Computer Science 352 Spring 2023
Programming Assignment 7
Due Friday 3/31/2023 by 7pm

In this PA you will be implementing a game similar to the popular 20 questions game (See http://20q.net/). In 20 questions, the 1 human player thinks of an object from everyday life. Then, the computer gets to ask the player up-to 20 questions about the object to try to narrow down what the object is. After asking questions, the computer must try to guess the object. In order to achieve this, the computer needs to have a database of objects, information about those objects, and some kind of data structure to allow the computer to narrow down the answer after each question is asked. There are multiple ways that this could be implemented. In this assignment, you will do so with a decision tree structure (https://en.wikipedia.org/wiki/Decision_tree).

For this assignment specifically, the game will be called “Tree-Questions” instead of “20-Questions”. You should divide the source up into three files. tq.h + tq.c will have the structs and functions for creating data structures to store the information in. game.c should have a main function and the primary game logic.

```
pa7
├── tq.h
├── tq.c
├── game.c
└── makefile
```

The **makefile** should have three rules: one for **tq.o**, one for **game**, and one for **clean**. The **tq.o** rule should compile **tq.c** with the required gcc flags and produce a library file named **tq.o** in the current working directory. The **game** rule should compile **game.c** with the required gcc flags and produce an executable named **game** in the current working directory. The **clean** rule should delete the **tq.o** and **game** files from the current working directory.

Input File Format

The Tree-Questions game will get information about what questions to ask, the items, and the answer to each question for each item, from a database file that is formatted similarly to a CSV file. working directory. Here is one example of a file that the program will be expected to read in:

```
10
does it have 4 legs?,does it have wheels?,does it have a motor?,does it have fur?
car,0,1,1,0
fox,1,0,0,1
elephant,1,0,0,0
trailer,0,1,0,0
tarantula,0,0,0,1
mirror,0,0,0,0
work bench,1,1,0,0
standing desk,1,1,1,0
bear,1,0,0,1
lion,1,0,0,1
```

- The first line should be a number, which tells the program how many total items this file has in it. In the example above, there are 10 total items: car, fox, elephant, trailer, tarantula, mirror, work bench, standing desk, bear, lion.
- The second line will have a comma-separated list of the questions that this file has information about. In the example above, there are four questions.
- From the third line on, each line will have information about one item in a comma-separated list. The first element will be the name of the item. For example, the item **car** is on line 3. Following this is a comma separated list of 1s and 0s. A 1 corresponds to a "yes" answer to a question, and a 0 corresponds to "no". For example, for the car row, the numbers are **0,1,1,0**. This means that the answer to the first and fourth questions from row 2 are "no" and the answer to the second and third questions from row 2 are "yes". This makes sense, because a card does NOT have 4 legs and does NOT have fur, but it DOES have four wheels and it DOES have a motor.

Of course, this is just one example. The number of items and questions may vary. The program should support *any* number of items and questions. The game should support any number of questions.

The database file should be provided as the first command-line argument to the program. For example, if the contents of that file were stored in a file named **small.tq** then you should run the game with this file like so:

```
$ ./game small.tq
```

TQ Files

The primary data structure this library will need to work with is a decision tree. In the case of this assignment, a decision tree is basically a binary search tree slightly modified to work better for the Tree-Questions game. This structure will require two structs:

```
typedef struct TQDecisionTreeNode {
  char text[50];
  int num_answers;
  char** answers;
  struct TQDecisionTreeNode* yes;
  struct TQDecisionTreeNode* no;
} TQDecisionTreeNode;

typedef struct TQDecisionTree {
  TQDecisionTreeNode* root;
} TQDecisionTree;
```

A **TQDecisionTree** is very simple. It just contains a pointer to the root node of the tree, or NULL if it is an empty tree. Simple as that. The more complex component is the representation of a **TQDecisionTreeNode**. Each node of the tree represents one question *or* group of answers. Instead of referring to child nodes in the tree via left/right pointers, we use yes/no pointers instead. This indicates that all ancestor answers in the **yes** subtree are ones that have a "yes" answer to this node's question. All ancestor answers in the **no** subtree are ones that have a "no" answer to this node's question. Every node will either represent a question or a pool of answers. If it is a question node, the **text** field should be populated with the question, **num_answers** should be 0, and **answers** should be **NULL**. If it instead represents a pool of answers, the **text** field should not be populated, **answers** should be a pointer to an array of pointers to the logical answer(s) at this point in the tree, and **num_answers** should be given the correct count based on how many answers there are. Only leaf nodes

will have answers. All other nodes will be question nodes. There are several functions that you should implement associated with this data structure. The first function:

```
void TQ_print_tree(TQDecisionTree* root);
```

This function should print out a text representation of a decision tree. Examples of the output of this will be shown throughout this spec. The next function to implement is:

```
TQDecisionTree* TQ_build_tree(char* file_name);
```

This function should accept a **char*** which represents the name of the file to load the data from. The function should open up the file, iterate through the data within (perhaps creating some intermediate data structure to store it in) and populate a decision tree. During this process, the answers should not be added to the tree, just the question structure and leaf nodes with no questions or answers. It should return a **TQDecisionTree** that has been populated with all of the questions. Every level of this totally-full tree should have the same question. One particular path through the tree from root to leaf represents one possible sequence of the user answers yes/no to the presented questions. For example, if we created a tree based on the data from the **small.tq** file and then called the **TQ_print_tree** function, we should get:

```
[does it have 4 legs?]
-y-> [does it have wheels?]
    -y-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n->
        -n-> [does it have fur?]
            -y->
            -n->
    -n-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n->
        -n-> [does it have fur?]
            -y->
            -n->
-n-> [does it have wheels?]
    -y-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n->
        -n-> [does it have fur?]
            -y->
            -n->
    -n-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n->
        -n-> [does it have fur?]
            -y->
            -n->
```

Pay attention to how this output is formatted. Your program must use the same formatting. Every level of the tree has the same question in all nodes. For example, the root node is **[does it have 4 legs?]** (the first question from the input file). Both of its children are questions **[does it have wheels?]** (the second question) one for the “yes” response to the previous question, and one for a “no” response. Finally you must also implement:

```
void TQ_populate_tree(TQDecisionTree* tree, char* file_name);
```

This should take a previously-built tree and a **char*** with the name of the data file. The function should populate the tree with the answers within the correct leaf nodes based on the data from the file, following the correct yes/no paths through the tree. This function should not *add* any new nodes to the tree. Rather, it should only add possible answers to the leaf nodes of the tree that is passed into it. For example, if we had already built the tree shown before, and then called this function with the **small.tq** file to add in answers, the print function should now display the same tree but with answers at the leaf nodes:

```
[does it have 4 legs?]
-y-> [does it have wheels?]
    -y-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n-> | standing desk |
        -n-> [does it have fur?]
            -y->
            -n-> | work bench |
    -n-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n->
        -n-> [does it have fur?]
            -y-> | fox | bear | lion |
            -n-> | elephant |
-n-> [does it have wheels?]
    -y-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n-> | car |
        -n-> [does it have fur?]
            -y->
            -n-> | trailer |
    -n-> [does it have a motor?]
        -y-> [does it have fur?]
            -y->
            -n->
        -n-> [does it have fur?]
            -y-> | tarantula |
            -n-> | mirror |
```

You also need to implement:

```
void TQ_free_tree(TQDecisionTree* tree);
```

This function should free all memory associated with this tree.

You can (and perhaps should) use other intermediate data structure(s) to store the information from the input file. I do so in my solution, and you are welcome to also. Though, you are not told an exact way you must do this, so the implementation detail there is up to you. You are welcome to include helper functions within the **tq.c**, but these four functions must have prototypes in the header and be implemented in the .c source file.

The construction of your tree will be tested both by printing out your tree structure, and by actually playing the game, so you should ensure these tree-related functions work properly.

Implementing the Tree-Questions Game

After you get the data structures and data loading working, you need to implement the NO-Questions game in the **game.c** file. To play the game, the user will run the **game** executable and provide the name of the data file:

```
$ ./game small.tq
```

The program should load in the data, build + populate the decision tree, and then use this structure to control the playing of the game. The program should start by asking the user the question at the root of the tree, and then continue asking questions following the yes / no route through the tree, depending on the answers that the user provides to each question. Eventually, a leaf node will be reached that does not have a question, has both the yes and no child pointers set to null, and will have 0 or more possible answers stored via the **`char** answers`** pointer array. When this leaf is reached, the program should iterate through the answers in-order, asking the user if it is the correct object. Once the user tells the program that it is, it can end. If the program exhausts all possible answers at that node without a yes response, the program should give up and then exit. Several example runs are shown in the section below:

```
$ ./game small.tq
does it have 4 legs? (y/n)
y
does it have wheels? (y/n)
n
does it have a motor? (y/n)
n
does it have fur? (y/n)
y
is it a fox? (y/n)
n
is it a bear? (y/n)
y
I guessed it!
```

```
$ ./game small.tq
does it have 4 legs? (y/n)
n
does it have wheels? (y/n)
y
does it have a motor? (y/n)
y
does it have fur? (y/n)
n
is it a car? (y/n)
n
You got me :)


$ ./game small.tq
does it have 4 legs? (y/n)
y
does it have wheels? (y/n)
n
does it have a motor? (y/n)
n
does it have fur? (y/n)
y
is it a fox? (y/n)
n
is it a bear? (y/n)
n
is it a lion? (y/n)
n
You got me :)
```

Submitting Your PA

Ensure that your program compiles and runs correctly on lectura and that it has no memory leaks or other memory-related errors. You should test the program thoroughly for functionality, as well as for memory leaks / errors with valgrind. Make sure you don't include any extra files other than the ones specified on the first page. After you have completed this PA, double check that the file structure and file / directory names match what is shown in the project overview on the first page.

Once you are ready to submit, zip up the **`pa7`** directory by running:

```
$ zip -r pa7.zip ./pa7
```

Then, turn this file to the PA 7 dropbox on gradescope.

There will be some public test cases on gradescope, and you should ensure you pass those. There will also be some hidden test cases that will not be revealed until after grades are published.