71. Simplify Path

Medium Stack String

Problem Description

In the given problem, we are asked to take an absolute path for a file or directory in a Unix-style file system and convert it to a simplified canonical path.

An absolute path is a full path from the root of the file system and starts with a slash (`'/'`). The path may contain several special components:

- A single period (`.`) represents the current directory.
- A double period (`..`) represents moving up one directory level.
- Multiple consecutive slashes (`//`) are considered identical to a single slash (`/`).

The goal is to take such a path and simplify it according to the rules of Unix file systems so that:

- The simplified path must begin with a single slash (`/`).
- Each pair of directories must be separated by a single slash (`/`).
- The path must not end with a trailing slash (`/`).
- The path should not contain any `.` or `..`, as they should be resolved to actual directories on the path from the root to the target.

For example, given the path `"/a//b////c/d//././/.."`, the simplified canonical path would be `"/a/b/c"`.

Intuition

The intuition behind the solution involves using a stack to process each component of the path from left to right. A stack is ideal for this task because it allows us to add and remove directories in the correct order - the last directory we moved into is the first one we'll move out of when we encounter a `..` directive.

Here is how we can break down the problem and use a stack to solve it:

1. Split the path by slashes, which gives us a list of directory names and other components like `.` and `..`. We can then iterate through this list.
2. Ignore any empty strings resulting from consecutive slashes, as well as any `.` components, since they don't change the current directory.
3. If a `..` is encountered, check if there are any directories in the stack that we can "move up" from. If the stack is not empty, we pop the last directory off, effectively moving up a directory level.
4. Add any other directory names to the stack, as they represent moving into a new directory.
5. Once we've processed all components of the path, we combine the directories in the stack to form the simplified canonical path. To adhere to Unix-style paths, we ensure that the path begins with a slash and each directory is separated by a single slash.
6. We do not add a trailing slash, because the canonical path format specifies that the path should not end with one.

Using this approach allows us to handle complex, redundant, or relative paths and convert them into their simplest form, which is the essence of simplifying a canonical path in a Unix-style file system.

Solution Approach

The implementation of this solution relies on using a stack data structure, which fits perfectly for scenarios where we need to process items in a last-in, first-out manner. In the context of file paths, this method is beneficial for handling directories and the `..` component that implies moving up one directory level. Below is the step-by-step breakdown of the algorithm based on the solution approach given:

1. The `path` is split into components using the `'/'` as a delimiter using the `split()` function, which gives us a list of directories and possibly some `'.'` and `'..'` components.
2. An empty stack `stk` is initialized to keep track of the directory names that we encounter as we iterate through the list of path components.
3. We begin iterating over each component from the list. There are a few possible cases for each component `s`:
   - If `s` is an empty string or `'.'`, which can happen if we have consecutive slashes or a period that represents the current directory, we do nothing and continue to the next component.
   - If `s` is `'..'`, we check if there is anything in the stack. If the stack is not empty, which means there are previous directories we can move up from, we `pop()` the top element from the stack.
   - For all other cases, the component `s` is a directory name and is pushed onto the stack using `append(s)`.
4. After processing all components, we need to construct the canonical path from the elements in the stack. We do this by joining the elements of the stack with a `'/'` delimiter between them and prepend a `'/'` to represent the root directory, ensuring that our resulting path correctly starts with a single slash.

The final return statement `'/' + '/'.join(stk)` effectively builds our simplified canonical path from the stack. It's important to note that the stack enables us to handle backtrack operations caused by `'..'` components efficiently, allowing us to simplify the path correctly as we iterate through the components only once. This solution ensures that we avoid any redundant or unnecessary operations and achieve a clean, concise path as the output.

Example Walkthrough

Let's apply the solution approach to a small example path: `"/home//foo/./bar/../baz/"`

According to the approach:

1. Split the path by slashes to get the components: `["home", "", "foo", ".", "bar", "..", "baz", ""]`.
2. Initialize an empty stack `stk`: `[]`.
3. Iterate over each component:
   - Skip `""` and `"."`.
   - `home`: Push onto the stack `["home"]`.
   - `foo`: Push onto the stack `["home", "foo"]`.
   - `bar`: Push onto the stack `["home", "foo", "bar"]`.
   - `..`: Pop from the stack to get `["home", "foo"]`.
   - `baz`: Push onto the stack `["home", "foo", "baz"]`.
   - Skip `""` at the end, since the path should not end with a trailing slash.
4. Construct the canonical path by joining the elements in the stack with `'/'`, and prepend a `'/'` to the result. The canonical path is `"/home/foo/baz"`.
5. Return the final simplified canonical path: `"/home/foo/baz"`.

In this example, the stack has allowed us to keep track of the directories we have moved into and efficiently handle the case when we needed to move back up a directory due to the `".."` component. The resulting path follows all the rules for a simplified canonical path and gives us the correct simple path from a more complex and redundant one.

Solution Implementation

Python

```python
class Solution:
    def simplifyPath(self, path: str) -> str:
        # Initialize an empty list to use as a stack
        stack = []

        # Split the path by "/", iterate over each part
        for part in path.split('/'):
            # If the part is an empty string or a ".", simply continue to the next part
            if not part or part == '.':
                continue
            # If the part is "..", pop from the stack if it's not empty
            elif part == '..':
                if stack:
                    stack.pop()
            # Otherwise, add the part to the stack
            else:
                stack.append(part)

        # Join the stack elements to form the simplified path and prepend with "/"
        simplified_path = '/' + '/'.join(stack)
        return simplified_path
```

Java

```java
class Solution {
    public String simplifyPath(String path) {
        // Use a deque as a stack to hold the directory names.
        Deque<String> stack = new ArrayDeque<>();

        // Split the path by "/" and iterate over the segments.
        for (String segment : path.split("/")) {
            // If the segment is empty or a single ".", just ignore it.
            if (segment.isEmpty() || ".".equals(segment)) {
                continue;
            }
            // If the segment is "..", pop an element from the stack if available.
            if ("..".equals(segment)) {
                if (!stack.isEmpty()) {
                    stack.pollLast();
                }
            } else {
                // Push the directory name onto the stack.
                stack.offerLast(segment);
            }
        }

        // Join all the elements in the stack with "/", prepended by a "/" to form the simplified path.
        String simplifiedPath = "/" + String.join("/", stack);

        // Return the simplified absolute path.
        return simplifiedPath;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to simplify the given Unix-like file path.
    string simplifyPath(string path) {
        deque<string> directoryNames; // Use a deque to store the directory names after parsing.
        stringstream ss(path); // Create a stringstream to separate the elements by '/'.
        string token; // String to store the separated elements.

        // Process each part of the path separated by '/'.
        while (getline(ss, token, '/')) {

            // Continue if the element is empty or a dot, which means stay in the current directory.
            if (token == "" || token == ".") {
                continue;
            }

            // If element is a double dot, move up to the parent directory if possible.
            if (token == "..") {
                // Only pop if the stack is not empty (cannot go above root).
                if (!directoryNames.empty()) {
                    directoryNames.pop_back();
                }
            } else {
                // Otherwise, it's a valid directory name; add to our list.
                directoryNames.push_back(token);
            }
        }

        // If directory stack is empty, we're at root.
        if (directoryNames.empty()) {
            return "/";
        }

        // Build the simplified path from the directory stack.
        string result;
        for (const auto& dirName : directoryNames) {
            result += "/" + dirName; // Prefix each directory name with a slash.
        }

        // Return the final simplified path.
        return result;
    }
};
```

TypeScript

```typescript
function simplifyPath(path: string): string {
    // Initialize an empty stack to store the parts of the simplified path
    const pathStack: string[] = [];

    // Split the input path by '/' and iterate through the segments
    for (const segment of path.split('/')) {
        // Skip empty segments and current directory references '.'
        if (segment === '' || segment === '.') {
            continue;
        }
        // If segment is the parent directory reference '..'
        if (segment === '..') {
            // Pop the last element from the stack if it's not empty
            if (pathStack.length) {
                pathStack.pop();
            }
        } else {
            // Push the current directory segment onto the stack
            pathStack.push(segment);
        }
    }

    // Join the stack elements with '/' to form the simplified path
    // Ensure to start the path with the root directory '/'
    return '/' + pathStack.join('/');
}
```

```python
class Solution:
    def simplifyPath(self, path: str) -> str:
        # Initialize an empty list to use as a stack
        stack = []

        # Split the path by "/", iterate over each part
        for part in path.split('/'):
            # If the part is an empty string or a ".", simply continue to the next part
            if not part or part == '.':
                continue
            # If the part is "..", pop from the stack if it's not empty
            elif part == '..':
                if stack:
                    stack.pop()
            # Otherwise, add the part to the stack
            else:
                stack.append(part)

        # Join the stack elements to form the simplified path and prepend with "/"
        simplified_path = '/' + '/'.join(stack)
        return simplified_path
```

Time and Space Complexity

The time complexity of the given code is `O(n)`. This is because we are traversing the entire input `path` once with the `path.split('/')` operation, and each of the split operations (inserting into stack and popping from stack) run in constant time `O(1)`. We join the stack at the end to form the simplified path but joining is also linear to the number of elements in the stack, which is at most `n`.

The space complexity is `O(n)` as we are potentially storing all the parts of the path in the stack `stk`. In the worst case scenario, the path does not contain any `".."` or `"."` and is not optimized thus we would have to store each part of the path in the stack.