CS2020: Data Structures and Algorithms (Accelerated)

Problem Set 4

Due: Tuesday March 5, 11:59pm

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** submit a file "collaborators.txt" containing the list of your collaborators, or the word "NONE." Your collaborators include every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. (Twenty Questions)

The Game of 20 Questions is a fun game for the whole family! It can be played by children from the age of five and upwards, and teaches important principles of basic logic and computer science. In this problem, you will be implementing a program to play 20 Questions (the advanced version).

In the typical game of 20 Questions, one player thinks of an object. The second player then asks *yes/no* questions about that object, until the object can be identified. For example, the game might proceed as follows:

> Alice: I'm thinking of an animal.
>
> Bob: Is it yellow?
> Alice: No.
>
> Bob: Is it brown?
> Alice: No.
>
> Bob: Is it gray?
> Alice: Yes.
>
> Bob: Is it bigger than a breadbox?
> Alice: Yes.
>
> Bob: Is it bigger than a house?
> Alice: No.
>
> Bob: Is it an elephant?
> Alice: Yes.

In this example, you might imagine that Alice and Bob each have in mind a set of animals, each with some associated properties:

1. Lion: Yellow, bigger than a breadbox, carnivore
2. Rat: Gray, smaller than a broadbox, omnivore
3. Elephant: Gray, bigger than a breadbox, herbivore
4. Etc.

With each question, Bob narrows down the possible set of objects that are consistent with the answers that Alice has already given, until he can successfully guess what she is thinking of.

Your job, in this problem set, is to implement a program to determine the best questions for Bob to ask. Let's say you have n objects with k properties. In that case, it is quite easy to come up with a strategy for finding the object using only k questions. What happens, however, when the number of objects is large? What happens when the number of properties k is large? Your goal is to find an algorithm that asks only $O(\log n)$ questions, no matter how large k is.

As we know from our study of binary search, the optimal strategy here is to ask a question that eliminates about half of the objects with each question. Unfortunately, there may be no one property that evenly divides the objects. For example, imagine you had only the following information on the animals:

1. Lion: Yellow
2. Rat: Gray
3. Giraffe: Long-necked
4. Elephant: Gray, Herbivore
5. Zebra: Striped
6. Tiger: Yellow, striped

Notice that here we are only listing the properties that an object *does* have. We are not listing the properties that an object does *not* have. For example, a lion is not gray, not long-necked, not an herbivore, and not striped.

So what question should you ask? No one property provides sufficient information. For the purpose of this game, we will allow you to ask compound questions, for example, *Is it yellow and striped?* or *Is it not gray and not striped?* Notice that this latter query would match two of the animals above (the lion and the giraffe). It may be difficult to devise a good query. The various properties may intersect in complicated ways, so it is not at all obvious which question will identify approximately half the objects.

As we will see, it is always possible to find a question that will identify at least 1/3 of the objects. (In fact, you can do somewhat better than that.)

QuestionGame Code

Attached to this problem set, you will find several Java classes that will help you to solve this problem. Begin by familiarizing yourself with this code.

- *QuestionGameBase*: This is the main class for controlling the game. It loads the object database, instantiates the player, chooses an object from the database, and then plays the game.

  Note that it is an `abstract` class, meaning that it cannot be instantiated directly. To use this class with your `QuestionTree`, subclass it and override `CreateTree()`. The `CreateTre` method should create an *empty* tree (which you will be writing later). In your subclass, you can also write code to test the game.

- *QuestionPlayer*: This is you, the player. The player is given the entire database of objects to begin with. Then, the player asks a sequence of queries until ready to guess. Most of the work, however, is delegated to the *QuestionTree*.

- *QuestionObject*: This class encapsulates an object which contains a name and a list of properties. Note that the class contains an iterator which lets you access the object's properties, as well as a method to determine whether a given property is contained in that object.

- *Query*: This class encapsulates a query. A query consists of both positive properties and negative properties. In the above example, asking whether the animal is *yellow* is a query about a positive property, while asking whether the animal is *not yellow* is a query about a negative property. The query consists of the conjunction ("and") of all the positive and negative properties.

  Note that the object contains two iterators, one that iterates the positive properties and one that iterates the negative properties. It also contains two methods to determine whether a given property or negative property is contained in this query.

- *QuestionTreeBase*: This is the main class for keeping track of the available objects, based on the questions that have already been asked. Again, this is an `abstract` class that cannot be instantiated directly. There are methods you have to implement as part of this problem set: you should implement them in a subclass. This will be discussed in more detail later.

- *TreeNode*: This is a node class that encapsulates one node in the QuestionTree. TreeNodes are used to hold object properties.

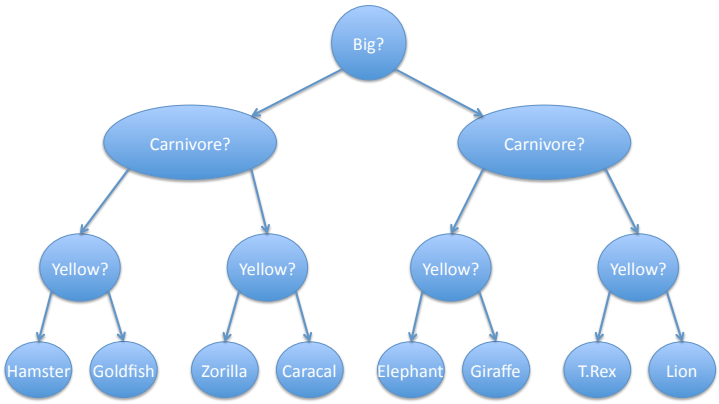
- *LeafNode*: This is a node class that is designed specifically to be a leaf of the tree. It cannot have any children. It it used to hold the object name.

In general, you should not modify any of the class files provided—all your changes should be contained in subclasses that you create.

Question Tree

We will organize the objects and their properties using a binary tree. (Note that it will *not* be a typical binary *search* tree.) Each node in the interior of the tree will represent a property, and each leaf will represent an object. If you walk the tree from the root to an object at the leaf, then every property node that you pass will correctly describe that object.

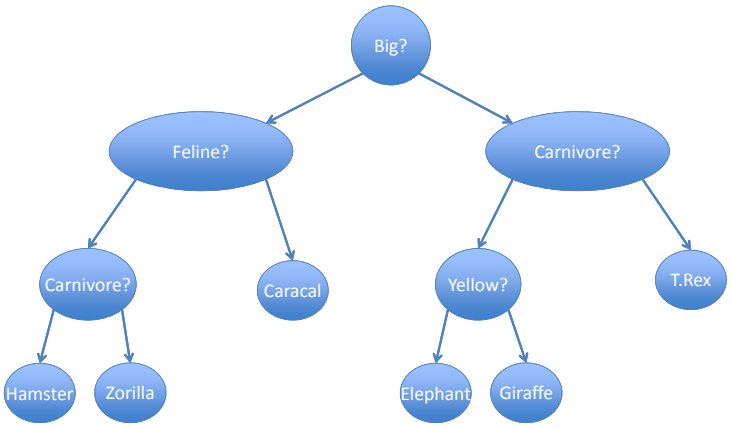
For example, imagine we have exactly three properties: *big*, *carnivore*, *yellow*. Then we can construct the following binary tree:



The root node asks whether the animal is *big*? If it is, then go right. Otherwise, go left. The next level (i.e., the children of the root) asks whether the animal is a *carnivore*? Again, if it is, go right, otherwise, go left. Finally, the third level nodes ask whether the animal is *yellow*? The leaves specify animals that satisfy the relevant properties.

Each property appears in the tree multiple times, in this case, at every node in the same level of the tree. This is because we need one leaf for each of the possible combinations of properties. If there are k properties, than there could be 2^k different objects, each of which has some subset of the properties.

Notice that the properties may not be as neatly organized. Different parts of the tree may have different properties, and they may be organized in different ways. For example, consider the following tree which has only a subset of the animals in the above tree:



What happens when the Goldfish (which is not feline) is inserted?

You will be writing the rest of the code for the `QuestionTree` class. Notably, you will be writing `buildTree`, which builds the tree, and `findQuery`, which determines the next query.

We have provided you with the remainder of the code in the abstract class[1] `QuestionTreeBase`, along with some additional methods that you might find useful:

- `Query constructQuery(TreeNode<String> node)`: Constructs a query based on the given node in the tree. If the chosen object satisfies the query, then it is guaranteed to be in the subtree of this node. If the chosen object does not satisfy the query, then it is guaranteed *not* not to be in the subtree indicated by this node.

  Beware that if you construct a query at node v containing property s, then the resulting query does not include s (or the negation of s). The query describes the path to v, but does not specify what happens at node v. In the example above, a query for Elephant includes *Big, not Carnivore, not Yellow*. A query for the parent of Elephant includes *Big, not Carnivore*.

[1] An abstract class is one that does not include all of the specified functionality. You must extend this class, adding the required functionality. See `http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html` for more details on abstract classes.

- `TreeNode<String> search(Query query)`: Given a query, this returns the matching node in the tree. If the chosen object satisfies the query, then it is guaranteed to be in this subtree. If the chosen object does not satisfy this query, then it is guaranteed to not be in this subtree.

  Beware that if the search returns a node v containing property s, then the resulting property s and the negation of s are not included in the query. Notice also that this method only returns the farthest node that can be reached walking the tree. If the treewalk reaches a node not included in the query (as either a positive or negative property), then the search returns that node—even if there are other properties in the query.

  For example, in the example above, a search for *not Big, not Feline* returns the parent of Zorilla. A search for *not Big, not Carnivore* returns the *Feline* node (as the search cannot determine from the query whether to proceed left or right).

- `void updateTree(Query query, boolean answer)`: Updates the tree based on the response to the query. If the query is satisfied (i.e., `answer` is `true`), then the `m_searchRoot` is updated to the matching node in the tree. Otherwise, the matching node in the tree is eliminated and the tree is re-trimmed.

- `int countObjects(TreeNode<String> node)`: Counts the number of objects that are descendents of a node.

- `TreeNode<String> getOneObject(TreeNode<String> node)`: Returns any one object from the tree. This is most useful if there is only one object left in the tree.

- `PrintTree()`: Prints all the objects in the tree and the associated root-to-leaf path. This may be useful for debugging.

Problem 1.a. Your first job is to build the *QuestionTree*. The Java class `QuestionTreeBase` contains an abstract method `buildTree` which takes as input an array of *QuestionObject*s and builds a tree as described above.

The requirements are that: (i) Every interior node should be a property. (ii) Every leaf should identify an object. (iii) All the child and parent pointers should be correctly in place. and (iv) All the properties for an object should be found on the root to leaf path to that object. A rightward edge indicates that the object has that property, while a leftward edge indicates that the object does not have that property.

Notice that `buildTree` takes an `ArrayList<QuestionObject>` as an input, that is, a list of `QuestionObject`s. See the Java reference for information on how to use an ArrayList.

One challenge in building the tree is that when you add a new object, it may require moving an existing object. (See the example above, where you insert a Goldfish.) One way to avoid this is to sort the items: if you insert objects with more properties first, then you will never have to move an object that has been previously inserted. (Why?) To this end, the `QuestionObject` implements the `Comparable<QuestionObject>` interface. It orders the items such that objects with more properties come before objects with fewer properties.

Hint: you can use `java.util.Collections.sort` *to sort the* `ArrayList<QuestionObject>`*, since the* `QuestionObject` *implements* `Comparable`*.*

Problem 1.b. When deciding what question to ask, you want to choose a query that will eliminate about half the nodes, whether the answer is yes or no. Since each query is associated with a node in the tree, we want to find a node v that has about half the objects as its descendents. If the answer to the question is *no*, then we can delete node v (removing half the objects). If the answer is *yes*, then we can ignore the rest of the tree, searching only in the subtree at v. (We do this by making v the root of the tree.) We need to determine how to find such a node v.

For a node v in the question tree, define $count(v)$ as the number of objects that are descendents of node v. For example, in the figure above, the *Feline* node has a count of three. Prove the following theorem:

Theorem 1 *Let n be the count at the root and assume $n > 1$. Then there exists a node v in the tree such that $n/3 < count(v) \leq 2n/3$.*

Submit your explanation.

Problem 1.c. Your next job is to implement `Query findQuery()`, the method that returns the next query to ask in the game. Notice that each query is equivalent to a node in the tree, and we have provided you with methods for converting back and forth from nodes to queries: `constructQuery` converts a node to a query, while `search` converts a query to a node.

Notice that as the game progresses, we will be updating the tree—deleting portions of the tree and/or updating the root, in response to the answers from the questions. Your method should work, despite these updates.

Write the code for `findQuery` such that each query will eliminate at least $n/3$ objects.

Problem 1.d. Once you have correctly implemented the above methods, you should be able to run the game. This can be achieved by subclassing the `QuestionGameBase` class. You can experiment with the attached databases. Choose one of the sample databases and determine how many queries it takes, both the average and maximum number of questions asked.

Problem 1.e. (Bonus) Extend the game semantics in some interesting way. (What other types of questions might you want to ask? Are there more complicated queries? What if the judge can think of more than one object—and you have to guess any of them or all of them? Etc.) More interesting extensions get more credit. Brag on the facebook group about any interesting things you try.