COMSC-165 Lab 16
Applied Recursion

Modify the **Animal.cpp** program that you wrote in lab 15, so that is "remembers" what it learns. To do so, write the binary tree data structure to a disk file at the end of a session, and read it at the beginning of a new one.

This modification involves the use of recursion. It is a difficult task to write the logic to save a binary tree structure, because of all the branches. But it is fairly easy to formulate logic that saves a single node, thus reducing the problem to the saving of the branches attached to that node -- a sequence that lends itself well to recursion.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab16** folder provided.

LAB 16: An Improved Artificial Intelligence Program `[ Animal.cpp, Animal.dat ]`
STEP 1 of 2: Add Persistence
Enhance lab 15's **Animal.cpp** program, by adding *persistence*. Store the binary tree to a binary disk file in your working folder, e.g. **Animal.dat**, and initialize the tree from that file if it exists. Write two recursive functions -- one for reading a node and its branches and one for writing a node and its branches.

Do *not* ask the user if they want to load the DAT file -- open it, and if successful, load it. Otherwise skip loading it and continue with a 1-node tree.

Since the functions will be recursive, and will also use `fstream` objects, you will have to create those objects in `main` and pass them by reference to the functions (which in turn pass them to themselves recursively). That way, all calls to the functions use the same file I/O objects.

The code for binary tree persistence will be developed in lecture, so consult your class notes for details.

STEP 2 of 2: Add Editing Features
Further enhance lab 15's **Animal.cpp** program, by adding *editing*. Since persistence has been added, any error in spelling or logic are remembered forever! So this creates the need for some editing capability.

Allow the user to enter **Y** or **N** in response to the program's questions, per the normal animal program from lab 15. But *allow* the possible responses **D** and **E**, so that the user can delete or edit the current question (or edit the current animal answer). Do *not* list these options for the use, however! Only "knowledgable users" are supposed to know of these options.

The final program is to include the features listed below:

1. Allow the user to delete a *question*. Call the **`deallocateTree`** function (that you wrote in the lab 15 version) for each of the deleted node's **`yes`** and **`no`** pointers. Then either (1) replace the node's **`text`**

string with "Elephant" for later editing, or (2) prompt for an answer that is to replace the deleted question, and copy that answer into the node's **`text`** string. Set the **`yes`** and **`no`** pointers to 0. Do *NOT* call `deallocateTree(p);` -- we need the p-node. Do *NOT* allow answers to be deleted.

2. Allow the user to edit a *question*. Prompt for a new question, and copy it into the node's **`text`** string. Do not break from the loop that moves the **`p`** pointer through the tree -- instead, simply continue so that the newly edited question appears.

3. Allow the user to edit an *answer*. Prompt for a new answer, and copy it into the node's **`text`** string. Do not break from the loop that moves the **`p`** pointer through the tree -- instead, simply continue so that the newly edited answer appears.

NOTE
Do *not* reset the `p` pointer after an edit or a delete.

*Post **Animal.cpp** and your **Animal.dat** to the COMSC server.*

NOTE: The data file location is the "working folder". Do not use any other file location specification.

GRADING POLICY
Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.
9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.

15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.