Thinking of an Animal

Paul R. Potts

March 2023

Getting Back to BASIC with ANIMAL.BAS

Here's a super-nerdy topic for you: a BASIC computer game I first came across about 45 years ago called "ANIMAL." This appeared in several different books of BASIC computer games edited by David H. Ahl. The book I've got, *BASIC Computer Games: TRS-80 Edition*, says that Animal was "originally developed by Arthur Luehrmann at Dartmouth College" and "subsequently shortened and modified by Nathan Teichholtz at DEC and Steve North at Creative Computing."

It would have been back in 1979 or so when I bought my copy of *BASIC Computer Games: TRS-80 Edition* at my local Radio Shack store in Harborcreek, Pennsylvania. My original copy is long-gone. In the past, I had occasionally searched for copies, but only found copies that were very expensive or in poor condition. But just recently one of my saved searches found a listing for both the original book and the harder-to-find sequel, *BASIC Computer Games, Volume Two: TRS-80 Edition*, for a price I thought was quite reasonable, so I bought them immediately. Here's the first one:

Here's a bit of page 4 showing the start of Animal:

I did a little demo of this game for Sam and Pippin, using a TRS-80 emulator. I was trying to show them what it was like to program early home computers back in the day, warts and all. As a kid, I spent a lot of time typing in programs from magazines and books, and then debugging them, because getting a program typed in perfectly, from a shrunken reproduction of a dot-matrix printout, on the first try, is nearly impossible. Here's what part of the code looks like in the book:

Typing in these programs, and then having to debug them to find all the errors, both in the printed code and introduced when typing them in, forced many hobbyists with no previous background in programming to become experts in BASIC. My TRS-80 BASIC is pretty rusty and my eyes are worse than they were in 1978, so I was slow at this, and was only able to get it partially working during that little demo, although I did manage to fix it later.

If you want to give it a try, you can find the source code here. I know that David Ahl has made at least one version of his source code available to the developer

Figure 1: "BASIC Computer Games: TRS-80 Edition"

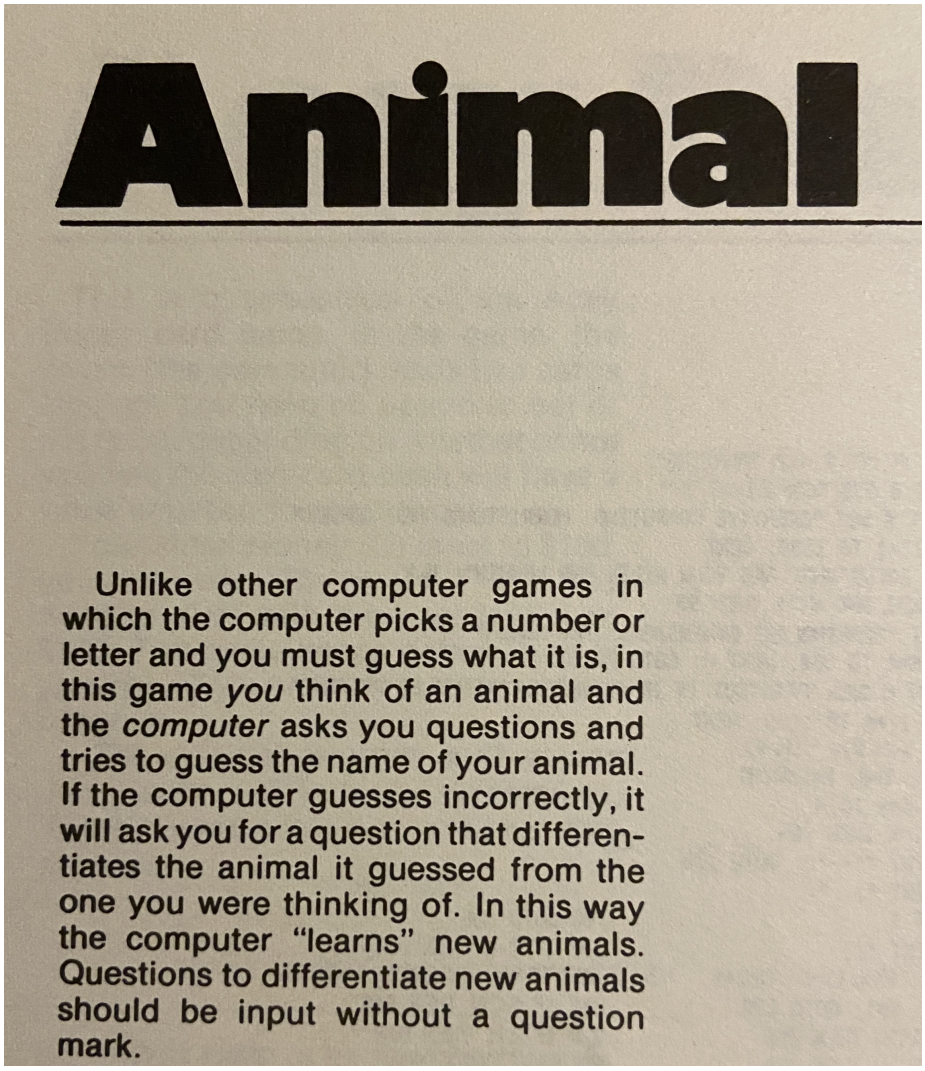
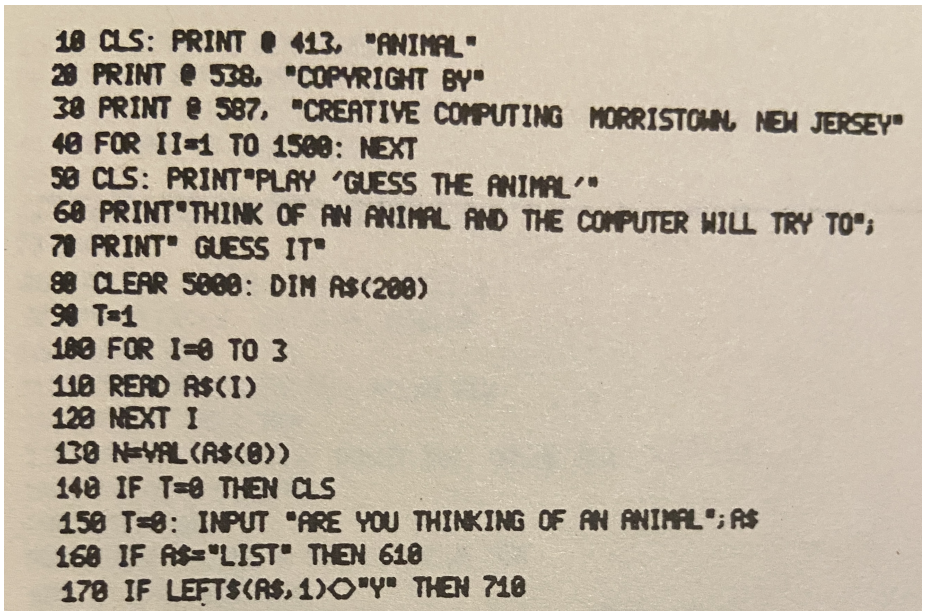


Figure 2: "Page 4: Animal"



```
10 CLS: PRINT @ 413, "ANIMAL"
20 PRINT @ 538, "COPYRIGHT BY"
30 PRINT @ 587, "CREATIVE COMPUTING  MORRISTOWN, NEW JERSEY"
40 FOR II=1 TO 1500: NEXT
50 CLS: PRINT"PLAY 'GUESS THE ANIMAL'"
60 PRINT"THINK OF AN ANIMAL AND THE COMPUTER WILL TRY TO";
70 PRINT" GUESS IT"
80 CLEAR 5000: DIM A$(200)
90 T=1
100 FOR I=0 TO 3
110 READ A$(I)
120 NEXT I
130 N=VAL(A$(0))
140 IF T=0 THEN CLS
150 T=0: INPUT "ARE YOU THINKING OF AN ANIMAL";A$
160 IF A$="LIST" THEN 610
170 IF LEFT$(A$,1)<>"Y" THEN 710
```

Figure 3: "Part of the Animal BASIC Code"

of Vintage BASIC for distribution, but I haven't seen a clear indication that he has placed all the BASIC games, in all their various versions, in the public domain, so I will not include the full source code from the TRS-80 edition in this article. If you want to see the exact version of the code that I've been running, the Internet Archive has it here. Unfortunately, it's a pretty fuzzy scan. Here's a sample session playing the TRS-80 version of the game using an online TRS-80 emulator to give you an idea of why I found the game interesting, and still do. The original TRS-80 only supported uppercase letters, unless you installed a clever third-party product, but for clarity, in this log, I have left the computer's output uppercase, but changed my input into lowercase:

```
PLAY 'GUESS THE ANIMAL'
THINK OF AN ANIMAL AND THE COMPUTER WILL TRY TO GUESS IT
ARE YOU THINKING OF AN ANIMAL? yes
DOES IT SWIM? yes
IS IT A FISH? yes
WHY NOT TRY ANOTHER ANIMAL

ARE YOU THINKING OF AN ANIMAL? yes
DOES IT SWIM? yes
IS IT A FISH? no
THE ANIMAL YOU ARE THINKING OF WAS A? penguin
PLEASE TYPE A QUESTION THAT WOULD DISTINGUISH A PENGUIN FROM A FISH
```

```
? does it look like it is wearing a tuxedo
FOR A PENGUIN THE ANSWER WOULD BE: yes

ARE YOU THINKING OF AN ANIMAL: yes
DOES IT SWIM? yes
DOES IT LOOK LIKE IT IS WEARING A TUXEDO? yes
IS IT A PENGUIN? yes
WHY NOT TRY ANOTHER ANIMAL
```

etc.

How the Animal Program Stores Data

As you can see, the game cleverly builds up a database of animal names along with questions it can use to filter the database. Each question must have a simple "yes" or "no" answer. With each question the game presents, it eliminates animals until there is a single candidate remaining. If the remaining candidate is not the right answer, it prompts the user to enter a new animal into its database.

It's not entirely obvious from reading the source code how the database is stored, as old versions of BASIC required variable names that were quite opaque, and the original game did not contain any comments. So, allow me to clarify it a little bit. The initial database can be seen by typing "LIST" at the "ARE YOU THINKING OF AN ANIMAL?" prompt:

```
ANIMALS I ALREADY KNOW ARE:
FISH    BIRD
```

Looking at the BASIC source code, you can find where these two starter database entries are defined, and this gives the reader a starting point from which to unpack the way the program works:

```
600 DATA"4","/QDOES IT SWIM/Y2/N3/","/AFISH","/ABIRD"
```

We see that the following strings are defined:

```
"4"
"/QDOES IT SWIM/Y2/N3"
"/AFISH"
"/ABIRD"
```

These are used by the program's startup code:

```
80 CLEAR 5000: DIM A$(200)
90 T=1
100 FOR I=0 TO 3
110 READ A$(I)
120 NEXT I
130 N=VAL(A$(0))
```

These strings are put into the array of strings called **A$**. In this early dialect of BASIC, we know that **A$** is an array because of the **DIM** (dimension) statement, which indicates it is a one-dimensional array of 201 elements, and because it is usually accessed using a subscript in parentheses. Why 201 and not 200? That's another oddity of BASIC; the first element of the array is accessed with the subscript 0, but **DIM** allocates one extra element, so that the highest usable array index is the same as the number you specify in the DIM statement. So, I can access elements 0 through 200, or 201 elements in total.

Confusingly, plain old **A$** is also used in this program to hold user input, and it is a single string, not an array.

Also confusingly, the initial value of **N** is placed, in string form, in the first string in the array of strings, at subscript 0. This string is updated when new animals are added to the array. Why is it kept here, instead of in a separate variable? I don't know.

BASIC Considered Painful

It's funny to think back on it, but from my perspective in 2023, this BASIC code from the 1970s reads more like a assembly language, with a few macros to support commonly used functions such as printing. This BASIC programming is not nearly as bad an example of "spaghetti code" as many of the published programs I used to type in, but even so, it's more opaque than most assembly-language programs, because most assembly-language programs are thoroughly commented.

Comments ate up valuable memory and required extra typing, so they were often omitted. Many things were sacrificed in the name of saving memory. And don't get me wrong, I really admire what the folks who developed code for these early home computers were able to accomplish! But let's not pretend that the surviving artifacts of that era are all that easy to understand.

Aside from the **FOR...NEXT** loops, TRS-80 BASIC code doesn't even follow the basic tenets of Structured Programming, much less functional programming or object-oriented programming. Perhaps it's not surprising that I found it relatively easy to start teaching myself Z-80 assembly language programming, alongside BASIC, back in the late 1970s as well; it wasn't necessarily harder to understand than highly condensed BASIC code, and was often easier to understand.

I should point out that there are more modern versions of BASIC that *do* support more modern programming styles. You can get rid of line numbers, and factor your code into functions. The original Microsoft Visual BASIC, which I used back in the early 1990s, was much easier to use, and the programs I wrote in it were much more readable and maintainable.

A Dynamic Data Structure

As the program runs, it builds up its database of questions and guesses. "DOES IT SWIM?" is our initial question, stored without the question mark, with the header "/Q" indicating that it is a question, and followed by indexes for which table entry to look at next for "YES" and "NO" answers. This follows the initial behavior: if you answer "YES" to the first question, the program will ask whether the animal is a fish (index 2), or else whether the animal is a bird (index 3).

Inserting a few lines to dump out the contents of the **A$** array, it becomes clearer what is happening as the program runs:

```
A$ NOW CONTAINS:
4
/QDOES IT SWIM/Y2/N3
/AFISH
ABIRD

ARE YOU THINKING OF AN ANIMAL? yes
DOES IT SWIM? no
IS IT A BIRD? no
THE ANIMAL YOU WERE THINKING OF WAS A ? BAT
PLEASE TYPE A QUESTION THAT WOULD DISTINGUISH A
BAT FROM A BIRD
? does it catch bugs using sonar
FOR A BAT THE ANSWER WOULD BE ? no

A$ NOW CONTAINS
 6
/QDOES IT SWIM/Y2/N3
/AFISH
/QDOES IT CATCH BUGS USING SONAR/Y 5/N 4
/ABIRD
/ABAT

ARE YOU THINKING OF AN ANIMAL? yes
DOES IT SWIM? yes
IS IT A FISH? no
THE ANIMAL YOU WERE THINKING OF WAS A ? seal
PLEASE TYPE A QUESTION THAT WOULD DISTINGUISH A
SEAL FROM A FISH
? does it breathe water
FOR A SEAL THE ANSWER WOULD BE ? no

A$ NOW CONTAINS
 8
```

```
/QDOES IT SWIM/Y2/N3
/QDOES IT BREATHE WATER/N 7/Y 6
/QDOES IT CATCH BUGS USING SONAR/Y 5/N 4
/ABIRD
/ABAT
/AFISH
/ASEAL
```

Note that the program is *rearranging* the elements of **A$**. It's doing this to maintain a tree structure, hidden in the array, that reflects the order in which the questions should be asked. This ensures that it doesn't ask redundant questions. The questions are the inner nodes of the tree, starting from the root, which is always "Does it swim?" The animals to guess are the terminal nodes, or leaves of the tree.

Most of the opaque logic of the program becomes clear when you realize that it handles inserting items into an array, moving items around to make space for the new items. This becomes inefficient as the number of array entries grows larger, but given that we don't have enough memory to add thousands or millions of questions to the tree, inserting a new question probably won't take more than a few seconds, and asking questions should never take very long, because for each question the program just follows an index to read one question out of the array.

Not Quite a Binary Search Tree

Above, I wrote that the data structure is a "tree structure." I did not write that the data structure is a "binary search tree," because it isn't. The nodes are heterogenous, of two types, not homogenous like a typical binary search tree. The pointers from node to node don't point to nodes that belong, in some sorted order, *before* or *after* the current node. The pointers don't indicate an *order* to the keys. Instead, they always represent a fixed "yes" or "no" relationship. We don't maintain an ordering as we insert nodes. Nodes aren't inserted into the tree where they fit, via some ordering, as we descend through the question nodes. Nodes are only inserted when we reach a leaf, containing an animal name, and the animal name doesn't match the animal the user is thinking of.

Because it's not a binary search tree that we can keep balanced as we insert nodes, the tree can become quite unbalanced:

Some nodes are much further down the tree than others, which means that reaching some of the animals requires answering a lot more questions than other animals. In the example above, we only have to answer one question to reach the "Is it a bird?" guess, but we have to answer four questions to reach the "Is it a frog?" guess.

It seems like there ought to be a way to re-balance the tree structure, the way we can re-balance a true binary search tree. But let's imagine we tried picking a new "root" question — for example "Is it amphibious?" To re-balance the tree

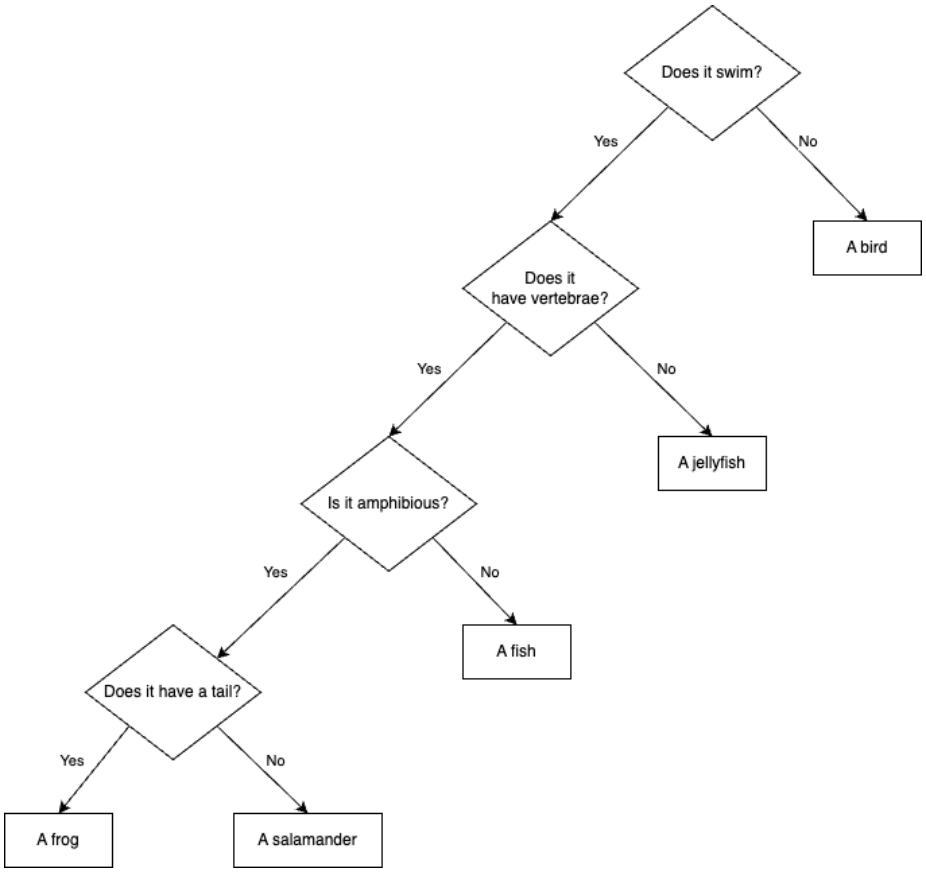


Figure 4: “An Unbalanced Tree Structure”

structure we'd need to hang the existing questions "Does it swim?" and "Does it have vertebrae?" under "Is it amphibious." But to flatten the height of the tree, they'd have to be arranged under the "no" branch of "Is it amphibious?" We'd wind up with a tree of questions question where "Does it swim?" would *only* be asked if "Is it amphibious?" is answered "no," which breaks the less-specific to more-specific question ordering. If we added a feature that re-balances the tree structure, allowing that ordering to be lost, the program's sequence of questions and guesses could become pretty nonsensical. For example, we can imagine a case where answering "no" to a question that asks whether your swimming animal has gills could lead you to a question about whether it flies south for the winter, while answering "yes" might lead the program to ask you if the animal is a dog.

For now, the only way to optimize the program's behavior is to add questions initially with an eye towards keeping the tree balanced. But I can think of a few enhancements that would make this easier. We could allow the user to clear the database, and then ask the user to enter the first question and the two guesses connected to the first question. This feature might allow the user to try following an existing taxonomy, although modern taxonomies of animals rendered as trees do not form binary trees, so unfortunately cannot be efficiently searched by answering simple yes or no questions. Oh well. This is supposed to be a simple game, not an homework problem in biological taxonomy! I can think of another possible enhancement that might help the user keep the tree structure balanced: the program could allow the user to *replace* existing questions or animal guesses one at a time, rather than inserting an additional question and animal guess only when we hit a leaf node and the guess is incorrect. We could call this an advanced feature, and make it the user's responsibility to maintain a sensible less-specific to more-specific ordering. But this is getting pretty far away from the spirit of the original program; it's not an actual database. A more user-friendly feature might be to add a way to save the tree structure to a file, so we can reload it later. The version of the program in the *BASIC Computer Games* book does not offer this feature, although the text suggests that it could be done by saving the **A$** array.

And Onward to Python

Just to see how hard it would be, I wrote a Python version. I did not attempt to replicate the original program's method of storing a tree structure in an array, because it was completely unnecessary; I can just create objects and build a tree of them, which was not possible to do in old BASIC programs.

For simplicity, I just used hash tables for my objects, not idiomatic Python objects. I could have created classes for the different types of nodes, question nodes, containing questions, with references to left and right child nodes, and guess nodes, which contain the names of animals and are leaves, with no children, but I did not bother. I wanted to avoid the extra boilerplate that is required for object-oriented programming in Python; I like many things about Python, but

not the way it handles OOP.

It only took me a couple of hours to get a program that worked correctly, but I spent a few more hours over the next couple of days tidying up the code, refactoring it into smaller functions, and improving the comments; I wanted the chance to sleep on it and come back to it with fresher eyes.

I first started by writing the routine to print out the tree, and once I had that, the rest was very easy, except for the slightly tricky logic that inserts new questions. The function that inserts a new question into the tree needs to keep track of whether the guess that failed is the "yes" leaf or the "no" leaf of the parent question. In the Python version I handle this by the **insert_new_yes_branch_q_node()** and **insert_new_no_branch_q_node()** functions. In my first version, I used a smaller number of longer functions, with more branching, but I have refactored it so that the functions are very short, to try to make the logic as easy to follow as possible.

Following the behavior of the BASIC program, the root question is "DOES IT SWIM?" and the first two guesses are "a fish" and "a bird." The program does not provide a way to change these starting nodes.

My program is over twice as long as the original BASIC program, if you count lines of text, but a lot of the lines in my program are comments and whitespace. The tiny amounts of memory available in early PCs did not allow much space for comments!

My code is also broken into explicit functions, which old BASIC dialects did not support.

I know that I'd much rather type the Python version into a computer than the BASIC version — and modern tools like Visual Studio Code, which does syntax highlighting and indicate a lot of problems and possible solutions while you're typing the code, makes it far, far easier to debug than the BASIC version. I'm not going to claim that Python is the ultimate language — there are a lot of things about it I don't like, mainly having to do with the lack of strict typing, which leads to failure to catch many errors prior to runtime. But at least I have source-level debugging and can watch variables change as I step through code. Looking back at how I used to have to write and debug programs, modern tools are better in many ways, and kids today *do* have it easier!

You can find this source code on GitHub.

```
"""
A very simple version of the classic Animal guessing game, originally published
in various editions of David H. Ahl's _BASIC Computer Games_ books back in the
1970s.
"""

import sys
```

```
# The original program used a BASIC DATA statement to populate an array of
# strings that serve as tree nodes, and refer to each other by array indices.
# This Python version implements the same initial tree nodes using
# dictionaries. There are two kinds of nodes: question nodes, which always
# appear in the tree with two leaf nodes, and guess nodes, which are the
# leaves.
#
# Following a path through question nodes, the program asks a series of
# questions to narrow down the options, until it reaches a guess node, and
# presents an animal name to the user. If the guess is incorrect, the program
# replaces the original guess node with a new question node that has as its
# leaves the original guess node, and a new guess node created with the animal
# name the user types.
#
# In this way, as the user interacts with the program, the tree will grow. The
# original program never replaces the root node, and the tree is never
# re-balanced to minimize the number of questions it takes to get to a guess.
# "Does it swim?" is always the first question. I have kept that behavior, but
# support for re-balancing the tree would be an interesting future upgrade to
# the program.
g_node_fish = {"A":"a fish"}
g_node_bird = {"A":"a bird"}
q_node_root = {"Q":"Does it swim?",
               "Y":g_node_fish,
               "N":g_node_bird}

# Convenience functions for reading and formatting user input of various kinds.
# The original program didn't do much to clean up user input, so we don't
# either, assuming a cooperative user, but we're not as obsessed with saving
# every possible byte, so we do a few things differently:
# - We keep whatever the user types as the animal name, so you shouldn't see
#   incorrect grammar like "a octopus" unless the user typed it that way.
# - We turn animal names into lowercase, so the computer won't ask "Is it An
#   octopus." This could results in incorrect capitalization for animal names
#   containing proper nouns; for example, if the user types in "a Jack Russell
#   terrier," the guess will be "Is it a jack russell terrier?" The alternative
#   would be some kind of validation of animal names, which is out of scope for
#   this simple program.
# - We make sure that when we request a question from the user, it starts with
#   a capital letter and ends with a question mark, so if the user typed "does
#   it have four wings and fly," the question will be stored as "Does it have
#   four wings and fly?"
def get_one_word_answer() -> str:
    return sys.stdin.readline().strip().lower()
```

```
def is_answer_affirmative(answer_str:str) -> bool:
    return answer_str[0] == 'y'

def is_answer_tree(answer_str:str) -> bool:
    return answer_str[0] == 't'

def get_answer() -> str:
    return sys.stdin.readline().strip()

def get_animal() -> str:
    return get_answer().lower()

def get_question() -> str:
    q_str = get_answer().lower().capitalize()
    if q_str[-1] != '?':
        return q_str + '?'
    else:
        return q_str

def get_q_answer(new_a_str:str) -> bool:
    print('For ' + new_a_str + ', what is the answer?')
    return is_answer_affirmative(get_one_word_answer())

# Make a new guess node with a new animal name.
def make_g_node() -> dict:
    print('What animal were you thinking of? ')
    return {'A':get_animal()}

# Make a new incomplete question node, with a new question to distinguish
# between the correct animal and the animal we incorrectly guessed.
def make_q_node_with_q_only(correct_a_str:str, incorrect_a_str:str) -> dict:
    print('Please type a yes-or-no question that would distinguish ' \
          + correct_a_str + \
          ' from ' + incorrect_a_str + ':')
    return {'Q':get_question(),
            'Y':None,
            'N':None}

# Complete the question node by setting the old and new guess child nodes. To
# do this we need the answer to the question that distinguishes between the
# correct animal and the animal we incorrectly guessed.
def attach_g_nodes(new_q_node:dict, incorrect_g_node:dict,
                   correct_g_node:dict) -> None:
    # Arrange the guess nodes according to whether the new animal should be
    # be reached by a "yes" answer or a "no" answer.
    if get_q_answer(correct_g_node['A']):
```

```
        new_q_node['Y'] = correct_g_node
        new_q_node['N'] = incorrect_g_node
    else:
        new_q_node['Y'] = incorrect_g_node
        new_q_node['N'] = correct_g_node

# Make a new question node, along with its children. This requires asking for
# a new animal name, a new question to distinguish between the correct animal
# and the animal we incorrectly guessed, and the answer to that question.
def make_new_q_node(q_node:dict, g_node:dict) -> dict:
    new_g_node = make_g_node()
    new_q_node = make_q_node_with_q_only(new_g_node['A'], g_node['A'])
    # Complete the question node by setting the old and new guess child nodes.
    attach_g_nodes(new_q_node, g_node, new_g_node)
    # The new question node will be inserted into the tree in place of the old
    # guess node.
    return new_q_node

# Add a new question node to the "yes" branch of an existing question node.
def insert_new_yes_branch_q_node(q_node:dict, g_node:dict):
    new_q_node = make_new_q_node(q_node, g_node)
    # Change the parent's "yes" dictionary entry.
    q_node['Y'] = new_q_node

# Add a new question node to the "no" branch of an existing question node.
def insert_new_no_branch_q_node(q_node:dict, g_node:dict):
    new_q_node = make_new_q_node(q_node, g_node)
    # Change the parent's "no" dictionary entry.
    q_node['N'] = new_q_node

# This function handles a guess node, which is always a leaf node, containing
# an animal name. If the guess is not correct, ask for the correct animal name
# and a question we can use in the future to distinguish between the two
# animals. This is how the game "learns."
def play_g_node(q_node:dict, g_node:dict, followed_yes_path:bool) -> None:
    print("Is it " + g_node['A'] + '?')
    if is_answer_affirmative(get_one_word_answer()):
        print('Great! Try another animal!')
    else:
        # Add a new question node to the existing question node's "yes" or "no"
        # branch, depending on the path we took to reach the guess node.
        if followed_yes_path:
            insert_new_yes_branch_q_node(q_node, g_node)
        else:
            insert_new_no_branch_q_node(q_node, g_node)
```

```
# This function handles any pair of parent and child nodes once we've asked at
# least one question, and so know if we're following the "yes" branch or not.
# Gameplay proceeds with mutual recursion between the pair of functions
# play_q_node() and play_node() until a guess node is reached.
def play_node(q_node:dict, node:dict, followed_yes_path:bool) -> None:
    if (node.get('Q')):
        play_q_node(node)
    else:
        play_g_node(q_node, node, followed_yes_path)

# This function handles any question node including the root. Ask the question,
# and then we know whether we're following the "yes" branch or not, and can
# call play_node().
def play_q_node(q_node) -> None:
    print(q_node['Q'])
    if is_answer_affirmative(get_one_word_answer()):
        play_node(q_node, q_node["Y"], True)
    else:
        play_node(q_node, q_node["N"], False)

def play_game(root_q_node:dict) -> None:
    # The root node is always a question node.
    play_q_node(root_q_node)

def get_indent_str(level) -> str:
    return " " * level * 3

# This is a pre-order, depth-first binary tree traversal in disguise. The basic
# algorithm is expressed something like this (in pseudocode):
#
# function handle_node(node)
#     do_something(node)
#     handle_node(node.left)
#     handle_node(node.right)
#
# Our traversal to print the game tree differs in the following ways:
#
# - We have two types of nodes, and they are handled differently, rather than
#   the usual design in which all nodes are the same type, and leaf nodes have
#   null left and right child pointers or references.
#
# - We report the current node question before each recursive call to handle
#   the left and right subtrees (in our case, the "yes" and "no" subtrees),
#   instead of just once before both calls. This is to make it clearer which
#   combination of question and answer brings us to each question or guess
#   node, because when printing a large tree there can be a large number of
```

```
#   lines from the subtrees printed between the two branches of a question
#   node.
#
# - We have a level parameter, used for indentation.
def print_game_tree(node:dict, level:int):
    if (node.get('Q')):
        print(get_indent_str(level) + 'Question: ' + node['Q'] + ' -> yes:')
        print_game_tree(node['Y'], level + 1)
        print(get_indent_str(level) + 'Question: ' + node['Q'] + ' -> no:')
        print_game_tree(node["N"], level + 1)
    else:
        print(get_indent_str(level) + "Guess: " + node['A'])
    return

print()
print('Play "Guess the Animal." Think of an animal and the computer will')
print('attempt to guess it. The game gets smarter over time as you teach it')
print('about more animals! This program by Paul R. Potts is based on the')
print('original BASIC game as it appears in the book Basic Computer Games:')
print('TRS-80 Edition, edited by David H. Ahl.')
print()
print('If you would like to see the internal tree of questions and animal')
print('names, type "tree" instead of "yes" or "no" when the program asks')
print('"Are you thinking of an animal?"')

while True:
    print()
    print('Are you thinking of an animal?')
    answer_str = get_one_word_answer()
    if is_answer_affirmative(answer_str):
        # Note that the root node is never replaced; the initial question is
        # always the same. Therefore, we don't need to pass a parent node to.
        # play_root().
        play_game(q_node_root)
    elif is_answer_tree(answer_str):
        print("Game tree:")
        print_game_tree(q_node_root, 1)
    else:
        print("Goodbye for now!")
        break
```

And here's a sample log from the above program:

```
Play "Guess the Animal." Think of an animal and the computer will
attempt to guess it. The game gets smarter over time as you teach it
about more animals! This program by Paul R. Potts is based on the
original BASIC game as it appears in the book Basic Computer Games:
```

```
TRS-80 Edition, edited by David H. Ahl.

If you would like to see the internal tree of questions and animal
names, type "tree" instead of "yes" or "no" when the program asks
"Are you thinking of an animal?"

Are you thinking of an animal?
tree
Game tree:
   Question: Does it swim? -> yes:
      Guess: a fish
   Question: Does it swim? -> no:
      Guess: a bird

Are you thinking of an animal?
yes
Does it swim?
yes
Is it a fish?
no
What animal were you thinking of?
a seal
Please type a yes-or-no question that would distinguish a seal from a fish:
Does it breathe water?
For a seal, what is the answer?
no

Are you thinking of an animal?
tree
Game tree:
   Question: Does it swim? -> yes:
      Question: Does it breathe water? -> yes:
         Guess: a fish
      Question: Does it breathe water? -> no:
         Guess: a seal
   Question: Does it swim? -> no:
      Guess: a bird

Are you thinking of an animal?
yes
Does it swim?
no
Is it a bird?
no
What animal were you thinking of?
a bat
```

```
Please type a yes-or-no question that would distinguish a bat from a bird:
Does it use echolocation?
For a bat, what is the answer?
yes

Are you thinking of an animal?
tree
Game tree:
   Question: Does it swim? -> yes:
      Question: Does it breathe water? -> yes:
         Guess: a fish
      Question: Does it breathe water? -> no:
         Guess: a seal
   Question: Does it swim? -> no:
      Question: Does it use echolocation? -> yes:
         Guess: a bat
      Question: Does it use echolocation? -> no:
         Guess: a bird

Are you thinking of an animal?
yes
Does it swim?
yes
Does it breathe water?
yes
Is it a fish?
no
What animal were you thinking of?
an octopus
Please type a yes-or-no question that would distinguish an octopus from a fish:
Does it have eight tentacles?
For an octopus, what is the answer?
yes

Are you thinking of an animal?
tree
Game tree:
   Question: Does it swim? -> yes:
      Question: Does it breathe water? -> yes:
         Question: Does it have eight tentacles? -> yes:
            Guess: an octopus
         Question: Does it have eight tentacles? -> no:
            Guess: a fish
      Question: Does it breathe water? -> no:
         Guess: a seal
   Question: Does it swim? -> no:
```

```
      Question: Does it use echolocation? -> yes:
         Guess: a bat
      Question: Does it use echolocation? -> no:
         Guess: a bird

Are you thinking of an animal?
no
Goodbye for now!
```

There Are Lots More

If you found this algorithm interesting, I recommend looking at the large body of old BASIC games — many are quite imaginative and clever. Try converting one of them into your favorite language. Half the battle is understanding what the original BASIC code was doing, as implementing algorithms in BASIC yielded code that can be pretty hard to read. But if you persist, you can bring lost gems back to life!
