588. Design In-Memory File System

May 20, 2017 | 31K views

Average Rating: 4.94 (36 votes)

Design an in-memory file system to simulate the following functions:

`ls`: Given a path in string format. If it is a file path, return a list that only contains this file's name. If it is a directory path, return the list of file and directory names **in this directory**. Your output (file and directory names together) should in **lexicographic order**.

`mkdir`: Given a **directory path** that does not exist, you should make a new directory according to the path. If the middle directories in the path don't exist either, you should create them as well. This function has void return type.

`addContentToFile`: Given a **file path** and **file content** in string format. If the file doesn't exist, you need to create that file containing given content. If the file already exists, you need to **append** given content to original content. This function has void return type.

`readContentFromFile`: Given a **file path**, return its **content** in string format.

Example:

```
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:
```

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.

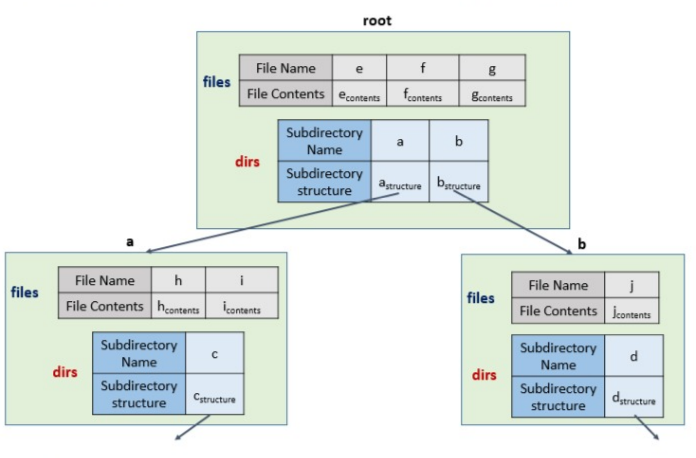
Note:

1. You can assume all file or directory paths are absolute paths which begin with `/` and do not end with `/` except that the path is just `"/"`.
2. 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.
3. You can assume that all directory names and file names only contain lower-case letters, and same names won't exist in the same directory.

Solution

Approach #1 Using separate Directory and File List[Accepted]

We start our discussion by looking at the directory structure used. The root directory acts as the base of the directory structure. Each directory contains two hashmaps namely $dirs$ and $files$. The $dirs$ contains data in the form $[(subdirectory_1\_name : subdirectory_{1\_structure}), (subdirectory_2\_name : subdirectory_{2\_structure})...]$. The $files$ contains data in the form $[(file_1 : file_{1\_contents}), (file_2 : file_{2\_contents})...]$. This directory structure is shown below with a sample showing just the first two levels.



Now, we'll discuss how we implement the various commands required.

1. `ls`: In this case, we start off by initializing t, a temporary directory pointer, to the root directory. We split the input directory path based on `/` and obtain the individual levels of directory names in a d array. Then, we traverse over the tree directory structure based on the individual directories found and we keep on updating the t directory pointer to point to the new level of directory(child) as we go on entering deeper into the directory structure. At the end, we will stop at either the end level directory or at the file name depending upon the input given. If the last level in the input happens to be a file name, we simply need to return the file name. So, we directly return the last entry in the d array. If the last level entry happens to be a directory, we can obtain its subdirectory list from the list of keys in its $dirs$ hashmap. Similarly, we can obtain the list of files in the last directory from the keys in the corresponding $files$ hashmap. We append the two lists obtained, sort them and return the sorted appended list.

2. `mkdir`: In response to this command, as in case of `ls`, we start entering the directory structure level by level. Whenever we reach a state where a directory mentioned in the path of `mkdir` doesn't exist, we create a new entry in the last valid directory's $dirs$ structure and initialize its subdirectory list as an empty list. We keep on doing so till we reach the end level directory.

3. `addContentToFile`: In response to this command as well, as in case of `ls`, we start entering the directory structure level by level. When we reach the level of the file name, we check if the file name already exists in the $files$ keys. If it exists, we concatenate the current contents to the contents of the file(in the value section of the corresponding file). If it doesn't exist, we create a new entry in the current directory's $files$ and initialize its contents with the current contents.

4. `readContentFromFile`: As the previous cases, we reach the last directory level by traversing through the directory structure level by level. Then, in the last directory, we search for the file name entry in the corresponding $files$' keys and return its corresponding value as the contents of the file.

Java Copy

```java
public class FileSystem {
    class Dir {
        HashMap < String, Dir > dirs = new HashMap < > ();
        HashMap < String, String > files = new HashMap < > ();
    }
    Dir root;
    public FileSystem() {
        root = new Dir();
    }
    public List < String > ls(String path) {
        Dir t = root;
        List < String > files = new ArrayList < > ();
        if (!path.equals("/")) {
            String[] d = path.split("/");
            for (int i = 1; i < d.length - 1; i++) {
                t = t.dirs.get(d[i]);
            }
            if (t.files.containsKey(d[d.length - 1])) {
                files.add(d[d.length - 1]);
                return files;
            } else {
                t = t.dirs.get(d[d.length - 1]);
            }
        }
        files.addAll(new ArrayList < > (t.dirs.keySet()));
        files.addAll(new ArrayList < > (t.files.keySet()));
        Collections.sort(files);
        return files;
```

Performance Analysis

- The time complexity of executing an `ls` command is $O(m + n + klog(k))$. Here, m refers to the length of the input string. We need to scan the input string once to split it and determine the various levels. n refers to the depth of the last directory level in the given input for `ls`. This factor is taken because we need to enter n levels of the tree structure to reach the last level. k refers to the number of entries(files+subdirectories) in the last level directory(in the current input). We need to sort these names giving a factor of $klog(k)$.

- The time complexity of executing an `mkdir` command is $O(m + n)$. Here, m refers to the length of the input string. We need to scan the input string once to split it and determine the various levels. n refers to the depth of the last directory level in the `mkdir` input. This factor is taken because we need to enter n levels of the tree structure to reach the last level.

- The time complexity of both `addContentToFile` and `readContentFromFile` is $O(m + n)$. Here, m refers to the length of the input string. We need to scan the input string once to split it and determine the various levels. n refers to the depth of the file name in the current input. This factor is taken because we need to enter n levels of the tree structure to reach the level where the files's contents need to be added/read from.

- The advantage of this scheme of maintaining the directory structure is that it is expandable to include even more commands easily. For example, `rmdir` to remove a directory given an input directory path. We need to simply reach to the destined directory level and remove the corresponding directory entry from the corresponding $dirs$ keys.

- Renaming files/directories is also very simple, since all we need to do is to create a temporary copy of the directory structure/file with a new name and delete the last entry.

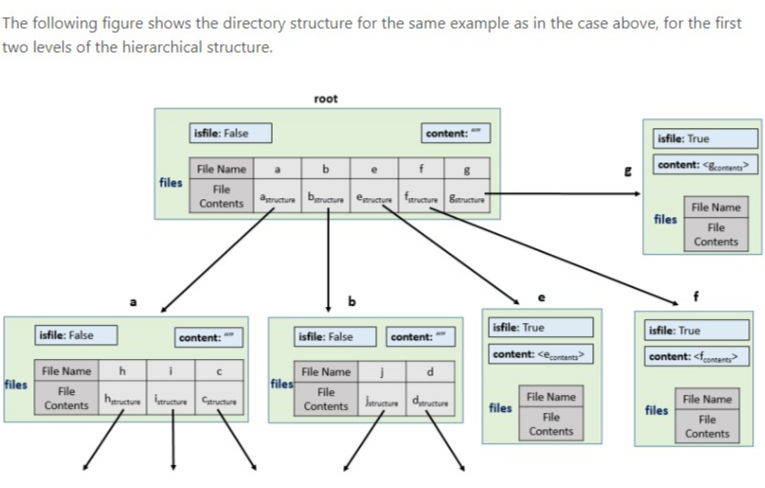
- Relocating a hierarchichal subdirectory structure from one directory to the other is also very easy, since, all we need to do is obtain the address for the corresponding subdirectory class, and assign the same at the new positon in the new directory structure.

- Extracting only directories or files list on any path is easy in this case, since we maintain separate entires for $dirs$ and $files$.

Approach #2 Using unified Directory and File List[Accepted]

This design differs from the first design in that the current data structure for a Directory contains a unified $files$ hashmap, which contains the list of all the files and subdirectories in the current directory. Apart from this, we contain an entry $isfile$, which when True indicates that the current $files$ entry is actually corresponding to a file, otherwise it represents a directory. Further, since we are considering the directory and files' entries in the same manner, we need an entry for $content$, which contains the contents of the current file(if $isfile$ entry is True in the current case). For entries corresponding to directories, the $content$ field is kept empty.

The following figure shows the directory structure for the same example as in the case above, for the first two levels of the hierarchical structure.



The implementation of all the commands remains the same as in the last design, except that we need to make entries in the same $files$ hashmap for both files and directories, corresponding to `addContentToFile` and `mkdir` respectively. Further, for `ls`, we need not extract entries separately for the files and directories, since they are unified in the current case, and can be obtained in a single go.

This approach is inspired by @shawngao

Java Copy

```java
public class FileSystem {
    class File {
        boolean isfile = false;
        HashMap < String, File > files = new HashMap < > ();
        String content = "";
    }
    File root;
    public FileSystem() {
        root = new File();
    }

    public List < String > ls(String path) {
        File t = root;
        List < String > files = new ArrayList < > ();
        if (!path.equals("/")) {
            String[] d = path.split("/");
            for (int i = 1; i < d.length; i++) {
                t = t.files.get(d[i]);
            }
            if (t.isfile) {
                files.add(d[d.length - 1]);
                return files;
            }
        }
        List < String > res_files = new ArrayList < > (t.files.keySet());
        Collections.sort(res_files);
        return res_files;
    }
```

Performance Analysis

- The time complexity of executing an `ls` command is $O(m + n + klog(k))$. Here, m refers to the length of the input string. We need to scan the input string once to split it and determine the various levels. n refers to the depth of the last directory level in the given input for `ls`. This factor is taken because we need to enter n levels of the tree structure to reach the last level. k refers to the number of entries(files+subdirectories) in the last level directory(in the current input). We need to sort these names giving a factor of $klog(k)$.

- The time complexity of executing an `mkdir` command is $O(m + n)$. Here, m refers to the length of the input string. We need to scan the input string once to split it and determine the various levels. n refers to the depth of the last directory level in the `mkdir` input. This factor is taken because we need to enter n levels of the tree structure to reach the last level.

- The time complexity of both `addContentToFile` and `readContentFromFile` is $O(m + n)$. Here, m refers to the length of the input string. We need to scan the input string once to split it and determine the various levels. n refers to the depth of the file name in the current input. This factor is taken because we need to enter n levels of the tree structure to reach the level where the files's contents need to be added/read from.

- The advantage of this scheme of maintaining the directory structure is that it is expandable to include even more commands easily. For example, `rmdir` to remove a directory given an input directory path. We need to simply reach to the destined directory level and remove the corresponding directory entry from the corresponding $dirs$ keys.

- Renaming files/directories is also very simple, since all we need to do is to create a temporary copy of the directory structure/file with a new name and delete the last entry.

- Relocating a hierarchichal subdirectory structure from one directory to the other is also very easy, since, all we need to do is obtain the address for the corresponding subdirectory class, and assign the same at the new positon in the new directory structure.

- If the number of directories is very large, we waste redundant space for $isfile$ and $content$, which wasn't needed in the first design.

- A problem with the current design could occur if we want to list only the directories(and not the files), on any given path. In this case, we need to traverse over the whole contents of the current directory, check for each entry, whether it is a file or a directory, and then extract the required data.

Analysis written by: @vinod23

Rate this article: