Directory To Tree

Walking a directory tree on a file system programmatically is a commonplace task. In this challenge, you'll use Node.js's file system library to produce an object tree containing information for a directory and all its subdirectories up to a specified depth. This structure could be useful for manipulating in memory or transmitting and displaying a file system on a web page or terminal application.  
   
For the purposes of this challenge, assume a Linux file system.  
   
Complete the function  
   
directoryToTree(rootDir, depth)  
Parameters  
rootDir: String - A path to the root directory to be processed (not the root directory of the file system). This is always relative to process.cwd().  
   
depth: Number - the number of levels in the directory tree to include in the result structure; must be >= 0 where 0 represents the root level, inclusive  
   
Return Value  
an object corresponding to the tree structure  
   
The return value of this function should be a hierarchical tree-like object that contains two types of nodes: file nodes and directory nodes, corresponding to a file or directory entity in the file system. Nodes in the output structure must be objects containing 4 or 5 properties as described below:  
   
name: the file or directory name, including extensions, and without leading path or trailing slash (in the case of directories).

path: the string that represents the path to the file or directory from and including the origin rootDir directory parameter as well as the file or directory name. This should be a relative path and should not be prefixed with "./" or postfixed with a trailing "/".

type: for the purposes of this challenge, either "dir" (for directory) or "file". No symbolic links or other file types need be handled.

size: an integer number of bytes the file or directory occupies on disk.

children: this is the only conditional property. This property must be present for all type: "dir" nodes (even if empty), and must be absent for all type: "file" nodes. This property contains a list of child nodes that exist within the current directory, which are recursively defined according to the same structural requirements as the parent.

All input strings will point to a valid directory. There is no need to worry about error handling in this challenge, although you're welcome to make your code robust to failures if you wish. You're also welcome to consult Node documentation for locating functions and modules that will help you obtain the needed information.  
   
Examples  
Examples below will utilize the same directory structure as in your project:  
   
.  
|-- dummy\_dir  
|   |-- a\_dir  
|   |   `-- test\_file1.md  
|   |-- b\_dir  
|   |   `-- test\_file2.md  
|   |-- test\_file0.md  
   
3 directories, 3 files  
The three markdown files contain the following content:  
   
test\_file0.md: # Hello world  
test\_file1.md: (empty)  
test\_file2.md: # Hi  
Example 1  
serializeDirTree("dummy\_dir/a\_dir", 5);  
This path points to the a\_dir directory, which contains only one file containing no data. It should return:

{  
    "path": "dummy\_dir/a\_dir",  
    "name": "a\_dir",  
    "type": "dir",  
    "size": 4096,  
    "children": [  
        {  
            "path": "dummy\_dir/a\_dir/test\_file1.md",  
            "name": "test\_file1.md",  
            "type": "file",  
            "size": 0  
        }  
    ]  
}  
Example 2  
serializeDirTree("dummy\_dir", 5);  
This example has a sufficiently large depth limitation that it includes all files and directories in the tree structure. Take note of the file sizes. It should return:  
   
{  
    "path": "dummy\_dir",  
    "name": "dummy\_dir",  
    "type": "dir",  
    "size": 4096,  
    "children": [  
        {  
            "path": "dummy\_dir/a\_dir",  
            "name": "a\_dir",  
            "type": "dir",  
            "size": 4096,  
            "children": [  
                {  
                    "path": "dummy\_dir/a\_dir/test\_file1.md",  
                    "name": "test\_file1.md",  
                    "type": "file",  
                    "size": 0  
                }  
            ]  
        },  
        {  
            "path": "dummy\_dir/b\_dir",  
            "name": "b\_dir",  
            "type": "dir",  
            "size": 4096,  
            "children": [  
                {  
                    "path": "dummy\_dir/b\_dir/test\_file2.md",  
                    "name": "test\_file2.md",  
                    "type": "file",  
                    "size": 4  
                }  
            ]  
        },  
        {  
            "path": "dummy\_dir/test\_file0.md",  
            "name": "test\_file0.md",  
            "type": "file",  
            "size": 13  
        }  
    ]  
}  
Example 3  
serializeDirTree("dummy\_dir", 1);  
Here, the depth has been limited to 1 level (the 0-th level is the root level). In other words, only return the root node, dummy\_dir, along with the immediate children of dummy. Were the depth argument set to 0, we'd only return the root node (depth will never be negative).  
   
Output:  
   
{  
    "path": "dummy\_dir",  
    "name": "dummy\_dir",  
    "type": "dir",  
    "size": 4096,  
    "children": [  
        {  
            "path": "dummy\_dir/a\_dir",  
            "name": "a\_dir",  
            "type": "dir",  
            "size": 4096,  
            "children": []  
        },  
        {  
            "path": "dummy\_dir/b\_dir",  
            "name": "b\_dir",  
            "type": "dir",  
            "size": 4096,  
            "children": []  
        },  
        {  
            "path": "dummy\_dir/test\_file0.md",  
            "name": "test\_file0.md",  
            "type": "file",  
            "size": 13  
        }  
    ]  
}  
Feel free to add and manipulate directories as you wish to create additional tests.  
  
//Below is the code for test cases. Good luck

//Test using Jest

require("util").inspect.defaultOptions.depth = null;

const {assert, config} = require("chai");

const {runTest} = require("./test\_runner");

config.truncateThreshold = 0;

describe("Example Test Cases", () => {

it("should work on a simple example", () => {

const expected = {

path: "dummy\_dir/a\_dir",

name: "a\_dir",

type: "dir",

size: 4096,

children: [

{

path: "dummy\_dir/a\_dir/test\_file1.md",

name: "test\_file1.md",

type: "file",

size: 0

}

]

};

return runTest("dummy\_dir/a\_dir", 5, expected);

});

it("should work on a more complex example", () => {

const expected = {

path: "dummy\_dir",

name: "dummy\_dir",

type: "dir",

size: 4096,

children: [

{

path: "dummy\_dir/a\_dir",

name: "a\_dir",

type: "dir",

size: 4096,

children: [

{

path: "dummy\_dir/a\_dir/test\_file1.md",

name: "test\_file1.md",

type: "file",

size: 0

}

]

},

{

path: "dummy\_dir/b\_dir",

name: "b\_dir",

type: "dir",

size: 4096,

children: [

{

path: "dummy\_dir/b\_dir/test\_file2.md",

name: "test\_file2.md",

type: "file",

size: 4

}

]

},

{

path: "dummy\_dir/test\_file0.md",

name: "test\_file0.md",

type: "file",

size: 13

}

]

};

runTest("dummy\_dir", 5, expected);

});

it("should work on a more complex example with depth limiting", () => {

const expected = {

path: "dummy\_dir",

name: "dummy\_dir",

type: "dir",

size: 4096,

children: [

{

path: "dummy\_dir/a\_dir",

name: "a\_dir",

type: "dir",

size: 4096,

children: []

},

{

path: "dummy\_dir/b\_dir",

name: "b\_dir",

type: "dir",

size: 4096,

children: []

},

{

path: "dummy\_dir/test\_file0.md",

name: "test\_file0.md",

type: "file",

size: 13

}

]

};

return runTest("dummy\_dir", 1, expected);

});

});