

# Animals – Twenty Questions

Part 1 due Tuesday April 21, 2015

Part 2 due Tuesday April 28, 2015

---

## Objectives

---

For this assignment, you will:

- Gain experience programming in C++
- Gain experience working with heap-allocated memory in classes
- Gain more experience with tree data structures and recursive methods
- Implement a basic Artificial Intelligence algorithm

---

## Description

---

In this assignment we will play a guessing game (somewhat similar to 20 questions), with the computer doing the guessing—and learning at the same time. In the sample below, the human’s responses are shown after the question marks.

Welcome to the Animals game!

```
Shall we play a game? y
Were you thinking of a elephant? n
Doh! What was the animal? cow
What question separates cow from elephant? Does it moo?
What is the correct answer for this animal? y
```

```
Shall we play a game? y
Does it moo? y
Were you thinking of a cow? y
Great!
```

```
Shall we play a game? y
Does it moo? n
Were you thinking of a elephant? n
Doh! What was the animal? gnat
What question separates gnat from elephant? Is it bigger than a breadbox?
What is the correct answer for this animal? n
```

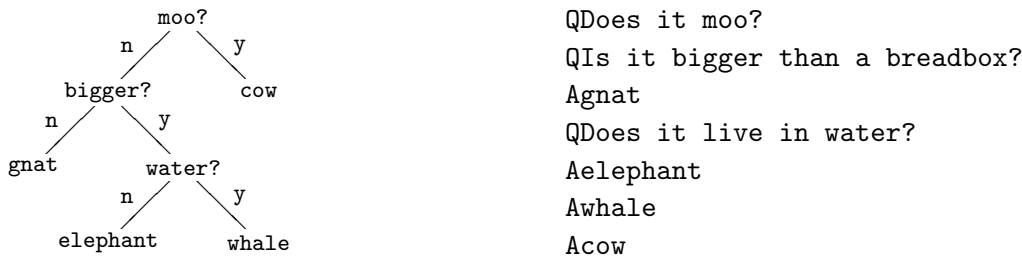
```
Shall we play a game? y
Does it moo? n
Is it bigger than a breadbox? y
Were you thinking of a elephant? n
Doh! What was the animal? whale
```

What question separates whale from elephant? Does it live in the water?  
What is the correct answer for this animal? y

Shall we play a game? n  
Bye!

The program maintains a binary tree whose internal nodes contain questions and whose leaves contain the names of animals. The left and right children of an internal node correspond to the responses “no” and “yes” (left being no and right be yes). When the program makes a wrong guess, it collects enough information to create a new node. The original leaf (the one with a wrong answer) is replaced by a new internal node that contains a new question and whose children are the old wrong answer and the new right answer.

*Notice that the tree has the property that a node is either a leaf or has two children.* That property makes it easy to distinguish answers from questions. It also allows us to store trees as text files, with the nodes listed in *preorder*. Here is a picture and the corresponding textual representation of a simple tree:



The lines in the file follow a preorder traversal of the tree. A question is prefixed by the letter Q and an answer by A. Do not specify empty trees in the file as they do not provide useful information. The shortest possible file has one line, an answer.

---

## Classes

---

This program has three cpp files `BinaryTreeIO.cpp`, `BinaryTree.cpp`, and `AnimalGame.cpp`. The corresponding header files have been provided. It may be helpful to look at Bailey’s implementation of a `BinaryTree` (source code is available on the course “Documentation and Handouts” page).

In general, all the nodes/trees will be created with `new` and all the variables will be of type `BinaryTree*`. The main C++ libraries that you will use are `string`, `iostream`, and `fstream`. Look at the link on the course resources page for C++ documentation on these classes.

Make sure there is a file called `animals.tree` in the directory where you are running your program (see `AnimalGame.cpp`) below or you will get an error!

### `BinaryTreeIO.cpp`

The file `BinaryTreeIO.cpp` provides functions to read and write a data file named `animals.tree` which stores the binary tree. The function declarations (i.e. prototypes) are provided in the file

## BinaryTreeIO.h.

This file will be the shortest and simplest. The header file specifies two functions whose actions should be straightforward: open a file named `animaltree`, call a recursive function to do the work, and then close the file. The equivalent Java methods are in the class `AnimalTree`.

As in our Java version, the easiest way to do this is to write recursive helper methods. The recursive functions will reflect the steps in a preorder traversal. The only parts to fill in are the file operations and the `BinaryTree` manipulations. You can write your own auxiliary functions as necessary.

As an example, here is a function that carries out a preorder traversal and counts the number of characters in all the strings:

```
int charCount(BinaryTree* root) {
    if (root == NULL){ // just in case
        return 0;
    }else if (root->isEmpty()){
        return 0;
    }else{
        return root->getString().length() +
            charCount(root->leftChild()) +
            charCount(root->rightChild());
    }
}
```

Notice that there is no class named `BinaryTreeIO`! This file simply provides two functions. There are no instance variables. In Java, these two functions would have to be attached to a class and would be declared `static`.

## BinaryTree.cpp

The file `BinaryTree.cpp` provides the implementation for the `BinaryTree` class. The class declaration is provided in the header file `BinaryTree.h`. Each node in the binary tree contains a single string.

There are many functions to implement in the `BinaryTree` class but most of them are short and simple. Completing this part is mostly a task in translating your knowledge of binary trees into C++. As a result, you will likely want to refer to the implementation of the class `BinaryTree.java` in Bailey's structure library. Note: you may not have to use some of the functions, like `height()` and `depth()`, but include them anyway (we may test them).

You must be very clear on how the different types of nodes are identified. Here is one possible convention:

- Recall that for our application, a node either has two children or is a leaf node. Thus, if one of `left` or `right` is `NULL`, then they both are `NULL`.

- An empty tree is a node whose `left` and `right` members are `NULL`.
- An empty tree has a `NULL` `parent`. That way, we can use just one empty node throughout the program, which you can define in your C++ class as a standalone, static variable, say `empty`.
- A leaf is a non-empty node whose left and right children are empty (*not `NULL!`* – being empty and being null are two separate states).
- A root is a non-empty node whose `parent` is `NULL`.

One of the consequences of this design decision is that each node has a unique parent and may therefore appear in only one tree. It leads to the “deep copy” requirement discussed below.

The tree nodes contain answers and questions. Answers are at leaf nodes and questions are at non-leaves. As in the Java version, do not add any additional information to the nodes to tell them apart. In particular, do not store in a node the leading character, A or Q, from the `animals.tree` file.

Notice that there is no `setString()` function. We want you to construct the trees by detaching subtrees and reassembling them. The greatest challenge is to maintain the parent pointers properly when executing `setLeftChild()` and `setRightChild()`. As we discussed in class when we covered binary trees, the `setParent()` method is private to insure the integrity of the parent pointers. Again, it *may* be useful to look at the Java `BinaryTree` implementation in the `structure5` library, which can be found via the link on the class handouts web page.

Finally, we have included code below for some of the methods of this class. You should read through and make sure to understand what is going on, but you may copy and use this code in the version you submit.

### `AnimalGame.cpp`

This file, as in the Java version, implements the `main` method that interacts with the human player to play the game and maintains the game tree. One variation from our Java version is that we won't deal with input and output from the command-line. Instead, when your game starts it should read from a file named “`animals.tree`” and when the game finishes it should update this file, that is, write the final tree to “`animals.tree`”.

`AnimalGame.cpp` will have more include directives than the other files.

```
#include <iostream>
#include <string>
#include "BinaryTree.h"
#include "BinaryTreeIO.h"
```

---

## Getting Started

---

Read through this entire document to make sure you understand the assignment. To make it easier to develop your code incrementally, we have provided:

- object files (.o files) for each class.
- A Java implementation of the Animal game is available at

`/common/cs/cs062/assignments/assignment11/JavaAnimals/`

The focus of your work will be to rewrite this Java program into C++. The starter files for this assignment (which include the header files and the object files) can be found at:

`/common/cs/cs062/assignments/assignment11/`

Look at the different header files and make sure you understand the basics of what the interface for each class is doing. Note: you may *NOT* change the header files provided and your program *must* compile with these header files.

You are now ready to start coding. You will likely work on `BinaryTreeIO.cpp` first, as it is due first. After that pick the file that seems easiest to you.

Once you have one of the .cpp files implemented, you can check it by linking with our provided binaries. For example, when you finish writing `BinaryTreeIO.cpp`, then typing the command

```
g++ -c BinaryTreeIO.cpp
```

will generate a new object file (.o) for your implementation<sup>1</sup>. Then, you can link all of the object files to create an executable file:

```
g++ -o animalgame BinaryTreeIO.o BinaryTree.o AnimalGame.o
```

This will generate a runnable version of your program in file `animalgame`.

You should also consider writing your own main method either in that class or in a separate file to test the functionality. Make sure when you're done though, to remove any main methods in your .cpp files except the one in `AnimalGame.cpp`.

---

## Grading

---

You will be graded based on the following criteria:

critierion	points
BinaryTreeIO	4
BinaryTree	6
AnimalGame	5
assert error checking	2
appropriate comments (including JavaDoc)	2
style and formatting	1
submitted correctly	1

---

<sup>1</sup>Note that if you ever want to use our implementation again, you will need to copy it again from `/common`

---

## Submitting Your Work

---

Part 1, consisting of your `BinaryTreeIO.cpp` class, will be due on Tuesday, April 21. Part 2, consisting of the rest of the program will be due one week later, on Tuesday, April 28.

For part 1, put your `BinaryTreeIO.cpp` file in a folder named “Assignment11P1\_LastNameFirstName” and drag it to the dropbox. *Notice the “P1” which stands for part 1.*

For part 2, submit all three `.cpp` files (including `BinaryTreeIO.cpp`) in a folder named “Assignment11P2\_LastNameFirstName” and drag it to the dropbox. *Notice the “P2” which stands for part 2.*

Make sure your code for both parts compiles with the included files *as they are provided* and that your files are well-documented files in the usual way with Javadoc comments, etc.

---

## Memory management in the BinaryTree class

---

One of most delicate aspects of C++ programming is making sure that all objects created with `new` are properly deleted. One must avoid stale pointers and memory leaks. The copy constructor, the destructor, and the assignment operator are the critical functions for these tasks.

In our `BinaryTree` class, the only job of the destructor is to delete all dynamically-constructed subtrees. We can do that with a simple call to `clear()`.

```
BinaryTree::~BinaryTree() {
    clear();
}
```

Although nice, this solution pushes the hard work off onto `clear()`, which recursively deletes the left and right subtrees and leaves the current object empty.

Below is the code for `clear()`. The implementation is mostly straightforward; there is just one subtlety. In the sample code, we keep a special empty tree, called `empty`, and use it wherever we can to avoid creating lots of empty trees. We must be careful, however, not to delete the tree named `empty`.

```
void BinaryTree::clear() {
    // don't change an empty tree
    if (isEmpty()){
        return;
    }

    // detach this tree from its parent, if any,
    // to avoid stale pointers from the parent
    if (isLeftChild()){
        parent->setLeft(empty);
    }
}
```

```

} else if (isRightChild()){
    parent->setRight(empty);
}

// delete the children recursively
if (left != empty){
    delete left;
}if (right != empty){
    delete right;
}

// make this an empty tree
value = "";
left = NULL;
right = NULL;
parent = NULL;
}

```

## Assignment operator

The assignment operator “copies” data into the (already constructed) current object. The first task is to recycle all the current descendants—which is exactly what `clear()` does. The next step is to assign new values to the fields, calling the copy constructor when necessary to obtain a deep copy.

```

const BinaryTree & BinaryTree::operator=(const BinaryTree & rhs) {
    // avoid self-copies
    if (this != &rhs) {

        // destroy the nodes in the current tree
        clear();

        // assign default values
        value = rhs.getString();
        left = NULL;
        right = NULL;
        parent = NULL;

        // copy the children, as necessary
        if (!rhs.isEmpty()) {
            if (rhs.getLeftChild()->isEmpty()){
                left = empty;
            }else{
                setLeft(new BinaryTree(*rhs.getLeftChild()));
            }

            if (rhs.getRightChild()->isEmpty()){

```

```

        right = empty;
    }else{
        setRight(new BinaryTree(*rhs.getRightChild()));
    }
}
}

// return the newly-copied object
return *this;
}

```

## Copy constructor

The copy constructor is nearly identical. In fact, similar enough, that we can use our trick of:

```

BinaryTree::BinaryTree(const BinaryTree & other){
    *this = other;
}

```

If you wanted to implement it for some reason, the only major difference is that the object under construction is a brand new object, so there is no need to call `clear()`.

As mentioned earlier, every non-empty node has a unique parent and can trace its ancestry to a unique root. This requirement leads to a “no sharing” policy and forces us to make deep copies. When we copy a subtree, either through the copy constructor or assignment, the copy is the root of its own tree. It cannot, and does not, share a parent with the object being copied.

Other implementations are possible, of course, and often the overhead of deep copying would be intolerable. Think about the options.