Programming Assignment 2

CS 311 Data Structures
Spring Quarter, 2006

$\boxed{\textbf{Due: Wednesday 4/26/05, 11:59pm}}$

Part 0

From now on, we will be paying particular attention to good commenting. All the programs that you submit should be commented according to the standard described below:

- **File headers**

  Each file should have a comment at the top, containing:

  - The name of the file
  - A *brief* description of the file's contents
  - The author's name
  - The date the file was created
  - If a file has been revised, then there must be comments explaining the changes. For example:

```
/*******************************************************
 * game.cpp                                            *
 *                                                     *
 * The main driver for a 20-questions game             *
 *                                                     *
 * Author: John Doe                                    *
 * Date:   20 September 2005                           *
 * Update: 25 September 2005 by Jane Smith             *
 *            -- fixed memory leak in destructor       *
 *******************************************************/
```

- **Class headers**
  Each class should be preceded by a comment containing:

  - The class name
  - The classes it is derived from (if applicable)
  - A *brief* description of the class
  - Update information (if applicable)

- **Function headers**
  Each function should be preceded by a comment containing:

  - The function name

  - Its inputs and the role of each input value
  - What the function accomplishes and returns
  - Any preconditions or assumptions about the input values

- **Inline comments**
  Add inline comments as necessary.

Part 1: Playing 20 questions

A decision tree is a model for classifying data. It represents the relationship between certain attributes and a class of objects. Below is an example of a decision tree that classifies animals:

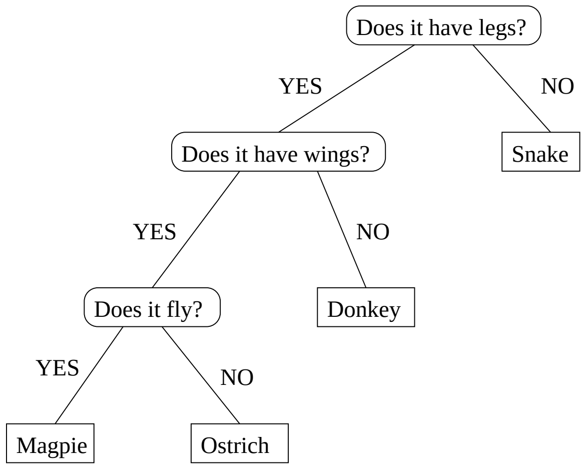

Figure 1: Sample decision tree.

An internal node of a decision tree contains an attribute test. The branches correspond to YES/NO answers. Finally, each leaf contains an object.

We can use the above tree to try and predict an animal based on certain attributes. Furthermore, we can provide additional information (new attributes and animals) which will allow the tree to grow and "learn". For example, the Ostrich and the Dodo have similar characteristics, but the Dodo is extinct. Given that, the tree can be updated thus:

For this part of the assignment, you will write a program that uses a decision tree to classify some initial data. The program will try to figure out what animal the user has guessed by asking yes/no questions. Furthermore, it will have the capability to "learn" more about the animal kingdom (and grow the tree).

Your program should start only with the knowledge that there exist dogs. The initial decision tree will have just one node containing "a dog", so its first guess about what animal the user has in mind will be "Is it a dog?".

When the program makes an incorrect prediction, it should ask the user for an attribute question: one whose YES/NO answer can give it a way to differentiate between the animal it predicted and the one the user had in mind. It will then create a new internal node for that attribute. Its children will be the two animals in question.

For example, if the user was thinking of a snake, then a suitable attribute question could be "Does it have legs?". This will then become the root of the tree, the YES child will be "a dog" and the NO child of the root will contain "a snake". Note that leaf nodes contain possible answers while internal nodes contain questions.

You are free to implement this any way you like, though well-designed programs will receive higher credit.

Your output should match that of the example below.

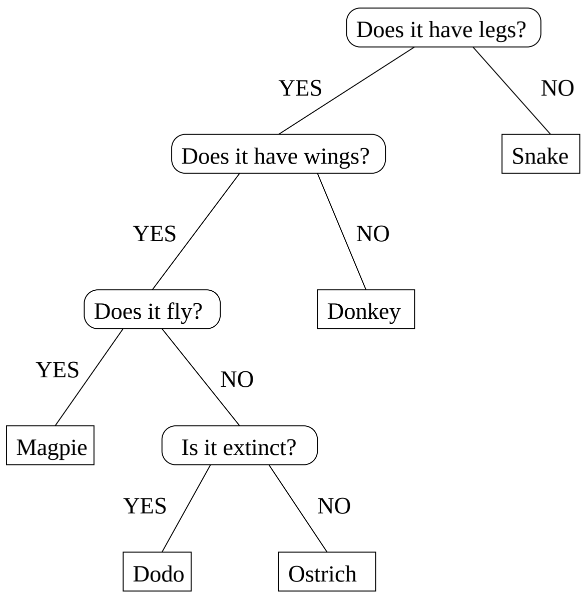

Figure 2: Sample decision tree that has "learned" new information.

A sample run of your program could generate the following dialog:

```
Welcome to the Guessing Game.
Think of an animal, and I will guess what it is.

===============================================================

THE ALL KNOWING COMPUTER: Is it a dog?
YOU: yes

THE ALL KNOWING COMPUTER: I am truly a genious!

===============================================

Would you like to play again? (y/n) y

Welcome to the Guessing Game.
Think of an animal, and I will guess what it is.

===============================================

THE ALL KNOWING COMPUTER: Is it a dog?
YOU: no

THE ALL KN^H^H^H^H^H^H HUMBLE COMPUTER: I give up. What is it?
YOU: a dodo

THE HUMBLE COMPUTER: Please type a question whose answer is yes for a dodo and no for a dog
YOU: Does it have wings?

THE ALL KNOWING COMPUTER: I knew that!
```

```
===============================================

Would you like to play again? (y/n) n

===============================================
```

When the game ends, your program should print (to stdout) a pre-order traversal of the tree, showing the contents of the nodes. For example, the tree corresponding to figure 1 would be printed as:

```
Does it have legs?
Does it have wings?
Does it fly?
a magpie
an ostrich
a donkey
a snake
```

Part 2: Playing with fire

You have been charged with writing a program that will help 911 dispatchers direct the closest fire engines to an incident.

You decide to use a two-dimensional k-d tree (i.e. a 2-d tree) to store the locations of the firehouses. This will allow a fast search in the tree for the closest firehouse to a given set of coordinates.

k-d trees

A k-d trees is a spatial data structure used to partition a k-dimensional space. In this application, we will concentrate on two dimensions. A k-d tree is a binary tree that is created by alternately partitioning the plane with vertical and horizontal lines. An internal node of the tree corresponds to a point that has split the plane either in an UP-DOWN or in a LEFT-RIGHT direction. A leaf node corresponds to a point that exists in a given "cell" (as defined by split lines) but does not split it yet.

Figure 3 shows a sequence of "insert point" operations, how they split the plane and how the corresponding tree is built.

Now, suppose there is a fire at the point marked with an X in figure 3. As we start traversing the tree (recursively, in preorder), we first examine the root, "a". At this time, it's the only firehouse we know about, so this is the closest one to the fire. We make note of the distance. Next, we examine "b". This is clearly closer than "a", so it becomes our current closest firehouse.

We cannot yet reject anything to the left of "b". If "f" was next to the vertical line, then it would be a closer neighbor.

Going left from "b", we find "c". This is way too far. Furthermore, we can guarantee that anything further up from "c" cannot possibly be closer to the fire than "b". If there was an UP branch off "c", we would not search in it at all. Similarly, we could reject a left branch off "f" if one existed.

Implementation

Implement a k-d tree as described above. You do not need to maintain its balance. You should implement the following operations:

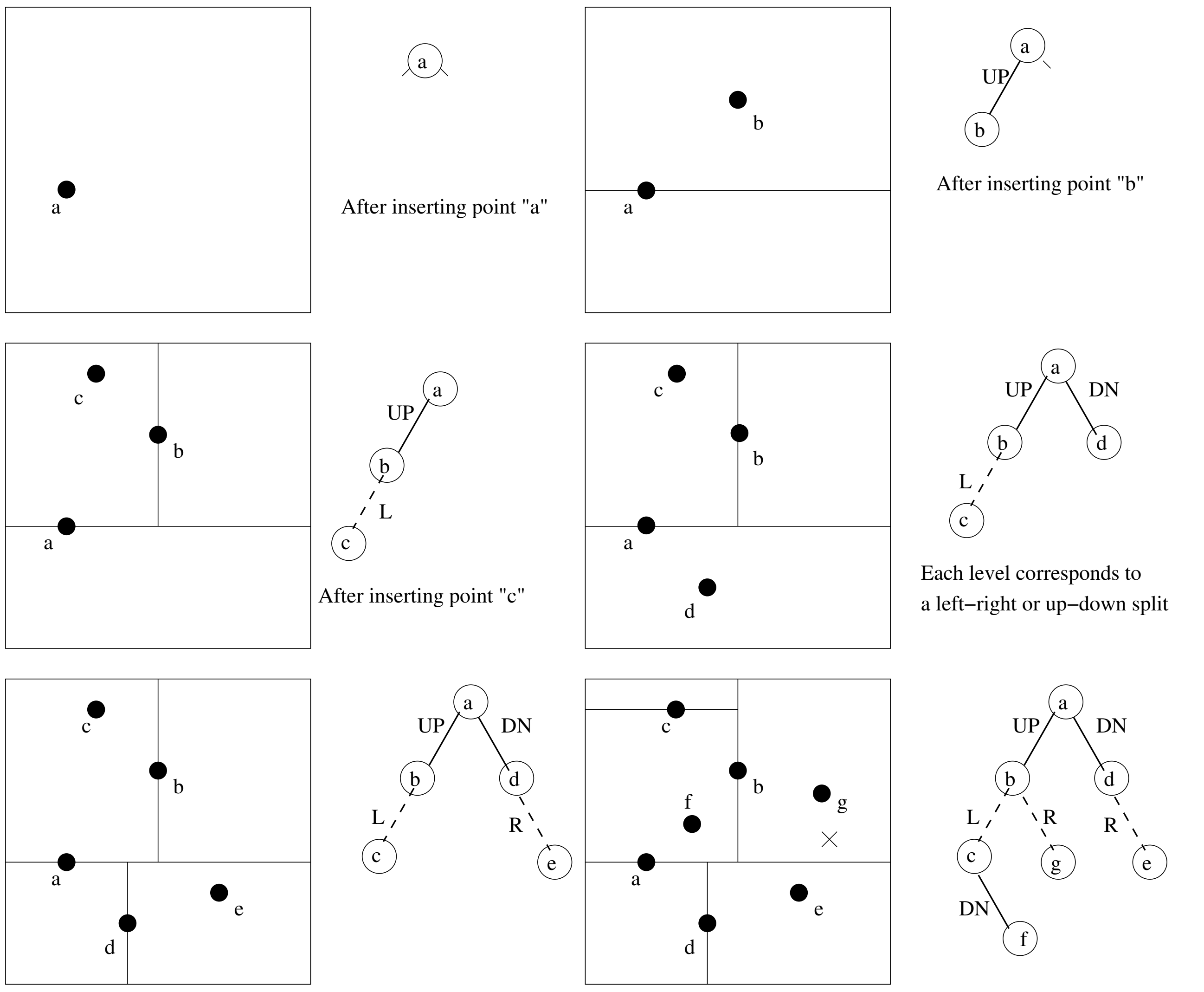
a
After inserting point "a"
a
b
UP
After inserting point "b"
a
b
c
UP
L
After inserting point "c"
a
b
c
d
UP
DN
L
Each level corresponds to
a left–right or up–down split
a
b
c
d
e
UP
DN
L
R
a
b
c
d
e
f
g
UP
DN
L
R
R
DN

Figure 3: *Points a,b,c,d,e,f,g are inserted in the given order into a 2-d tree.*

- **insert**. Given the name and coordinates of a point, insert it in the k-d tree.
- **find nearest neighbor**. Given the coordinates of a point (any point) find the point on the tree which is closest and print its name and coordinates. You should also print a trace of this operation that shows which nodes are examined and what is the current closest neighbor. For example:

```
Checking node "a" (-3, -1)
Nearest neighbor: "a", (-3, -1)
Checking node "b" (0, 2)
Nearest neighbor: "b", (0, 2)
Checking node "c" (-2, 4)
Nearest neighbor: "b", (0, 2)
etc.
```

  This way, we will be able to verify that your search algorithm does not go through the whole tree when it doesn't need to.
- **preorder traversal**. For each node, print its name and coordinates

Don't forget to also write a destructor that correctly destroys the tree at the end of the program. You may also need to implement several private methods that help with the implementation of the operations listed above.

Input and Interface

The initial firehouse info will be read from a file. Each line contains the name of a firehouse (a string) followed by two integers representing its coordinates on the cartesian plane.

The first point should always split the plane into a left and right part. (This is so we can test your results more easily)

Your program should provide the following interface:

```
1. Insert a new point
2. Find a nearest neighbor
3. Print the tree
4. Exit
Make a selection:
```

If the user selects 1, (s)he should be asked to provide a name followed by the coordinates (e.g. `firehouse1 30 50`).

If the user selects 2, (s)he will be asked to provide the coordinates of the fire (e.g. `10 20`).

If the user selects 4 the program will exit (don't forget to deallocate all memory!), otherwise it will print the menu of options again and continue as described above.

Assumptions

These assumptions simplify the coding. You are free to ignore any of them if you would rather have a more realistic program. Document any such decisions that you make.

- The city is on a grid and no two firehouses are on the same street.
- The earth is flat.
- Distance is computed in a straight line (as the crow flies).

Coding

Comment your code according to the guidelines described above.

Make certain you submit code that compiles and links with no warnings or errors. Your code will be graded on correctness, style and, where applicable, efficiency.

Provide a makefile for both parts.

Clarifications, etc.

If you require any clarifications, post your questions to the newsgroup. I have also reserved the Tech PC classroom for this Wednesday (the 19th) at 5:00pm to go over the basics of gdb and do some examples on memory allocation and deallocation. Please attend this as it will be helpful for the assignment.

Time management

START EARLY!

What to submit

When you finish your projects, remove any executables and `core*` files, use `tar` to archive everything and email the file to `cs311@cs.northwestern.edu` AND to yourself (to verify that the attachment worked). You will be sent an acknowledgement of receipt within 12 hours. Note that this will only verify that your email was received. We will not be checking whether the code was attached correctly.