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.

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: Recursion

Recursion is about breaking up a problem into steps that are the same, and following these steps over and over until some **terminating case** is triggered, causing the recursion to stop and begin back-tracking through all the levels.

Sometimes, you can solve the same sorts of problems with a simple loop, but other times, recursion is the most effective way to solve a problem. For example, we will work with a mock "file system" in this lab, and traverse through the filesystem using recursion.

1.2.1 Simple Examples of Recursion

Let's start off small. Say we want to calculate $n!$, n-factorial.

With a simple loop, we could calculate it like this:

```
1 int CalculateFactorial_Loop( int value )
2 {
3     int result = value;
4     for ( int i = value-1; i > 0; i-- )
5     {
6         result *= i;
7     }
```

```
8       return result;
9   }
```

But if we are breaking it down into a recursive problem, we think about it like this...

Recursing down:

What is $n!$	It is $(n-1)! \times n$.
OK, but what is $(n-1)!$	It is $(n-2)! \times (n-1)$.
OK, but what is $(n-2)!$	It is $(n-3)! \times (n-2)$.
...	
OK, but what is $4!$	It is $3! \times 4$.
OK, but what is $3!$	It is $2! \times 3$.
OK, but what is $2!$	It is $1! \times 2$.
OK, but what is $1!$	It is 1.

Oh, I have a number now! I can solve the rest!!

Returning up:

What is $2!$	It is $1! \times 2$, which is $1 \times 2 = 2$.
What is $3!$	It is $2! \times 3$, which is $2 \times 3 = 6$.
What is $4!$	It is $3! \times 4$, which is $6 \times 4 = 24$.
What is $5!$	It is $4! \times 5$, which is $24 \times 5 = 120$.

And so on, until we reach n, and return finally, once and for all.

Code-wise, this function would look like this:

```
1   int CalculateFactorial( int n )
2   {
3       if ( n == 1 )
4       {
5           // Terminating case
6           return 1;
7       }
8
9       return n * CalculateFactorial( n-1 );
10  }
```

Similarly, a function to figure out the nth term of a Fibonacci sequence looks like this:

```
1  int Fib( int n )
2  {
3      if ( n == 0 )
4      {
5          return 0;
6      }
7
8      if ( n == 1 )
9      {
10         return 1;
11     }
12
13     return Fib( n-1 ) + Fib( n-2 );
14 }
```

(Where the Fibonacci sequence is 1 1 2 3 5 8 13, ...)

1.2.2 Filesystem

In this lab, we will be traversing a filesystem. Let's think about Windows for a minute, and how a path might look like this:

`C:\Users\Bob\Games\PickinSticks.exe`

At each level of the directory, there may be 0 or more folders, as well as 0 or more files. Let's say we wanted to display all items in the Games folder, as well as any subfolders. How could we do this?

Let's say we have a directory like this:

- Games\
  - Jazz\
    - Jazz.exe
    - Jazz.cfg
  - Keen\
    - Keen1
      - Keen1.exe
    - Keen4
      - Keen4.exe
    - Register.bat
  - PickinSticks.exe

And we are coming up with a function to list out all files and folders in the Games directory...

1. Item 1 in Games: "Jazz". This is a folder; display its contents.
   (a) Item 1 in Jazz: "Jazz.exe". This is not a folder.
   (b) Item 2 in Jazz: "Jazz.cfg". This is not a folder.
2. Item 2 in Games, display its name. "Keen". This is a folder; display its contents.
   (a) Item 1 in Keen: "Keen1". This is a folder; display its contents.
      i. Item 1 in Keen1: "Keen1.exe". This is not a folder.
   (b) Item 2 in Keen: "Keen4". This is a folder; display its contents.
      i. Item 1 in Keen4: "Keen4.exe". This is not a folder.
   (c) Item 3 in Keen: "Register.bat"
3. Item 3 in Games: "PickinSticks.exe". This is not a folder.
4. Done.

1.3 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.

1.3.1 File

```
1  struct File
2  {
3      File();
4      ~File();
5
6      vector<File*> childrenPtrs;
7      File* ptrParent;
8
9      bool isDirectory; // traversible if true
10     string name;
11 };
```

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.

1.3.2 Filesystem

```
1  class Filesystem
2  {
3  public:
4      // public interfaces
5      File* Find( const string& filename );
6
7      File* GetFile( const list<string>& path,
8          const string& filename );
9
10 private:
11     /* You implement these */
12     File* Find( const string& filename,
13         File* ptrLookAt );
14
15     File* GetFile( list<string> path,
16         const string& filename, File* current );
17
18 public:
19     // Constructor / destructor
20     Filesystem();
21     ~Filesystem();
22
23     string GetPath( File* ptrFile );
24     int GetSize();  // Get size
25
26 private:
27     // Create data for our lab
28     void CreateFilesystem();
29
30 private:
31     File* m_ptrRoot;
32     int m_nodeCount;
33 };
```

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.

1.3.3 Functions to implement

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.

Step-by-step:

1. Terminating cases:
   (a) If the name of the File being pointed to by ptrLookAt matches the filename parameter, then return ptrLookAt.
   (b) If the size of the childrenPtrs is 0 (`ptrLookAt->childrenPtrs.size() == 0`), then return nullptr.
2. Recursion: If we haven't hit an endpoint (found the file, or no file exists), then investigate the next set of children.

3. Iterate through all the children with a for-loop like this:
   `for ( unsigned int i = 0; i < ptrLookAt->childrenPtrs.size(); i++ )`

   Within the for loop...

   (a) Create a File pointer.
   (b) Call the Find function again, assigning its return value to your File pointer. Pass in the same `filename` parameter, but pass in the current child as the next node to check.
   `File* ret = Find( filename, ptrLookAt->childrenPtrs[i] );`
   (c) If the return value is NOT nullptr, then return the result.
   (d) Otherwise, continue looping.

4. Once the end of the for loop is reached, it means that the file was not found. In this case, return nullptr.

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.)

Step-by-step:

1. First, pop the front-most item from the `path`; this is the root folder.

   TERMINATING CASE - DONE TRAVERSING THE FOLDERS:

2. If the path size `if ( path.size() == 0 )` is now 0, that means we've reached the end of the folder list, and now we look for the file.

   (a) Loop through all the child pointers of the `current` File pointer...
   `for ( unsigned int i = 0; i < current->childrenPtrs.size(); i++ )`

      i. If the name of the current child pointer is equal to the `filename`, `if ( current->childrenPtrs[i]->name == filename )` then return this child.
      `return current->childrenPtrs[i];`

   (b) If the end of the loop has been reached, that means we couldn't find the file at this path. Therefore, return nullptr.

   RECURSION - TRAVERSING THE FOLDERS:

3. Loop through all the child pointers to look for the next folder to traverse...
   `for ( unsigned int i = 0; i < current->childrenPtrs.size(); i++ )`

   (a) Check if this item's name matches the next folder in the path.
   `if ( current->childrenPtrs[i]->name == path.front() )`

      i. Call the GetFile function recursively, with the path, the filename, and the current pointer's children. Return this function call.
      `return GetFile( path, filename, current->childrenPtrs[i] );`

4. At the end of the for loop, if no next folder to traverse is found, then return nullptr.

1.4 Example output

```
 / ROOT
 / ROOT / file-0.txt
 / ROOT / file-1.txt
 / ROOT / file-2.txt
 / ROOT / folder-0
 / ROOT / folder-1
 / ROOT / folder-0 / folder-D
 / ROOT / file-0.txt / file-A.txt
 / ROOT / file-1.txt / file-B.txt
 / ROOT / file-2.txt / file-C.txt
 / ROOT / folder-0 / folder-D / file-D.txt
 / ROOT / folder-1 / folder-E
 / ROOT / file-0.txt / file-A.txt
 / ROOT / file-1.txt / file-B.txt
 / ROOT / file-2.txt / file-C.txt
 / ROOT / folder-1 / folder-E / file-E.txt
Total nodes: 16

Find file1.txt ...
File: file-1.txt, Parent: ROOT,
     Path:  / ROOT / file-1.txt

Find file2.txt ...
File: file-E.txt, Parent: folder-E,
     Path:  / ROOT / folder-1 / folder-E / file-E.txt

Get file ROOT / folder-0 / folder-D / file-D.txt ...
Front: ROOT
GetFile, 2, file-D.txt, ROOT
Front: folder-0
GetFile, 1, file-D.txt, folder-0
Front: folder-D
GetFile, 0, file-D.txt, folder-D
file-D.txt
```