Assignment 3

COMP 250 Fall 2020

posted:	Tuesday, Nov. 17, 2020
due:	Thursday, Dec. 3, 2020 at 23:59

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 kept. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and codePost may be overloaded during rush hours).
  - This is the file you should be submitting on codePost:
    - `DogShelter.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- Please note that the class you submit should be part of a package called `assignment3`.
- **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD CODE HERE" block.** You may add helper methods to the `DogShelter` class, but you are not allowed to modify any other class.
- 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. **Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to codepost, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, 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 highly encourage you to test your code thoroughly before submitting your final version.

- In a week we will share with you a `Minitester` class that you can run to test if your methods are correct. This class is equivalent to the exposed tests on codePost. Please note that these tests are only a subset of what we will be running on your submissions. We encourage you modify and expand these classes. You are welcome to share your tester code with other students on Piazza. Try to identify tricky cases. Do **not** hand in your tester code.
- 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.

Learning Objectives

In this assignment we will bring together binary search trees and max heaps. Working on this problem will allow you to better understand how to manipulate trees and how to use recursion to exploit their recursive structure.

A Dog Shelter

In this assignment you will be implementing a class called `DogShelter` that defines a data structure which can use to maintain information regarding the dogs being sheltered at a given facility. Before a dog joins the shelter, it is assigned an estimate of its age which will be kept on file. To have an easier time searching through the files, these are kept in order based on the dogs' age. Moreover, to keep all the occupants healthy, information regarding their vet appointments is also kept on each dog's file. Since life at the shelter is not always easy, it is of the highest importance to find a family for each occupant as soon as possible. To reduce the time that each dog spend at the facility, the policy at the shelter is that the dog that has been here the longest is the one with highest priority for being adopted. Hence, a number indicating the days a dog has been spending at the shelter is also kept on file and it indicates the priority to adoption that the dog has.

For this assignment, you can assume that there will not be dogs in the shelter with the same age and/or the same number of days spent at the shelter.

For this assignment (as for any coding project you decide to undertake) it is extremely important that you test your code as you write it using small tests. **Do NOT start testing your code only after you are done writing the entire assignment. It will be extremely hard to debug your program and you should not expect us (instructor, TAs, and mentors) to be able to fix your code just by looking at an error for few minutes during office hours.** If you need help with a bug you need to make sure to put in the time necessary to work on it so that you'll be able to identify a very specific reason and place in your code where the error originates (note that I do not mean the line displayed by the stack trace!).

The Dog class

To represent a dog, we have provided you with a class called `Dog`. This class contains the following fields:

- A name for the dog (`String`)
- An age estimate of the dog (`int`)
- The number of days they have spent at the shelter (`int`)
- The number of days until their next vet appointment (`int`)
- The expected cost (in dollars) of their next vet appointment (`double`)

Note that this class implements `Comparable` and several `public` methods are available to you. Please note that **you must not modify this class**.

The Shelter

A template for the `DogShelter` class has been provided to you. This class contains a `DogNode` inner class, and a field `root` which contains a reference to the root node of the tree representing the shelter. This class also implements the interface `Iterable` and an inner class called `DogShelterIterator` is also included. Inside the `DogShelter` class you will notice several `public` methods that have already been implemented. These are all non-static methods which will be called on objects of type `DogShelter`. Most of these methods call methods from the `DogNode` class that you will need to implement.

The tree we are using to represent a shelter is a tree where we store elements of type `Dog`. It is a binary search tree when we look at the dog's ages, and a max heap when we look at the number of days the dogs have been spending at the shelter.

The `DogNode` inner class is used to represent a node in the tree and it contains the following fields:

- `Dog d`: a reference to an object of type `Dog`. This is the element/key stored in the node. It contains information regarding a dog in the shelter.
- `DogNode younger`: this node is the root of the left subtree which stores information about dogs that are younger than the dog stored in the current node.
- `DogNode older`: this node is the root of the right subtree which stores information about dogs that are older than the dog stored in the current node.
- `DogNode parent`: this node is the parent node of the current node. Hence it stores information about a dog that has been at the shelter longer than the dog stored in the current node and has therefore higher adoption priority that the dog at the current node.

For this assignment you need to implement the following methods/classes. Please note that your code will be tested on time efficiency, so make sure to implement the methods exploiting the structure of the tree and using what we learned in class. Note also that you might want to implement the following in a different order than the one in which the tasks are listed.

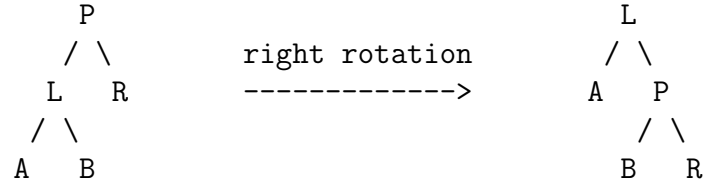
- `DogNode.shelter()` [25 points]
  This method takes as input a `Dog` object `d` and adds it to the tree that has as root the `DogNode` on which the method was called. The method returns the root to the new tree that now contains `d`.

  As explained above the tree we are trying to implement is a binary search tree when we look at the dog's ages, and a max heap when we look at the number of days the dogs have been spending at the shelter. Adding a new dog `d` to the tree has to be done in two steps:

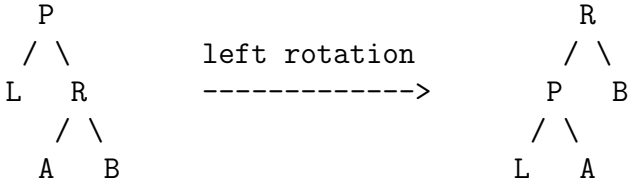
  1. Add a new node to the tree in the leaf position where binary search determines a node for `d` should be inserted. (see `add()` for binary search tree learned in class)
  2. Fix the trees so that the properties of the max heap are maintained (i.e. the parent node must have higher adoption priority 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. 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.

- `DogNode.adopt()` [25 points]
  This method takes as input a `Dog` object `d` and removes the key *equals* to `d` from the tree that has as root the `DogNode` on which the method was called. The method returns the root to the new tree that now does not contain the dog `d`.

  As explained above, the tree we are trying to implement is a binary search tree when we look at the dog's ages, and a max heap when we look at the number of days the dogs have been spending at the shelter. Hence, removing a dog `d` from the tree has to be done trying to maintain both properties.

  1. Remove the node that contains a dog equal to `d` 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 oldest dog `oldD` in the left subtree, use `oldD` to replace the dog to be removed, and remove `oldD` 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 saw in class, since we need to make sure to maintain 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.

- `DogNode.findOldest()` [5 points]
  This method returns the oldest dog in the tree that has as root the `DogNode` on which the method was called.

- `DogNode.findYoungest()` [5 points]
  This method returns the youngest dog in the tree that has as root the `DogNode` on which the method was called.

- `DogNode.findDogToAdopt()` [5 points]
  This method takes as input two integers `minAge` and `maxAge`. You can assume that `minAge` is less than or equal to `maxAge`. The method returns the dog (in the tree that has as root the `DogNode` on which the method was called) with age within the given range (both boundaries inclusive) who has the highest adoption priority. If no such dog exists, the method returns `null`.

- `DogNode.budgetVetExpenses()` [5 points]
  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 shelter will need to spend for the vet in the next `numDays` (inclusive) for the dogs that belong to the tree that has as root the `DogNode` on which the method was called.

- `DogNode.getVetSchedule()` [10 points]
  This method takes no inputs and returns an array list of array lists of dogs. The array list at index 0 contains all the dogs that should see the vet 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 dogs that should see the vet 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 dogs that should see the vet in i weeks from now. In each sublist, the dogs are ordered from youngest to oldest.

- `DogShelterIterator` [20 points]
  Implement the inner class `DogShelterIterator`. The method `DogShelter.iterator()` returns a `DogShelterIterator` object which can be used to iterate through all the dogs in the tree. The iterator should access the dogs from the youngest to the oldest. 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.

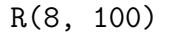
  You can implement the iterator however you like. We will test it by examining the order in which the dogs are accessed. It is ok if your iterator uses an `ArrayList` to store references to all the dogs in the tree. We have imported the class in the template. There are more space efficient ways to implement 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 dogs:

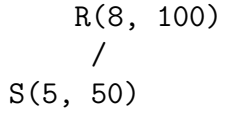
```
Dog R = new Dog("Rex", 8, 100, 5, 50.0);
Dog S = new Dog("Stella", 5, 50, 2, 250.0);
Dog L = new Dog("Lessie", 3, 70, 9, 25.0);
Dog P = new Dog("Poldo", 10, 60, 1, 35.0);
Dog B = new Dog("Bella", 1, 55, 15, 120.0);
Dog C = new Dog("Cloud", 4, 10, 23, 80.0);
Dog A = new Dog("Archie", 6, 120, 18, 40.0);
Dog D = new Dog("Daisy", 7, 15, 12, 35.0);
```

Assume we have created an object of type shelter `s` using the dog `R` as input. This means that at the moment the shelter is represented by a tree that has only a root node as follows:

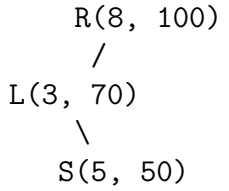


I indicated between brackets the age of the dog as well as its adoption priority (i.e. the number of days the dog has been at the shelter).

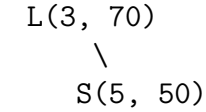
After executing the statement `s.shelter(S)`, the tree representing the shelter will look as follows:



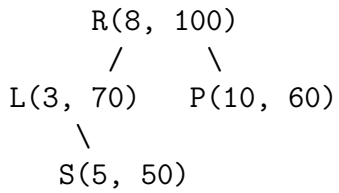
After executing the statement `s.shelter(L)`, the tree representing the shelter will look as follows (note that here one rotation was necessary in order to maintain the max heap property of the tree):



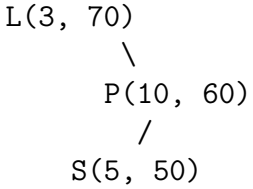
After executing the statement `s.adopt(R)`, the tree representing the shelter will look as follows:



After executing the statements `s.shelter(R)` and `s.shelter(P)` , the tree representing the shelter will look as follows:



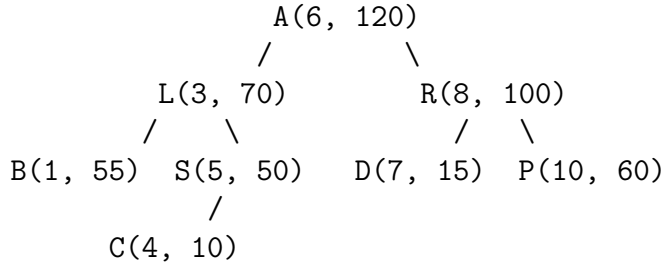
After executing the statement `s.adopt(R)`, the tree representing the shelter will look as follows:



After executing the following statements

```
s.shelter(R);
s.shelter(B);
s.shelter(C);
s.shelter(A);
s.shelter(D);
```

the tree representing the shelter `s` will now look as follows:



We can also test the other methods as follows:

```
System.out.println(s.findOldest()); // displays Poldo(10 , 60)

System.out.println(s.findYoungest()); // displays Bella(1 , 55)

System.out.println(s.findDogToAdopt(3, 5)); // displays Lessie(3 , 70)

System.out.println(s.budgetVetExpenses(15)); // displays 515.0
```

```
System.out.println(s.getVetSchedule());
/*
 * displays [[Stella(5 , 50), Rex(8 , 100), Poldo(10 , 60)],
 *   [Lessie(3 , 70), Daisy(7 , 15)],
 *   [Bella(1 , 55), Archie(6 , 120)],
 *   [Cloud(4 , 10)]]
 */
```

Finally, after executing the statement `s.adopt()`, the tree representing the shelter will look as follows: