ECE 361 Fall 2020
Homework #3
Binary Trees, File I/O, and makefiles

THIS ASSIGNMENT SHOULD BE SUBMITTED TO D2L BY 10:00 PM ON **SAT, 14-NOV-2020** (YES, THIS IS A WEEK LATER THAN THE DATE IN THE SYLLABUS, BUT I FEEL THE EXTRA TIME IS WARRANTED). THE ASSIGNMENT WILL BE GRADED AND IS WORTH 100 POINTS. IT IS EASIEST TO GRADE, AND I BELIEVE EASIEST FOR YOU SUBMIT YOUR SOURCE CODE FILES AS TEXT FILES INSTEAD OF TRYING TO CUT/PASTE YOUR CODE INTO A `.doc` OR `.docx` FILE.

WE WILL BE USING GITHUB CLASSROOM FOR THIS ASSIGNMENT SO PLEASE PUSH YOUR FINAL CODE AND TRANSCRIPT TO YOUR HOMEWORK #3 REPOSITORY. WE'D ALSO LIKE YOU TO SUBMIT AN ARCHIVE OF THE REPOSITORY TO YOUR D2L HOMEWORK #3 DROPBOX.

NOTE: This assignment is notably more complex than the first two assignments. As I mentioned in class before the midterm, I am providing less detail on the assignments going forward, leaving more of the design and implementation up to you. I imagine this makes some (maybe many) of you nervous, but I have confidence in your abilities.

Using Abstract Data Types

This question gives you an opportunity to implement and experiment with Binary trees and with parsing text files. Our target application builds and operates on a database of MLS (Major League Soccer) team statistics.

The background

The MLS 2020 season is/was (hopefully) a singular, never to be repeated, occurrence caused by the Covid-19 pandemic. MLS was one of the first professional sports league to adapt to the "new normal" starting with a closed *MLS is Back* tournament held in Orlando, FL in the late Spring, followed by a shortened "closed gate" (i.e. no fans in the stadium) schedule culminating in playoffs that start on November 20 and end around Mid-December. Our local team, the Portland Timbers, won the *MLS is Back* tournament and have qualified for the playoffs, as have our arch rivals, the Seattle Sounders. The 3rd team in the Cascadia cup rivalry, the Vancouver Whitecaps, are currently below the playoff line, but have a very good chance of making the playoffs. Since the US/Canadian border is closed due to Covid-19, the Whitecaps have been playing their "home" games in Portland. In fact, the Timbers and the Whitecaps played Sunday night with the Timbers emerging victorious in a hard fought 1-0 win.

The application

The application you are going to build reads a file containing the win/loss/games statistics for each of the 26 MLS teams, parses the file and adds the information to a binary tree using an API that you are going to write. The application traverses the trees (one for each conference) to display the "Tables" (i.e. the standings) for teams in the Eastern and Western conferences. The current "Tables" can be found at https://www.mlssoccer.com/standings. The Tables are updated after every game is completed so the data file included with the release is already obsolete.

As a Binary Tree ADT, this one is pretty simple. There is no searching and no deleting from the tree – only creating the tree, inserting nodes into the tree, and doing an in-order traversal of the tree to produce the standings. The application will also determine and "announce" the Shield winner. The math is simple:

```
PPG = #Games played / ((#wins * 3pts/win) + (#losses * 0 pts/loss) +
     (#ties * 1 pt/tie))
```

Your application doesn't need to do the math. PPG is one of the pieces of information provided in the file. The PPG for each team is the "data" that is used to insert the node into the tree. The top 8 (of 12) teams in the Western Conference and the top 10 (of 14) teams in the Eastern Conference qualify for the playoffs. The team with the highest PPG in both conferences wins "the Shield".

The database file

Information for the database will come from a text file called `MLS2020.txt` that your program will read. You should place a copy of this file in your working directory for the assignment. I have included a "helper" function that will parse a line from the file and place the fields in a `struct` that your code can operate on. Each record in the file has the following fields with each field separated by a comma (.csv format):

- A `char[] array` containing the name of the team. The full name can be scanned by `fscanf()` by using the format field [^,].
- An `int` specifying the conference the team plays in (0 for Eastern Conference, 1 for Western Conference)
- An `int` specifying the points the team earned
- A `float` specifying the PPG the team earned
- An `int` specifying the number of games the team played
- An `int` specifying the number of wins the team earned
- An `int` specifying the number of losses the team earned
- An `int` specifying the number of ties (draws) the team earned

For example, The line for the Portland Timbers is:

```
Portland Timbers,1,35,1.75,20,10,5,5
```

Which means that the Portland Timbers play in the Western Conference and have earned 35 points (10 wins*3 + 5 losses*0 + 5 ties*1). Their PPG is 35 / 20 games played = 1.75. Lines that start with `//` are comments and are ignored.

I have provided the function (in `MLSapp_Helpers.c`):

```
TeamInfoPtr_t parseTeamInfo(const char *buf, TeamInfoPtr_t info_ptr)
```

That takes as input a char string containing a line of text from the file. The function returns a pointer to a `TeamInfo_t struct` and can be used to copy the results from the parsed line back to a `TeamInfo struct` called `info_ptr` in the function it is called from.

The `TeamInfo_t` is defined in `MLStree.h`.

The binary tree ADT

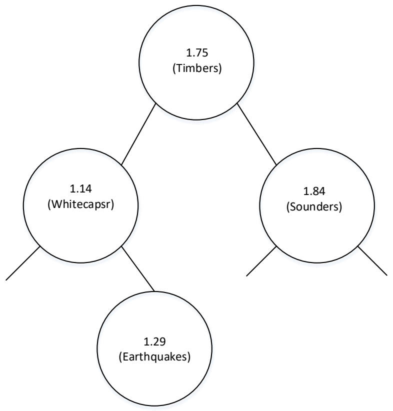
The application uses one instance of your binary tree ADT for each conference. Because of this you need to build your binary tree ADT in a way that can independently support more than one tree. You've seen this before in the linked list and queue ADT's. Those ADT's support a `create()` function that returns a "handle" (pointer) to the instance of the ADT. This is the way the MLStree is architected. You will write a create function with the following prototype:

- `MLStreePtr_t createMLStree(MLSconf_t conf, int numTeams, int numPlayoffTeams)` – Creates a new `MLStree` instance. Returns an `MLStreePtr_t` (a pointer to the `MLStree struct`) as the "handle" for the tree.

Other than creating a binary tree instance, your ADT needs to implement two other functions:

- `TreeNodePtr_t insert(MLStreePtr_t tree, TeamInfo_t info)` – This function inserts a new `TeamInfo_t struct` into the correct place in `tree`. The tree is ordered based on the `PPG` field in the `TeamInto_t struct`.

The following figure illustrates the tree as it would exist if the teams were entered in this order: Timbers, Sounders, Whitecaps, and the San Jose Earthquakes. Yes, the insertion order does matter – worst case would be populating a tree with a sorted list…random is better.



- `void printStandings(MLStreePtr_t tree)` – This function does an in-order traversal of the tree and prints the standings. An in-order traversal of a properly populated binary tree results in a sorted output. Many algorithms assume an ascending sort, but in this case your algorithm should do a descending sort since you want the highest PPG to be listed first. The difference in the algorithm is that the right subtree is visited before the left subtree. For the tree pictured above the nodes would be visited in this order:

- Start at root and descend the right tree until NULL. Print the standings line for the root of the NULL node which would be the Sounders.
- Descend the left subtree from the Sounders node which is NULL so nothing is printed.
- Return to the Sounder's parent which is the Timbers and print its standing line.
- Descend the left subtree of the Timbers node to the Whitecaps subtree and descend its right subtree. Print the Earthquake's standings line.
- Descend the left subtree from the Whitecaps which is NULL and print the Whitecap's standings line.

There are examples of both recursive and non-recursive in-order traversals in the `sample_code` folder.

Although getting an in-order traversal algorithm based on the PPG field of the `TeamInfo_t` is not a trivial task(although you can find sample code that almost meets the needs of your particular tree structure), the challenge for this function is to produce a nicely formatted standings line. For example, your `printStandings()` function must also indicate whether a team qualifies for the playoffs. In my application my `printStandings()` function keeps track of the number of nodes that have been printed and compares it to the number of playoff teams (saved in the `NumPlayoffTeams` in the `MLStree_t struct`. While you don't have to mimic the standings line my application prints (see one of the results files), it does show the information I am expecting to see.

The hw 3 application

Pseudocode for the HW3 application is as follows:

1. Create two instances of your binary tree ADT, one for the Eastern conference and one for the Western conference. The Eastern conference has 14 teams with the top 10 teams qualifying for the playoffs. The Western conference has 12 teams with the top 8 teams qualifying for the playoffs.
2. Build/populate an MLStree for both conferences by opening the `MLS2020.txt` file and parsing each line in the file. I called this function `buildMLStrees()` in my application. Insert a new node into the correct (by conference) tree, populating the `TeamInfo_t struct` using the `parseTeamInfo()` function provided in `MLSapp_helpers.c`. Keep track of the team in each conference with the best `PPG`; you will need that information to determine the Shield winner. You may find the sample code in `sample_code/read_string.c` helpful in designing your `buildMLStrees()` function.
3. Use the `printStandings()` function you wrote to display the Tables for each conference.
4. Determine the Shield winner. This is the team with the highest `PPG` of all of the MLS teams. Display the winner with a suitable congratulatory message. Print the statistics of the Shield winner using the `printTeamInfo()` function in `MLSapp_Helpers.c` A transcript of my hw3 output is in the file `test_results/hw3_test_results.txt`.

Task list:

- Download the hw3 release from D2L and/or accept the Homework #3 assignment in GitHub classroom.
- Read through this write-up (which is likely to be overwhelming at first brush) and remember Dr Hall's Panic rule. It's OK to panic for a few minutes but after that take a deep breath, get out your highlighter and decide on a design and implementation plan.

  It is OK to collaborate with your colleagues on this assignment – in fact, I encourage you to do so. There is a lot to absorb and a goodly amount of design work to do before you start writing code. Remember, though, there is a big difference between collaborating and copying. I encourage the former and strongly discourage the latter.

- Study the source code in the `starter_code` and `sample_code` directories. Look at the test results in the `test_results` folder. Review Karumanchi and the lecture notes for binary trees. Search online for examples. I'm a big fan of geeksforgeeks.org, tutorialspoint.com and, of course, stackoverflow.com. There is a wealth of useful information and sample code available online. Acknowledge the source if you copy more than a few functions.
- Design and implement your binary tree ADT, paying attention to the requirements for each of the three functions you need to write. The `MLStree.h` file and the `MLStree_starter.c` file provide code you can build on if you'd like. Use the `_VERBOSE_` debug flag in `MLStree.h` to add debug messages to your code. Even though my code "mostly worked" the first time, I made use of _VERBOSE_ `printf()` messages to root cause the problems I did have. This code is not difficult, but Ready-Fire-Aim is not the best approach to follow. Design your code, then write it. Include /** function preambles, structured blocks of code and meaningful signal names. Build some error checking logic into the program up front. It's easier than adding it later or doing without.
- Test your binary tree ADT with the `test_MLStree.c` starter code I gave you. I did most of my debug using that. Once my MLStree code was stable it didn't take a huge amount of work to create and debug my `hw3.c`
- Design and implement your `buildMLStrees()` function. This is the key function in your hw3 application. It needs to read the data, parse it, insert nodes into the proper tree and keep track of the stats you need to determine the Shield winner.
- Implement your final `main()`. It needs to create and populate the trees, print the standings for both conferences and "announce" the Shield winner.
- Create a `makefile` to manage the build process for your application
- Integrate the functionality and debug and test your final application. If you've done a good job on design and have tested the individual components of your applicaition this final may be easier than you think.
- Build your application using the `make file` you wrote. Execute your program one last time. Include a transcript (log) that shows your `makefile` at work to build the

application. The transcript should also demonstrate that your program is working correctly. Identify your transcript with your user name or some other identification.

- Push all of your source code (`.c` and `.h`) files and transcript(s) to your GitHub repository for the assignment. Upload an archive of your repository to your D2L Homework #3 dropbox.

LOOKING AHEAD: ORGANIZE, SAVE, AND DOCUMENT YOUR WORK. WE MAY REVISIT THIS SCENARIO IN HOMEWORK #4