToNodes);

        for (const nodes of subtreeToNodes.values()) {
            if (nodes.length > 1) {
                for (const node of nodes) {
                    node.deleted = true;
                }
            }
        }

        this.constructPath(root, path, ans);
        return ans;
    }

    buildSubtreeToNodes(node, subtreeToNodes) {
        let subtree = "(";
        for (const [s, child] of node.children.entries()) {
            subtree += s + this.buildSubtreeToNodes(child, subtreeToNodes);
        }
        subtree += ")";
        if (subtree !== "()") {
            if (!subtreeToNodes.has(subtree))
                subtreeToNodes.set(subtree, []);
            subtreeToNodes.get(subtree).push(node);
        }
        return subtree;
    }

    constructPath(node, path, ans) {
        for (const [s, child] of node.children.entries()) {
            if (!child.deleted) {
                path.push(s);
                this.constructPath(child, path, ans);
                path.pop();
            }
        }
        if (path.length > 0)
            ans.push(Array.from(path));
    }
}
```