Problem 1166: Design File System

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are asked to design a file system that allows you to create new paths and associate them with different values.

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

FileSystem

class:

bool createPath(string path, int 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

.

int get(string path)

Returns the value associated with

path

or returns

-1

if the path doesn't exist.

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

.

Code Snippets

C++:

```
class FileSystem {
public:
FileSystem() {

}

bool createPath(string path, int value) {

}

int get(string path) {

}
};

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem* obj = new FileSystem();
* bool param_1 = obj->createPath(path,value);
* int param_2 = obj->get(path);
*/
```

Java:

```
class FileSystem {

public FileSystem() {

}

public boolean createPath(String path, int value) {
```

```
}

public int get(String 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);
*/
```

Python3:

```
class FileSystem:

def __init__(self):


def createPath(self, path: str, value: int) -> bool:


def get(self, path: str) -> int:



# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.createPath(path,value)
# param_2 = obj.get(path)
```

Python:

```
class FileSystem(object):

def __init__(self):


def createPath(self, path, value):
```

```
        """
        :type path: str
        :type value: int
        :rtype: bool
        """


    def get(self, path):
        """
        :type path: str
        :rtype: int
        """



# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.createPath(path,value)
# param_2 = obj.get(path)
```

JavaScript:

```
var FileSystem = function() {

};

/**
* @param {string} path
* @param {number} value
* @return {boolean}
*/
FileSystem.prototype.createPath = function(path, value) {

};

/**
* @param {string} path
* @return {number}
*/
FileSystem.prototype.get = function(path) {
```

```
};

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.createPath(path,value)
* var param_2 = obj.get(path)
*/
```

TypeScript:

```
class FileSystem {
constructor() {

}

createPath(path: string, value: number): boolean {

}

get(path: string): number {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.createPath(path,value)
* var param_2 = obj.get(path)
*/
```

C#:

```
public class FileSystem {

public FileSystem() {

}

public bool CreatePath(string path, int value) {
```

```
}

public int Get(string path) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* bool param_1 = obj.CreatePath(path,value);
* int param_2 = obj.Get(path);
*/
```

C:

```
typedef struct {

} FileSystem;


FileSystem* fileSystemCreate() {

}

bool fileSystemCreatePath(FileSystem* obj, char* path, int value) {

}

int fileSystemGet(FileSystem* obj, char* path) {

}

void fileSystemFree(FileSystem* obj) {

}

/**
* Your FileSystem struct will be instantiated and called as such:
```

```
* FileSystem* obj = fileSystemCreate();
* bool param_1 = fileSystemCreatePath(obj, path, value);

* int param_2 = fileSystemGet(obj, path);

* fileSystemFree(obj);
*/
```

Go:

```
type FileSystem struct {

}


func Constructor() FileSystem {

}


func (this *FileSystem) CreatePath(path string, value int) bool {

}


func (this *FileSystem) Get(path string) int {

}


/**
* Your FileSystem object will be instantiated and called as such:
* obj := Constructor();
* param_1 := obj.CreatePath(path,value);
* param_2 := obj.Get(path);
*/
```

Kotlin:

```
class FileSystem() {

fun createPath(path: String, value: Int): Boolean {
```

```
}

fun get(path: String): Int {

}

}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = FileSystem()
* var param_1 = obj.createPath(path,value)
* var param_2 = obj.get(path)
*/
```

Swift:

```
class FileSystem {

init() {

}

func createPath(_ path: String, _ value: Int) -> Bool {

}

func get(_ path: String) -> Int {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem()
* let ret_1: Bool = obj.createPath(path, value)
* let ret_2: Int = obj.get(path)
*/
```

Rust:

```
struct FileSystem {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl FileSystem {

fn new() -> Self {

}

fn create_path(&self, path: String, value: i32) -> bool {

}

fn get(&self, path: String) -> i32 {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem::new();
* let ret_1: bool = obj.create_path(path, value);
* let ret_2: i32 = obj.get(path);
*/
```

Ruby:

```
class FileSystem
def initialize()

end


=begin
:type path: String
```

```
:type value: Integer
:rtype: Boolean
=end
def create_path(path, value)

end


=begin
:type path: String
:rtype: Integer
=end
def get(path)

end



end

# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem.new()
# param_1 = obj.create_path(path, value)
# param_2 = obj.get(path)
```

PHP:

```
class FileSystem {
/**
*/
function __construct() {

}

/**
* @param String $path
* @param Integer $value
* @return Boolean
*/
function createPath($path, $value) {

}
```

```
/**
* @param String $path
* @return Integer
*/
function get($path) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* $obj = FileSystem();
* $ret_1 = $obj->createPath($path, $value);
* $ret_2 = $obj->get($path);
*/
```

Dart:

```
class FileSystem {

FileSystem() {

}

bool createPath(String path, int value) {

}

int get(String path) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = FileSystem();
* bool param1 = obj.createPath(path,value);
* int param2 = obj.get(path);
*/
```

Scala:

```
class FileSystem() {

def createPath(path: String, value: Int): Boolean = {

}

def get(path: String): Int = {

}

}

/**
* Your FileSystem object will be instantiated and called as such:
* val obj = new FileSystem()
* val param_1 = obj.createPath(path,value)
* val param_2 = obj.get(path)
*/
```

Elixir:

```
defmodule FileSystem do
@spec init_() :: any
def init_() do

end

@spec create_path(path :: String.t, value :: integer) :: boolean
def create_path(path, value) do

end

@spec get(path :: String.t) :: integer
def get(path) do

end
end

# Your functions will be called as such:
# FileSystem.init_()
# param_1 = FileSystem.create_path(path, value)
# param_2 = FileSystem.get(path)
```

```
# FileSystem.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang:

```
-spec file_system_init_() -> any().
file_system_init_() ->
.

-spec file_system_create_path(Path :: unicode:unicode_binary(), Value :: integer()) -> boolean().
file_system_create_path(Path, Value) ->
.

-spec file_system_get(Path :: unicode:unicode_binary()) -> integer().
file_system_get(Path) ->
.


%% Your functions will be called as such:
%% file_system_init_(),
%% Param_1 = file_system_create_path(Path, Value),
%% Param_2 = file_system_get(Path),

%% file_system_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket:

```
(define file-system%
(class object%
(super-new)

(init-field)

; create-path : string? exact-integer? -> boolean?
(define/public (create-path path value)
)
; get : string? -> exact-integer?
(define/public (get path)
)))
```

```
;; Your file-system% object will be instantiated and called as such:
;; (define obj (new file-system%))
;; (define param_1 (send obj create-path path value))
;; (define param_2 (send obj get path))
```

Solutions

C++ Solution:

```
/*
* Problem: Design File System
* Difficulty: Medium
* Tags: string, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class FileSystem {
public:
FileSystem() {

}

bool createPath(string path, int value) {

}

int get(string path) {

}
};

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem* obj = new FileSystem();
* bool param_1 = obj->createPath(path,value);
* int param_2 = obj->get(path);
*/
```

Java Solution:

```
/**
* Problem: Design File System
* Difficulty: Medium
* Tags: string, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class FileSystem {

public FileSystem() {

}

public boolean createPath(String path, int value) {

}

public int get(String 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);
*/
```

Python3 Solution:

```
"""
Problem: Design File System
Difficulty: Medium
Tags: string, hash

Approach: String manipulation with hash map or two pointers
```

```
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class FileSystem:

def __init__(self):


def createPath(self, path: str, value: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class FileSystem(object):

def __init__(self):


def createPath(self, path, value):
"""
:type path: str
:type value: int
:rtype: bool
"""


def get(self, path):
"""
:type path: str
:rtype: int
"""



# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.createPath(path,value)
# param_2 = obj.get(path)
```

JavaScript Solution:

```
/**
* Problem: Design File System
* Difficulty: Medium
* Tags: string, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/


var FileSystem = function() {

};

/**
* @param {string} path
* @param {number} value
* @return {boolean}
*/
FileSystem.prototype.createPath = function(path, value) {

};

/**
* @param {string} path
* @return {number}
*/
FileSystem.prototype.get = function(path) {

};

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.createPath(path,value)
* var param_2 = obj.get(path)
*/
```

TypeScript Solution:

```
/**
* Problem: Design File System
* Difficulty: Medium
* Tags: string, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class FileSystem {
constructor() {

}

createPath(path: string, value: number): boolean {

}

get(path: string): number {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.createPath(path,value)
* var param_2 = obj.get(path)
*/
```

C# Solution:

```
/*
* Problem: Design File System
* Difficulty: Medium
* Tags: string, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/
```

```
public class FileSystem {

public FileSystem() {

}

public bool CreatePath(string path, int value) {

}

public int Get(string path) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* bool param_1 = obj.CreatePath(path,value);
* int param_2 = obj.Get(path);
*/
```

C Solution:

```
/*
* Problem: Design File System
* Difficulty: Medium
* Tags: string, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/




typedef struct {

} FileSystem;
```

```
FileSystem* fileSystemCreate() {

}

bool fileSystemCreatePath(FileSystem* obj, char* path, int value) {

}

int fileSystemGet(FileSystem* obj, char* path) {

}

void fileSystemFree(FileSystem* obj) {

}

/**
* Your FileSystem struct will be instantiated and called as such:
* FileSystem* obj = fileSystemCreate();
* bool param_1 = fileSystemCreatePath(obj, path, value);

* int param_2 = fileSystemGet(obj, path);

* fileSystemFree(obj);
*/
```

Go Solution:

```
// Problem: Design File System
// Difficulty: Medium
// Tags: string, hash
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type FileSystem struct {

}
```

```
func Constructor() FileSystem {

}


func (this *FileSystem) CreatePath(path string, value int) bool {

}


func (this *FileSystem) Get(path string) int {

}


/**
* Your FileSystem object will be instantiated and called as such:
* obj := Constructor();
* param_1 := obj.CreatePath(path,value);
* param_2 := obj.Get(path);
*/
```

Kotlin Solution:

```
class FileSystem() {

fun createPath(path: String, value: Int): Boolean {

}

fun get(path: String): Int {

}

}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = FileSystem()
```

```
* var param_1 = obj.createPath(path,value)
* var param_2 = obj.get(path)
*/
```

Swift Solution:

```
class FileSystem {

init() {

}

func createPath(_ path: String, _ value: Int) -> Bool {

}

func get(_ path: String) -> Int {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem()
* let ret_1: Bool = obj.createPath(path, value)
* let ret_2: Int = obj.get(path)
*/
```

Rust Solution:

```
// Problem: Design File System
// Difficulty: Medium
// Tags: string, hash
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct FileSystem {
```

```
}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl FileSystem {

fn new() -> Self {

}

fn create_path(&self, path: String, value: i32) -> bool {

}

fn get(&self, path: String) -> i32 {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem::new();
* let ret_1: bool = obj.create_path(path, value);
* let ret_2: i32 = obj.get(path);
*/
```

Ruby Solution:

```
class FileSystem
def initialize()

end


=begin
:type path: String
:type value: Integer
:rtype: Boolean
```

```
=end
def create_path(path, value)

end


=begin
:type path: String
:rtype: Integer
=end
def get(path)

end



end

# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem.new()
# param_1 = obj.create_path(path, value)
# param_2 = obj.get(path)
```

PHP Solution:

```
class FileSystem {
/**
*/
function __construct() {

}

/**
* @param String $path
* @param Integer $value
* @return Boolean
*/
function createPath($path, $value) {

}

/**
```

```
* @param String $path
* @return Integer
*/
function get($path) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* $obj = FileSystem();
* $ret_1 = $obj->createPath($path, $value);
* $ret_2 = $obj->get($path);
*/
```

Dart Solution:

```
class FileSystem {

FileSystem() {

}

bool createPath(String path, int value) {

}

int get(String path) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = FileSystem();
* bool param1 = obj.createPath(path,value);
* int param2 = obj.get(path);
*/
```

Scala Solution:

```
class FileSystem() {

def createPath(path: String, value: Int): Boolean = {

}

def get(path: String): Int = {

}

}

/**
* Your FileSystem object will be instantiated and called as such:
* val obj = new FileSystem()
* val param_1 = obj.createPath(path,value)
* val param_2 = obj.get(path)
*/
```

Elixir Solution:

```
defmodule FileSystem do
@spec init_() :: any
def init_() do

end

@spec create_path(path :: String.t, value :: integer) :: boolean
def create_path(path, value) do

end

@spec get(path :: String.t) :: integer
def get(path) do

end
end

# Your functions will be called as such:
# FileSystem.init_()
# param_1 = FileSystem.create_path(path, value)
# param_2 = FileSystem.get(path)
```

```
# FileSystem.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang Solution:

```
-spec file_system_init_() -> any().
file_system_init_() ->
.

-spec file_system_create_path(Path :: unicode:unicode_binary(), Value :: integer()) -> boolean().
file_system_create_path(Path, Value) ->
.

-spec file_system_get(Path :: unicode:unicode_binary()) -> integer().
file_system_get(Path) ->
.


%% Your functions will be called as such:
%% file_system_init_(),
%% Param_1 = file_system_create_path(Path, Value),
%% Param_2 = file_system_get(Path),

%% file_system_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket Solution:

```
(define file-system%
(class object%
(super-new)

(init-field)

; create-path : string? exact-integer? -> boolean?
(define/public (create-path path value)
)
; get : string? -> exact-integer?
(define/public (get path)
```

```
)))

;; Your file-system% object will be instantiated and called as such:
;; (define obj (new file-system%))
;; (define param_1 (send obj create-path path value))
;; (define param_2 (send obj get path))
```