Assignment 3

COMP 250 Fall 2025

Posted on:	Sunday, Nov. 16, 2025
Suggested Deadline:	Monday, Dec. 1, 2025 at 23:59

General Instructions

- **Submission instructions**
  - We encourage you to start early and aim to meet the suggested deadline (Dec. 1st), which is designed to be reasonable under standard circumstances. Re-submissions will be allowed until December 10th, but to receive Mastery you must submit a Proficient version of the assignment by the suggested deadline. Each time you submit, the results of some of the tests will be exposed, and you will know immediately what competency level you have achieved. Note that questions during office hours and on the discussion board will be prioritized based on upcoming deadlines.
  - 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**. The level of mastery you achieve with your latest submission is what will be considered when computing your final letter grade.
  - Your task is to complete and submit the following file:
    - `Catfeinated.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks.
  - **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD YOUR CODE HERE" block.** You may add private helper methods to the class you have to submit (and in fact you are highly encouraged to do so), but you are not allowed to modify any other class.
- Please make sure that the file you submit is part of a package called `assignment3`.
- The assignment shall be graded automatically. 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 beside those already imported in the code provided. **Any failure to comply with these rules will give you an automatic 0.**

- Whenever you submit your files to Ed, you will see the results of certain exposed tests along with the competency level you've achieved. A small subset of these tests will also be shared with you to help with debugging. We highly encourage you to write your own tests and thoroughly test your code before submitting your final version. Learning how to test and debug your code is a fundamental skill to develop.

  You are welcome to share your tester code with other students on Ed and collaborate with others in developing it.

- Your submission will receive an "Inconclusive" if the code does not compile.

- Failure to comply with any of these rules may result in penalties. If anything is unclear, it is your responsibility to seek clarification, either by asking during office hours or posting your question on the Ed.

- IMPORTANT: Do NOT wait until you've finished writing the entire assignment to start testing your code. Debugging will be extremely difficult if you do so. If you need help with debugging, feel free to reach out to the teaching staff. When asking for help, be sure to mention:

  - The bug you're trying to fix.
  - What steps you've already taken to resolve it.
  - Where you've isolated the error.

Learning Objectives

This assignment combines concepts from binary search trees and max heaps, providing an opportunity to deepen your understanding of tree-based data structures. By working through this problem, you will develop a stronger grasp of how to manipulate trees, implement recursive algorithms, and leverage the inherent recursive structure of these data structures to solve complex problems efficiently.

Additionally, this assignment offers practical experience with interfaces like `Comparable` and `Iterable`. You will gain hands-on practice in implementing your own iterator, reinforcing your understanding of how interfaces can be used to enhance the functionality and flexibility of your code.

Managing the Catfeinated Empire

You have just been named CEO of a new chain of Cat Cafes, *Catfeinated*. These cozy coffee shops, where cats roam freely and interact with customers, have been a hit with feline fans. However, upon stepping into your new role, you're shocked to discover that the chain has no proper records of its most important employees - the cats! As a forward-thinking boss, you recognize that these feline employees deserve salaries and working conditions that improve with seniority, just like any other staff.

To fix this, you reach out to a close friend who once claimed to be a computer science expert. Excitedly, you ask them to create a sophisticated database for the 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 full-fledged database. However, they recently learned about binary search trees and how they can be leveraged to efficiently access data. They explain they have also heard about heaps and they'd like to combine the two ideas together in an attempt to utilize trees to their full potential. They email you a draft, which looks promising (after all, cats *love* trees), until they abruptly mention they're busy with a new venture and try to sell you on joining their "business" by buying a bunch of knives to re-sell. Suspecting it's a pyramid scheme, you push back, but they hang up and block your number.

Realizing you'll have to take matters into your own hands, you decide to build this system using the draft as a starting point.

The Cat Database

The cats' information will be stored in a binary tree, represented by the class `Catfeinated`. This tree has a single field, `root`, which refers to the root node. Nodes are defined using a nested class, `CatNode`, which is made `public` for testing purposes. Each `CatNode` contains data stored in a `Cat` object.

The company tracks time month-to-month, starting from January 1, 2000 (month 0). For instance, September 2015 corresponds to month 189, and November 2024 is month 299. You keep track of the month in which each cat was hired to work at Catfeinated in each cat's file. To have an easier time searching through the files, these are kept in order based on the cats' seniority.

These cats are truly your most valuable asset, and regular grooming appointments are a must to keep them healthy and happy. The company provides regular trips to the cat salon and covers all the expenses. To schedule the trips and help you keeping your budget on track, you decided to store all the relevant information on each of these cats' files.

Fluffy cats, with their luxurious coats, are a hallmark of *Catfeinated*. As a devoted customer-turned-CEO, you've decided to honor these extraordinary cats with a Hall of Fame. The thickness of each cat's fur (measured professionally in millimeters!), is recorded upon hiring.

Note that, due to budget constraints, you are never able to hire more than one cat per month. That is, you can assume that no two cats have the same seniority.

Testing and Debugging

To succeed in this assignment—and any coding project—it is crucial to test your code incrementally. Write small tests and validate each method before moving to the next.

Do NOT wait until the entire assignment is written to begin testing your code. Debugging becomes exponentially harder when everything is interconnected. If you encounter a bug, isolate the issue to a specific piece of code, and ensure you've thoroughly attempted to resolve it before seeking help. When asking for support, clearly state:

The problem you're facing. The steps you've taken to debug the issue. The specific part of your code where the error occurs (beyond the stack trace).

This proactive approach will save you time and frustration while helping you develop strong debugging skills.

The Cat class

To represent a cat, we have provided you with a pre-defined class called `Cat`. This class captures the essential details of each feline employee through the following fields:

- `String name`: The cat's name.
- `int monthHired`: The month the cat was hired, measured as the number of months since January 1, 2000.
- `int furThickness`: The thickness of the cat's fur in millimeters.
- `int daysUntilGrooming`: The number of days remaining until the cat's next grooming appointment.
- `double groomingCost`: The expected cost (in dollars) of the upcoming grooming session.

This class implements the `Comparable` interface, enabling comparisons between cats based on their fields. Several `public` methods are also provided for your use.

Important: You must not modify this class under any circumstances. It is provided as-is, and altering it will result in your program failing to work when uploaded on Ed.

The Cafe: Catfeinated Class

A template for the `Catfeinated` class has been provided to you. This class is the foundation of your Cat Cafe database and includes the following key components:

- A nested class, `CatNode`, which represents each node in the tree structure.
- A field, `root`, which stores a reference to the root node of the tree representing the cafe.
- An inner class, `CatfeinatedIterator`, which supports iteration over the cafe's cats.

The `Catfeinated` class also implements the `Iterable` interface, allowing you to iterate through its nodes in ascending order of seniority. Within this class, you will find several `public` methods

that have already been implemented. These methods are all non-static and are designed to operate on objects of type `Catfeinated`. Most of these methods rely on the functionality provided by the `CatNode` class, which you will need to complete.

The database for *Catfeinated* is represented by a binary tree that simultaneously adheres to two properties:

1. Binary Search Tree property on seniority: the nodes are ordered based on the cats' seniority, making it efficient to find cats hired in a specific month.
2. Max Heap property on fur thickness: when comparing the thickness of the cats' fur, the tree ensures that each parent node has a thicker coat than its children.

The `CatNode` nested class is at the heart of the `Catfeinated` tree. Each node represents a single cat and contains the following fields:

- `Cat catEmployee`: a reference to an object of type `Cat`. This is the element/key stored in the node and contains all relevant information about a specific cat employed at the cafe.
- `CatNode junior`: a reference to the root of the left subtree which stores information about cats that are more junior (i.e. hired later) than the cat stored in the current node.
- `CatNode senior`: a reference to the root of the right subtree which stores information about cats that are more senior (i.e. hired earlier) than the cat stored in the current node.
- `CatNode parent`: a reference to the parent node. Hence it stores information about a cat with thicker fur than the current cat (and who therefore has therefore higher chances of getting in that hall of fame of yours ;)).

For this assignment, you need to implement the following methods and classes. Your code will be tested for time efficiency, so ensure that you take advantage of the tree's structure and properties while applying the techniques we learned in class. This assignment challenges you to think critically about where to begin and how to build upon your progress. The tasks are not listed in order of difficulty, so we encourage you to evaluate which methods to tackle first based on your familiarity with the concepts.

Here are some tips to help you get started:

- *Think strategically.* Some methods are interdependent, meaning that implementing one could simplify the implementation of another. For instance, a fully functional `CatfeinatedIterator` might be helpful for completing other tasks.
- *Start small.* Begin with smaller, self-contained tasks like `findMostSenior()` to warm up and get comfortable with the tree structure. These tasks provide a solid foundation and are a great way to build confidence before tackling more complex methods.
- *Focus on what you know now.*
  - Methods, like `findMostSenior()`, `findMostJunior()`, and `CatfeinatedIterator`, can already be implemented using concepts we've covered in class. These methods are a great starting point to familiarize yourself with the tree structure and gain momentum on the assignment.

- For methods like `hire()`, `retire()`, and `buildHallOfFame()`, you'll need a solid understanding of heaps and how the binary search tree and max-heap properties interact. We'll cover heaps in depth during Thursday's class (Nov. 21st), so consider focusing on the other methods now and revisit the more advanced ones after the lecture.

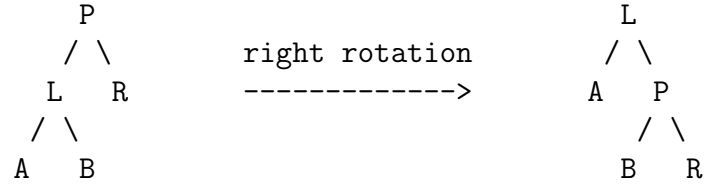
- `Catfeinated.Catfeinated()`
  This constructor takes as input a `Catfeinated` object `cafe` and creates a *shallow* copy of it. The new `Catfeinated` object is a tree storing exactly the same objects of type `Cat`, but with different `CatNode`s.

- `CatNode.hire()`
  This method takes as input a `Cat` object `c` and adds it to the tree rooted at the `CatNode` on which the method was called. The method returns the root to the new tree that now contains `c`.

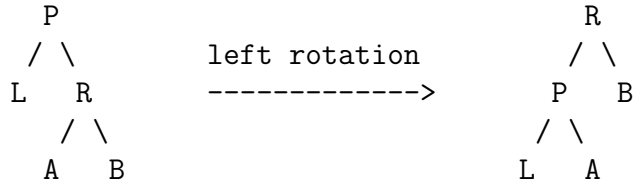
  As explained above the tree we are trying to implement is a binary search tree when we look at the cats' seniority, and a max heap when we look at the cats' fur thickness. To preserve the tree's properties, adding a new cat `c` to the tree has to be done in two steps:

  1. Add a new node to the tree in the leaf position where binary search determines a node for `c` should be inserted. (see `add()` for binary search tree learned in class)
  2. Fix the trees so that the properties of the max heap are maintained (i.e. the parent node must have thicker fur than its children). This means that we need to perform upheap if needed. Note that since this is also a binary search tree, we need to make sure that when performing upheap we don't break the properties of the binary search tree. To ensure this, instead of performing upheap as seen in class, we will need to implement a tree rotation that reverses the parent-child relationship whenever necessary. Depending if the child that has to be swap in the parent position is the left or the right child, we will need to perform a right rotation or a left rotation respectively. This is how the rotations work:

     – To swap the left child in the parent position, we perform a right rotation as follows:



     – To swap the right child in the parent position, we perform a left rotation as follows:



The tricky part about implementing this is to make sure to fix all the necessary pointers.

For a couple of examples, see the end of the pdf.

- `CatNode.retire()`
  This method takes as input a `Cat` object `c` and removes the key *equals* to `c` (e.g. compare them using `equals()`) from the tree rooted at the `CatNode` on which the method was called. The method returns the root of the tree obtained by removing `c` from the latter.

  As explained above the tree we are trying to implement is a binary search tree when we look at the cats' seniority, and a max heap when we look at the cats' fur thickness. Hence, removing a cat `c` from the tree has to be done while preserving both properties.

  1. Remove the node that contains a cat equal to `c` similarly to how we did it in class. **Differently from the algorithm seen in class**, when the node to be removed has both children, find the most senior cat `seniorC` in the left subtree, use `seniorC` to replace the cat to be removed, and remove `seniorC` from the left subtree (the algorithm we have seen in class was doing the same thing, but using instead the smallest key from the right subtree).
  2. If the changes made from removing the node above broke the properties of the max heap, then perform downheap to fix the tree. Here you want to think about which are the situations in which removing a node would break the properties of the max heap and tackle only such situations. *You do not want to blindly perform downheap through the entire tree.* As for upheap, also downheap will have to be implemented differently from how we have seen in class, since we need to make sure to also preserve the properties of the binary search tree. Once again, you will have to determine which of the two children should be swapped in the parent position and based on that, perform the correct tree rotation as explained above.

  For a couple of examples, see the end of the pdf.

- `CatNode.findMostSenior()`
  This method returns the most senior cat in the tree rooted at the `CatNode` on which the method is called.

- `CatNode.findMostJunior()`
  This method returns the most junior cat in the tree rooted at the `CatNode` on which the method is called.

- `Catfeinated.buildHallOfFame()`
  This method takes as input one integer `numOfCatsToHonor`. You can assume that the input is a number greater than or equal to zero. The method returns a list of containing `numOfCatsToHonor` cats from the cafe with the thickest fur. The cats should be appearing in the list in descending order of fur thickness. If there are less than `numOfCatsToHonor` in the cafe, then all of the cats from the cafe should appear in the list.

- `Catfeinated.budgetGroomingExpenses()`
  This method takes as input an integer `numDays` indicating the number of days for which we

want to budget the expenses. It returns a double indicating the expected amount of dollars that the cafe will need to spend for grooming its cat employees in the next `numDays` (inclusive).

- `Catfeinated.getGroomingSchedule`
  This method takes no inputs and returns an array list of array lists of cats. The array list at index 0 contains all the cats that need grooming in next week (i.e. within the next 7 days, including today which is represented by a 0). The array list at index 1 contains all the cats that need grooming in a week from now (i.e. within the next x days, where x is greater than or equal to 7 and less than 14). In general, the array list at index i contains all the cats that need grooming in i weeks from now. In each sublist, the cats appear in ascending order of seniority.

- `CatfeinatedIterator`
  Implement the inner class `CatfeinatedIterator`. The method `Catfeinated.iterator()` returns a `CatfeinatedIterator` object which can be used to iterate through all the cats in the cafe. The iterator should access the cats in ascending order of seniority. Please note that, as per documentation, the method `next()` must raise a `NoSuchElementException` if called when there are no more elements to iterate on. The exception has been imported for you in the template.

  You can implement the iterator however you like. We will test it 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 tested on space efficiency.

Examples

Let's now look at a couple of examples. Consider the following cats:

```
Cat B = new Cat("Buttercup", 45, 53, 5, 85.0);
Cat C = new Cat("Chessur", 8, 23, 2, 250.0);
Cat J = new Cat("Jonesy", 0, 21, 12, 30.0);
Cat JJ = new Cat("JIJI", 156, 17, 1, 30.0);
Cat JTO = new Cat("J. Thomas O'Malley", 21, 10, 9, 20.0);
Cat MrB = new Cat("Mr. Bigglesworth", 71, 0, 31, 55.0);
Cat MrsN = new Cat("Mrs. Norris", 100, 68, 15, 115.0);
Cat T = new Cat("Toulouse", 180, 37, 14, 25.0);
```

Assume we have created an object of type `Catfeinated` `cafe` using the constructor that takes no inputs. This means that to begin with, the cafe is empty. That is, the field `root` of `cafe` contains `null`.

We then call `hire()` on `cafe` with input `B` (i.e. `cafe.hire(B)`). After its execution, the cafe is represented by a tree that has only a root node as follows:

```
B(45, 53)
```

I indicated between brackets the month in which the cat was hired, followed by its fur thickness in millimeters.

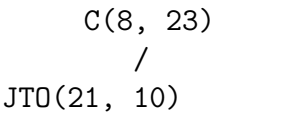
After executing the statement `cafe.hire(JTO)`, the tree representing the cafe will look as follows:

```
B(45, 53)
       \
    JTO(21, 10)
```

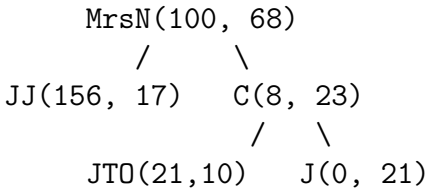
After executing the statement `cafe.hire(C)`, the tree representing the cafe will look as follows (note that here one rotation was necessary in order to maintain the max heap property of the tree):

```
B(45, 53)
       \
     C(8,23)
       /
 JTO(21, 10)
```

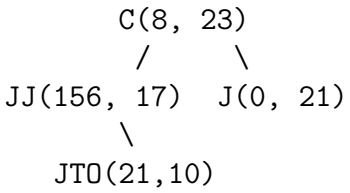
After executing the statement `cafe.retire(B)`, the tree representing the cafe will look as follows:



After executing the statements `cafe.hire(JJ)`, `cafe.hire(J)`, and `cafe.hire(MrsN)` , the tree representing the cafe will look as follows:

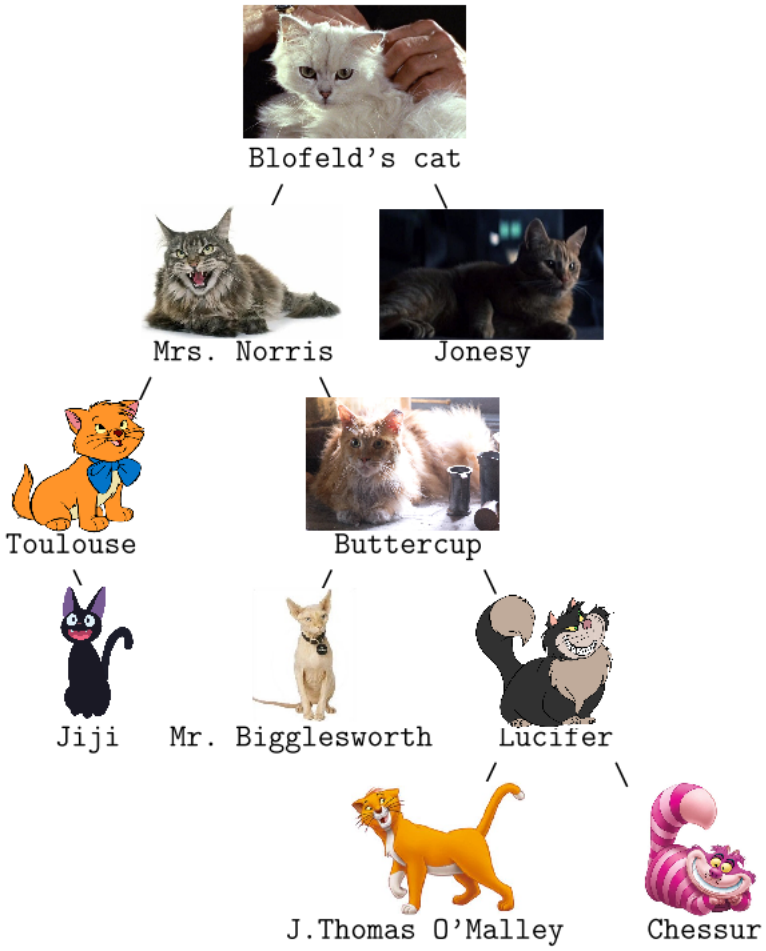


After executing the statement `cafe.retire(MrsN)`, the tree representing the cafe will look as follows:



In the following page, you can find the tree representing the cafe after the following statements have been executed:

```
cafe.hire(B);
cafe.hire(T);
cafe.hire(MrB);
cafe.hire(MrsN);
cafe.hire(new Cat("Blofeld's cat", 6, 72, 18, 120.0));
cafe.hire(new Cat("Lucifer", 10, 44, 20, 50.0));
```



We can also test the other methods as follows:

```
System.out.println(cafe.findMostSenior()); \\ displays Jonesy(0 , 21)

System.out.println(cafe.findMostJunior()); \\ displays Toulouse(180 , 37)

System.out.println(cafe.buildHallOfFame(3));
\\ displays [Blofeld's cat(6 , 72), Mrs. Norris(100 , 68), Buttercup(45 , 53)]

System.out.println(cafe.budgetGroomingExpenses(13)); \\ displays 415.0

System.out.println(cafe.getGroomingSchedule());
/*
 * displays
 * [[JIJI(156 , 17), Buttercup(45 , 53), Chessur(8 , 23)],
 *  [J. Thomas O'Malley(21 , 10), Jonesy(0 , 21)],
 *  [Toulouse(180 , 37), Mrs. Norris(100 , 68), Lucifer(10 , 44), Blofeld's cat(6 , 72)],
 *  []
 *  [Mr. Bigglesworth(71 , 0)]]
 */
```