# Assignment 3

COMP 250 - Winter 2018

Due date : March 26th, 2018

## 1 General Instructions (read carefully)

- You are provided some starter code that you should fill in as requested. **Add your code only where you are instructed to do so**. You can add some helper methods. Do not modify the code in any other way and in particular : do not change the methods or constructors that are already given to you, do not import extra code and do not touch the method headers. **Any failure to comply with these rules will give you an automatic 0**.

- **Submission instructions**

  - Assignment is due on March 26th 11:59 PM.
  - 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).
  - Submit only the file `Building.java` to the myCourses Assignment 3 folder.
  - Around one day before due date : we will run whatever you had submitted last and give you some feedback on it, mostly to warn you if you currently have a grade of zero, so you can fix it if there is a problem. Take advantage of this by submitting early !
  - Morning after due date : we will run your last submission and warn you again if your code does not compile.
  - After this, you can resubmit at any moment for a period of three days after the initial due date, with a penalty of 20% per day.

  - The last acceptable file you submit will be used to determine your final grade for this assignment.

- The starter code includes a tester class. 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 more challenging set of examples. We therefore highly encourage you to modify that tester class, expand it and share it with other students on the myCourses discussion board. Do not include it in your submission.

- You will automatically get 0 if your code does not compile.

- **Failure to comply with any of those 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 myCourses.

# 2 Building Management

## 2.1 Introduction

You are having coffee with one of your friends named Paul. Paul recently graduated and is now working for a Town Housing Office. He seems to be enjoying his work : the buildings that they are managing are quite old now, so they need to be renovated. He tells you more about this and then asks what you are up to. So you tell him about that amazing class (COMP250) that you are following. You tell him about binary search trees.

Paul looks puzzled and interrupts you to ask : "That's cute, but what's the point of this ?". You answer : "you are using software and database in your everyday work life, I am learning how to write them". But Paul is looking as if he does not believe you, so you add "let me show you. I will show you how the database you use at the office works".

"Ok, so you talked a great deal about how old your buildings are, this is how I am going to organize the database. What else is important on your buildings ?". Paul gets overly enthusiastic, so you restrict the list of important features to just a few : "Paul, I am not going to include the guy who designed it or what techniques were used to build the building. I only want to give you an idea of how the software you are using everyday is working !".

## 2.2 Organization of the code you are provided with :

The database of all buildings is going to be represented by a tertiary tree. We have separated the data associated to one building and the tree structure.

The class `OneBuilding` represents the data associated to a specific building. A `OneBuilding` has several fields : first its `name` (it can represent the address or the name for instance). A `OneBuilding` also has a year of construction (`yearOfConstruction`), a height (`height`), a year when it is likely to need to be repaired (`yearForRepair`) and the projected cost of those repairs (`costForRepair`).

A `Building` bears the tree structure. A `Building` has a field `data` bearing the information. A `Building` has also three fields of type `Building` : `older`, `same` and `younger`. These are going to allow you to construct a tertiary tree representing all the buildings. As the names suggest, given a `Building b`, the `Building b.older` and all its children were built before `Building b` was built, the `Building b.younger` and all its children were built after `Building b` was built and the `Building b.same` and all its children were built the same year as `Building b` was built. We also ask that

the buildings of same year of construction are sorted by decreasing height.

You are also provided with a tester. The tester is **VERY** partial. Even if your code seems to work, it does not mean that your code is correct : it only means that your code is working **ON AN EXAMPLE**. You **NEED** to expand the tester.

You do not need to import any other code (and if you do, you will get 0). All the material needed for this assignment was covered during class.

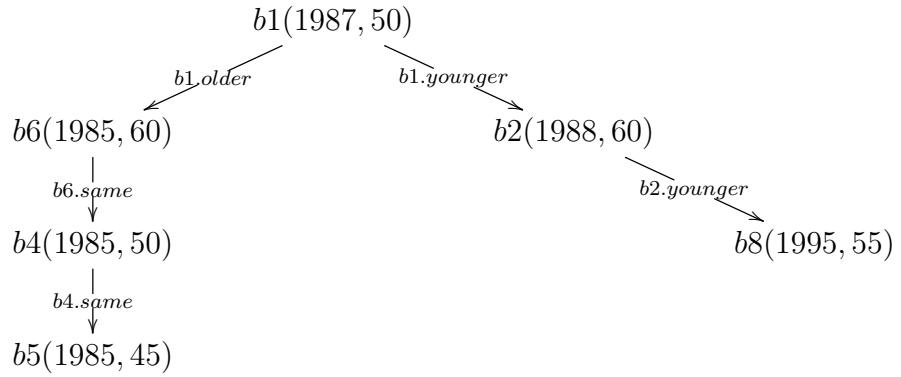
## 2.3 Building your database

First thing to do is to add buildings in your tree structure.

**Question 1.** First, code `addBuilding`. The argument is the `OneBuilding` that you want to add to the tree structure.

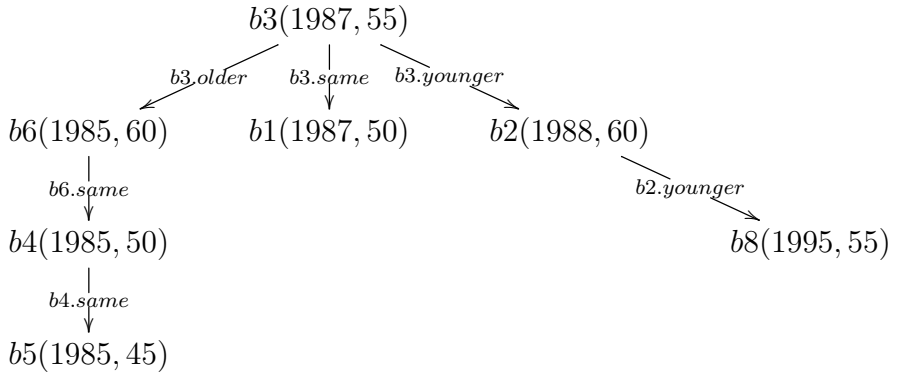
Given a `Building b` and a `OneBuilding obToAdd`, `b.addBuilding(obToAdd)` adds the `OneBuilding obToAdd` in the tree with root `b` accordingly to the rules explained before : if `obToAdd` is older (resp. younger) than `b.data`, it has to be added (in the appropriate place) to the subtree `b.older` (resp. `b.younger`). if `obToAdd` is the same age as `b.data`, then it depends on the height : if `obToAdd` is strictly higher than `b.data`, it becomes the new root and you have to adapt the other fields accordingly.

To clarify, let us give some examples. At the nodes of the tree, you will see (for instance) $b1(1987, 50)$. This means `Building b1` has `data.yearOfConstruction` 1987 and `data.height` 50. When the fields `older`, `same` or `younger` are not null, we represent them with an arrow to the corresponding `Building`. Let us start with the following tree :

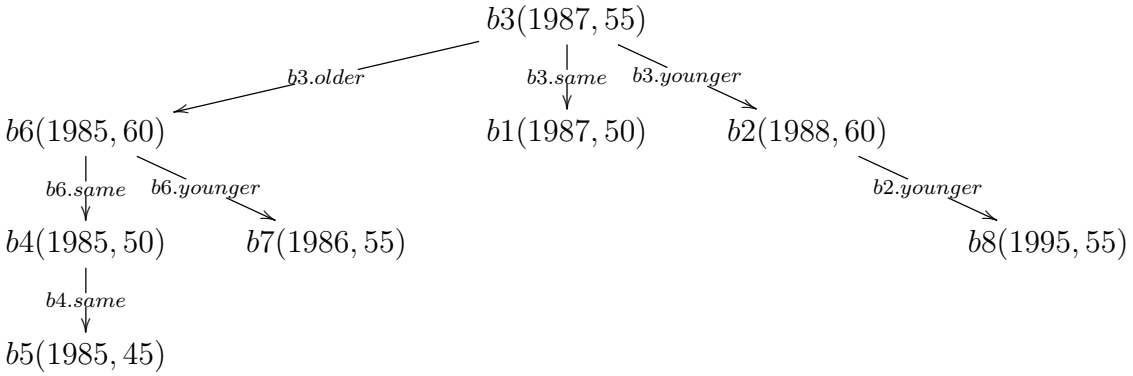



Now, if we have a `One Building ob3` with `yearOfConstruction` 1987, `height` 55, the command `b1.addBuilding (ob3)` should return a `Building` where

the root `b3` has data `ob3` and the fields should have been updated to :



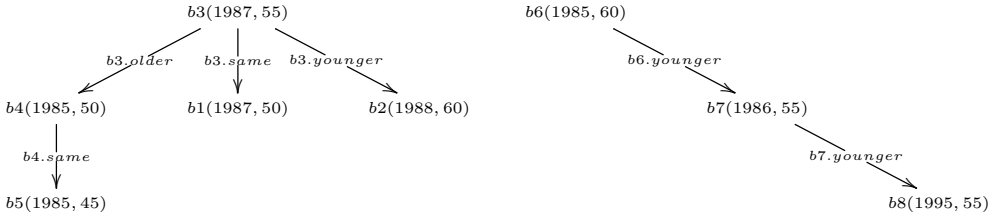

If `OneBuilding ob7` has `yearOfConstruction` 1986 and `height` 55, then the command `b3.addBuilding(ob7)` should return `b3` and the tree structure should be updated to :



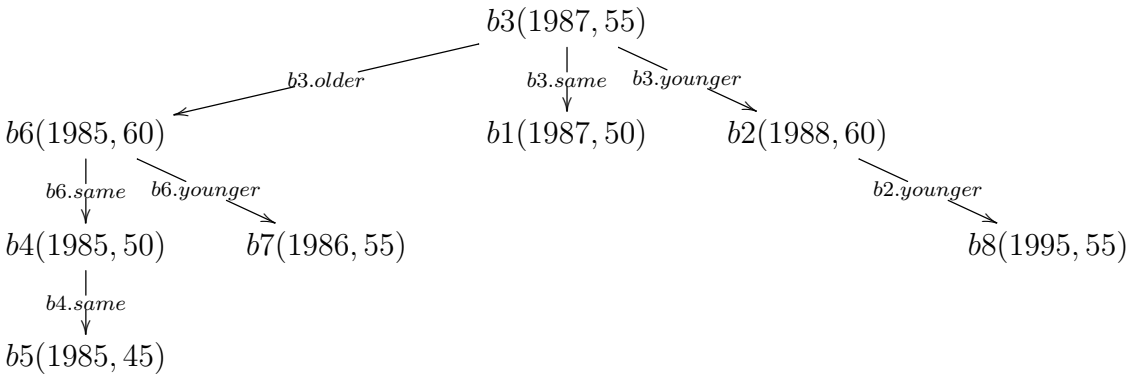

with `b7.data = ob7`.

**Question 2.** You can now code `addBuildings`. The argument is a `Building`. Given a `Building b` and a `Building bToAdd`, the command `b.addBuildings(bToAdd)` first adds the `OneBuilding bToAdd.data` to `b`, then goes on by adding the `Building bToAdd.older`, then `bToAdd.Same` and finally `bToAdd.younger` and returns the root of the tree structure as in the case for `addBuilding`.

For instance, if we have the two following `Buildings` :



The command `b1.addBuildings(b6)` will return `b1` and update the tree structure to :

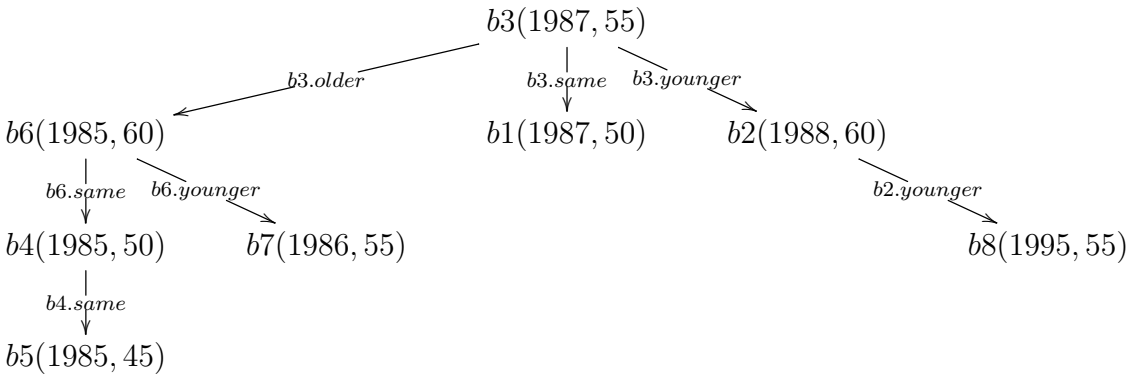



**Question 3.** Now you can code `removeBuilding`. Its argument is a `OneBuilding`. The command `b.removeBuilding (ob)` should return either `b` if no children of `b` has `data` corresponding to `ob`, or the root of the tree corresponding to `b` where the `OneBuilding ob` has been removed otherwise.

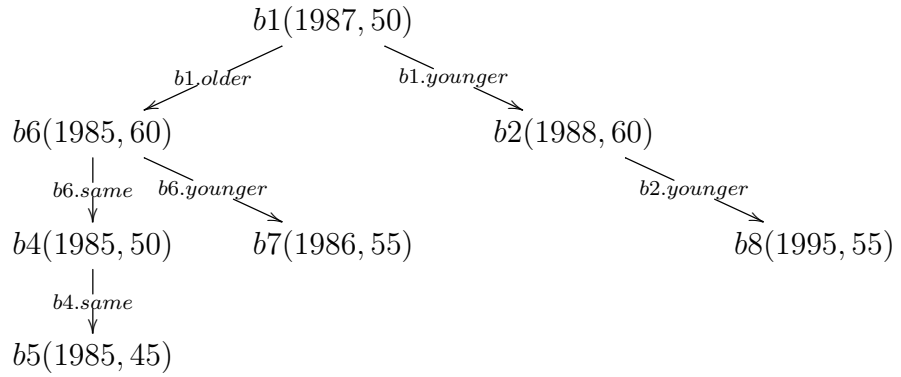
You should not remove any of the children of the `Building` with `data` corresponding to `ob`. For the rest of the explanation, let us assume that `b.data == ob` (you can easily adapt what is going to be said to the case where the `Building` to remove is not the root). In this case, there are three cases to consider. First, if `b.same != null`, then the new root becomes `b.same` and the subtrees of `b` should be correctly attached to the new root. Second, if `b.same == null` and `b.older != null`, then the new root becomes `b.older` and you need to add the `OneBuilding`s in `b.younger` to this new root. Finally, if `b.same == null` and `b.older == null`, then the new root is `b.younger` (note that this also covers the case when `b` is a leaf).

*hint : you just coded* `addBuildings`

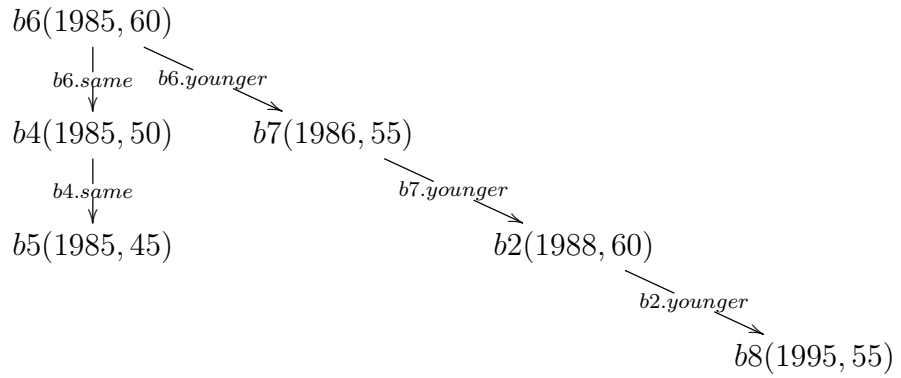
On an example, consider



The command `b3.removeBuilding(b3.data)` should return the new root `b1` with fields updated to :




The command `b1.removeBuilding(b1.data)` should now return the new root `b6` with fields updated to :




## 2.4 Showing Paul how powerful this is

Congratulations, you can now construct a database, you are now going to exploit this to find out useful information. **In all the remaining questions, you should not modify the tree structure anymore.**

Paul : "Those numbers are ridiculous, that's not how high a building is or how much it costs to repair a building !"

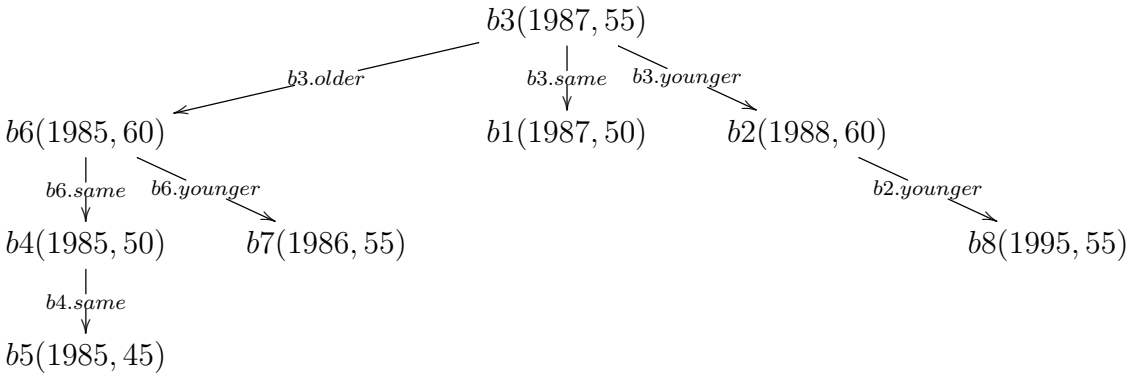
you : "I do not work on buildings, if you are unhappy about those numbers, you can either change them or imagine they are in a different unit."

Paul : "Oh, OK. So what now ?"

you : "well, I can compute things. For instance, I can easily tell you when the oldest building was constructed."

**Question 4.** Code `oldest`. When called on a `Building b`, the integer `b.oldest()` should be the year when the oldest `Building` in the tree with root `b` was built. Do not modify the tree structure.

Let us give you an example. At the nodes of the tree, you will see for instance $b1(1950, 10)$. This means `Building b1` has `data.yearOfConstruction` 1950 and `data.height` 10.




`b3.oldest()` should return 1985 and `b1.oldest()` should return 1987.

Paul : "Can you tell me how high is the highest building then ?"

you : "Yes. But it is missing the point of having my database organized by year of construction though."

**Question 5.** Code `highest`. When called on a `Building b`, the integer `b.highest()` should be the height of the highest `Building` in the tree with root `b`. Do not modify the tree structure.

On the example given to you in question 4, `b3.highest()` should return 60 and `b4.highest()` should return 50.

Paul : "You said that I was missing a point on the organization of the database."

you : "Well, I decided to organize buildings by their years of construction, so it is very efficient to answer questions based on this parameter. For instance, instead of how high the highest building is, I can tell you which one is the highest building constructed on a certain year."

**Question 6.** Code `highestFromYear`. When called on a `Building b`, the `OneBuilding b.highestFromYear(year)` should be the `OneBuilding` that was the highest built in year `year` and that is in the tree with root `b` (and `null` if there are none). Do not modify the tree structure.

On the example from question 4, `b3.highestFromYear(1985)` should return `b6.data`, and so should `b6.highestFromYear(1985)`, but `b4.highestFromYear(1985)` should return `b4.data`.

Paul : "Can you answer questions based on multiple years ?"

you : "Yes, for instance, I can tell you how many buildings were built during the time from one year to another one."

**Question 7.** Code `numberFromYears`. When called on a `Building b`, the integer `b.numberFromYears(yearMin, yearMax)` should be the number of `Building`s built from year `yearMin` to `yearMax` (including those two years) that is in the tree with root `b`. If `yearMin > yearMax`, this should return 0. Do not modify the tree structure.

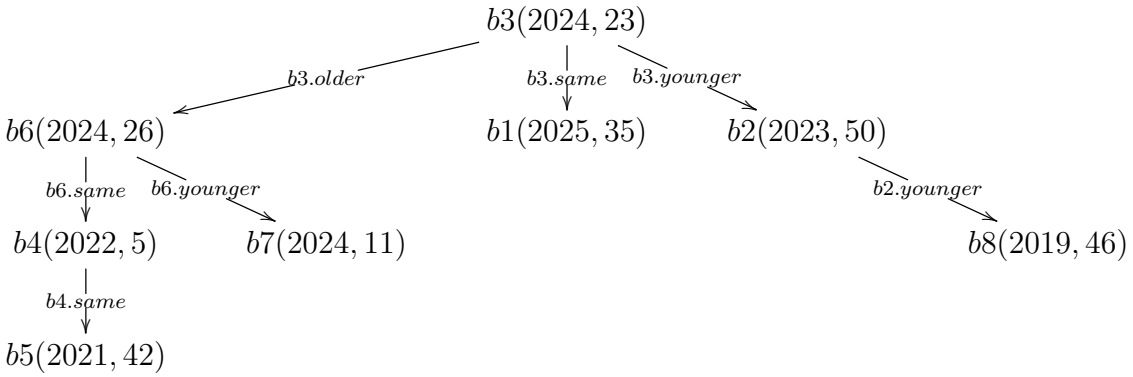
On the example from question 4, `b3.numberFromYears(1986, 1988)` should return 4 (corresponding to buildings `b7`, `b3`, `b1` and `b2`).

Paul : "That's pretty cool ! Can this also help us plan the renovation of the buildings ?"

you : "yes"

**Question 8.** Code `costPlanning`. When called on a `Building b`, the array `b.costPlanning(n)` should have in its i-th cell, how much you should spend on year (2018 + i) to repair all the scheduled buildings in the tree with root `b`. The argument `n` will always be a positive integer. Do not modify the tree structure.

Let us modify the example given in question 4 to only display the year planned for repairs and the costs for such repairs.




On this example, `b3.costPlanning(7)` should return the array `{0, 46, 0, 42, 5, 50, 60}`: in year 2018, there is no cost to plan for repairing the buildings, but in year 2024, buildings `b3`, `b6` and `b7` will need to be repaired, for a cost of 23 + 26 + 11 = 60. Similarly, `b2.costPlanning(7)` should return the array `{0, 46, 0, 0, 0, 50, 0}`.