I. Problem Description

This problem involves taking data of 14 Premier League soccer teams, and telling the user based on their input whether or not the two teams they inputted are rivals. The data that the program starts with is comma separated values. In 2018, fans of the 20 teams in the top flight of English soccer were surveyed and asked who was their "most disliked" team. For the purposes of this project, teams are considered rivals if fans of both teams said that the other team was one of their top 5 most disliked adversaries. So for example if fans of team A say team B is one of their most disliked teams, but fans of team B do not rank team A as one of their most disliked teams, they are *not* considered to be rivals. The data was trimmed to include only teams that had a rivalry within the 20 teams of the Premier League. Furthermore, rival teams are only included in the data if they are in the Premier League (as opposed to being in a lower division). The starting data is the `Vector<string> teams`. Each entry is a different team followed by the names of all relevant teams that they considered to be rivals.

[0] Manchester United,Chelsea,Manchester City,Liverpool,Arsenal
[1] Arsenal,Chelsea,Liverpool,Tottenham,Manchester United
[2] Manchester City,Tottenham,Liverpool,Chelsea,Arsenal,Manchester United

Manchester City and Manchester United would be considered to be rivals, but Manchester City and Arsenal would not (because Arsenal fans did not rank Manchester City as one of their top five disliked). The student is given the data with each line as a new string entry in the vector `teams`. The user framework is given to the student, and the goal is to write two functions for either solution type. The first solution uses a map, the second uses a graph. For the map solution the goal is to write `fillMap` and `areRivals`, and the code below is provided

```
// provided function takes care of the framework operations
void teamSearchMap(string filepath) {
   // The provided code opens the file with the given name
   // and then reads the lines of that file into a vector.
   ifstream in;
   Vector<string> teams;

   if (openFile(in, filepath)) {
       readEntireFile(in, teams);
   }
   cout << "Read file " << filepath << ", "
        << teams.size() << " teams found." << endl;

   // uses student function to populate the map
   Map<string, Vector<string>> filledMap = fillMap(teams);

   // lists the teams so the user can see their options
   for (string key : filledMap.keys()) {
       cout << key << endl;
```

```
    }

    string name = "";
    while (true) {
        cout << endl;

        // asks the user for input of two teams
        string name = getLine("Enter two team names (i.e.
'Arsenal,Chelsea' (no spaces separating teams)) (RETURN to quit):");
        // ends if user hits RETURN
        if (name == "") {
            cout << "All done!";
            break;
        }
        // splits the two inputted teams into a vector
        Vector<string> currentTeams = stringSplit(name, ',');

        // uses the student function to check the map if the two teams
are rivals
        if (areRivals(filledMap, currentTeams)) {
            cout << currentTeams[0] << " and " << currentTeams[1] << "
are rivals!" << endl;
        } else {
            cout << currentTeams[0] << " and " << currentTeams[1] << "
are not rivals." << endl;
        }
    }
}
```

The second solution uses a graph, so the above code is slightly modified to use list nodes instead. Student makes these functions:

```
// fillMap takes in a vector with each of the lines from the teams
file and creates a map with each team as the key and each
// of its fans' most disliked teams (rivals) in a vector as the value
Map<string, Vector<string>> fillMap(Vector<string> teams) {
    // TODO
    return {};
}
```

```
// areRivals takes in the filledMap, and the user input vector of the
two current teams. Returns true if both teams consider the
// other to be a rival, false otherwise.
bool areRivals(Map<string, Vector<string>> filledMap, Vector<string>
currentTeams) {
      // TODO
      return false;
}
```

And these variations for the graph solution:

```
// fillGraph takes in a vector with each of the lines from the teams
file and creates a graph with each team as node connected to the
// root and its children are every other team which it considers to be
a rival
ListNode* fillGraph(Vector<string> teams) {
    // TODO
   return ;
}

// areRivals takes in the filled graph root, and the user input vector
of the two current teams. Returns true if both teams consider the
// other to be a rival, false otherwise.
bool areRivals(ListNode* root, Vector<string> currentTeams) {
   // TODO
   return false;
}
```

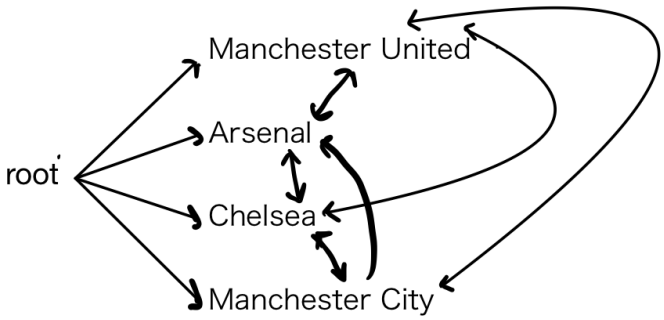
The student is also given the standard ListNode constructor except the `next` field is a vector of values that can be added to. This can be found in the `listnode.h` header file. Finally the graph solution gives three provided helper functions:

```
// getPositionOnRoot takes in the root of the team tree and a team
name, and returns the index of the team on the roots "next" vector
int getPositionOnRoot(ListNode* root, string teamName) {
   // loop over every child of the root
   for (int n = 0; n < root->next.size(); n++) {
       // if the team name matches the node we are on, return the
index
       if (teamName == root->next[n]->data) {
           return n;
       }
   }
   error("team not found!");
}
```

```
// isChild takes in a parent ListNode* and a child string, and returns
true if the string is one of the children of the parent
// node, false otherwise
bool isChild(ListNode* parent, string child) {
    // loop over every child node of the given parent node
    for (ListNode* teamNode : parent->next) {
        // if the child team is found, return true
        if (teamNode->data == child) return true;
    }
    return false;
}

// deallocates the graph to avoid leaks in test cases
void deallocateGraph(ListNode* root) {
    for (ListNode* node : root->next) {
        delete node;
    }
    delete root;
}
```

The goal is to create a graph that connects all teams to a common root, and then create a pointer that connects every rival to each other. A condensed version is below.



The pointers from Man. United to Man. City go in two directions, so they are rivals. Man. City has a pointer to Arsenal, but Arsenal does not have one to Man. City. So they are not rivals.

II. Solutions and Test Cases

Map solution:

```
// fillMap takes in a vector with each of the lines from the teams
file and creates a map with each team as the key and each
// of its fans' most disliked teams (rivals) in a vector as the value
Map<string, Vector<string>> fillMap(Vector<string> teams) {
   Map<string, Vector<string>> filledMap;

   for (string line : teams) {
       // break each team in the line into a new item in the vector
       Vector<string> workingTeams = stringSplit(line, ',');
       // make the name of the first team the key, and a vector of the
rest of the teams (the rivals) is the value
       filledMap[workingTeams[0]] = workingTeams.subList(1);
   }

   return filledMap;
}

// areRivals takes in the filledMap, and the user input vector of the
two current teams. Returns true if both teams consider the
// other to be a rival, false otherwise.
bool areRivals(Map<string, Vector<string>> filledMap, Vector<string>
currentTeams) {
   if (filledMap[currentTeams[0]].contains(currentTeams[1])
           && filledMap[currentTeams[1]].contains(currentTeams[0])) {
       return true;
   } else {
       return false;
   }
}

PROVIDED_TEST("fillMap and are rivals tests") {
   Vector<string> testVector = {"red,blue,green",
                                "blue,green",
                                "yellow,red,green",
                                "green,blue,red,yellow"};

   Vector<string> entry1 = {"red", "green"};
   Vector<string> entry2 = {"red", "blue"};
   Vector<string> entry3 = {"red", "yellow"};
   Vector<string> entry4 = {"yellow", "green"};
   Vector<string> entry5 = {"green", "green"};
```

```
    Map<string, Vector<string>> testMap = fillMap(testVector);

    EXPECT(areRivals(testMap, entry1));
    EXPECT(!areRivals(testMap, entry2));
    EXPECT(!areRivals(testMap, entry3));
    EXPECT(areRivals(testMap, entry4));
    EXPECT(!areRivals(testMap, entry5));

}

PROVIDED_TEST("fillMap and areRivals tests 2") {
    Vector<string> testVector = {"Stanford,Berkeley,UCLA",
                                 "Berkeley,Stanford,UCLA",
                                 "UCLA,Stanford,Berkeley",
                                 "Oregon"};

    Vector<string> entry1 = {"Stanford", "Berkeley"};
    Vector<string> entry2 = {"Stanford", "UCLA"};
    Vector<string> entry3 = {"Berkeley", "UCLA"};
    Vector<string> entry4 = {"Oregon", "Stanford"};

    Map<string, Vector<string>> testMap = fillMap(testVector);

    EXPECT(areRivals(testMap, entry1));
    EXPECT(areRivals(testMap, entry2));
    EXPECT(areRivals(testMap, entry3));
    EXPECT(!areRivals(testMap, entry4));
}

PROVIDED_TEST("fillMap and areRivals tests 3") {
    Vector<string> testVector = {"Raptors",
                                 "Lakers",
                                 "Jazz",
                                 "Hawks"};

    Vector<string> entry1 = {"Raptors", "Jazz"};
    Vector<string> entry2 = {"Raptors", "Hawks"};
    Vector<string> entry3 = {"Raptors", "Lakers"};
    Vector<string> entry4 = {"Jazz", "Lakers"};

    Map<string, Vector<string>> testMap = fillMap(testVector);

    EXPECT(!areRivals(testMap, entry1));
    EXPECT(!areRivals(testMap, entry2));
    EXPECT(!areRivals(testMap, entry3));
```

```
    EXPECT(!areRivals(testMap, entry4));
}
```

This solution is unique in the sense that it makes use of a map. Where N is the number of teams. The Big-O of `fillMap` is $O(N^2)$. For `areRivals` it is O(N). We simply built up a map by using string split and built up a map that mapped the first team in the vector to the sublist beginning with the second team. For `areRivals`, we check both map entries to see if the teams are rivals of each other. This approach is the simpler of the two, and it should be more efficient as well because it has far fewer lines/operations. The only drawback is it is much less complex so there is less problem-solving for the student, but in the real world this is actually a good thing.

```
// fillGraph takes in a vector with each of the lines from the teams
file and creates a graph with each team as node connected to the
// root and its children are every other team which it considers to be
a rival
ListNode* fillGraph(Vector<string> teams) {
    ListNode* root = new ListNode("root");

    // populate the root with all fourteen teams first, each as its own
node. We do this first so that we have all to nodes make rival
    // connections to!
    for (string line : teams) {
        // find the first comma and use the substring function to find
the first team name in the line
        string headTeam = line.substr(0, stringIndexOf(line, ","));
        // create new node that contains the team name and attach it to
the root
        ListNode* newTeam = new ListNode(headTeam);
        root->next.add(newTeam);
    }

    // instead of using the getPositionOnRoot function each time, we
simply create a variable called headTeamPosition which we set
    // to zero, and then increase by one each line we move down. We
know that this will be the correct index.
    int headTeamPosition = 0;
    for (string line : teams) {
        // break each team in the line into a new item in the vector
        Vector<string> workingTeams = stringSplit(line, ',');
        string headTeam = workingTeams[0];

        // loop over every RIVAL now. So i starts at one, and increases
until we have hit every rival on the line
        for (int i = 1; i < workingTeams.size(); i++) {
            string currentRival = workingTeams[i];
```

```
            // use given function to get the index of the rivalTeam on
the root's next vector
            int rivalPosition = getPositionOnRoot(root, currentRival);
            // add a pointer that points from the headTeam to the
rivalTeam

            root->next[headTeamPosition]->next.add(root->next[rivalPos
            ition]);
        }
        // once we have added every rival for one team, increase the
index and go to the next line
        headTeamPosition++;
    }
    return root;
}

// areRivals takes in the filled graph root, and the user input vector
of the two current teams. Returns true if both teams consider the
// other to be a rival, false otherwise.
bool areRivals(ListNode* root, Vector<string> currentTeams) {
    // use given function to find the position of the user inputted
teams on the root
    int team1Position = getPositionOnRoot(root, currentTeams[0]);
    int team2Position = getPositionOnRoot(root, currentTeams[1]);

    // use the given isChild function to check if the second user
inputted team is a child of the first, and vice-versa. If so,
    // return true.
    if (isChild(root->next[team1Position], currentTeams[1])
            && isChild(root->next[team2Position], currentTeams[0])) {
        return true;
    } else {
        return false;
    }
}

PROVIDED_TEST("fillGraph and areRivals tests") {
    Vector<string> testVector = {"red,blue,green",
                                 "blue,green",
                                 "yellow,red,green",
                                 "green,blue,red,yellow"};

    Vector<string> entry1 = {"red", "green"};
    Vector<string> entry2 = {"red", "blue"};
    Vector<string> entry3 = {"red", "yellow"};
```

```
    Vector<string> entry4 = {"yellow", "green"};
    Vector<string> entry5 = {"green", "green"};

    ListNode* testRoot = fillGraph(testVector);

    EXPECT(areRivals(testRoot, entry1));
    EXPECT(!areRivals(testRoot, entry2));
    EXPECT(!areRivals(testRoot, entry3));
    EXPECT(areRivals(testRoot, entry4));
    EXPECT(!areRivals(testRoot, entry5));

    deallocateGraph(testRoot);
}

PROVIDED_TEST("fillGraph and areRivals tests 2") {
    Vector<string> testVector = {"Stanford,Berkeley,UCLA",
                                 "Berkeley,Stanford,UCLA",
                                 "UCLA,Stanford,Berkeley",
                                 "Oregon"};

    Vector<string> entry1 = {"Stanford", "Berkeley"};
    Vector<string> entry2 = {"Stanford", "UCLA"};
    Vector<string> entry3 = {"Berkeley", "UCLA"};
    Vector<string> entry4 = {"Oregon", "Stanford"};

    ListNode* testRoot = fillGraph(testVector);

    EXPECT(areRivals(testRoot, entry1));
    EXPECT(areRivals(testRoot, entry2));
    EXPECT(areRivals(testRoot, entry3));
    EXPECT(!areRivals(testRoot, entry4));

    deallocateGraph(testRoot);
}

PROVIDED_TEST("fillGraph and areRivals tests 3") {
    Vector<string> testVector = {"Raptors",
                                 "Lakers",
                                 "Jazz",
                                 "Hawks"};

    Vector<string> entry1 = {"Raptors", "Jazz"};
    Vector<string> entry2 = {"Raptors", "Hawks"};
    Vector<string> entry3 = {"Raptors", "Lakers"};
    Vector<string> entry4 = {"Jazz", "Lakers"};
```

```
    ListNode* testRoot = fillGraph(testVector);

    EXPECT(!areRivals(testRoot, entry1));
    EXPECT(!areRivals(testRoot, entry2));
    EXPECT(!areRivals(testRoot, entry3));
    EXPECT(!areRivals(testRoot, entry4));

    deallocateGraph(testRoot);
}
```

This solution is different because it uses a graph built from list nodes. The Big-O of `fillGraph` is $O(N^2)$. For `areRivals` it is O(N). This solution is interesting because it modifies the linked list format that was introduced in class to create a simple graph, which was not explored in depth. For `fillGraph`, the approach is to first create a node for every team, then add a pointer that goes from the root to each team. Then we go back and create pointers from every team to its now existing rival nodes. `areRivals` makes use of the provided `isChild` function to check the user inputted nodes to see if they are rivals. An interesting quirk of this solution is that since the graph nodes are constructed such that they have a `next` vector which contains every *listnode* they point to, once we add the teams to the root, we have no way of finding them again strictly with a team's name. So we have to use the provided `getPositionOnRoot` function, which is an O(N) operation. If it were not for this inefficiency, the `fillGraph` function would be O(N) rather than $O(N^2)$.
For testing, I tried a bunch of varied scenarios, simply on a smaller scale. I built up a vector like the one the student would receive, and then used the student functions to build up a map/graph and then used areRivals to test that the entries were accurate for both solution paths. I also wanted to make sure that we would be able to deal with a scenario in which there are no rivals (third test case). This could be a common oversight.

III. Problem Motivation

This problem is complex and a teacher might want to use it because it involves the usage of a lot of ADTs. A strong working knowledge of pointers and vectors is essential for this problem as they are used in some capacity in nearly every operation. The first solution also makes use of maps, and the second solution uses graphs, which are not fully explored in the course. Thus, using the graphs in the problem requires a nuanced understanding of not only the general graph theory, but also the vectors and linked list foundation which makes it up. This is an appropriate challenge because the first one, while simpler, allows a strong understanding of maps, and shows how making the value field of a map into a vector, rather than simply a string, allows one to store multiple values. The second one is challenging because it involves a lot of thinking and deep understanding through the use of nested pointers. Also the student has to take chronology into account; a node has to be first created for every team before one can begin linking them up. I chose this problem because linked lists was a topic that I struggled a lot with, so I decided to upgrade the back-end to allow for numerous children to take it to the next level and demonstrate a nuanced understanding. This touches in some way almost every skill that we learned this quarter, including strings, vectors, ADTS, maps, Big-O, pointers, linked lists, trees, and graphs. I also

chose this because it connects to soccer and the history/culture of the sport, which I have been a lot more interested in to stay entertained during the pandemic.

IV. Concept mastery, common misconceptions

This project touches on a variety of the learning goals of CS106B. First, it should get students excited to use programming to solve real-world problems, as the interface that the solution creates is pretty fun, and many students are interested in sports, so this is a relevant connection. Second, the student learns to recognize and understand common abstractions, as they have to understand the functioning of the provided functions as well as their own code. Finally, this problem involves breaking down a complex problem, as they have to code two separate functions that should be able to play nicely together.

The first issue that came up for me was figuring how to wire all of the team nodes together correctly. I realized that I had to first loop over the teams vector to create a node for each team, and then start connecting them together. After I had created a node for every team, another issue that I had was finding the right nodes to connect to each other. As previously stated, we can't simply search for a list node using its name. So I had to make use of the `getPositionOnRoot` function, and then link them up using the resulting index in the `root->next` vector.

Another misconception that might come is determining whether teams are rivals. Just because fans of one team considers another team to be rivals (has the latter team in their map entry or has a pointer to them), does not necessarily mean they are rivals. *Both* teams have to consider the other team to be a rival. I tried to be explicit about this in the project description. Students might also get caught up in the interface, so I tried to be clear that they don't have to worry about how any of the provided code works. They just need to implement the student functions as described. Another bug might be not properly using pointers to edit the `next` vector. To correctly add a child to a node, for example the root node, one needs to write: `root->next.add(ListNodeGoesHere)`. This might be challenging for students to identify at first, as they have to realize how to access the vector, but also that it is a vector that they can use all the standard vector operations to work on. There is no given function to add children either. So `root.add(ListNodeGoesHere)` and `root->ListNodeGoesHere` are both not valid.

Finally, for the map solution, another misconception might be, after we have split up the team names in a single line, to add this entire vector to the map value/graph. This would result in a team being listed as its own rival in the first case, and each team having a pointer that goes to itself for the second solution. We only want to consider every team from the second team on the line to the end to be a rival. There is a test case in the first test for each solution which tests to see if "green" and "green" are rivals to account for this. It should always return false.