# Lab 13: Intro to Trees

## 1.1 Information

**Topics:** Trees, Recursion

**Turn in:** All source files (.cpp and .hpp).

**Starter files:** Download on GitHub or D2L.

```
Lab 13 - Dictionary/
 ├── lab13_main.cpp
 ├── lab13_Filesystem.hpp
 └── StringUtil.hpp
```

**Penalties:** The following items will negatively impact your score.

- **Program doesn't build**
  Your program should always build. Programs turned in that don't build will automatically receive a grade of 50%. Additionally, I build your code from the command line in Linux; Your code should be portable. Certain features are allowed in Visual Studio or Windows but don't work for all compilers.
  Avoid: `#pragma once`, `system("pause");`, ignoring filename cases

- **Missing source files**
  If your .hpp or .cpp files are missing, they cannot be graded and will result in a 0%. Always double-check to make sure you're submitting all your files.

- **Visual Studio files**
  I don't want these. I ONLY want your .hpp and .cpp files. I won't count

off if you turn it in, but do me a favor (and help me grade quickly) by not turning in junk files.

- **Zipped files**
  I don't want this. Just submit your source files. I won't count off if you turn in a zip, but when I download assignments they're already zipped so it just makes more work for me.

# Contents

## 1.2 About: Trees

## 1.3 About: Recursion

## 1.4 Lab specifications

A file system of a computer is usually represented in a tree format, where each node represents a folder or a file. If a node is a folder, it may have children. Files do not have children.

In this lab, we will work with trees to model a file system. A Filesystem and File class are already created, and you will add functionality.

### 1.4.1 File

You do not have to update anything with File. The File has a constructor and a destructor - the destructor is responsible for deleting any of its children. A File also has a vector of pointers to child Files, and a pointer to its parent File. It also has a boolean to describe if it is a directory or not, and it has a name string.

### 1.4.2 Filesystem

The file system has two member variables: `m_ptrRoot`, a File pointer, and `m_nodeCount`. Similarly to a linked list, we only need to store a pointer to one node; to get to something else deeper in the tree, we will need to traverse the tree. However, trees are not linear like a linked list is.

The Filesystem has a function CreateFilesystem - You do not need to understand this function, it just creates files and folders that we will work with. There is a recursive GetPath function, which will take a file pointer, and traverse through that File's parents to build a string that contains the path to that File.

You will need to implement the recursive Find and GetFile functions. Both of these functions are overloaded; the ones under `// public interfaces` are for outside functions to call. Within these functions, they call the recursive version, passing in the `m_ptrRoot` as the starting point.

### 1.4.3 Functions

**File* Find( const string& filename, File* ptrLookAt )**

For this function, we look for a File object, whose name is filename. If that File is found, we return it as File*. If it is not found, we return nullptr.

Conceptually, first the root is searched. If the filename isn't found at the root, then we look at each of the root's children: We do the same search in each SUBTREE of that child.

Therefore, you will need to iterate over the list of childrenPtrs of the ptrLookAt current File, and run the same routine on each of those children.

If one of the calls covering the child subtree found the file we're looking for, continue passing that found File* "upward".

**Terminating cases:** We will either be returning nullptr if we've searched all the children of a node, down to the leaf level, and have not found the item. Or, we will be returning the file itself when we find it, at any level of the tree.

**Recursive cases:** We will need to recurse (call Find with new arguments) for all child Files of the ptrLookAt current File. If the child call gives us something that isn't a nullptr, we return that upward.

### File* GetFile( list path, const string& filename, File* current )

For this function, a path is passed in. At that path, the filename passed in is expected to be found. We start at `m_ptrRoot` on the first call.

If the ROOT is the only item in the path, this is the directory we search, and look at all the children to try to find a file with the same name as filename.

However, if we are not at the current path, we need to call GetFile again with a new path and a new current pointer. We will need to move forward one directory (as given by the first item, or front() item, of our path), and call the function again.

You will need to utilize `path.pop_front()` and `path.size()` and `path.front()` in this function.

**Terminating cases:** If there is nothing left in our path list, then we are at the directory that our file should be in.

Check all the children of the current File. If any of the files match the filename, return that child. Otherwise, return nullptr - because we are in the correct directory, but that file is not found.

**Recursive cases:** If we are not at the correct path (i.e., path.size() is not 0), then we need to go to the next folder.

Search through the current File's children, looking for a folder that matches the path.front()'s name. Once you've found the directory, call GetFile again with the updated arguments.

(Again, if we're at a directory and can't find a subfolder with the name given in the path, return nullptr because the search failed.)