Problem 588: Design In-Memory File System

Problem Information

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

Problem Description

Design a data structure that simulates an in-memory file system.

Implement the FileSystem class:

FileSystem()

Initializes the object of the system.

List<String> ls(String path)

If

path

is a file path, returns a list that only contains this file's name.

If

path

is a directory path, returns the list of file and directory names

in this directory

.

The answer should in

lexicographic order

.

void mkdir(String path)

Makes a new directory according to the given

path

. The given directory path does not exist. If the middle directories in the path do not exist, you should create them as well.

void addContentToFile(String filePath, String content)

If

filePath

does not exist, creates that file containing given

content

.

If

filePath

already exists, appends the given

content

to original content.

String readContentFromFile(String filePath)

Returns the content in the file at

filePath

.

Example 1:

Operation	Output	Explanation
FileSystem fs = new FileSystem()	null	The constructor returns nothing.
fs.ls("/")	[]	Initially, directory `/` has nothing. So return empty list.
fs.mkdir("/a/b/c")	null	Create directory `a` in directory `/`. Then create directory `b` in directory `a`. Finally, create directory `c` in directory `b`.
fs.addContentToFile("/a/b/c/d","hello")	null	Create a file named `d` with content `"hello"` in directory `/a/b/c`.
fs.ls("/")	["a"]	Only directory `a` is in directory `/`.
fs.readContentFromFile("/a/b/c/d")	"hello"	Output the file content.

Input

["FileSystem", "ls", "mkdir", "addContentToFile", "ls", "readContentFromFile"] [[], ["/"], ["/a/b/c"], ["/a/b/c/d", "hello"], ["/"], ["/a/b/c/d"]]

Output

[null, [], null, null, ["a"], "hello"]

Explanation

FileSystem fileSystem = new FileSystem(); fileSystem.ls("/"); // return [] fileSystem.mkdir("/a/b/c"); fileSystem.addContentToFile("/a/b/c/d", "hello"); fileSystem.ls("/"); // return ["a"] fileSystem.readContentFromFile("/a/b/c/d"); // return "hello"

Constraints:

1 <= path.length, filePath.length <= 100

path

and

filePath

are absolute paths which begin with

'/'

and do not end with

'/'

except that the path is just

"/"

.

You can assume that all directory names and file names only contain lowercase letters, and the same names will not exist in the same directory.

You can assume that all operations will be passed valid parameters, and users will not attempt to retrieve file content or list a directory or file that does not exist.

You can assume that the parent directory for the file in

addContentToFile

will exist.

1 <= content.length <= 50

At most

300

calls will be made to

ls

,

mkdir

,

addContentToFile

, and

readContentFromFile

.

Code Snippets

C++:

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

}

vector<string> ls(string path) {

}

void mkdir(string path) {

}

void addContentToFile(string filePath, string content) {

}

string readContentFromFile(string filePath) {

}
```

```
};

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem* obj = new FileSystem();
* vector<string> param_1 = obj->ls(path);
* obj->mkdir(path);
* obj->addContentToFile(filePath,content);
* string param_4 = obj->readContentFromFile(filePath);
*/
```

Java:

```
class FileSystem {

public FileSystem() {

}

public List<String> ls(String path) {

}

public void mkdir(String path) {

}

public void addContentToFile(String filePath, String content) {

}

public String readContentFromFile(String filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* List<String> param_1 = obj.ls(path);
* obj.mkdir(path);
* obj.addContentToFile(filePath,content);
```

```
* String param_4 = obj.readContentFromFile(filePath);
*/
```

Python3:

```
class FileSystem:

def __init__(self):


def ls(self, path: str) -> List[str]:


def mkdir(self, path: str) -> None:


def addContentToFile(self, filePath: str, content: str) -> None:


def readContentFromFile(self, filePath: str) -> str:



# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.addContentToFile(filePath,content)
# param_4 = obj.readContentFromFile(filePath)
```

Python:

```
class FileSystem(object):

def __init__(self):


def ls(self, path):
"""
:type path: str
:rtype: List[str]
"""
```

```
def mkdir(self, path):
"""
:type path: str
:rtype: None
"""


def addContentToFile(self, filePath, content):
"""
:type filePath: str
:type content: str
:rtype: None
"""


def readContentFromFile(self, filePath):
"""
:type filePath: str
:rtype: str
"""



# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.addContentToFile(filePath,content)
# param_4 = obj.readContentFromFile(filePath)
```

JavaScript:

```
var FileSystem = function() {

};

/**
* @param {string} path
* @return {string[]}
```

```
*/
FileSystem.prototype.ls = function(path) {

};

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

};

/**
* @param {string} filePath
* @param {string} content
* @return {void}
*/
FileSystem.prototype.addContentToFile = function(filePath, content) {

};

/**
* @param {string} filePath
* @return {string}
*/
FileSystem.prototype.readContentFromFile = function(filePath) {

};

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* var param_4 = obj.readContentFromFile(filePath)
*/
```

TypeScript:

```
class FileSystem {
constructor() {

}

ls(path: string): string[] {

}

mkdir(path: string): void {

}

addContentToFile(filePath: string, content: string): void {

}

readContentFromFile(filePath: string): string {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* var param_4 = obj.readContentFromFile(filePath)
*/
```

C#:

```
public class FileSystem {

public FileSystem() {

}

public IList<string> Ls(string path) {

}
```

```
public void Mkdir(string path) {

}

public void AddContentToFile(string filePath, string content) {

}

public string ReadContentFromFile(string filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* IList<string> param_1 = obj.Ls(path);
* obj.Mkdir(path);
* obj.AddContentToFile(filePath,content);
* string param_4 = obj.ReadContentFromFile(filePath);
*/
```

C:

```
typedef struct {

} FileSystem;


FileSystem* fileSystemCreate() {

}

char** fileSystemLs(FileSystem* obj, char* path, int* retSize) {

}

void fileSystemMkdir(FileSystem* obj, char* path) {
```

```
}

void fileSystemAddContentToFile(FileSystem* obj, char* filePath, char*
content) {

}

char* fileSystemReadContentFromFile(FileSystem* obj, char* filePath) {

}

void fileSystemFree(FileSystem* obj) {

}

/**
* Your FileSystem struct will be instantiated and called as such:
* FileSystem* obj = fileSystemCreate();
* char** param_1 = fileSystemLs(obj, path, retSize);

* fileSystemMkdir(obj, path);

* fileSystemAddContentToFile(obj, filePath, content);

* char* param_4 = fileSystemReadContentFromFile(obj, filePath);

* fileSystemFree(obj);
*/
```

Go:

```
type FileSystem struct {

}


func Constructor() FileSystem {

}


func (this *FileSystem) Ls(path string) []string {
```

```
}


func (this *FileSystem) Mkdir(path string) {

}


func (this *FileSystem) AddContentToFile(filePath string, content string) {

}


func (this *FileSystem) ReadContentFromFile(filePath string) string {

}


/**
* Your FileSystem object will be instantiated and called as such:
* obj := Constructor();
* param_1 := obj.Ls(path);
* obj.Mkdir(path);
* obj.AddContentToFile(filePath,content);
* param_4 := obj.ReadContentFromFile(filePath);
*/
```

Kotlin:

```
class FileSystem() {

fun ls(path: String): List<String> {

}

fun mkdir(path: String) {

}

fun addContentToFile(filePath: String, content: String) {
```

```
}

fun readContentFromFile(filePath: String): String {

}

}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = FileSystem()
* var param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* var param_4 = obj.readContentFromFile(filePath)
*/
```

Swift:

```
class FileSystem {

init() {

}

func ls(_ path: String) -> [String] {

}

func mkdir(_ path: String) {

}

func addContentToFile(_ filePath: String, _ content: String) {

}

func readContentFromFile(_ filePath: String) -> String {

}
}
```

```
/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem()
* let ret_1: [String] = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath, content)
* let ret_4: String = obj.readContentFromFile(filePath)
*/
```

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 ls(&self, path: String) -> Vec<String> {

}

fn mkdir(&self, path: String) {

}

fn add_content_to_file(&self, file_path: String, content: String) {

}

fn read_content_from_file(&self, file_path: String) -> String {

}
```

```
}

/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem::new();
* let ret_1: Vec<String> = obj.ls(path);
* obj.mkdir(path);
* obj.add_content_to_file(filePath, content);
* let ret_4: String = obj.read_content_from_file(filePath);
*/
```

Ruby:

```
class FileSystem
def initialize()

end


=begin
:type path: String
:rtype: String[]
=end
def ls(path)

end


=begin
:type path: String
:rtype: Void
=end
def mkdir(path)

end


=begin
:type file_path: String
:type content: String
:rtype: Void
=end
```

```
def add_content_to_file(file_path, content)

end


=begin
:type file_path: String
:rtype: String
=end
def read_content_from_file(file_path)

end



end

# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem.new()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.add_content_to_file(file_path, content)
# param_4 = obj.read_content_from_file(file_path)
```

PHP:

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

}

/**
* @param String $path
* @return String[]
*/
function ls($path) {

}

/**
* @param String $path
```

```
* @return NULL
*/
function mkdir($path) {

}

/**
* @param String $filePath
* @param String $content
* @return NULL
*/
function addContentToFile($filePath, $content) {

}

/**
* @param String $filePath
* @return String
*/
function readContentFromFile($filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* $obj = FileSystem();
* $ret_1 = $obj->ls($path);
* $obj->mkdir($path);
* $obj->addContentToFile($filePath, $content);
* $ret_4 = $obj->readContentFromFile($filePath);
*/
```

Dart:

```
class FileSystem {

FileSystem() {

}

List<String> ls(String path) {
```

```
}

void mkdir(String path) {

}

void addContentToFile(String filePath, String content) {

}

String readContentFromFile(String filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = FileSystem();
* List<String> param1 = obj.ls(path);
* obj.mkdir(path);
* obj.addContentToFile(filePath,content);
* String param4 = obj.readContentFromFile(filePath);
*/
```

Scala:

```
class FileSystem() {

def ls(path: String): List[String] = {

}

def mkdir(path: String): Unit = {

}

def addContentToFile(filePath: String, content: String): Unit = {

}

def readContentFromFile(filePath: String): String = {
```

```
}

}

/**
* Your FileSystem object will be instantiated and called as such:
* val obj = new FileSystem()
* val param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* val param_4 = obj.readContentFromFile(filePath)
*/
```

Elixir:

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

end

@spec ls(path :: String.t) :: [String.t]
def ls(path) do

end

@spec mkdir(path :: String.t) :: any
def mkdir(path) do

end

@spec add_content_to_file(file_path :: String.t, content :: String.t) :: any
def add_content_to_file(file_path, content) do

end

@spec read_content_from_file(file_path :: String.t) :: String.t
def read_content_from_file(file_path) do

end
end
```

```
# Your functions will be called as such:
# FileSystem.init_()
# param_1 = FileSystem.ls(path)
# FileSystem.mkdir(path)
# FileSystem.add_content_to_file(file_path, content)
# param_4 = FileSystem.read_content_from_file(file_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_ls(Path :: unicode:unicode_binary()) ->
[unicode:unicode_binary()].
file_system_ls(Path) ->
.

-spec file_system_mkdir(Path :: unicode:unicode_binary()) -> any().
file_system_mkdir(Path) ->
.

-spec file_system_add_content_to_file(FilePath :: unicode:unicode_binary(),
Content :: unicode:unicode_binary()) -> any().
file_system_add_content_to_file(FilePath, Content) ->
.

-spec file_system_read_content_from_file(FilePath ::
unicode:unicode_binary()) -> unicode:unicode_binary().
file_system_read_content_from_file(FilePath) ->
.


%% Your functions will be called as such:
%% file_system_init_(),
%% Param_1 = file_system_ls(Path),
%% file_system_mkdir(Path),
%% file_system_add_content_to_file(FilePath, Content),
```

```
%% Param_4 = file_system_read_content_from_file(FilePath),

%% 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)

; ls : string? -> (listof string?)
(define/public (ls path)
)
; mkdir : string? -> void?
(define/public (mkdir path)
)
; add-content-to-file : string? string? -> void?
(define/public (add-content-to-file file-path content)
)
; read-content-from-file : string? -> string?
(define/public (read-content-from-file file-path)
)))

;; Your file-system% object will be instantiated and called as such:
;; (define obj (new file-system%))
;; (define param_1 (send obj ls path))
;; (send obj mkdir path)
;; (send obj add-content-to-file file-path content)
;; (define param_4 (send obj read-content-from-file file-path))
```

Solutions

C++ Solution:

```
/*
* Problem: Design In-Memory File System
* Difficulty: Hard
```

```
* Tags: string, graph, hash, sort
*
* 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() {

}

vector<string> ls(string path) {

}

void mkdir(string path) {

}

void addContentToFile(string filePath, string content) {

}

string readContentFromFile(string filePath) {

}
};

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem* obj = new FileSystem();
* vector<string> param_1 = obj->ls(path);
* obj->mkdir(path);
* obj->addContentToFile(filePath,content);
* string param_4 = obj->readContentFromFile(filePath);
*/
```

Java Solution:

```
/**
* Problem: Design In-Memory File System
* Difficulty: Hard
* Tags: string, graph, hash, sort
*
* 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 List<String> ls(String path) {

}

public void mkdir(String path) {

}

public void addContentToFile(String filePath, String content) {

}

public String readContentFromFile(String filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* List<String> param_1 = obj.ls(path);
* obj.mkdir(path);
* obj.addContentToFile(filePath,content);
* String param_4 = obj.readContentFromFile(filePath);
*/
```

Python3 Solution:

```
"""
Problem: Design In-Memory File System
Difficulty: Hard
Tags: string, graph, hash, sort

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 ls(self, path: str) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class FileSystem(object):

def __init__(self):


def ls(self, path):
"""
:type path: str
:rtype: List[str]
"""


def mkdir(self, path):
"""
:type path: str
:rtype: None
"""


def addContentToFile(self, filePath, content):
```

```
        """
        :type filePath: str
        :type content: str
        :rtype: None
        """


    def readContentFromFile(self, filePath):
        """
        :type filePath: str
        :rtype: str
        """



# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.addContentToFile(filePath,content)
# param_4 = obj.readContentFromFile(filePath)
```

JavaScript Solution:

```
/**
* Problem: Design In-Memory File System
* Difficulty: Hard
* Tags: string, graph, hash, sort
*
* 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
```

```
* @return {string[]}
*/
FileSystem.prototype.ls = function(path) {

};

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

};

/**
* @param {string} filePath
* @param {string} content
* @return {void}
*/
FileSystem.prototype.addContentToFile = function(filePath, content) {

};

/**
* @param {string} filePath
* @return {string}
*/
FileSystem.prototype.readContentFromFile = function(filePath) {

};

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* var param_4 = obj.readContentFromFile(filePath)
*/
```

TypeScript Solution:

```
/**
* Problem: Design In-Memory File System
* Difficulty: Hard
* Tags: string, graph, hash, sort
*
* 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() {

}

ls(path: string): string[] {

}

mkdir(path: string): void {

}

addContentToFile(filePath: string, content: string): void {

}

readContentFromFile(filePath: string): string {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = new FileSystem()
* var param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* var param_4 = obj.readContentFromFile(filePath)
*/
```

C# Solution:

```
/*
* Problem: Design In-Memory File System
* Difficulty: Hard
* Tags: string, graph, hash, sort
*
* 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 IList<string> Ls(string path) {

}

public void Mkdir(string path) {

}

public void AddContentToFile(string filePath, string content) {

}

public string ReadContentFromFile(string filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = new FileSystem();
* IList<string> param_1 = obj.Ls(path);
* obj.Mkdir(path);
* obj.AddContentToFile(filePath,content);
* string param_4 = obj.ReadContentFromFile(filePath);
*/
```

C Solution:

```
/*
* Problem: Design In-Memory File System
* Difficulty: Hard
* Tags: string, graph, hash, sort
*
* 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() {

}

char** fileSystemLs(FileSystem* obj, char* path, int* retSize) {

}

void fileSystemMkdir(FileSystem* obj, char* path) {

}

void fileSystemAddContentToFile(FileSystem* obj, char* filePath, char*
content) {

}

char* fileSystemReadContentFromFile(FileSystem* obj, char* filePath) {

}

void fileSystemFree(FileSystem* obj) {
```

```
}

/**
* Your FileSystem struct will be instantiated and called as such:
* FileSystem* obj = fileSystemCreate();
* char** param_1 = fileSystemLs(obj, path, retSize);

* fileSystemMkdir(obj, path);

* fileSystemAddContentToFile(obj, filePath, content);

* char* param_4 = fileSystemReadContentFromFile(obj, filePath);

* fileSystemFree(obj);
*/
```

Go Solution:

```
// Problem: Design In-Memory File System
// Difficulty: Hard
// Tags: string, graph, hash, sort
//
// 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) Ls(path string) []string {

}
```

```
func (this *FileSystem) Mkdir(path string) {

}


func (this *FileSystem) AddContentToFile(filePath string, content string) {

}


func (this *FileSystem) ReadContentFromFile(filePath string) string {

}


/**
* Your FileSystem object will be instantiated and called as such:
* obj := Constructor();
* param_1 := obj.Ls(path);
* obj.Mkdir(path);
* obj.AddContentToFile(filePath,content);
* param_4 := obj.ReadContentFromFile(filePath);
*/
```

Kotlin Solution:

```
class FileSystem() {

fun ls(path: String): List<String> {

}

fun mkdir(path: String) {

}

fun addContentToFile(filePath: String, content: String) {

}

fun readContentFromFile(filePath: String): String {
```

```
}

}

/**
* Your FileSystem object will be instantiated and called as such:
* var obj = FileSystem()
* var param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* var param_4 = obj.readContentFromFile(filePath)
*/
```

Swift Solution:

```
class FileSystem {

init() {

}

func ls(_ path: String) -> [String] {

}

func mkdir(_ path: String) {

}

func addContentToFile(_ filePath: String, _ content: String) {

}

func readContentFromFile(_ filePath: String) -> String {

}
}

/**
```

```
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem()
* let ret_1: [String] = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath, content)
* let ret_4: String = obj.readContentFromFile(filePath)
*/
```

Rust Solution:

```
// Problem: Design In-Memory File System
// Difficulty: Hard
// Tags: string, graph, hash, sort
//
// 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 ls(&self, path: String) -> Vec<String> {

}

fn mkdir(&self, path: String) {

}
```

```
fn add_content_to_file(&self, file_path: String, content: String) {

}

fn read_content_from_file(&self, file_path: String) -> String {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* let obj = FileSystem::new();
* let ret_1: Vec<String> = obj.ls(path);
* obj.mkdir(path);
* obj.add_content_to_file(filePath, content);
* let ret_4: String = obj.read_content_from_file(filePath);
*/
```

Ruby Solution:

```
class FileSystem
def initialize()

end


=begin
:type path: String
:rtype: String[]
=end
def ls(path)

end


=begin
:type path: String
:rtype: Void
=end
def mkdir(path)
```

```
end


=begin
:type file_path: String
:type content: String
:rtype: Void
=end
def add_content_to_file(file_path, content)

end


=begin
:type file_path: String
:rtype: String
=end
def read_content_from_file(file_path)

end


end

# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem.new()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.add_content_to_file(file_path, content)
# param_4 = obj.read_content_from_file(file_path)
```

PHP Solution:

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

}

/**
```

```
* @param String $path
* @return String[]
*/
function ls($path) {

}

/**
* @param String $path
* @return NULL
*/
function mkdir($path) {

}

/**
* @param String $filePath
* @param String $content
* @return NULL
*/
function addContentToFile($filePath, $content) {

}

/**
* @param String $filePath
* @return String
*/
function readContentFromFile($filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* $obj = FileSystem();
* $ret_1 = $obj->ls($path);
* $obj->mkdir($path);
* $obj->addContentToFile($filePath, $content);
* $ret_4 = $obj->readContentFromFile($filePath);
*/
```

Dart Solution:

```
class FileSystem {

FileSystem() {

}

List<String> ls(String path) {

}

void mkdir(String path) {

}

void addContentToFile(String filePath, String content) {

}

String readContentFromFile(String filePath) {

}
}

/**
* Your FileSystem object will be instantiated and called as such:
* FileSystem obj = FileSystem();
* List<String> param1 = obj.ls(path);
* obj.mkdir(path);
* obj.addContentToFile(filePath,content);
* String param4 = obj.readContentFromFile(filePath);
*/
```

Scala Solution:

```
class FileSystem() {

def ls(path: String): List[String] = {

}
```

```
def mkdir(path: String): Unit = {

}

def addContentToFile(filePath: String, content: String): Unit = {

}

def readContentFromFile(filePath: String): String = {

}

}

/**
* Your FileSystem object will be instantiated and called as such:
* val obj = new FileSystem()
* val param_1 = obj.ls(path)
* obj.mkdir(path)
* obj.addContentToFile(filePath,content)
* val param_4 = obj.readContentFromFile(filePath)
*/
```

Elixir Solution:

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

end

@spec ls(path :: String.t) :: [String.t]
def ls(path) do

end

@spec mkdir(path :: String.t) :: any
def mkdir(path) do

end
```

```
@spec add_content_to_file(file_path :: String.t, content :: String.t) :: any
def add_content_to_file(file_path, content) do

end

@spec read_content_from_file(file_path :: String.t) :: String.t
def read_content_from_file(file_path) do

end
end

# Your functions will be called as such:
# FileSystem.init_()
# param_1 = FileSystem.ls(path)
# FileSystem.mkdir(path)
# FileSystem.add_content_to_file(file_path, content)
# param_4 = FileSystem.read_content_from_file(file_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_ls(Path :: unicode:unicode_binary()) ->
[unicode:unicode_binary()].
file_system_ls(Path) ->
.

-spec file_system_mkdir(Path :: unicode:unicode_binary()) -> any().
file_system_mkdir(Path) ->
.

-spec file_system_add_content_to_file(FilePath :: unicode:unicode_binary(),
Content :: unicode:unicode_binary()) -> any().
file_system_add_content_to_file(FilePath, Content) ->
.
```

```
-spec file_system_read_content_from_file(FilePath ::
unicode:unicode_binary()) -> unicode:unicode_binary().
file_system_read_content_from_file(FilePath) ->
.


%% Your functions will be called as such:
%% file_system_init_(),
%% Param_1 = file_system_ls(Path),
%% file_system_mkdir(Path),
%% file_system_add_content_to_file(FilePath, Content),
%% Param_4 = file_system_read_content_from_file(FilePath),

%% 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)

; ls : string? -> (listof string?)
(define/public (ls path)
)
; mkdir : string? -> void?
(define/public (mkdir path)
)
; add-content-to-file : string? string? -> void?
(define/public (add-content-to-file file-path content)
)
; read-content-from-file : string? -> string?
(define/public (read-content-from-file file-path)
)))

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

```
;; (send obj add-content-to-file file-path content)
;; (define param_4 (send obj read-content-from-file file-path))
```