Previous Next

71. Simplify Path

Feb. 16, 2020 | 8.6K views

Average Rating: 4.91 (11 votes)

Given an **absolute path** for a file (Unix-style), simplify it. Or in other words, convert it to the **canonical path**.

In a UNIX-style file system, a period `.` refers to the current directory. Furthermore, a double period `..` moves the directory up a level.

Note that the returned canonical path must always begin with a slash `/`, and there must be only a single slash `/` between two directory names. The last directory name (if it exists) **must not** end with a trailing `/`. Also, the canonical path must be the **shortest** string representing the absolute path.

Example 1:

```
Input: "/home/"
Output: "/home"
Explanation: Note that there is no trailing slash after the last directory name.
```

Example 2:

```
Input: "/../"
Output: "/"
Explanation: Going one level up from the root directory is a no-op, as the root level
```

Example 3:

```
Input: "/home//foo/"
Output: "/home/foo"
Explanation: In the canonical path, multiple consecutive slashes are replaced by a sir
```

Example 4:

```
Input: "/a/./b/../../c/"
Output: "/c"
```

Example 5:

```
Input: "/a/../../b/../c//.//"
Output: "/c"
```

Example 6:

```
Input: "/a//b////c/d//././/.."
Output: "/a/b/c"
```

Solution

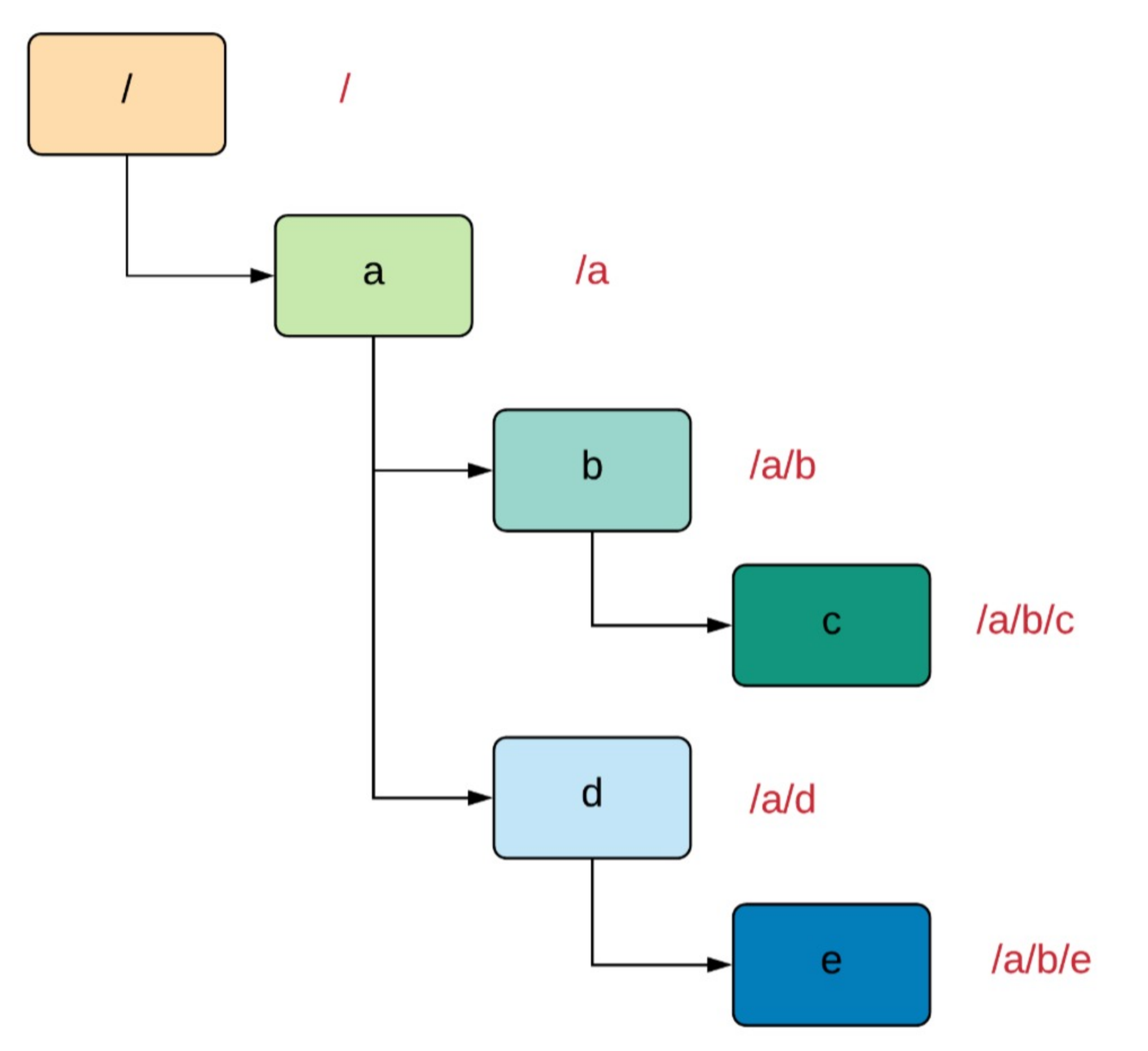
Approach: Using Stacks

Intuition

This is a direct implementation of one of the most common commands used (in one form of the other) on all the famous operating systems. For e.g. this directly translates to a part of the implementation of the `cd` command functionality in the Unix OS. It basically helps us to change directory. Obviously, there's much more to it than simply figuring out the smallest path of the final directory where the user wants to go. There is kernel level implementation involved that actually uses the underlying file system to `change` the directory you are in to the one where you want to go. You might argue that nobody would go crazy on the command line and want to run something like:

```
cd /a/b/c/.././././//d
```

However, our code needs to be able to handle the different scenarios and all the special characters like `.`, `..`, and `/`. All these weird scenarios will be taken care of by a couple of checks here and there. The main idea for solving this problem would remain the same. The heading of this approach mentions the data structure that we are going to use. However, we are going to work our way through the problem so as to understand as to why a `stack` fits in here. Let's look at a tree-ish representation of a simple directory path.



Now suppose that to a path like `/a/b/c`, we add another component like `/a/b/c/..`. Now, this is interesting because the `..` is no longer a sub-directory name. It has a special meaning and an indication to the operating system to move up one level in the directory structure thus transforming the overall path to just `/a/b`. It's as if we `popped out` the subdirectory `c` from the overall path. That's the core idea of this problem. If you think about it, the only actionable special character is `..`. The single dot is kind of a no-op because it simply means the current directory. So nothing changes in the overall path as such. Now that we have a general idea about how this problem can be potentially solved using the stack, let's get right into the algorithm and discuss the solution.

Algorithm

1. Initialize a stack, `S` that we will be using for our implementation.
2. Split the input string using `/` as the delimiter. This step is really important because no matter what, the given input is a `valid` path and we simply have to shorten it. So, that means that whatever we have between two `/` characters is either a directory name or a special character and we have to process them accordingly.
3. Once we are done splitting the input path, we will process one component at a time.
4. If the current component is a `.` or an empty string, we will do nothing and simply continue. Well if you think about it, the split string array for the string `/a//b` would be `[a,,b]`. yes, that's an empty string in between `a` and `b`. Again, from the perspective of the overall path, it doesn't mean anything.
5. If we encounter a double-dot `..`, we have to do some processing. This simply means go one level up in the current directory path. So, we will pop an entry from our stack if it's not empty.
6. Finally, if the component we are processing right now is not one of the special characters, then we will simply add it to our stack because it's a legitimate directory name.
7. Once we are done processing all the components, we simply have to connect all the directory names in our stack together using `/` as the delimiter and we will have our shortest path that leads us to the same directory as the one provided as an input.

Java Python Copy

```python
class Solution:
    def simplifyPath(self, path: str) -> str:

        # Handle empty string
        if not str:
            return str

        # Initialize a stack
        stack = []

        # Split the input string on "/" as the delimiter
        # and process each portion one by one
        for portion in path.split("/"):

            # If the current component is a "..", then
            # we pop an entry from the stack if it's non-empty
            if portion == "..":
                if stack:
                    stack.pop()
            elif portion == "." or not portion:
                # A no-op for a "." or an empty string
                continue
            else:
                # Finally, a legitimate directory name, so we add it
                # to our stack
                stack.append(portion)

```

Complexity Analysis

- Time Complexity: $O(N)$ if there are N characters in the original path. First, we spend $O(N)$ trying to split the input path into components and then we process each component one by one which is again an $O(N)$ operation. We can get rid of the splitting part and just string together the characters and form directory names etc. However, that would be too complicated and not worth depicting in the implementation. The main idea of this algorithm is to use a stack. How you decide to process the input string is a personal choice.
- Space Complexity: $O(N)$. Actually, it's $2N$ because we have the array that contains the split components and then we have the stack.

Rate this article: ★★★★★

Previous Next

Comments: 8

Sort By

Type comment here... (Markdown is supported)

Preview Post

Zbeeee ★62 March 4, 2020 3:11 AM

Can't figure out what is canonical path.

13 Share Reply

SHOW 1 REPLY

beasr ★17 May 7, 2020 7:17 AM Report

Not sure why this question is medium level difficulty, it's pretty straightforward.

17 Share Reply

albine ★5 March 17, 2020 6:32 PM

How is triple dots "..." specified? On my Mac it's like go up 2 directories, but the test case expects leaving it as is. Is it a legal directory name or a legal directive or anything else? Also the link "Absolute path vs relative path in Linux/Unix" in the problem desciption is broken.

5 Share Reply

SHOW 2 REPLIES

r1ce ★3 May 24, 2020 1:25 AM

Better to use a deque rather than a stack I think. Java's implementation of a stack allows iteration in the wrong direction which this solution is taking advantage of.

3 Share Reply

SHOW 1 REPLY

rishabhpandita ★26 May 2, 2020 7:09 AM Report

"..." should be an invalid test case

2 Share Reply

takenpilot ★2 May 5, 2020 7:41 AM

JavaScript solution:

```
var simplifyPath = function(path) {
  const split = path.split('/');
  const finalPath = [];
```

Read More

1 Share Reply

geekidharsh ★1 March 23, 2020 3:15 PM Report

In the example tree above, absolute path of "e" should be **/a/d/e and not /a/b/e** , otherwise this is a great explanation. Thanks

1 Share Reply

manojkrajasekar ★35 March 19, 2020 8:17 PM

Awesome Explanation !

1 Share Reply