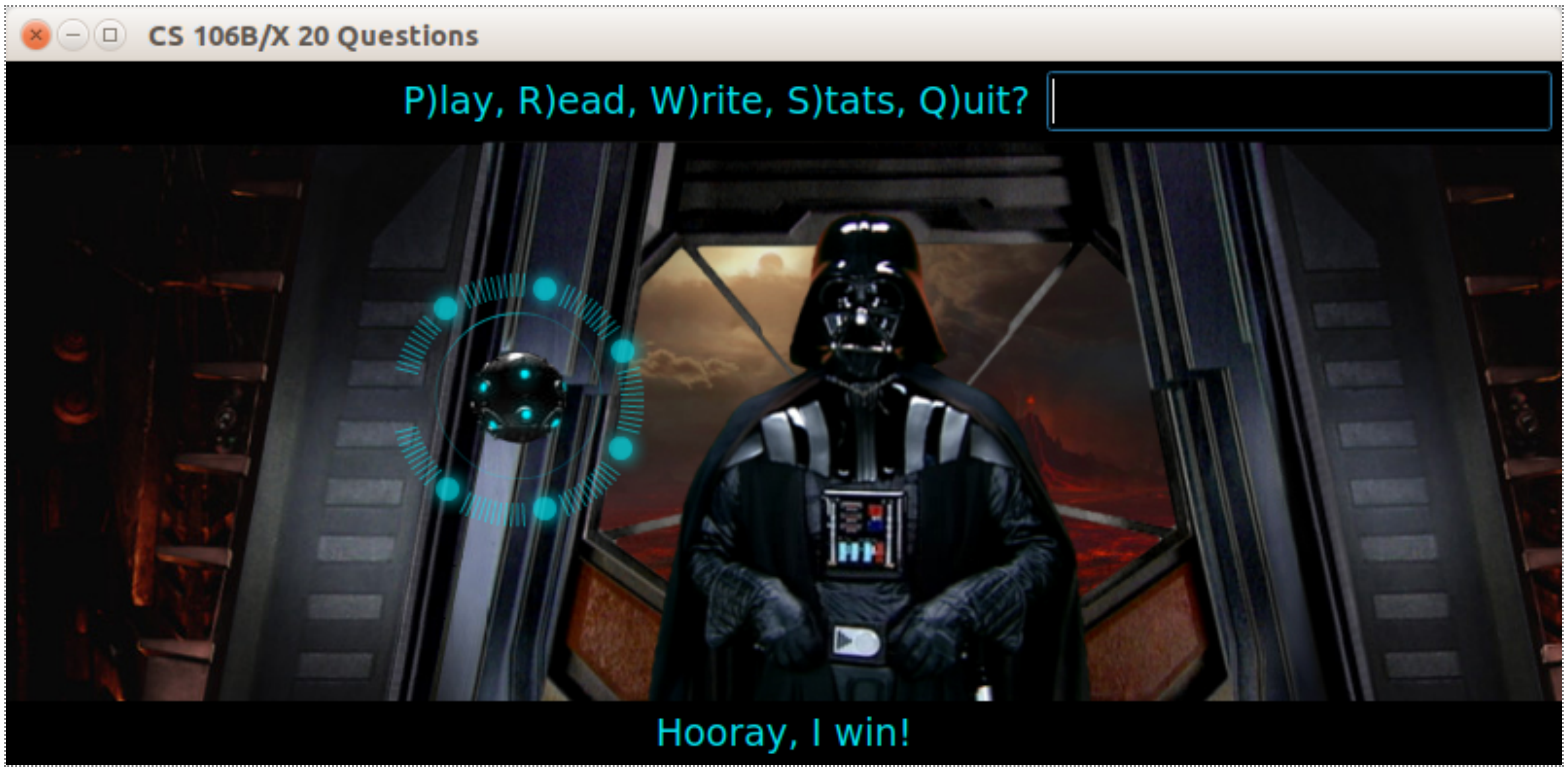
Stanford CS 106B: 20 Questions

Assignment by Marty Stepp. Based on a problem by Stuart Reges of U of Washington.

Links   Description   Implementation   Style



This problem focuses on implementing a game using a binary tree.

This is a **pair assignment**. You are allowed to work individually or work with a single partner. If you work as a pair, **comment both members' names** on top of every submitted code file. Only one of you should submit the assignment; do not turn in two copies.

Links:

starter code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following two files. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified.

- **questiontree.cpp**, the C++ implementation for your class
- **questiontree.h**, the C++ header for your class
- **myquestions.txt**, your custom input file

turn in

When you are finished, submit your assignment using our **Paperless** web system. You can turn in all parts of the assignment together, or turn in each problem separately; it is up to you.

output logs:

- log #1
- log #2
- log #3
- log #4
- log #5

Try pressing **Ctrl+1** .. **Ctrl+4** in the console window to automatically type the user input and compare your output to these logs!

Problem Description:

In this problem you will implement a yes/no guessing game called "20 Questions." Each round of the game begins by you (the human player) thinking of an object. The computer will try to guess your object by asking you a series of **yes-or-no questions**. Eventually the computer will have asked enough questions that it thinks it knows what object you are thinking of, so it will make a final guess about what your object is. If this guess is correct, the computer wins; if not, you win. *(Though the game is called "20 Questions," our game will not limit the game to a max of 20.)* For example, the computer might decide to ask the following series of questions to figure out what the player is thinking of:

```
Is it an animal? y
Can it fly? n
Does it have a tail? y
Does it squeak? n
Are you thinking of: lion? y
Hooray, I win!
```

log of execution, single game, computer wins (partial)

The computer stores its knowledge of questions and answers in a **binary tree** whose nodes represent the game's questions and answers. Every node's data is a string representing the text of the question or answer. A "question" node contains a left "yes" subtree and a right "no" subtree. An "answer" node is a leaf in the tree. The idea is that this tree can be traversed to ask the human player a series of questions.

For example, in the tree below, the computer would begin the game by asking the player, "Is it an animal?" If the player says "yes," the computer goes left to the "yes" subtree and then asks the user, "Can it fly?" If the user had instead said "no," the computer would go right to the "no" subtree and then ask the user, "Does it have wheels?"

This pattern continues until the game reaches a leaf "answer" node. Upon reaching an answer node, the computer asks whether that answer is the correct answer. If so, the computer wins.

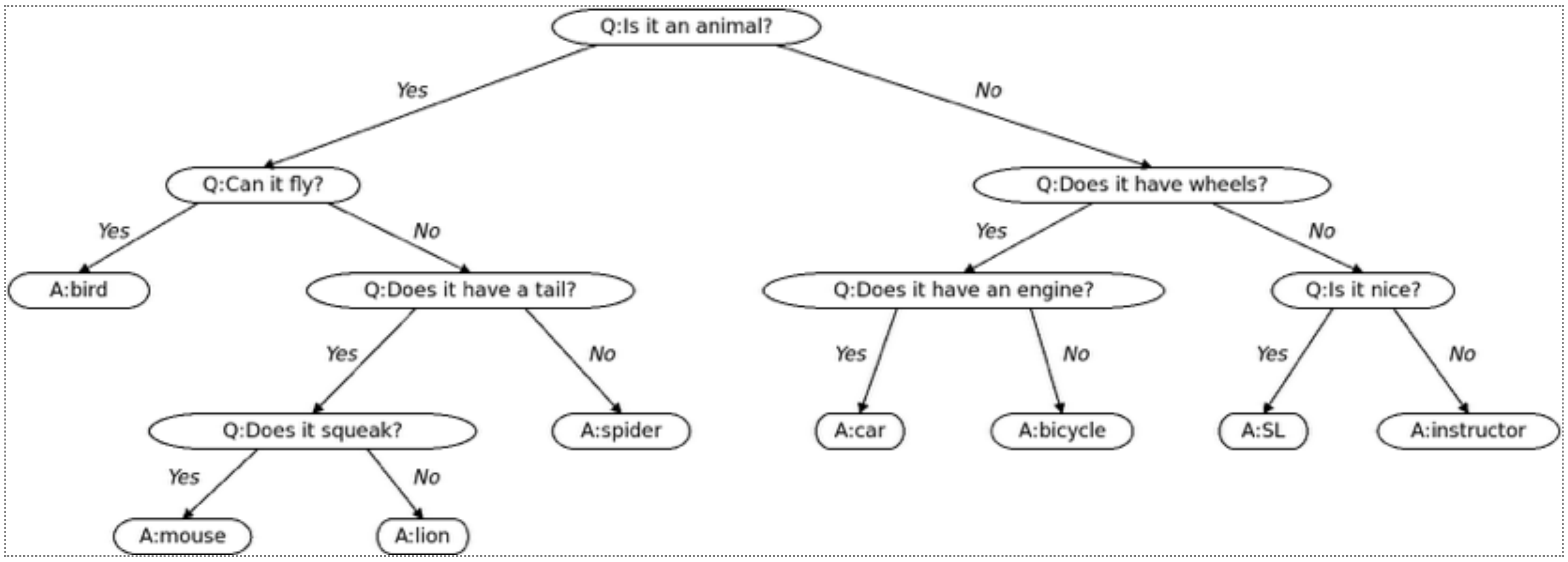


*diagram of binary tree of game questions and answers (**questions.txt**)*

The computer **expands its question tree** by one question each time it loses a game, and therefore it gets better at the game over time. Specifically, if the computer's answer guess is incorrect, you the human player must tell it a new question it can ask to help it do better in future games. For example, suppose in the preceding log that the player was not thinking of a lion, but of an elephant. The game log would look like this:

```
Is it an animal? y
Can it fly? n
Does it have a tail? y
Does it squeak? y
Are you thinking of: lion? n
Drat, I lost. What was your object? elephant
Type a Y/N question to distinguish elephant from lion: Does it have a trunk?
And what is the answer for elephant? yes
```

log of execution, single game, computer loses and grows its tree (partial)

The computer takes the new information from the lost game and uses it to **grow its tree** of questions and answers. You must replace the old incorrect answer node with a **new question node** that has the old incorrect answer and new correct answer as its children. For example, after the game represented by the preceding log, the computer's overall game tree would be the following:

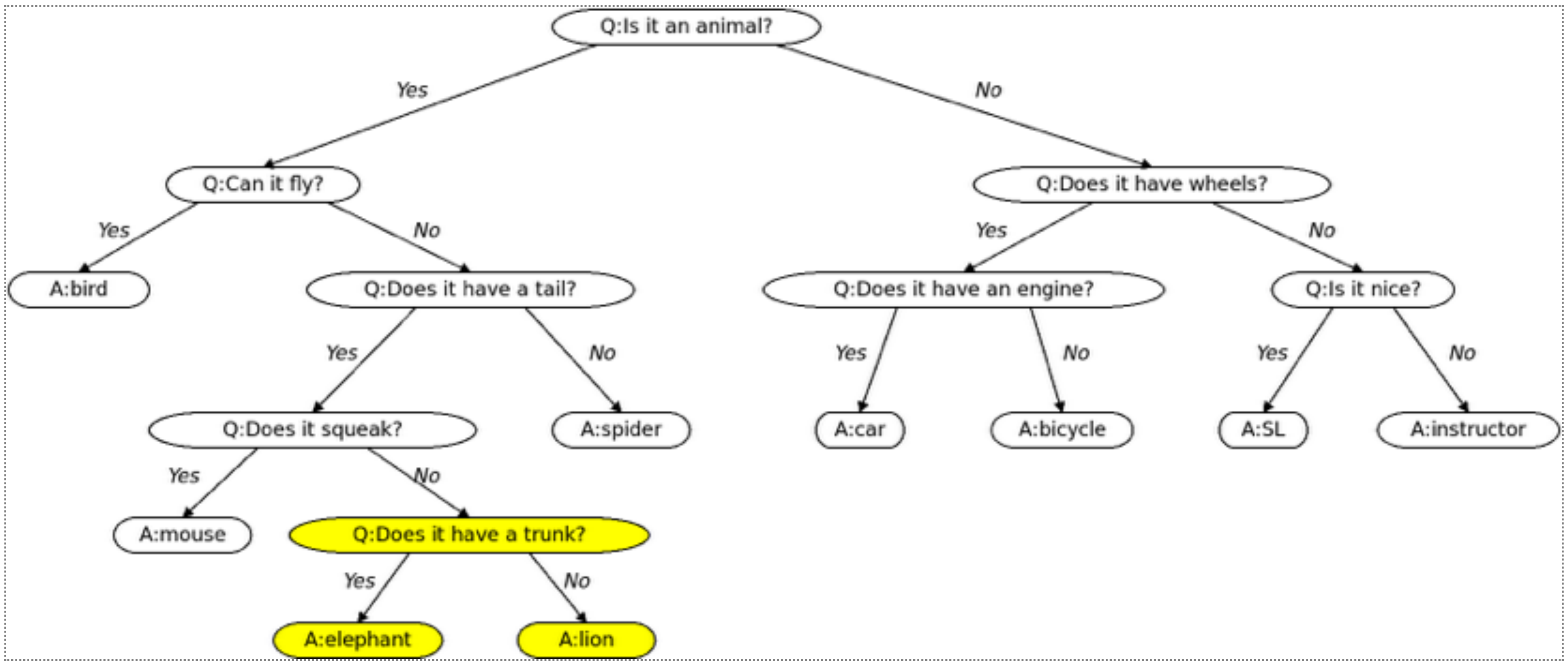


diagram of question tree after "elephant" is added

We provide a main client program that handles user interaction and calls your tree's member functions to play games. There is a simple console user interface, which is better for comparing your output, and also a graphical user interface that makes it so that Darth Vader is trying to guess your item. This is loosely based on an old online game called *"The Sith Sense"* that was essentially a web-based version of this program launched as part of a movie promotional campaign.

You will implement your code in a class named `QuestionTree`. You will use a provided structure `QuestionNode` to represent each individual node of the question tree.

The computer can **save and load** its tree of questions and answers from the disk so that it will retain its improvements over time, even after the program exits and reloads later. Your question tree uses **input files** in the format shown in the example below. The lines occur in a specific order representing a tree-like structure of questions and answers. Each question has two follow-up options that we'll call its *children*, which are either answer guesses or other questions to be asked. The first child of a question is its "yes" child, which comes immediately after it. The second child of a question is its "no" child, which comes immediately after the "yes" child and any of its children. Note that the "yes" and "no" child could be an answer or another question. This tree-like structure is illustrated in the diagram below at right, which shows an indented and annotated version of the file's contents. The order in which the lines are stored represents a **preorder traversal** of the question binary tree.

```
Q:Is it an animal?
Q:Can it fly?
A:bird
Q:Does it have a tail?
Q:Does it squeak?
A:mouse
A:lion
A:spider
Q:Does it have wheels?
Q:Does it have an engine?
A:car
A:bicycle
Q:Is it nice?
A:section leader
A:teacher
```

*input file **questions.txt***

```
Q:Is it an animal?
|
y-- Q:Can it fly?
|    |
|    y-- A:bird
|    |
|    n-- Q:Does it have a tail?
|         |
|         y-- Q:Does it squeak?
|         |    |
|         |    y-- A:mouse
|         |    |
|         |    n-- A:lion
|         |
|         n-- A:spider
|
n-- Q:Does it have wheels?
     |
     y-- Q:Does it have an engine?
     |    |
     |    y-- A:car
     |    |
     |    n-- A:bicycle
     |
     n-- Q:Is it nice?
          |
          y-- A:section leader
          |
          n-- A:teacher
```

*indented version of **questions.txt***
(actual input files will not be indented!)

Implementation Details:

To help you implement your question tree, we provide a file **questionnode.h** that declares a structure called `QuestionNode` to represent one binary tree node. It has the following members. Do not modify **questionnode.h**.

```
struct QuestionNode {
    string data;           // question or answer text
    QuestionNode* yes;     // pointer to "yes" subtree (left)
    QuestionNode* no;      // pointer to "no"  subtree (right)

    // constructor (all parameters are optional)
    QuestionNode(string data = "", QuestionNode* yes = nullptr, QuestionNode* no = nullptr);
};
```

You will implement a class named `QuestionTree` that uses `QuestionNode` structures to build a tree of questions and answers. Your tree class must have the following public members listed in the table below. Make sure to read the next section for important style constraints for this program.

```
QuestionTree()
```

In this constructor you should initialize your question tree. At first the tree contains only a single node storing the answer "computer". In other words, if you were to create a question tree and then immediately begin playing the game, the computer would ask the following question:

```
Are you thinking of: computer?
```

```
~QuestionTree()
```

In this destructor you should free all dynamically allocated memory for your tree and its question nodes.

```
bool playGame(UI& ui)
```

In this member function you should play one complete game of 20 Questions with the user, walking your question tree from its root to an answer leaf node by asking a series of Yes/No questions. This function must be written **recursively** and must not use any loops.

If the computer loses the game, you must prompt for a new question and answer node and modify your question tree as appropriate. That behavior is part of the responsibility of this function. Note that the user can type any text they like for their new question and answer; the user won't type the `Q:` or `A:` prefixes, and the new question they type need not end with a question mark; you should accept whatever text they enter, including a blank string.

After the game is over, the provided `main` program will prompt the user whether or not to play again; this is not part of your `playGame` member function. Leave this functionality to the provided program and don't implement that part yourself.

You should also return a `bool` value indicating the outcome of the game. Return `true` if the computer won and `false` if the human won.

UI parameter: You should not perform user input/output by directly talking to `cout` or `cin`, nor by calling library functions such as `getLine`. Instead, we will pass you a parameter of type `UI` that represents a general user interface for performing user input/output. We are doing this so that your game can function properly whether it is used with a graphical UI, console UI, or any other form of user interaction. The `UI` object passed to your function has the following public member functions that you should use for all input and output when playing your game:

UI member name	Description
`ui.print("message");`	prints a complete line of output (similar to: `cout << "message" << endl;`)
`bool b = ui.readBoolean("prompt");`	prompts the user with a yes/no question and returns the answer as a `bool` value of `true` (yes) or `false` (no) (similar to: `bool b = getYesOrNo("prompt");`)
`string s = ui.readLine("prompt");`	prompts the user with a question and returns the answer as a `string` (similar to: `string s = getLine("prompt");`)

The table above mentions console operations that are similar to the methods of the `UI` parameter. But the `UI` is meant to be general and could be an object that performs input/output using a graphical user interface.

```
int countAnswers() const
```

In this member function you should return the number of nodes in the tree that represent answers (leaves). If there are no answers in the tree, return `0`. (On each call of this member function you should perform a full traversal of the tree to count the answer nodes; do not store a count of such nodes in a private member variable.)

```
int countQuestions() const
```

In this member function you should return the number of nodes in the tree that represent questions (non-leaves). If there are no questions in the tree, return `0`. (On each call of this member function you should perform a full traversal of the tree to count the question nodes; do not store a count of such nodes in a private member variable.)

```
int gamesLost() const
```

In this member function you should return the number of games the computer has lost during this session. Initially 0 games have been played and won. A game is lost when the computer asks the user, "Are you thinking of: ___?" and the user says "no". (You are allowed to use a private member variable to keep count of the number of games lost, and return its value here.)

```
int gamesWon() const
```

In this member function you should return the number of games the computer has won during this session. Initially 0 games have been played and won. A game is won when the computer asks the user, "Are you thinking of: ___?" and the user says "yes". (You are allowed to use a private member variable to keep count of the number of games won, and return its value here.)

```
void answerRange(int& minDepth, int& maxDepth) const
```

In this member function you should traverse your tree and find the minimum and maximum level (also called depth) of any answer node in the tree. The idea is that this represents the minimum and maximum number of questions that the computer might need to ask to guess the user's item. You will store the minimum and maximum depths in the given integer output parameters passed by reference.

For example, if the tree consists of just a single answer node, both `minDepth` and `maxDepth` should be set to `1` at the end of your function. But for the initial tree shown near the start of this document, `minDepth` should be set to `3` (because that is the depth of the answer `"bird"`) and `maxDepth` should be set to `5` (because that is the depth of answers `"mouse"` and `"lion"`).

You should not make any assumptions about what values `minDepth` and `maxDepth` will have prior to a call to your function. It may help you to know that C++ has constants named `INT_MIN` and `INT_MAX` defined in the `<climits>` header. On each call of this member function you should perform a full traversal of the tree to find the min/max depth; do not store such information in private member variables.

```
void readData(istream& input)
```

In this member function you should read question tree data from the given input stream (`istream&`). The file format matches that described earlier in this document. You must read the file's data and turn it into a binary tree of question nodes as drawn earlier in this document, so that subsequent calls to the `playGame` member function will use this data for the series of questions to ask the player. This function must be written **recursively** and must not use any loops.

Assume valid input: You may assume that the input file exists, is readable, and is in the proper format described in this spec, with no blank lines or extraneous/invalid data. Assume that each question will have exactly two children representing its "yes" and "no" options. You may assume that each line of the file begins with a prefix of `"Q:"` or `"A:"`. The prefixes must be accounted for by your code; for example, you may need to strip these prefixes off of each line before asking the questions to the user.

Since you are loading new data and creating a new tree of question nodes, you must free memory for any previous nodes that were in the tree beforehand. Do not leak any memory from this function.

This member function loads the question tree's data of questions and answers, but it does not load any record of the number of games won or lost. Nor does it change your question tree's count of total games won/lost.

```
void writeData(ostream& output)
```

In this member function you should write question tree data to the given output stream (`ostream&`). You must write the data in the same correct file format and order to match that of the input files so that it can be read later by `readData` if desired. Each line must start with either `"Q:"` to indicate a question node or `"A:"` to indicate an answer (a leaf). All characters after these first two should represent the text for that node (the question or answer). The nodes should appear in the order produced by a *preorder traversal* of the tree. Another way of saying this is that if you read an input file using `readData` and then immediately call `writeData`, the newly written file should have exactly the same contents as the original that was read. You may assume that the file is writable. As with other tree-traversal functions, this member function must be written **recursively** and must not use any loops.

This function saves the question tree's current data of questions and answers, but it does not save any record of the number of games won or lost.

```
void mergeData(UI& ui, istream& input)
```

In this member function you should read question tree data from the given input stream (`istream&`) and merge it with your tree's existing data. You will prompt the user to enter a new question that will become the new root of your tree and will become a new question separating the previous root and the root of the data in the given file. For example, suppose your existing data represents a set of questions about animals, but you want to add knowledge about *Star Wars* characters. You could merge the file **starwars.txt** with your tree in the program's console UI. Your method should produce the given `UI` object (as described in the `playGame` section) to produce a user interaction in the following format:

```
Input file name? starwars.txt
Reading question tree ...
Type a Y/N question to distinguish my data from this file: Is it a Star Wars character?
And what is the answer for the new data? y
```

Since the new question is, "Is it a Star Wars character?" and the user specifies "yes" as the answer for the new file's data, the tree after this merge operation would look like the following diagram. If the user had specified "no" as the answer for the new file's data, the previous tree would instead be on the left and the new file's data would be on the right.

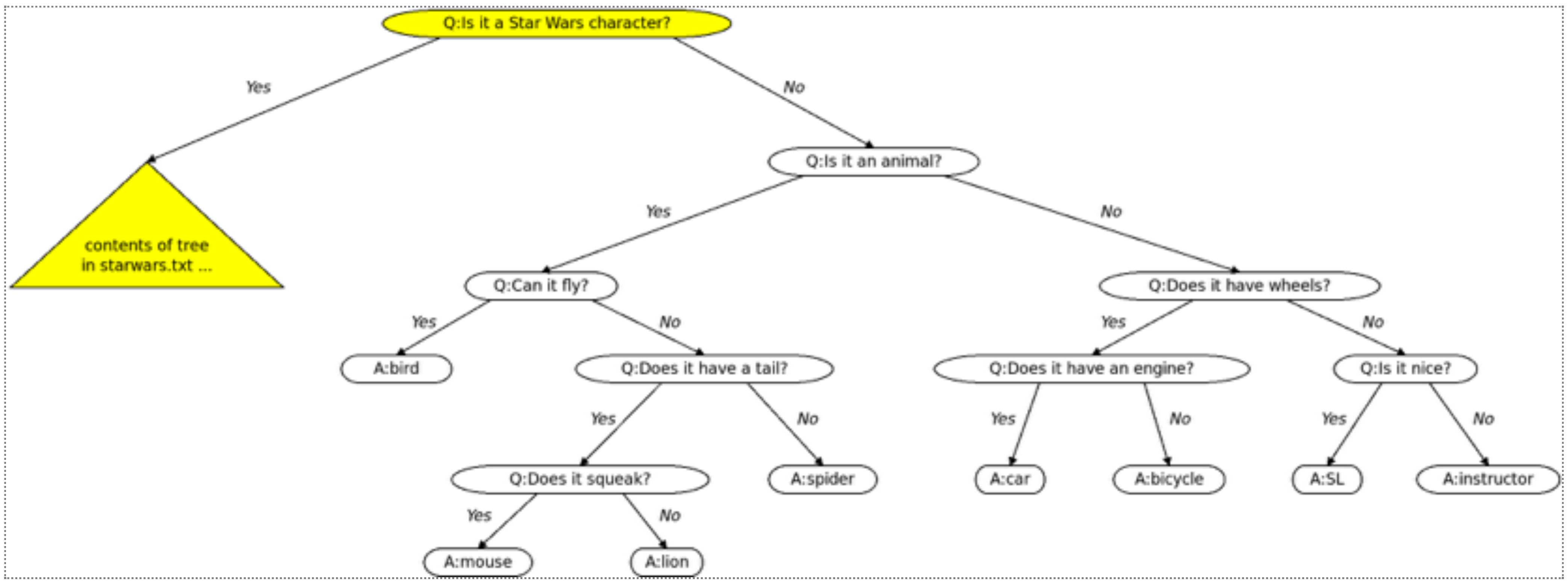


diagram of binary tree after merge with ***starwars.txt***

You may assume that the input file format matches that described earlier in this document. This function must be written **recursively** and must not use any loops. As with other functions, you may *assume valid input.*

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homeworks 1-5 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style contraints specific to this problem:

Private member variables: The only private member variables you should have on this problem are a pointer to the **root** of your question tree, and integer counters for the number of **games won and lost.** Do not declare any other instance variables besides these three. You also should not declare any static or 'global' variables. Local variables declared inside individual functions are allowed.

Binary tree usage: Part of your grade will come from appropriately utilizing binary trees and recursive algorithms to traverse them. Any functions that traverse a binary tree from top to bottom should implement that traversal **recursively**. Absolutely **no loops** are allowed on this program in any part of your code. We will check this particular constraint strictly; no exceptions!

Recursion: Part of your grade will come from appropriately utilizing recursion to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid "arm's length" recursion, which is where the true base case is not found and unnecessary code/logic is stuck into the recursive case. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Helpers: You will find that some of the member functions below do not have the parameters you'll need to implement the desired recursive behavior. In such cases you should not change our required function headers, but you may add extra "helper" functions as needed to help you implement the behavior. Any member functions you add, along with all private member variables, must be declared **private**.

Redundancy: If you find that multiple functions have repeated logic, you should find a way to avoid this redundancy. Perhaps one function should call the other, or both should call a shared helper function.

Collections: Do not use any **collections** (such as `Vector`, `Map`, arrays, etc.) anywhere in your code. Your internal binary tree of nodes is the only structured data storage you should have in this program.

Memory usage: Your code should have no memory leaks. Do not allocate dynamic memory (`new`) needlessly, and free the memory associated with any new objects you allocate internally once they are no longer being used.

Survey: After you turn in the assignment, we would love for you to fill out our anonymous CS 106B homework survey to tell us how much you liked / disliked the assignment, how challenging you found it, how long it took you, etc. This information helps us improve future assignments.

Honor Code Reminder: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.