<https://leetcode.ca/all/1166.html>

You are asked to design a[file system](https://leetcode.ca/all/1166.html#) which provides two functions:

**createPath(path, value)**: Creates a new path and associates a value to it if possible and returns True. Returns False if the path already exists or its parent path doesn't exist.

**get(path)**: Returns the value associated with a path or returns -1 if the path doesn't exist.

The format of a path is one or more concatenated strings of the form: / followed by one or more lowercase English letters.

For example, /leetcode and /leetcode/problems are valid paths while an empty string and / are not.

Implement the two functions.

Please refer to the examples for clarifications.

**Example 1:**

**Input:**

["FileSystem","createPath","get"]

[[],["/a",1],["/a"]]

**Output:**

[null,true,1]

**Explanation:**

FileSystem fileSystem = new FileSystem();

fileSystem.createPath("/a", 1); // return true

fileSystem.get("/a"); // return 1

**Example 2:**

**Input:**

["FileSystem","createPath","createPath","get","createPath","get"]

[[],["/leet",1],["/leet/code",2],["/leet/code"],["/c/d",1],["/c"]]

**Output:**

[null,true,true,2,false,-1]

**Explanation:**

FileSystem fileSystem = new FileSystem();

fileSystem.createPath("/leet", 1); // return true

fileSystem.createPath("/leet/code", 2); // return true

fileSystem.get("/leet/code"); // return 2

fileSystem.createPath("/c/d", 1); // return false because the parent path "/c" doesn't exist.

fileSystem.get("/c"); // return -1 because this path doesn't exist.

**Constraints:**

2 <= path.length <= 100

1 <= value <= 10^9

Each path is **valid** and consists of lowercase English letters and '/'.

At most 10^4 calls **in total** will be made to createPath and get.

**NOTE:** create method has been changed on August 29, 2019 to createPath. Please reset to default code definition to get new method signature.

**Attempt 1: 2025-3-15**

**Solution 1: Hash Table + Node structure + Design (60 min)**

import java.util.\*;

public class Solution {

class TrieNode {

Map<String, TrieNode> children;

int value;

public TrieNode(int value) {

this.children = new HashMap<>();

this.value = value;

}

}

class Trie {

TrieNode root = new TrieNode(-1);

// Creates a new path and associates a value to it if possible and returns True.

// Returns False if the path already exists or its parent path doesn't exist.

public boolean insert(String path, int value) {

// Problem: The code allows creating the root path ("/"), which is invalid.

// For example, createPath("/", 5) returns true, but it should return false.

// Cause: Splitting "/" results in ["", ""], and the code incorrectly treats

// this as a valid path.

// Explicitly reject creating the root path "/"

if (path.equals("/")) {

return false;

}

TrieNode node = root;

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

for(int i = 1; i < parts.length - 1; i++) {

// Returns false parent path doesn't exist

if(!node.children.containsKey(parts[i])) {

return false;

}

node = node.children.get(parts[i]);

}

// Returns false if the path already exists

if(node.children.containsKey(parts[parts.length - 1])) {

return false;

}

node.children.put(parts[parts.length - 1], new TrieNode(value));

return true;

}

public int search(String path) {

TrieNode node = root;

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

for(int i = 1; i < parts.length; i++) {

if(!node.children.containsKey(parts[i])) {

return -1;

}

node = node.children.get(parts[i]);

}

return node.value;

}

}

Trie trie = new Trie();

public boolean createPath(String path, int value) {

return trie.insert(path, value);

}

public int get(String path) {

return trie.search(path);

}

public static void main(String[] args) {

Solution so = new Solution();

boolean result = so.createPath("/", 1); // return false

boolean result1 = so.createPath("/leet", 1); // return false

boolean result2 = so.createPath("/leet/code", 2); // return true

int value1 = so.get("/leet/code"); // return 2

boolean result3 = so.createPath("/c/d", 1); // return false because the parent path "/c" doesn't exist.

int value2 = so.get("/c"); // return -1 because this path doesn't exist.

System.out.println(result);

System.out.println(result1);

System.out.println(result2);

System.out.println(value1);

System.out.println(result3);

System.out.println(value2);

}

}

Time: O(k) for both createPath and get, where k is the length of the input path.

Space: O(T), where T is the total number of components in all stored paths.

**Refer to**

<https://leetcode.ca/2019-02-08-1166-Design-File-System/>

Related to [588-Design-In-Memory-File-System](https://leetcode.ca/2017-07-10-588-Design-In-Memory-File-System/), and this question is simplified version of [588-Design-In-Memory-File-System](https://leetcode.ca/2017-07-10-588-Design-In-Memory-File-System/)

**Solution 1: Trie**

We can use a trie to store the paths, where each node stores a value, representing the value of the path corresponding to the node.

The structure of the trie node is defined as follows:

children: Child nodes, stored in a hash table, where the key is the path of the child node, and the value is the reference to the child node.

v: The value of the path corresponding to the current node.

The methods of the trie are defined as follows:

insert(w, v): Insert the path w and set its corresponding value to v. If the path w already exists or its parent path does not exist, return false, otherwise return true. The time complexity is O(|w|), where |w| is the length of the path w

search(w): Return the value corresponding to the path w. If the path w does not exist, return −1. The time complexity is O(|w|)

The total time complexity is O(∑w∈W|w|), and the total space complexity is O(∑w∈W|w|), where W is the set of all inserted paths.

class Trie {

Map<String, Trie> children = new HashMap<>();

int v;

Trie(int v) {

this.v = v;

}

boolean insert(String w, int v) {

Trie node = this;

var ps = w.split("/");

for (int i = 1; i < ps.length - 1; ++i) {

var p = ps[i];

if (!node.children.containsKey(p)) {

return false;

}

node = node.children.get(p);

}

if (node.children.containsKey(ps[ps.length - 1])) {

return false;

}

node.children.put(ps[ps.length - 1], new Trie(v));

return true;

}

int search(String w) {

Trie node = this;

var ps = w.split("/");

for (int i = 1; i < ps.length; ++i) {

var p = ps[i];

if (!node.children.containsKey(p)) {

return -1;

}

node = node.children.get(p);

}

return node.v;

}

}

class FileSystem {

private Trie trie = new Trie(-1);

public FileSystem() {

}

public boolean createPath(String path, int value) {

return trie.insert(path, value);

}

public int get(String path) {

return trie.search(path);

}

}

/\*\*

\* Your FileSystem object will be instantiated and called as such:

\* FileSystem obj = new FileSystem();

\* boolean param\_1 = obj.createPath(path,value);

\* int param\_2 = obj.get(path);

\*/

**Refer to Deepseek**

Okay, I need to find a Java solution for Leetcode problem 1166, which is about designing a file system. Let me read the problem statement again to make sure I understand it correctly.

So the task is to create a data structure that simulates a file system. There's a function createPath that adds a new path and assigns a value to it. But the path can only be created if its parent path already exists. For example, if we try to create "/a/b/c" without "/a/b" existing, it should return false. And if the path already exists, it should also return false. Otherwise, it creates the path and returns true. Then there's the get function, which returns the value associated with a path or -1 if the path doesn't exist.

Hmm, how to model this? Maybe a trie-like structure, where each node represents a directory or file, and has children. Each node can have a value. The root would be the empty path, then each part of the path is a node. For example, "/a/b/c" would be root -> a -> b -> c.

Wait, but the path is given as a string split by '/'. So first, we need to split the path into components. For example, "/a/b/c" becomes ["a", "b", "c"]. Then, starting from the root, we check each component step by step to see if the path exists.

So the data structure could be a tree, where each node has a map of children (like a dictionary of child nodes, keyed by the path component), and a value. The root node is the base.

For the createPath function: split the path into parts, check each part except the last one exists. Then add the last part to the parent's children, and assign the value. If any of the parent parts don't exist, return false. Also, if the last part already exists in the parent's children, return false.

For the get function: split the path into parts, traverse from the root, checking each part. If at any point a part is not found, return -1. Otherwise, return the value of the final node.

So how to implement this in Java.

Let's see. We can have a class for the Node. Each Node has a map from String to Node (for children), and an Integer value. The root node is initialized when the FileSystem is created.

Wait, but the problem says that the createPath function's path is a valid absolute path. So the path starts with '/', and the components are non-empty and valid. So when splitting the path, the first element after splitting would be empty (since the path starts with '/'), but then followed by the actual components.

Wait, for example, splitting "/a/b/c" by '/' would give ["", "a", "b", "c"]. So when processing the path, we can split on '/' and then ignore the first empty string. Then, the components would be "a", "b", "c".

So in code, when handling createPath(path, value), first split the path into parts. Let's say:

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

Then, we can iterate through the parts, skipping the first empty string (since the path starts with '/'), and then check if each part exists in the current node's children.

Wait, but maybe there are cases where parts have multiple slashes, but the problem states that the path is valid, so perhaps we don't have to handle that. So the split is safe.

So for createPath:

- Split the path into parts. Let's say parts = path.split("/");

- The parts array would have elements like ["", "a", "b", "c"] for path "/a/b/c".

- So the actual components are parts[1], parts[2], ..., parts[n-1].

- So we need to iterate from 1 to parts.length -1.

But wait, the problem says that the path has to be created, and the parent must exist. For example, creating "/a" would require that the parent is the root. So for the path "/a", the parent is the root. The path components after splitting would be ["", "a"], so the parts array length is 2. So the parent is parts[0] (empty string?), which is the root. Wait, maybe I should process the parts differently.

Wait, maybe for a given path, the parent is the path excluding the last component.

Wait, for example, the path "/a/b/c" has parent "/a/b". So when creating this path, we need to check if the parent exists. So how to split the path into components, check all except the last, and then add the last.

So steps for createPath:

1. Split the path into components, ignoring the leading empty string (since the path starts with '/'), and then check if the components array has at least one element (since the path can't be just "/", because the problem says the path is not empty and not just the root).

Wait, according to the problem statement, createPath parameters are (String path, int value). So path must be a valid absolute path, which starts with '/', and does not end with a '/' (unless it's the root). But the root path is "/", but according to the examples, the first example is creating "/a" with value 1. So perhaps the path is like "/a", "/a/b", etc.

So when processing the path, split by '/', and get the components. For example, "/a" splits into ["", "a"], and the components are "a". The parent of "/a" is the root. So to create "/a", we need to check that the root exists (which it does), then add "a" as a child of the root.

So for a general path:

components = path.split("/"); which gives something like ["", "a", "b", "c"].

Then, the parent components are all except the last element. So for "/a/b/c", the parent components are ["a", "b"], and the new component is "c". So the parent path is "/a/b".

To solve this problem, we need to design a file system that allows creating paths and associating values with them, provided the parent directories exist. Additionally, we need to retrieve values from existing paths.

**Approach**

The solution involves using a trie-like structure where each node represents a directory. Each node maintains a map of child directories and a value. The root node represents the root directory ("/"). Here's the detailed approach:

Node Structure: Each node contains a map of child nodes (directories) and an integer value. The root node is initialized with an empty map and a default value of -1.

Create Path:

Split the given path into components, ignoring empty strings resulting from splitting by "/".

Check if the path is valid (non-empty and not the root).

Traverse the trie from the root to the parent directory of the new path. If any parent directory does not exist, return false.

If the parent directories exist and the new directory does not, create it and set its value. Otherwise, return false.

Get Value:

Split the path into components and traverse the trie from the root.

If any directory in the path does not exist, return -1. Otherwise, return the value of the final directory.

**Solution Code**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

class FileSystem {

private static class Node {

Map<String, Node> children;

int value;

public Node() {

children = new HashMap<>();

value = -1;

}

public Node(int value) {

children = new HashMap<>();

this.value = value;

}

}

private Node root;

public FileSystem() {

root = new Node();

}

public boolean createPath(String path, int value) {

List<String> components = new ArrayList<>();

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

if (!part.isEmpty()) {

components.add(part);

}

}

if (components.isEmpty()) {

return false; // path is "/"

}

String lastComponent = components.get(components.size() - 1);

List<String> parentComponents = components.subList(0, components.size() - 1);

Node current = root;

for (String part : parentComponents) {

if (!current.children.containsKey(part)) {

return false;

}

current = current.children.get(part);

}

if (current.children.containsKey(lastComponent)) {

return false;

}

current.children.put(lastComponent, new Node(value));

return true;

}

public int get(String path) {

List<String> components = new ArrayList<>();

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

if (!part.isEmpty()) {

components.add(part);

}

}

if (components.isEmpty()) {

return -1;

}

Node current = root;

for (String part : components) {

if (!current.children.containsKey(part)) {

return -1;

}

current = current.children.get(part);

}

return current.value;

}

}

**Explanation**

Node Class: Each node has a map of child nodes and a value. The root node is initialized in the constructor of the FileSystem.

createPath Method: This method splits the path into components, checks if the parent path exists, and creates the new directory if valid.

get Method: This method splits the path into components, traverses the trie, and returns the value of the specified path or -1 if it doesn't exist.

This approach efficiently handles both creating paths and retrieving values using a trie structure, ensuring that each operation (create and get) runs in O(k) time complexity, where k is the number of components in the path. The space complexity is linear in the number of directories created.

**Time Complexity Analysis**

The solution uses a trie structure to manage paths. Here's the breakdown of time complexity for each operation:

createPath(path: String, value: int):

Splitting the Path: Splitting the string by / takes O(k), where k is the length of the path string.

Traversing the Trie: For a path with m components (e.g., /a/b/c has m = 3), traversal requires O(m) operations. Each step involves a hash map lookup/insertion, which is O(1) on average.

Total: O(k + m) → simplifies to O(k) since m ≤ k (each component is at least 1 character).

get(path: String):

Splitting the Path: Same as above, O(k).

Traversing the Trie: Again, O(m) steps for m components.

Total: O(k + m) → simplifies to O(k).

**Space Complexity**

Trie Structure: Each node in the trie occupies space proportional to the number of child nodes. For n valid paths, the worst-case space is O(T), where T is the total number of components across all paths (e.g., /a/b contributes 2 components).

**Summary**

Time: O(k) for both createPath and get, where k is the length of the input path.

Space: O(T), where T is the total number of components in all stored paths.

This makes the solution efficient for large-scale operations, as hash map operations dominate the time complexity and are highly optimized.

**Refer to**

[L588.Design In-Memory File System](note://6C9C569FCA024F08BBCF7D03FAD809BB)