# CSC148 Assignment 2:

# A Treemap Visualization

July 17th, 2023

## Table of contents

# Introduction

Trees are a fundamental data structure used to model hierarchical data. Often, a tree represents a hierarchical categorization of a base set of data, where the leaves represent the data values themselves, and internal nodes represent groupings of this data. Files on a computer can be viewed this way: regular files (e.g., PDF documents, video files, Python source code) are grouped into *folders*, which are then grouped into larger folders, etc.

Sometimes the data in a tree has some useful notion of size. For example, in a tree representing the departments of a company, the size of a node could be the dollar amount of all sales for that department, or the number of employees in that department. Or in a tree representing a computer's file system, the size of a node could be the size of the file.

A **treemap** is a visualization technique to show a tree's structure according to the weights (or sizes) of its data values. It uses rectangles to show subtrees, scaled to reflect the proportional sizes of each piece of data. Treemaps are used in many contexts such as news headlines, various kinds of financial information, and computer disk usage. Some free programs use treemaps to visualize the size of files on your computer, for example:

- (Windows) WinDirStat, (OSX) Disk Inventory X, (Linux) KDirStat

For this assignment, you will write an interactive treemap visualization tool that you can use to visualize hierarchical data. It will have a general API (implemented with inheritance, naturally!) and you will define a specific subclass that will allow you to visualize the **files and folders in your computer.**

## Setup and starter code

Please download the starter file zip located here and save it into your assignments/a2 folder.

Unzipping this file should give you the following files and folders:

- `tm_trees.py`
- `a2_sample_test.py`
- `example directory`
- `treemap_visualizer.py`

The file in which you will modify is the `tm_trees.py` file. There are two classes in this file: `TMTree` and `FileSystemTree`. The descriptions of these classes are the documentation sections: ***TMTree Class*** and ***FileSystemTree Class.***

The `a2_sample_test.py` file is for you to test your code and debug. Add your own test cases to this file and use it to debug your code.

The `example directory` is a directory containing folders and files which you will use as test data for your tree. It may be useful to familiarize yourself with this directory. There is a visualization of the structure in both a tree, and the treemap format in the section: ***Visualization Example: The Example Working Directory*** in the documentation section.

The `treemap_visualizer.py` file contains code which runs an interactive graphical window showing a treemap visualization of this data to the user. This component is provided to you and contains code that uses your data modeller to run the visualization.

# Tasks

## Task 0: Understand the assignment

Read the introduction, and the `TMTree` and `FileSystemTree` descriptions in the documentation section. Look through the starter files and the example directory.

**GENERL NOTE:** You may find it helpful to add helper methods, this is allowed. You may add parameters to the existing methods, as long as they are given a default value (i.e., our testing functions will be able to call it).

## Task 1: Complete the initializers

The classes `TMTree` and `FileSystemTree` are defined in the file called `tm_trees.py`. The starter code provides a skeleton of both classes. To complete the first task, you must complete the initializers of each class.

### 1.1 Complete the initializer of `TMTree`.

Use the docstring and the list of attributes in the documentation to guide you. The tree's will be instantiated through the `FileSystemTree` initializer starting with the leaf nodes. Leaf nodes will have their `data_size` set to the size of the file. Internal nodes will have the `data_size` parameter initially set to zero; this needs to be updated in the initializer based on the sum of the sizes of `_subtrees`. The tree in the ***Visualization Example: The Example Working Directory*** in the documentation section may be a useful reference.

Use the `get_colour()` function to set the colour of the tree node. This function selects a random colour in the format of `rgb` values which is not on the grey scale (i.e., it intelligently chooses a colour which is not on the grey scale).

### 1.2 Complete the initializer of `FileSystemTree`.

This method should be implemented recursively (i.e., it should involve calling the `FileSystemTree` initializer to build subtrees). To complete this method, **you will need to use the OS module**. It is already imported in the file. Here is the python documentation for the OS module. For your convenience, this information is summarized in ***OS Module Summary*** in the documentation section.

Idea for Algorithm

- Instantiate a `TMTree` instance if the path is a file.
- If the path is a folder, recursively instantiate each item in the folder and build a list of this folders subtrees. Use the list of subtrees when instantiating this folder as a `TMTree` instance.

**Progress Check!**

After this task is complete, the first two tests on the `a2_sample_test.py` file should pass.

`test_single_file():` This tests the instantiation of a `FileSystemTree` with a tree with a single node (a file).
`test_example_data():` This tests the instantiation of a `FileSystemTree` using the example directory.

## Task 2: Implementing the Treemap Algorithm

This task involves writing two methods which construct the layout of the 2D visualization window by mapping each tree node onto a rectangle in the visualization window, based on the tree structure and `data_size` attribute. There are two methods two write for this task.

### 2.1 Write the method: `update_rectangles`.

This method is a recursive algorithm which assigns the `rect` attribute to each node in the tree. It will be called on the root node of the tree after initialization. The input is the size of the visualization window. There is no return, instead it directly modifies the `rect` attribute for each node in the tree.

The algorithm is described in detail in the documentation section ***Treemap Algorithm***. Use the example directory illustration in the documentation section ***Visualization Example*** to work through what your rectangles should look like after using the algorithm.

### 2.2 Write the method: `get_rectangles`.

This method is used to display the `rect` and `_colour` attributes for each ***leaf** node in the visualization. It is returned as a list of tuples, each tuple has two elements: `(rect, colour)`. Pygame will take each of these elements and display it as a rectangle on the display. In the working directory example this would be:

```
[((0, 0, 94, 28), (r, g, b)),
 ((0, 28, 94, 69), (r, g, b)),
 ((0, 97, 94, 3), (r, g, b)),
 ((94, 0, 76, 100), (r, g, b)),
 ((170, 0, 30, 72), (r, g, b)),
 ((170, 72, 30, 28), (r, g, b))]
```

Where `r, b, g` are random integers between 0 and 255.

***leaf:** This will change in task 6 when you change the `_expanded` attribute. At that point, it will return information for each node which are either **not expanded** or **leaf nodes** from an expanded folder.

**Progress Check!**

After this task is complete the following tests on the `a2_sample_test.py` file should pass:

`test_single_file_rectangles(x, y, width, height):` This tests the `update_rectangles` and the `get_rectangles` methods with a tree with a single root node.
`test_example_data_rectangles():` This tests the `update_rectangles and get_rectangles` methods with the example directory.
**NOTE:** Since the file ordering may be different, the subtrees are sorted for testing purposes. With the sorting, the results are different from the visualization you may see. You don't need to preform any sorting anywhere in the assignment.

Next, run the `treemap_visualizer.py` file. You will see at the bottom of the file, we have set `PATH_TO_VISUALIZE` as the `example directory`. You may customize this for testing purposes. Running this file should display a window which looks like the image in the ***Visualization Example: The Example Working Directory*** except each rectangle should be coloured randomly.

## Task 3: Select a Tree

The first step to making our treemap interactive is to allow the user to select a node in the tree. There is one method to complete for this task:

### 3.1 write the method `get_tree_at_position`.

This method takes the position of the mouse click, in the form of an xy-coordinate and returns the ***leaf** corresponding to that position in the treemap.

**Hint:** the function should be recursive. upon each iteration, find the subfolder which contains the xy-coordinate of the mouse, then recurse on that that subtree. Once you find it, return that current tree. If the xy-coordinate is not within the boundaries of the visualization, return None.

***leaf:** This will change in task 6 when you change the `_expanded` attribute. At that point, it will return a tree node which is either **not expanded** or it is a **leaf node**.

**Progress Check!**

Try running `treemap_visualizer.py` again, and then click on a rectangle on the display. You should see the path to that file displayed at the bottom of the screen. Click again to unselect that rectangle and try selecting another one.

## Task 4: Making the Display Interactive

Now that we can select nodes, we can move on to making the treemap more interactive.

Inside the `event_loop` function in the `treemap_visualizer.py` module, we have provided a skeleton of a loop that repeatedly waits for a new event, and then processes the event based on its type. To support these events, complete the following methods:

### 4.1 write the method `update_data_sizes`.

In the following steps, you will write methods to change the size of tree nodes and delete tree nodes. In order for these reflections to be seen across the tree, you will first write a method which updates the `data_size` attributes throughout the whole tree.

This method will be called any time after the tree has been modified. Since some change has been made, the data sizes of parent nodes should be reflected. This method will update the `data_size` attributes for every internal node in the tree based on the sum of sizes of children.

The method should return the size of the node that was called.

### 4.2 write the method `change_size`.

Change the `data_size` attribute for the given tree by its parameter `<factor>` which will be a percentage in the form of a float. This method only works for leaf nodes.

**Note:** Factor may be negative. A leaf's `data_size` cannot decrease below 1. There is no upper limit. The 1% is always rounded up before applying the change.  See the examples below:

```
data_size = 150
factor = 0.01
The change is: 150 * 0.01 = 1.5
That becomes rounded up to 2. Therefore:
      -> data_size = 152
      -> If factor was -0.01, then data_size would be 148.

data_size = 150
factor = -1.2
The change is: 150 * -1.2 = -180
Since this would result in a negative data size:
      -> data_size = 1
```

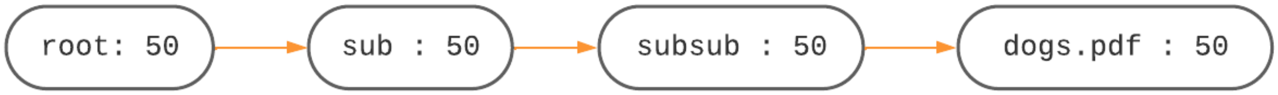
**4.3 write the method `delete_self`.**

Write a method that deletes the given tree node from the tree by removing it from its parents list of children. This method could be called on either leaf or internal nodes. If the given tree node is the only child of its parent (i.e., `cats.pdf` is the only child of `images` in the example directory), then you need to recursively delete the parent.

So, if you delete `cats.pdf`, both cat `cats.pdf` and `images` would be deleted.

If the method returns true, the visualizer will update the `data_size` attributes, thus, you don't need to adjust any data sizes. Note that if the recursion falls back to the root, the method should return false, and thus the root node will not be deleted, and the size not updated.

To illustrate this, suppose the file system was the following:



If we called delete on `dogs.pdf` (or either `sub` or `subsub`), the resulting visualization should show only the root with a size of 50, since all trees except root have been deleted, but `update_sizes` is not called (since that would result in data size of zero).

Do not set `self.parent_tree` to None, because it might be used by the visualizer to go back to the parent folder.

**Progress Check!**

One of the nice things about code with an interactive display is that it's usually straightforward to test basic correctness.

Run the treemap visualizer and see if you can resize by selecting a node and clicking **"up"** and **"down"** arrows on your keyboard. Each click should modify the size by 1%. Delete rectangles by selecting a node and clicking the **delete** button. Make sure the text displays updates when the selected rectangle changes.

## Task 5: Update Colours and Depths

Currently, all your tree nodes have a random colour assigned to them, and currently, in your visualization, it only displays the leaf nodes (i.e., the files in the directory). In the next section, we will change this so you can view the folders, and open and close folders. Before we do that, we are going to make a change to the colouring of the tree nodes, so that internal nodes (folders) are coloured on the grey scale. To complete this task, four methods will be written.

### 5.1 write the method `update_depths`.

This method will be called on the root node in the tree. The root node should have a depth of 0. The depth of the tree corresponds with the distance to the root node. This method should be recursive, and it updates the `_depth` attribute for each tree node (including leaves). There is no return value.

### 5.2 write the method `max_depth`.

This method is called after `update_depths` is called, and it returns the maximum depth of the tree. In the working directory example with workshop as the root node, the maximum depth would be 3.

### 5.3 write the method `update_colours`.

The colouring works in the following way:

- All leaf nodes are random colours, as assigned by the initializer.
- The internal nodes should be a shade of grey depending on their depth.
- The shade of the root node is black. The `rgb` value for black is (0, 0, 0).
- The shade of subsequent internal nodes takes on a shade between (0, 0, 0) and (200, 200, 200), with an equal step size between depths.

Recall that in `rgb` values: (0, 0, 0) is black, (255, 255, 255) is white and all numbers in between are shades of grey. See the illustrations below for examples of how the colouring would work at different depths:

```
    Let max_depth = 5
          -> The internal node shades will be the following colours:
                0 : (0, 0, 0)
                1 : (50, 50, 50)
                2 : (100, 100, 100)
                3 : (150, 150, 150)
                4 : (200, 200, 200)

    Let max_depth = 4
          -> The internal node shades will be the following colours:
                0 : (0, 0, 0)
                1 : (66, 66, 66)
                2 : (132, 132, 132)
                3 : (198, 198, 198)
```

The update colours method takes in a parameter for the in the `step_size` (50 and 66 respectively in the examples), and recursively updates the grey scale colour for each internal node. It should not change the leaf nodes.

### 5.4 write the method `update_colours_and_depths`.

This method puts it all together. It should first call your `update_depths` method, then find the maximum, and use that to determine the step size for the `update_colours` method, then call that method. This method is called after instantiation of a tree, or after changes are made to the tree.

**Progress Check!**

Since the tree is displaying only the leaf nodes, at this point you won't see the change in visualization. There is one final test on the `a2_sample_test.py` file which should pass. At this point, all test cases on this file should pass.

`test_update_colours_and_depths():` This test case tests the `update_colours_and_depths` function by running a tree traversal and returning the depths and colours of the internal nodes.

## Task 6: Implementing the displayed-tree

Now you will implement the displayed-tree functionality for the visualizer, so that the user can expand and collapse their view of the data. The operations **E** (expand), **A** (expand all), **C** (collapse), and **X** (collapse all), will be implemented in this section. Review the operations list in the documentation section ***Pygame Visualization Operations***. Note that the `update_data_sizes` and `update_rectangles` methods are called after any of these keys are pressed which allow the changes to be displayed on the visualization.

### 6.1 Update the `_expanded` attribute.

1. Edit the initializer for `TMTree` to set the initial value of `_expanded` to False for all trees.
2. Update `get_rectangles()` in a way such that it returns the tree node information for all nodes which are either leaves or trees which are not expanded.
3. Update `get_tree_at_position()` to return the correct node. Now, this node may either be a leaf, or an internal tree node which is not expanded.

### 6.2 Write the method `expand`.

This method sets this tree to be expanded, unless it is a leaf, since that cannot be expanded.

### 6.3 Write the method `expand_all`.

This method sets all descendants of the given tree node to be expanded. Note again that leaves should not be expanded.

### 6.4 Write the method `collapse`.

This method collapses the given folder. (i.e., the given node and all sibling nodes will be replaced by the folder which holds them)

**Note:** This method could be called with any node on the tree (i.e., it doesn't have to be a leaf node or a node which is not expanded). Thus, you need to collapse the parent of the given tree and **collapse every descendant of it**.

### 6.5 Write the method `collapse_all`.

This method collapses the tree down to the root node. This method could be called on any node in the tree. After this method is called, ALL nodes in the tree should be collapsed (i.e., the _expanded attribute is set to false for all tree nodes)

**Progress Check!**

Run the treemap visualizer. You should initially see a single black rectangle. You should be able to click on it and press the **E** to key to expand it. From there you can test out **E**, **A**, **C** and **X** operations. Make sure the text display always updates when the selected rectangle changes. The folders should be shades of grey based on the completion of your last task, and the leaf nodes should be random colours.

## Task 7: Move around files

This is the final task. The following methods provide functionality for the operations **M** (move), **D** (duplicate), and **V** (copy/paste). Note that the `update_data_sizes` and `update_rectangles` methods are called after any of these keys are pressed which allow the changes to be displayed on the visualization.

### 7.1 write the method `move`.

This method moves a file into a folder. The user will select a file by clicking it, and they will hover their cursor over the folder in where they want to move the file. To do this, the pointers of parents and children should be adjusted.

The method only works when the selected tree is a leaf, add destination is not a leaf.

### 7.2 write the method `duplicate`.

This method creates a duplicate of the file that is being selected. It only works with leaf nodes (i.e. files). The selected file should be duplicated, and the new version should be stored in the same folder. To do this, a new `FileSystemTree` instance should be created. Recall that `FileSystemTree` instances are instantiated using a path; Use the `get_full_path()` method to access this.

**Note:** if the `data_size` of this node was previously modified by `change_size`, the duplicated node will have the original `data_size` of the file, since you should instantiate it by the path, so in this case, they would have different sizes.

### 7.3 write the method `copy_paste`.

This method is similar to the move method, except it copies the file, and moves the new file to the destination folder. It only works when the selected tree is a leaf, add destination is not a leaf.

**Progress Check!**
Run the treemap visualizer. You should be able to use the functionality of **M**, **D** and **V**. Revisit the YouTube demo video if you are unsure how to use these operations. Make sure the text display always updates when the selected rectangle changes.

**Congratulations! You have completed A2.**

# Documentation

## TMTree Class

`TMTree` is a common interface. The treemap visualizer will expect for any sort of data that we want to visualize to have that same interface. Different types of hierarchical data are subclasses of `TMTree`.

**Each node in the tree is an instance of the `TMTree` class.** The root node points to its children through the `_subtrees` attribute. The leaf nodes have no children, therefore `_subtrees = []` for leaf nodes.

### Instantiation

The initializer of `TMTree` is considered private and should be *overridden* by each subclass. This means that you cannot initialize a `TMTree` instance independently, it must be done in the initializer of one of its subclasses.

Instantiate a `TMTree` by the following: `TMTree.__init__(self, name, subtrees, size)`

### Attributes

The following table shows a summary of the attributes in the `TMTree` class.

| Name | Type | Description |
|---|---|---|
| **data_size** | int | The size of the file or folder. If it is a file, it is the size of the file (in bytes). If it is a folder, it is the sum of all the sizes of the files and folders within it. An empty folder would have size zero. |
| **rect** | Tuple(int, int, int, int) | The rect describes the positioning of the tree node in the visualization. It is a tuple of four positive integers: (x, y, width, height), where (x, y) represents the upper-left corner of the rectangle. The width and height describe the rectangle's width and height. The upper-left corner of the visualization window is at coordinate (0, 0) and the y-axis points down. |
| **_colour** | Tuple(int, int, int) | The colour of the rectangle displayed in the visualization. It is a tuple of three integers which are the (R, G, B) values of the colour. Note that (R, G, B) values are integers ranging from 0 to 255 inclusive. |
| **_name** | str | The name of the file or folder. |
| **_parent_tree** | TMTree | A pointer to the TMTree object which is the tree's parent. |
| **_subtrees** | List[TMTree] | A list of the subtrees. |
| **_expanded** | bool | A Boolean that is true if the tree node is expanded in the visualization, and false if it is not. Leaf nodes should never be expanded (i.e., it should always be set to false for leaf nodes. In the starter code, _expanded is set to True for all nodes (except the leaf nodes). After task 6, it will me modified so that _expanded is initially set to False for all nodes. |

**Methods**

The following table shows a list of all the methods in the `TMTree` class. It is noted in the description if this is a method in which you will need to implement.

| Name | Parameters | Return | Description | Relevance |
|---|---|---|---|---|
| **__init__** | **name**<br>**subtrees:** a list of TMTree objects, which are children of the tree node.<br>**data_size:** the size of the tree node. | None | Initializes a TMTree object. The initial attributes are set, and the new instance sets itself as the parent for all children nodes. | TODO in Task 1<br>Started but incomplete. |
| **is_empty** | None | Bool | Returns True iff this tree is empty. | Prewritten. May be used if needed. |
| **get_parent** | None | TMTree (optional) | Returns the parent of this tree. | Prewritten. May be used if needed. |
| **update_rectangles** | **rect:** tuple with (x,y,width,height) | None | Assigns the _rect attribute to each node in the tree using the Treemap Algorithm. | TODO in Task 2 |
| **get_rectangles** | None | Tuple<br><br>(rect, colour) | Return a list with tuples for every leaf* in the displayed-tree rooted at this tree. Each tuple consists of a tuple that defines the appropriate pygame rectangle to display for a leaf, and the colour to fill it with.<br><br>*In task 5 this changes | TODO in Task 2 |
| **get_tree_at_position** | **pos:** the (x,y) coordinate where the mouse click occurs | TMTree (optional) | Returns the leaf* node on the visualization which is located at the inputted position <pos>.<br><br>*In task 5 this changes | TODO in Task 3 |
| **change_size** | factor | None | Changes the data size of a leaf node by the parameter <factor>. | TODO in Task 4 |
| **delete_self** | None | Bool | Removes the tree node from the visualization, and returns True if the removal was successful. | TODO in Task 4 |
| **update_data_sizes** | None | None | Updates the data_size attribute for all nodes in the tree. This is called anytime after a change is made, and it is called with the root of the tree. | TODO in Task 4 |

| | | | | |
|---|---|---|---|---|
| **update_depths** | None | None | Updates the _depth attribute for all nodes, starting at the root node with a depth of zero. | TODO in Task 5 |
| **max_depth** | None | int | Returns the maximum depth of the tree. It is called with the root of the tree. | TODO in Task 5 |
| **update_colours** | step_size | None | Updates the _colour attribute for all internal nodes in the tree. The root node is black (0, 0, 0), and each subsequent is a shade darker based on step_size. | TODO in Task 5 |
| **update_colours_and _depths** | None | None | Calls the update_depth method, the max_depth, and uses that as the step size to call update_colours. | TODO in Task 5 |
| **expand** | None | None | Set this tree node to be expanded unless it is a leaf. | TODO in Task 6 |
| **expand_all** | None | None | Sets all descendants of the selected tree node to be expanded except the leaves. | TODO in Task 6 |
| **collapse** | None | None | Collapses this folder by setting the parents _expanded attribute to false. Also sets all descendants _expanded attributes to false. | TODO in Task 6 |
| **collapse_all** | None | None | Collapses all nodes in the tree. | TODO in Task 6 |
| **move** | destination | None | Moves the selected node to the destination. | TODO in Task 7 |
| **duplicate** | None | TMTree (optional) | Duplicates the selected node. It must be a leaf node. If duplication is successful, returns the new TMTree object. | TODO in Task 7 |
| **copy_paste** | destination | None | Duplicates the selected node, and moves it to the destination. | TODO in Task 7 |
| **get_path_string** | none | str | Implemented in FileSystemTree class. Can be called in TMTree. | Prewritten. |
| **get_separator** | none | str | Implemented in FileSystemTree class. Can be called in TMTree. | Prewritten. |
| **get_suffix** | none | str | Implemented in FileSystemTree class. Can be called in TMTree. | Prewritten. |

## FileSystemTree

The `FileSystemTree` is a subclass of `TMTree`, which stores a directory from your computer as follows:

- Each *internal node* corresponds to a *folder*.
- Each *leaf* corresponds to a *regular file* (e.g., PDF document, movie file, or Python source code file).

The `_name` attribute will always store the **name** of the folder or file (e.g., preps or prep8.py).

The `_data_size` attribute of a leaf (i.e., a file) is simply how much space (in bytes) the file takes up on the computer.

The `_data_size` of an internal node (i.e., a folder) corresponds to the sum of the sizes (in bytes) of all files contained in the folder, including its subfolders.

### Instantiation

A `FileSystemTree` instance is instantiated with the following: `tree = FileSystemTree(path)`

Where path is a string with a pathname. For example:

```
path = ‘C:\Users\Andrea\Desktop\csc148\A2\example-directory\workshop’
```

### Description

After instantiation (i.e. `tree = FileSystemTree(path)`), the variable `tree` is an instance of the `FileSystemTree` object, which is a subclass of `TMTree`. This means that the initializer of `FileSystemTree` must instantiate a `TMTree` object.

Using the instantiation of the path above, the following may be called:

```
print(tree)
>>> <tm_trees.FileSystemTree object at 0x00000222CC6CCA10>

print(tree._name)
>>> workshop

print([subtree._name for subtree in tree._subtrees])
>>> ['activities', 'draft.pptx', 'prep']

print(tree.path)
>>> C:\Users\Andrea\Desktop\csc148\A2\example-directory\workshop

print(tree._subtrees[0].path)
>>> C:\Users\Andrea\Desktop\csc148\A2\example-directory\workshop\activities
```

**Attributes**

Since `FileSystemTree` is a subclass of `TMTree`, all the attributes are the same as the `TMTree` attribute. There is one additional attribute:

| Name | Type | Description |
|---|---|---|
| **path** | string | A string containing the path of the tree object. |

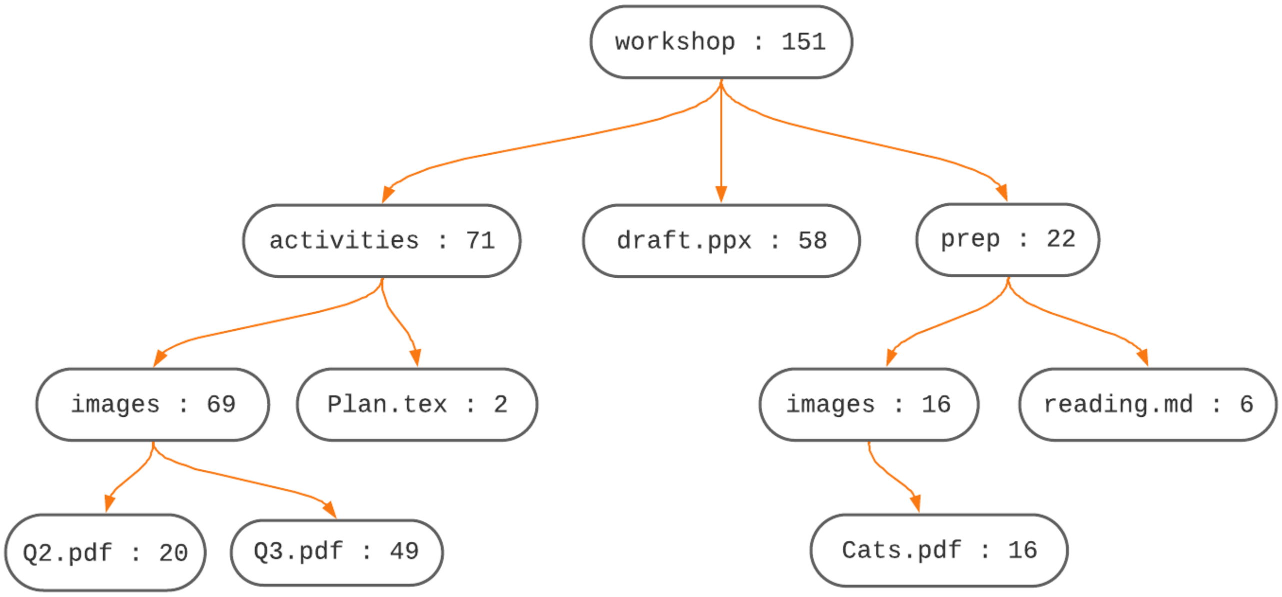
**Methods**

The following table contains a list of methods in the `FileSystemTree`.

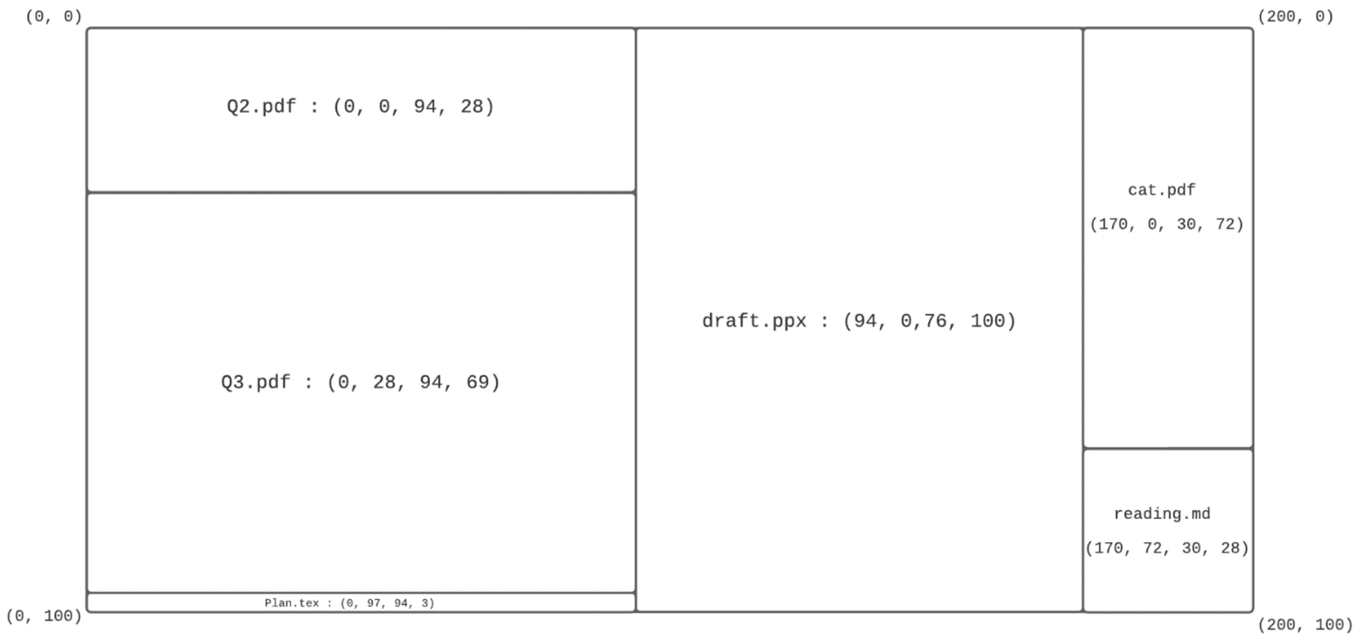
| Name | Input | Return | Description | Relevance |
|---|---|---|---|---|
| **__init__** | **path:** string of directory path | None | Recursively constructs a tree where the root of the tree is the base of the path parameter. | TODO in Task 1 |
| **get_full_path** | None | string | Returns the path attribute for this tree. | Prewritten. May be used if needed. |
| **get_separator** | None | string | Return the string used to separate names in the string representation of a path from the tree root to this tree. It returns either '/' or '\' depending on OS. | Prewritten. Not relevant to your tasks. |
| **get_suffix** | None | string | Return the string used at the end of the string representation which is the suffix for the size of data. | Prewritten. Not relevant to your tasks. |

## Visualization Example: The Example Working Directory

For your convenience, we have drawn the tree of the sample working directory, starting with workshop as the root of the tree. Notice how the size of each folder is a sum of the sizes of the items within the folders.



After completing Task 2 and implementing the `update_rectangles` and `get_rectangles` methods, a visualization window with width 200 and height 100 will have the shape seen below. Each rectangle should be a random colour. The rectangle data is shown here to assist in your understanding of the treemap algorithm.



## Treemap Algorithm

For all rectangles, we'll use the **pygame representation of a rectangle**, which is a tuple of four integers `(x, y, width, height)`, where `(x, y)` represents the upper-left corner of the rectangle. Note that in pygame, the origin is in the upper-left corner and the y-axis points *down*. Each unit on both axes is a pixel on the screen.

There are many variations of the Treemap Algorithm. For this assignment, we'll use a dynamic version – one that lets us see different visualizations of the same tree. However, the process for generating the tree is always the same, even as the displayed-tree changes.

For simplicity, we'll use "`size`" to refer to the `data_size` attribute in the algorithm below.

The `update_rectangles` method should work like so:

**Input**: A data tree that is an instance of some subclass of `TMTree`, and a pygame rectangle (the display area to fill).
**Output**: None - It will directly modify the `rect` attributes of the tree objects inside the data tree.

Algorithm:

Note: Unless explicitly written as "displayed-tree", all occurrences of the word "tree" below refer to a data tree.

1. If the tree has size 0, then it's `rect` should have an area of 0.
2. If the tree is a leaf in the displayed-tree, then it should take up the entire area of the given rectangle.
3. Otherwise, if the display area's **width is greater than its height**: divide the display area into **vertical rectangles**, one smaller rectangle per subtree of the displayed-tree, in proportion to the sizes of the subtrees. The last subtree should take up the entire remaining space. Example:

```
input_rectangle = (0, 0, 200, 100)

width = 200

Let tree X be a tree which has three subtrees: (a, b, c) with
sizes (10, 25, 15) respectively. The total size is therefore 50.

Subtree a:
-> 20% of total_size
-> width = 40 (20% of 200)
-> rect = (0, 0, 40, 100)

Subtree b:
-> 50% of total size
-> 50% of total_size
-> width = 100 (50% of 200)
-> rect = (40, 0, 100, 100)

Subtree c:
-> last subtree
-> width = 200 - 140 = 60
-> rect = (140, 0, 60, 100)
```

Note that the three rectangles' edges overlap, which is expected.

**Note:** You can assume that every non-zero-sized subtree will have a large enough size that its computed rectangle has a width and height >= 1.

4. If the **height is greater than or equal to the width**, then make **horizontal rectangles** instead of vertical ones, and do the analogous operations as in step 3.

5. Recurse on each of the subtrees in the displayed-tree, passing in the corresponding smaller rectangles.

**Note about rounding**: because you're calculating proportions here, the exact values will often be floats instead of integers. For all but the last rectangle, always truncate the value (i.e., round **down to the nearest integer**). In other words, if you calculate that a rectangle should be (0, 0, 10.9, 100), round the width down to (0, 0, 10, 100). Then a rectangle below it would start at (0, 100), and a rectangle beside it would start at (10, 0).

However, the *final* (rightmost or bottommost) edge of the last smaller rectangle should *always* be equal to the outer edge of the input rectangle. This means that **the last subtree might be a bit bigger than its true proportion** of the total size, but it won't be a noticeable difference in our application.

You will implement this algorithm in the `update_rectangles` method in `TMTree`.

## Pygame Visualization Operations

All changes described below should **only change the tree object**; so, if a rectangle is deleted or its size changes, it **DOES NOT** modify the actual file on your computer!

When the `treemap_visualizer.py` file is running, a window with the treemap visualization will appear. The following table summarizes the operations which will be possible. The operations on tree nodes are preformed by clicking on a node to select it, then pressing the keyboard button. Capital letters are not required.

| OPERATION | DESCRIPTION |
|---|---|
| **QUIT** (X BUTTON IN CORNER) | The user can close the window and quit the program by clicking the X icon (like any other window). |
| **LEFT CLICK ON A RECTANGLE** | The text display updates to show two things: the names on the path from the root of the data tree to the selected tree node and the selected leaf's data_size. Clicking on the same rectangle again unselects it. Clicking on a different rectangle selects that one instead. |
| **E** | Expand the folder.<br><br>**Details:** If the user selects a rectangle, and then presses **E**, the tree corresponding to that rectangle is expanded in the displayed-tree, thus its children become visible. If the tree is a leaf, nothing happens. |
| **A** | Expand the folder and all folders inside.<br><br>**Details:** If the user selects a rectangle, and then presses **A**, the tree corresponding to that rectangle, as well as all of its subtrees, are expanded in the displayed-tree. If the tree is a leaf, nothing happens. |
| **C** | Collapse the parent folder. |

| | |
|---|---|
| | **Details:** If the user selects a rectangle, and then presses **C**, the parent of that tree is unexpanded (or "collapsed") in the displayed-tree. If the parent is None because this is the root of the whole tree, nothing happens. |
| **X** | Collapse the entire display.<br><br>**Details:** If the user selects any rectangle, and then presses **X**, the entire displayed-tree is collapsed down to just a single tree node. If the displayed-tree is already a single node, nothing happens. |
| **Q** | Visualize the selected folder.<br><br>**Details:** If the user selects any rectangle, and then presses **Q**, the selected rectangle will replace the current displayed-tree. |
| **B** | Go back to parent folder (if **Q** was pressed).<br><br>**Details:** If the user presses **B** at any time, the selected rectangle's parent tree (if any) will replace the current displayed-tree. |
| **"UP" AND "DOWN" ARROW KEYS** | Change the size of a file (in the visualization).<br><br>**Details:** If the user presses the **Up Arrow** or **Down Arrow** key when a rectangle is selected, the selected leaf's data_size increases or decreases by 1% of its current value, respectively. |
| **M** | Move a file.<br><br>**Details:** If the user selects a rectangle that is a leaf, and hovers the cursor over another rectangle that is an internal node and presses **M**, the selected leaf should be moved to be a subtree of the internal node being hovered over. If the selected node is not a leaf or if the hovered node is not an internal node, nothing happens. |
| **DELETE KEY** | Delete a file or folder from the visualization.<br><br>**Details:** If the user selects a rectangle that is either a leaf or internal node and then presses the **delete** key, if it has a parent (i.e., it is not the root node), then that node should be deleted by removing it from the parent's subtrees (and hence in the visualization). |
| **D** | Duplicate a file.<br><br>**Details:** If a user selects a leaf, and then presses **D**, the selected leaf will be duplicated, and the duplicated leaf will be stored in the same folder and the selected leaf. If the selected rectangle is not a leaf, nothing happens. |
| **V** | Copy and paste a file.<br><br>**Details:** If the user selects a rectangle that is a leaf, then hovers the cursor over another rectangle that is an internal node and presses **V**, the selected leaf is duplicated, and moved be a subtree of the internal node being hovered over. If the selected node is not a leaf, or if the hovered node is not an internal node, nothing happens. |

## OS module documentation summary

To complete task 1.2, you will need use pythons OS Module to get data about your computer's files in a Python program. We recommend using the following functions:

- `os.path.isdir`
- `os.listdir`
- `os.path.join`
- `os.path.getsize`
- `os.path.basename`

You may **NOT** use the `os.path.walk` function.

**Warning**: don't use string concatenation (+) to try to join paths together. Because different operating systems use different separators for file paths, you run the risk of using a separator that is invalid on the DH lab machines on which we'll test your code. So instead, make sure to use `os.path.join` to build larger paths from smaller ones.

**You should add the subtrees in your `FileSystemTree` in the order they are produced from `os.listdir`.**

The following provides a demo for what you can expect:

```
my_path = os.getcwd()        # returns a path to the current working directory

print(my_path)
>>> 'C:\Users\Andrea\Desktop\csc148\A2'

print(os.listdir(my_path))   # returns a list of the items in the directory of the path
>>> ['example-directory', 'a2_sample_test.py', 'print_dirs.py', 'tm_trees.py',
   'treemap_visualizer.py']

print(os.path.join(my_path, 'example-directory', 'workshop'))  # combines path with parameters
>>> 'C:\Users\Andrea\Desktop\csc148\A2\example-directory\workshop'

path_to_file = 'C:\Users\Andrea\Desktop\csc148\A2\example-directory\workshop\draft.ppx'
path_to_directory = 'C:\Users\Andrea\Desktop\csc148\A2\example-directory\workshop'

print(os.path.isdir(path_to_directory))  # returns True if the path is a folder
>>> True

print(os.path.isdir(path_to_file))       # returns False if the path is a file
>>> False

os.path.getsize(path_to_file)            # returns the size of the file
>>> 58

os.path.basename(path_to_file)           # returns the base name of the file path
>>> 'draft.ppx'
```