Project 5

Due Date: Tuesday 22 November by 11:59 PM

General Guidelines.

The method signatures provided in the skeleton code and/or API in the handout indicate the required methods. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment, and you are welcome to add those into the classes.

Unless otherwise stated in this handout, you are welcome to add to/alter any provided java files as well as create new java files as needed, but make sure that your code works with the provided test cases as you will not be submitting the test code as part of your submission. Your solution must be coded in Java.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!
Importing the Math class is okay.

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will be subject to consequences according to the syllabus and university policy.

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.

There are three parts to this project. Part 2 depends partially on Part 1, but Part 3 is completely independent of the other two. Each part is worth 20 points, but Part 3 has an option for extra credit depending on your implementation.

Part 1. Hashtable

In this part, you must implement a generic Hashtable. A Pair class is provided for you, which is generic. I have also provided skeleton code with a few methods that will help.

Implementation Details

- To get full credit, you should implement the Hashtable using linear probing to handle collisions. The underlying array is tested in Test 2, which is worth 4 out of the 20 points. So if you would rather do separate chaining, you can, but you will fail Test 2. If you choose separate chaining, just make the getTable method return null. Also, if you decide to do separate chaining, you may use the ArrayList class for the chains, but make sure you are using it in an efficient way.
- When you resize the table, use the *getNextNum* method that is provided to make the new size a multiple of 6 plus or minus 1, so that it is more likely to be prime. For example, to make the table larger, you would pass in 2M (where M is the current table size) as the argument and use the result as the new table size.
- Your implementation should provide the expected runtimes that were discussed in class, and you should manage the load factor as discussed in class.

API: Required Methods for Hashtable.java

Signature	**Description**
`Hashtable()`	`constructor–start with a default table size of 11`
`Hashtable(int m)`	`constructor–start with a table size of m`
`V get(K key)`	`return the value associated with the given key; return null if the key is not in the table`
`void put(Key key, V val)`	`puts (key, val) into the table or updates the value to val if key is already in the table; resize`

	`as necessary`
`V delete(K key)`	`remove* and return the pair associated with key; return null if key is not in the table; resize as necessary`
`boolean isEmpty`	`return true if the table is empty and false otherwise`
`int size()`	`return the number of pairs in the table`
`Pair[] getTable()`	`return the underlying array–this is just for testing; if you decided to forgo the 4 points and do separate chaining instead, just make this return null`

*Keep in mind that “removing” from this table has to be done carefully so as to avoid breaking up the clusters.

Part 2. Patient Queue (MaxPQ)

In this part, you will implement a queue for handling patients coming in to an ER. Each patient has a name (String), an urgency level (int), and an arrival time (int) and patients with higher urgency levels should be given priority. If two patients have the same urgency level, they should be seen based on their arrival time (FIFO). Your Queue should be a Max Priority Queue. Note that there is already a Patient class provided for you, but you may need to add to it, so it is included in your submission.

Implementation Details

- The MaxPQ must be an array-based binary heap.
- The underlying array is not tested, so you can start at index 0 or at index 1.
- In case of “ties” when you implement sink, always swap with the left child over the right child (when they are the same).
- Never swap equal items in sink or swim.
- Use the provided *compareTo* method to compare Patients. It checks urgency levels first and then arrival times.
- As you should know, binary heaps are built to do *insert* and *delMax* in O(logN) time. They do not generally allow you to update items or remove arbitrary items in O(logN) time because you would need to find the item in the queue first. In this implementation, you must implement update and remove functions that are

O(logN) (amortized). To do this, you will need to have a way to quickly (in O(1) time) get the position of an item in the PQ. This can be done by using a Hashtable and by updating the Patient class.

API: Required Methods for MaxPQ.java

Signature	**Description**	**Amortized Runtime**
`MaxPQ()`	`constructor–start with a default array size of 10`	
`void insert(Patient item)`	`insert the new Patient into the PQ; resize if necessary`	`O(logN)`
`Patient delMax() throws EmptyQueueException`	`delete and return the max; throw the exception if the PQ is empty`	`O(logN)`
`Patient getMax() throws EmptyQueueException`	`return the max; throw the exception if the PQ is empty`	`O(1)`
`int size()`	`return the number of elements in the PQ`	`O(1)`
`boolean isEmpty()`	`return true if the PQ is empty and false otherwise`	`O(1)`
`public Patient remove(String s)`	`remove and return Patient (whose name is s) from the PQ; return null if the Patient is not in the PQ`	`O(logN)`
`void update(String s, int urgency)`	`update the urgency level of Patient (whose name is s) to urgency`	`O(logN)`

Part 3. Tree (BST or Right-Leaning Red-Black Tree)

In the file called `Tree.java`, implement a search tree. You have two options. You can implement a binary search tree, or you can implement a *right-leaning* red-black tree. In both cases, the required methods to be implemented are the same, but their implementations will be different because the structures are different. And (as you

probably guessed), option 2 is significantly more difficult, which is why that option can earn you extra credit.

Option 1.
If you correctly implement the functions for the binary search tree, you can get full credit, but your implementations must be correct and efficient. The BST tests are worth 5 points each (so 20 points total).

Option 2.
If you correctly implement the functions for the right-leaning red-black tree, you can get full credit + 12 extra points, but our implementations must be correct and efficient. The RBT tests are worth 8 points each (so 32 points total).

Guidelines

- Both trees should maintain BST order as discussed in class. Note that the K generic type extends Comparable, so you can use `compareTo`.
- In both cases, you need to be able to efficiently get the *height* and *size* of individual nodes in the tree, as well as the *height* and *size* of the tree itself. It is not a very efficient method to calculate this each time by searching the tree, and such a method will not be given full credit. Note that the Node class provided already has variables for *height* and for *N*. The key is to update these as you do the inserts.
- For the right-leaning red-black trees:
    - A right-leaning red-black tree is like a left-leaning red-black tree, but all the red links must go to the right instead of the left.
    - You may use the code provided in the slides. You will need the same basic transformations for this tree.
    - The first thing you should do is look at the cases for left-leaning inserts (Cases 1-4 in the slides) and change them to fit a right-leaning tree. (Think of this tree as a mirror image of the left-leaning tree.)
    - The second thing you should do is make sure you fully understand the code for the transformations. Like I said, you don't need new code for the rotations or the color flip, but you will have a very difficult time if you don't actually understand what the code is doing, so trace through it first.

Required Methods to be Implemented

Method Signature	**Description**	**Runtime**
`public void put(K key,`	`Insert (key, val) into`	`BST: expected O(logN)`

V val)	the tree or update val if key is already in the tree.	RBT: worst O(logN)
public V get(K key)	Get the value associated with key in the tree or return null if the key does not exist.	BST: expected O(logN) RBT: worst O(logN)
public boolean isEmpty()	Return true if the tree is empty or false otherwise. Should be simple.	BST: O(1) RBT: O(1)
public int size()	Return the number of nodes in the tree. Should be simple.	BST: O(1) RBT: O(1)
public int height()	Return the height of the tree. This should take O(1) time.	BST: O(1) RBT: O(1)
public int height(K key)	Return the height of the subtree whose root node contains the given key. If the key does not exist in the tree, return -1. The time for this should be the same as the time it takes to find the node.	BST: expected O(logN) RBT: worst O(logN)
public boolean contains(K key)	Return true if the key is in the tree and false otherwise.	BST: expected O(logN) RBT: worst O(logN)
public int size(K key)	Return the size of the subtree whose root contains the given key. Return -1 if the key does not exist. The size should include the node itself. Also, the time	BST: expected O(logN) RBT: worst O(logN)

	for this should not be worse than the time it takes to find the node.	

Part 3 Grading

The tests total 20-32 points depending on the option you choose, but the testing method here is different. You are provided with input and output files. The input files contain commands and the output files contain the results of running those commands. The test code does all this for you, but the way your code is evaluated is by comparing the output of your code with the expected output by looking at the difference between the two files.

- For debugging purposes, test 0 is the easiest one and can easily be drawn by hand (even as a RBT).
- Both versions use the same input files, but the output files are different.
- To test a BST implementation: type the following command into the terminal:

  ```
  ./runBSTTests.sh
  ```
- If your output looks like this, it is correct:

  ```
  Running Test 1...
  Running Test 2...
  Running Test 3...
  Running Test 4...
  ```
- Incorrect output will result in other things being printed out. Those things indicate differences between the expected output file and your output file. You should be able to compare them manually. Your output for a given test will be in a file called `output<testNum>.txt`. (For example, `output0.txt` for Test 0.) The expected output will be in a file called `test_bst_output<testNum>.txt.` (For example, `test_bst_output0.txt` for Test 0.)
- To test an RBT implementation, the only differences are:
  - The command to run the tests is: `./runRBTTests.sh`
  - The expected output will be in a file called `test_rbt_output<testNum>.txt` (e.g. `test_rbt_output0.txt` for Test 0).

Testing your code on lectura. (See specific instructions above for testing Part 3.)

Once you have transferred your files,

- you can compile a java file in the terminal with the following command:
  `javac <filename>`
- you can can compile all the .java files with this command:
  `javac *.java`
- you can run a compiled file with the following command:
  `java <filename>`

Submission Procedure.

To submit your code, please upload the files to **lectura** and use the **turnin** command below. Once you log in to lectura and transfer your files, you can submit using the following command:

`turnin cs345p5 Hashtable.java Patient.java MaxPQ.java Tree.java`

Upon successful submission, you will see this message:

Turning in:
`Hashtable.java -- ok`
`Patient.java -- ok`
`MaxPQ.java -- ok`
`Tree.java -- ok`
All done.

Note: Your code submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it runs and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues.

Grading.

The autograder should give you a good idea of your grade. However, your final grade will be determined by the TAs. Below are some possible deductions.

Possible Deductions

Item	**Max Deduction Possible**
Bad Coding Style	5 points
Not following instructions	100% of the points you would have earned (i.e. automatic 0)
Not submitted correctly on lectura	100% of the points you would have

	earned (i.e. automatic 0)
Does not compile/run on lectura	100% of the points you would have earned (i.e. automatic 0)

Other notes about grading:

- If you do not follow the directions (e.g. importing classes without permission, etc.), you may not receive credit for that part of the assignment.
- Good Coding Style includes: using good indentation, using meaningful variable names, using informative comments, breaking out reusable functions instead of repeating them, etc.
- If you implement something correctly but in an inefficient way, you may not receive full credit.
- In cases where efficiency is determined by counting something like array accesses, it is considered academic dishonesty to *intentionally* try to avoid getting more access counts without actually improving the efficiency of the solution, so if you are in doubt, it is always best to ask. If you are asking, then we tend to assume that you are trying to do the project correctly.
- If you have questions about your graded project, you may contact the TAs and set up a meeting to discuss your grade with them in person. Regrades on programs that do not work will only be allowed under limited circumstances, usually with a standard 15-point deduction. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.