<https://leetcode.com/problems/simplify-path/description/>

Given an absolute path for a Unix-style file system, which begins with a slash '/', transform this path into its simplified canonical path.

In Unix-style file system context, a single period '.' signifies the current directory, a double period ".." denotes moving up one directory level, and multiple slashes such as "//" are interpreted as a single slash. In this problem, treat sequences of periods not covered by the previous rules (like "...") as valid names for files or directories.

The simplified canonical path should adhere to the following rules:

1. It must start with a single slash '/'.
2. Directories within the path should be separated by only one slash '/'.
3. It should not end with a slash '/', unless it's the root directory.
4. It should exclude any single or double periods used to denote current or parent directories.

Return the new path.

**Example 1:**

Input: path = "/home/"

Output: "/home"

Explanation:

The trailing slash should be removed.

**Example 2:**

Input: path = "/home//foo/"

Output: "/home/foo"

Explanation:

Multiple consecutive slashes are replaced by a single one.

**Example 3:**

Input: path = "/home/user//../PiDocumentsctures"

Output: "/home/user/Pictures"

Explanation:

A double period ".." refers to the directory up a level.

**Example 4:**

Input: path = "/../"

Output: "/"

Explanation:

Going one level up from the root directory is not possible.

**Example 5:**

Input: path = "/.../a/../b/c/../d/./"

Output: "/.../b/d"

Explanation:

"..." is a valid name for a directory in this problem.

**Constraints:**

1. 1 <= path.length <= 3000
2. path consists of English letters, digits, period '.', slash '/' or '\_'.
3. path is a valid absolute Unix path.

**Attempt 1: 2024-05-29**

**Solution 1: Stack (30 min)**

class Solution {

    public String simplifyPath(String path) {

        Stack<String> stack = new Stack<>();

        for(String segment : path.split("/")) {

            // If the segment is empty or a single ".", just ignore it.

            if(segment.isEmpty() || segment.equals(".")) {

                continue;

            }

            // If the segment is "..", pop an element from the stack if available.

            if(segment.equals("..")) {

                if(!stack.isEmpty()) {

                    stack.pop();

                }

            } else {

                stack.push(segment);

            }

        }

        // Join all the elements in the stack with "/", prepended by a "/" to

        // form the simplified path.

        return "/" + String.join("/", stack);

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/simplify-path/solutions/1847526/best-explanation-ever-possible-not-a-clickbait/>

How's going Ladies - n - GentleMen, today we are going to solve another coolest problem i.e. Simplify Path

Alright, so let's understand this problem with an example :-

Input: path = "/a/./b/../../c/"

Output: "/c"

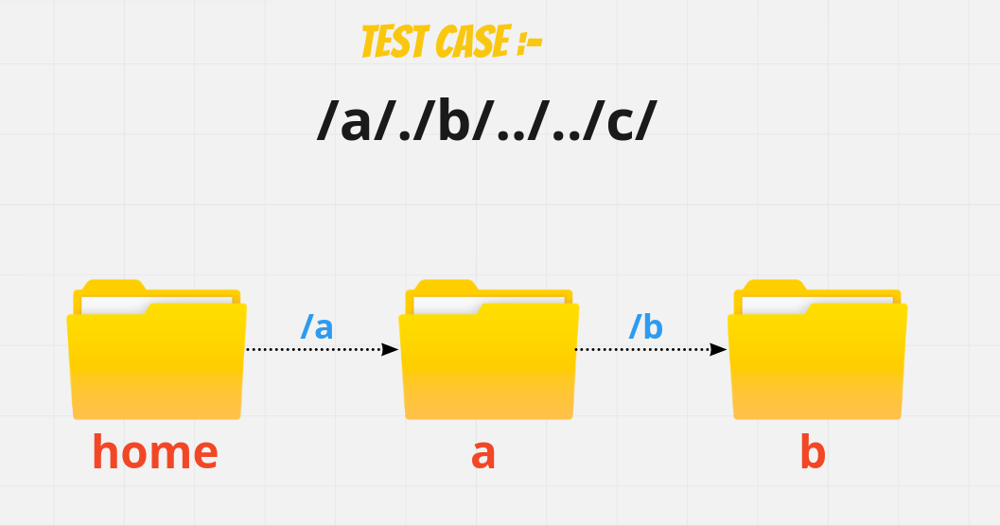
Let's understand what this mean's, so basically this seems like you a path of your folder, so generically we use this kind of command's in terminal. I hope u know a bit about that. Anyway's let's move further on.

okay, so the first command /a/ means get into the folder /a/

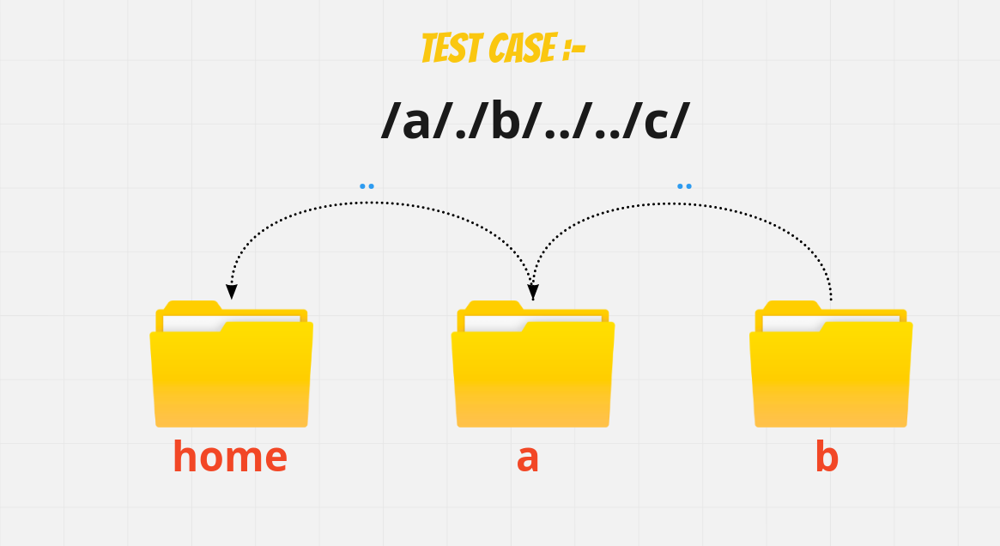
1. The next command is /./ means stay over there
2. The next command is /b/ means get into the folder /b/
3. The next command is /../ means come out from the folder /b/
4. The next command is /../ means come out from the folder /a/
5. Now we are kind of in home directory
6. The next command is /c/ means get into the folder /c/
7. And in the output we return what command we left with.

Let's understand it a bit visually.

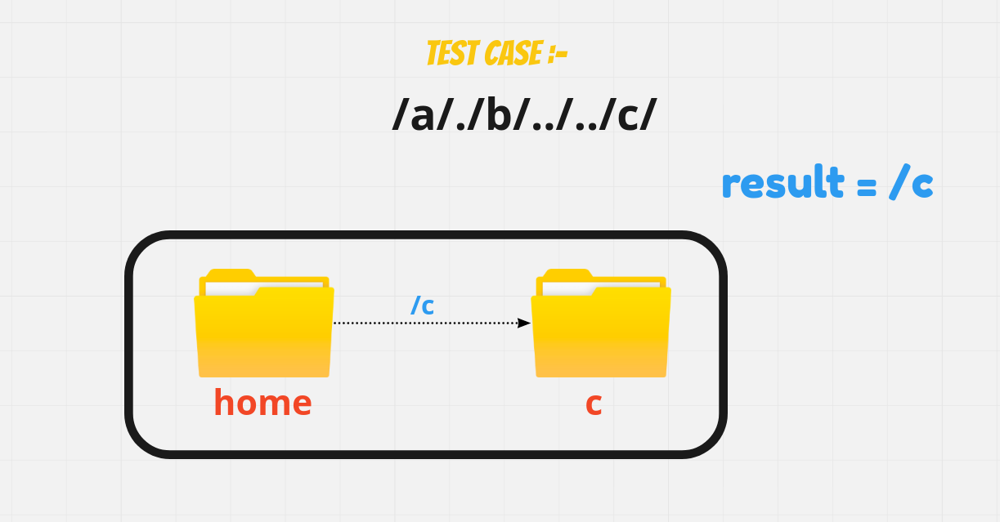
1. Going to /a/./b/



1. Coming out /../../



1. Going to /c/



So, basically what are we doing:-

Pushing and Popping directory names based on rules

And what are the rules :-

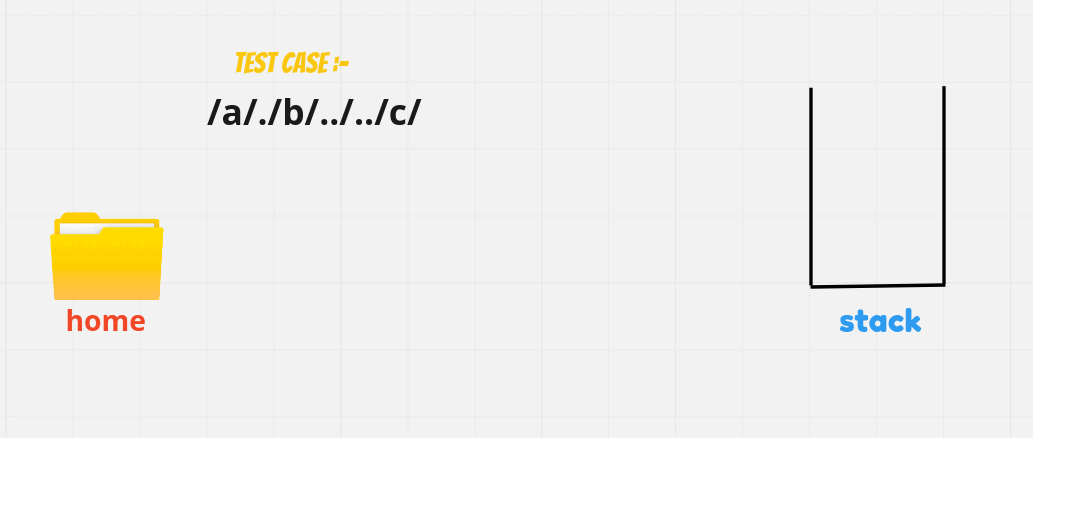
1. /.. come out from the directory
2. /nameOfDirectory going into directory

Now you say, Dude that's A-OK but which Data Structure do we use to solve this problem. Well. i had already gives you a hint if you remember Pushing and Popping. So where do we Generically used in Stack or Queue

We'll solve this problem using Stack. But if you want the solution of Queue as well, do lemmino (:

Alright, back to the problem - So, what we can do is by looking at the rules, split the directrory by the slash/ given and that will give us in the form of array e.g :- [a, ., b, .., .., c]

Let's understand it's working visually:-



But remeber when returning we have to go in the form of reverse order. Because Stack use LIFO order and the highest one will comes out. But we need the lowest once first. So, we need to append in the carefull manner.

I hope ladies and gentlemen approach is clear Let's code it up

class Solution {

public String simplifyPath(String path) {

Stack<String> s = new Stack<>();

StringBuilder res = new StringBuilder();

String[] p =path.split("/");

for(int i=0;i<p.length;i++){

if(!s.isEmpty() && p[i].equals("..")) s.pop();

else if(!p[i].equals("") && !p[i].equals(".") && !p[i].equals(".."))

s.push(p[i]);

}

if(s.isEmpty()) return "/";

while(!s.isEmpty()){

res.insert(0,s.pop()).insert(0,"/");

}

return res.toString();

}

}

**Refer to**

<https://algo.monster/liteproblems/71>

**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:

1. A single period (.) represents the current directory.
2. A double period (..) represents moving up one directory level.
3. 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:

1. The simplified path must begin with a single slash (/).
2. Each pair of directories must be separated by a single slash (/).
3. The path must not end with a trailing slash (/).
4. 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:
   1. 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.
   2. 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.
   3. 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:
   1. Skip "" and ".".
   2. home: Push onto the stack ["home"].
   3. foo: Push onto the stack ["home", "foo"].
   4. bar: Push onto the stack ["home", "foo", "bar"].
   5. ..: Pop from the stack to get ["home", "foo"].
   6. baz: Push onto the stack ["home", "foo", "baz"].
   7. 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**

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;

}

}

**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.