Assignment 3

COMP 250 Winter 2020

posted: Monday, March 9, 2020
due: Monday, March 23, 2020 at 23:59

Learning Objectives

This assignment aims at building on what we have seen in assignment 2, where we navigated through list-like objects, to start working with non-linear data structures. In this assignment, you will implement a derivative of a Binary Search Tree and execute several methods that iterate through trees. The objective is to familiarize yourself with the implementation of this Abstract data type, probably the most discussed one in the course. More specifically, you will be applying concepts seen through the in-class discussion of recursion, trees and how to navigate them, and binary search trees, as well as some principles we will formalize when we discuss heaps.

General Instructions

- **Submission instructions**
  - Late assignments will be accepted up to 2 days late and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format was submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
  - Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).
  - Please store all your files in a folder called "Assignment3", zip the folder and submit it to MyCourses. Inside your zipped .senior, there must be the following file.
    - `CatTree.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- It does not matter whether or not you create a package to store all these classes. It is up to you to decide whether you'd like to have a package or not.
- You are given class templates to complete. You can only change the code of these classes within the methods containing the comments "YOUR CODE GOES HERE". You cannot

change any method headers unless a comment specifies you can, and even then, the only changes you can make are to add exception handling. If you change a method header, we might not be able to grade you and you may receive a grade of zero. Read the comments in the templates carefully.

- Requests to evaluate the assignment manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class than what is already imported in the template. **Any failure to comply with these rules will give you an automatic 0.**
- We have included with these instruction a tester class. If your code fails the tests performed by the tester, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We therefore highly encourage you to modify the tester class and expand it.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.
- **If you submit just the java file (not in a zip), you will receive a grade of zero.**

A Crowded Cat Cafe Chain

You have just been named CEO of a chain of Cat Cafes. A cat cafe is a coffee shop in which cats are roaming and interact with customers. At the time of your promotion, you are informed that the chain does not have any repository of the cats, and maintains no information about their seniority. You are extremely upset upon hearing this as you understand those cats are your most important staff members and, like any other employees, deserve salary and working conditions that improve with seniority.

You immediately call your best friend, who recently told you they were a computer science expert. You expect them to build you a state-of-the-art database for all your cats. Upon hearing your request, they seem to hesitate a bit and, after a moment of wavering, confess that they are only taking their second computer science course and cannot yet build a professional database. However, they recently learned about binary search trees and how they can be leveraged to efficiently access data. They explain they have not yet learned about heaps and as such their idea does not utilize trees to their full potential, but it should still provide a solution to your problem. They email you a quick draft and you start getting excited about it (after all, cats do love trees), until they specify they cannot help you because they are currently busy with their new job, which they then begin describing. It sounds pretty cool until they try to convince you to invest in their "company" by buying a bunch of knives to re-sell. You try to tell them it sounds like a pyramid scheme but they start yelling something about those only ever occurring in Ancient Egypt, and they hang up on you. You try to call them back but it appears they blocked your number.

It looks like you are going to have to build your tree on your own. You have a second look at their email to see what you have to work with, and what is left to do.

The database of all cats is represented by a ternary tree `CatTree`. This tree has a single field `root` which contains a reference to the root node. The nodes are defined by a private nested class `CatNode`. Each `CatNode` is associated with data stored in a `CatInfo` object (code provided).

This `CatInfo` class stores data into the following fields:

- `String name`: the name of the cat.
- `int monthHired`: the time at which the cat was hired. You manage your business month to month, so for simplicity, since the company began operating on January 1st, 2000, you are identifying each month of operation with a simple integer associated with the number of months elapsed since then. Thus, January 12, 2000 was in month 0. November 25, 2015 was in month 191, and March 2020 is month 243.
- `int furThickness`: the average thickness of the cat's fur in millimeters (measured professionally), stored as an integer.
- `int nextGroomingAppointment`: the month in which we expect the next grooming appointment, once again stored as an integer.
- `int expectedGroomingCost`: the expected cost of the next grooming appointment, in dollars.

The `CatTree` class contains a `CatNode` inner class, and a field `root`, which contains the root node of the tree. It also implements `Iterable` and includes a `CatTreeIterator` inner class. `CatTree`

contains several already implemented public methods, as well as a constructor. Those methods, called on the tree, call the `CatNode` functions you will implement.

The `CatNode` inner class contains the information about the tree structure.

- `CatInfo data`: the `CatInfo` object with the data for this cat.
- `CatNode senior`: this cat and all the children of its node have more seniority than the current node's cat.
- `CatNode junior`: this cat and all the children of its node have less seniority than the current node's cat.
- `CatNode same`: this cat and all the children of its node have exactly the same seniority than the current node's cat.

In addition, in the structure of the tree, `CatNode`s with equal seniority should be sorted in decreasing order of fur thickness.

Build your database. Note: you will be tested on time efficiency, so make sure to implements methods based on what was seen in class!

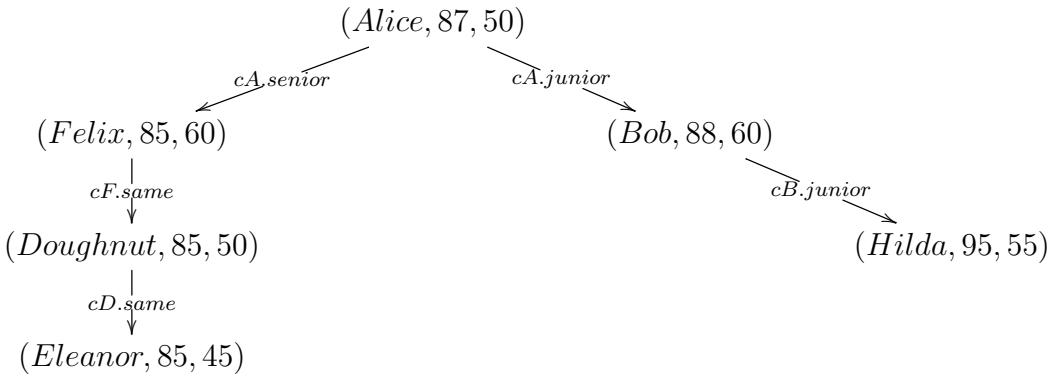
[**12 points**] First, consider `CatTree.addCat()`. This method takes as input a `CatInfo` object, converts it to a `CatNode` and calls `CatNode.addCat()` with the `CatNode` as input. You are asked to implement `CatNode.addCat()`.

Given a `CatNode c` and a `CatNode catToAdd`, `c.addCat(catToAdd)` adds the `CatNode catToAdd` in the tree with root `c` accordingly to the rules explained before : if the cat to be added is more senior than the cat at the root, then it has to be added to the subtree with root `c.senior`. In the case where the `CatNode` to be added has the same seniority as the root, the organization depends on fur thickness. If the cat to be added has thicker fur than the cat at the root, it should be stored in the root node, and the cat previously in the root node should be stored in the subtree with root `c.same`. The method should return the root of the tree it was called on.

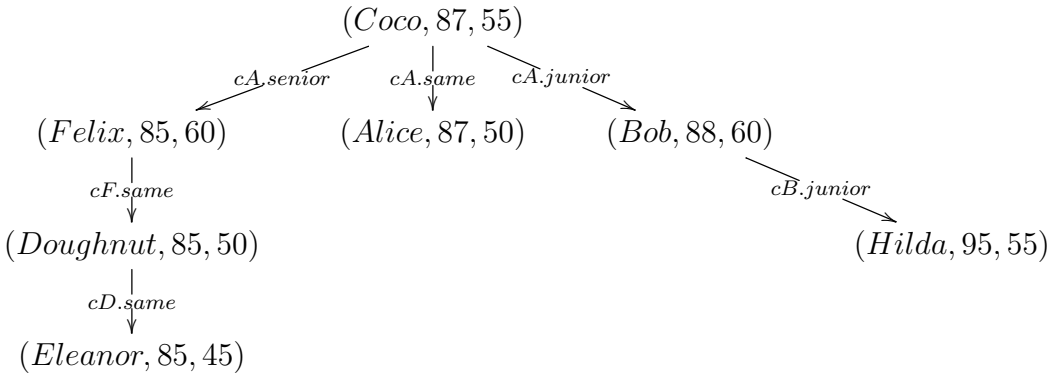
To clarify, let us give some examples. Let us define a `CatNode` cA, associated with `cA.data.name` `Alice`, `cA.data.monthHired` equal to 87, and `cA.data.furThickness` equal to 50. For simplicity, from here on, we will refer to this as declaring a `CatNode` $cA \longrightarrow$ (Alice, 87, 50). Let's build a tree with `cA` at its root, which also includes $cB \longrightarrow (Bob, 88, 60)$, $cD \longrightarrow (Doughnut, 85, 50)$, $cE \longrightarrow (Eleanor, 85, 45)$, $cF \longrightarrow (Felix, 85, 60)$, and $cH \longrightarrow (Hilda, 95, 55)$

When the fields `senior`, `same` or `junior` are not null, we represent them with an arrow to the corresponding `CatNode`. Note that we name the nodes base on the `CatInfo` they initially contain, but their contents can get swapped with other nodes, so `CatNode cA.data` might not

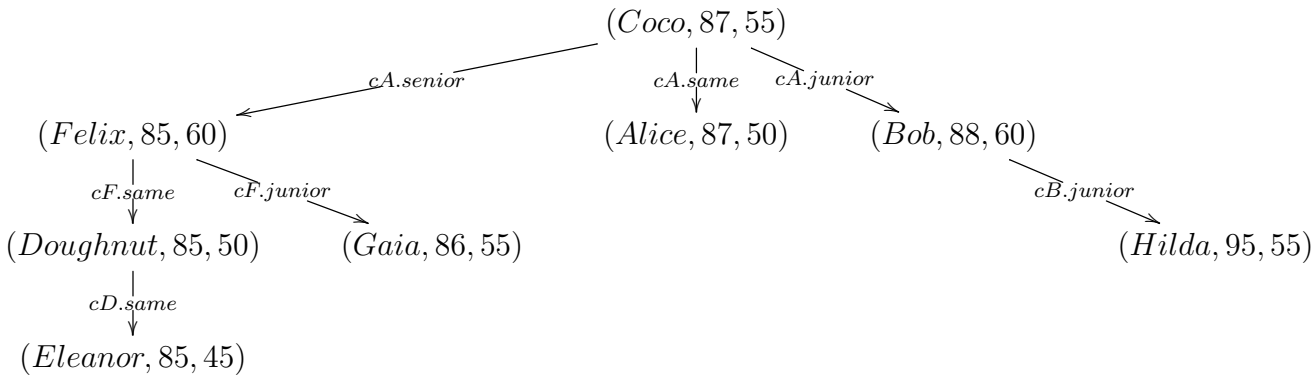
always contain the `CatInfo Alice`.



Now, if we have a `CatNode` $cC \longrightarrow (Coco, 87, 55)$, the command `cA.addCat(cC)` should return the root `CatNode cA`, but its content will have been swapped with the node we added:



If `CatNode cG` has `monthHired` equal to 86 and `furThickness` equal to 55, then the, given $cG \longrightarrow (Gaia, 86, 55)$ command `cC.addCat(cG)` should return the root `cA` and the tree structure should be updated to:

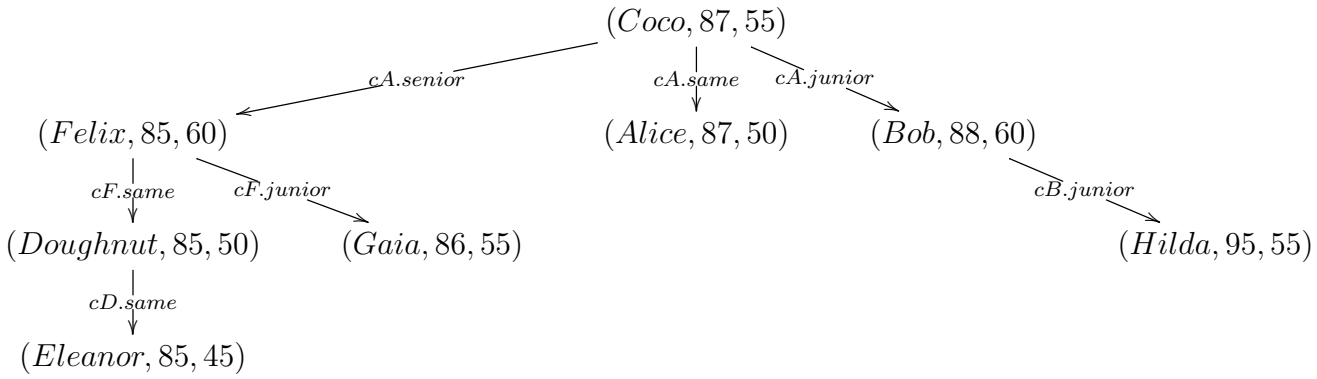


`addCat` is going to be used directly or indirectly in all your other methods. Make sure you pay extra attention to whether `addCat` works correctly before submitting your assignment. If it is defective, it will affect your marks for the whole assignment

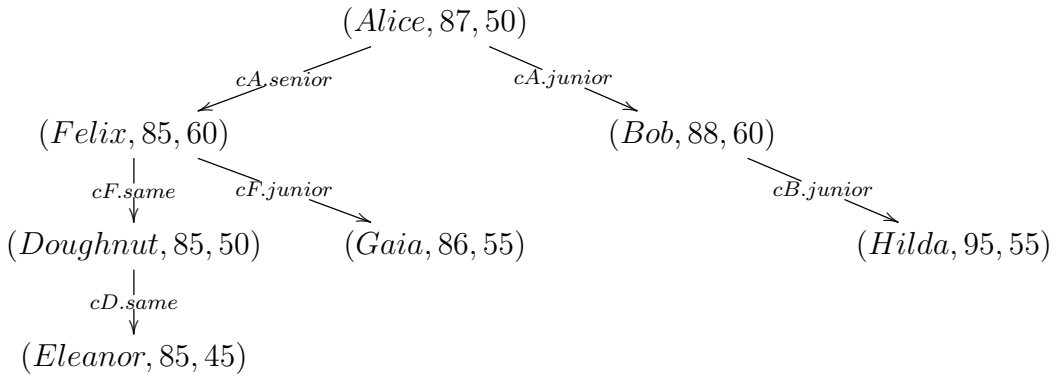
[22 points] Now you can code `CatNode.removeCat()`. Its argument is a `CatInfo`. The command `c.removeCat (cinf)` should attempt to remove the `CatNode` corresponding to the info in `cinf`, and return the `CatNode` it was called on, whether or not the tree was modified. You can follow the algorithm seen in class.

You should not remove any of the children of the `CatNode` with `data` corresponding to `cinf`. For the rest of the explanation, let us assume that `c.data == cinf` (you can easily adapt what is going to be said to the case where the `CatInfo` to remove is not the root). In this case, there are three cases to consider. First, if `c.same != null`, then the `CatInfo` object in `c.same` moves to the root, and the subtrees of `c` should be adjusted in relation to the new root. Second, if `c.same == null` and `c.senior != null`, then the new root becomes `c.senior` and you need to add the `CatInfo`s in `c.junior` to this new root. Finally, if `c.same == null` and `c.senior == null`, then the new root is `c.junior`.

Consider the following example. This is the tree we had in the previous task, so remember the contents of `cA` and `cC` were switched. `cA` now contains the cat named `Coco`, and `cC` now contains `Alice`.

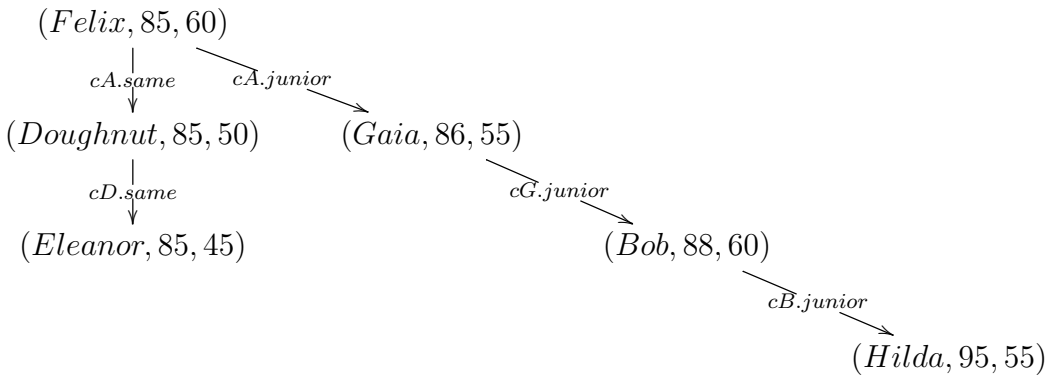


The command `cA.removeCat(cA.data)` should return `cA` with the tree updated to :

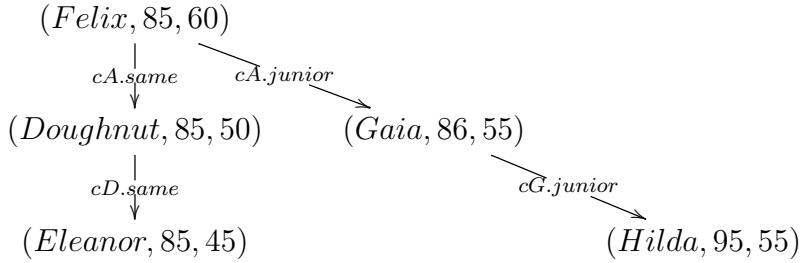


Notice the contents of `cA` were once again swapped. Now, the command `cA.removeCat(cA.data)`

should execute one more swap, and return `cA` with the tree updated to :



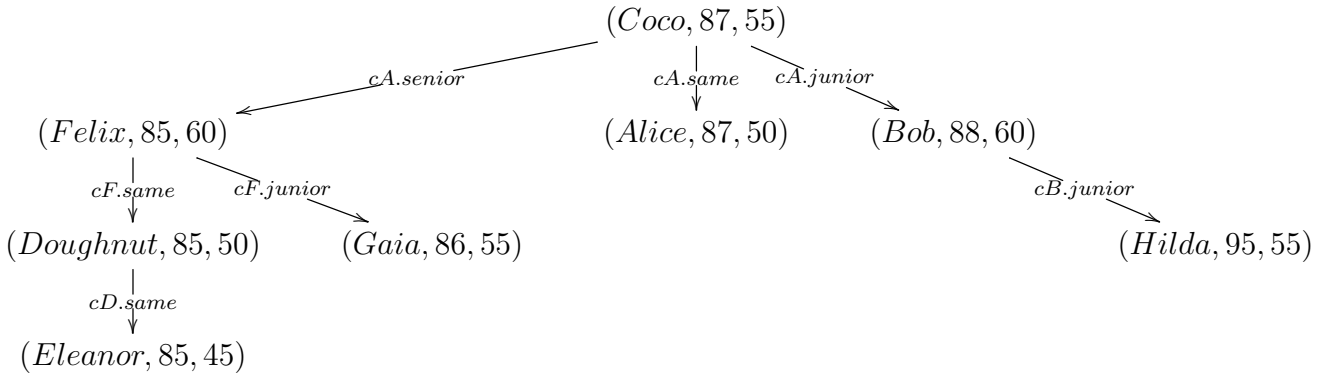
We could now continue with `cA.removeCat(cB.data)`, which will return `cA`, and update the tree to:



where the node containing the `CatInfo` named `Hilda` is `cB`.

[8 points] Code `mostSenior`. When called on a `CatNode c`, the integer `c.mostSenior()` should be the month when the most experienced `CatNode` in the tree with root `c` was hired. Do not modify the tree structure.

In this example,`cA.mostSenior()` should return 85 and `cC.mostSenior()` should return 87. Remember `cA` contains `Coco`, and `cC` contains `Alice`



[9 points] Implement `fluffiest`. When called on a `CatNode c`, the integer `c.fluffiest()` should be the fur thickness of the `CatNode` with greatest fur thickness in the tree with root `c`. Do not modify the tree structure.

On the example given to you in the previous question, `cA.fluffiest()` should return 60 and `cD.fluffiest()` should return 50.

[**8 points**] Implement `hiredFromMonths`. When called on a `CatNode c`, the integer `c.hiredrFromMonths(monthMin, monthMax)` should be the number of cats hired from month `monthMin` to `monthMax` (including those two months), that is in the tree with root `c`. If `monthMin > monthMax`, this should return 0. Do not modify the tree structure.

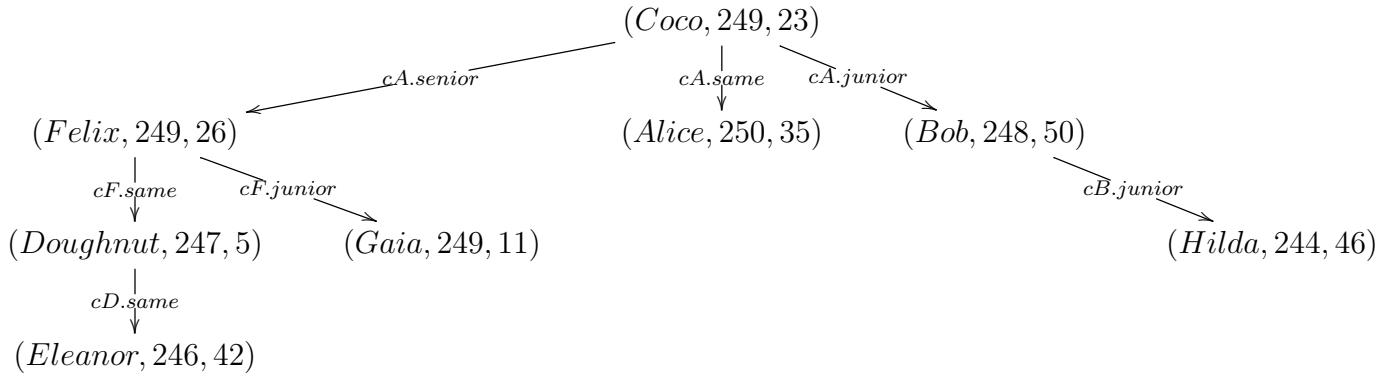
On the example from the question asking you to code `mostSenior`, `cA.hiredFromMonths(86, 88)` should return 4 (corresponding to `CatNode cG`, `cC`, `cA` and `cB`).

[**8 points**] Implement `fluffiestFromMonth`. When called on a `CatNode c`, `c.fluffiestFromMonth(month)` should return the `CatInfo` linked to the cat with thickest fur, hired in the month `month`, in the tree with root `c`. If no such cat is found, `null` should be returned. Do not modify the tree structure.

On the example from the question asking you to build `mostSenior`, `cA.fluffiestFromMonth(85)` should return `cF.data`, and so should `cF.fluffiestFromMonth(85)`, but `cD.fluffiestFromMonth(85)` should return `cD.data`.

[**15 points**] Implement `costPlanning`. When called on a `CatNode c`, the array `c.costPlanning(n)`, which should have length `n`, should have in its i-th cell, how much you should spend on month (243 + i) for all the scheduled grooming appointments over this time period, in the tree with root `c`. Here, month 243 is March 2020, so at i=0, you are planning the cost for this month, and at index 1, you are planning the costs for month 244, which is next month (April). The argument `n` will always be a positive integer. Do not modify the tree structure.

Let us modify the example given in the question asking you to implement `mostSenior` to display the month planned for the next grooming appointment and the associated cost.



On this example, `cA.costPlanning(7)` should return the array `{0, 46, 0, 42, 5, 50, 60}`: in month 243, there is no cost to plan for grooming, but in month 249, Cats `Coco`, `Felix` and `Gaia` will need to be groomed, for a cost of $23 + 26 + 11 = 60$. Similarly, `cB.costPlanning(7)` should return the array `{0, 46, 0, 0, 0, 50, 0}`. *Note: Bob and Hilda have higher grooming cost because they bite. Doughnut gets a discount because he no longer has teeth.*

You can use the iterator described in the next task to solve this problem

[**18 points**] Implement the inner class `CatTreeIterator`. The method `CatTree.iterator()` returns a `CatTreeIterator` object which can be used to iterate through all the cats in the tree. The iterator should access the cats from most senior to most junior. In the case in which cats

have been hired within the same month, the iterator should access cats based on fur thickness: cats with thicker fur should be access later.

We will test your iterator by examining the order in which the cats are accessed. It is ok if your iterator uses an `ArrayList` to store references to all the cats in the tree. We have imported the class in the template. There are more space efficient ways to implement an iterator for trees, but you will not be tests on space efficiency.

Good luck!