ASSIGNMENT WEEK 3

PART 1 (JAVA HEAP)

In class we implemented a max heap's **`heapify`** method in Java for generic keys, but lacking values. For this assignment, you will implement methods as outlined below in **Java**.

0. Use the code from class / Canvas as a starting point or start yourself from scratch to create a class **`MinHeap`** with generic key-value entries.
1. Create a class **`Data`** with generic attributes **`key`** and **`value`** of generic type. Define your **`MinHeap`** class such that it uses array members of type **`Data`**, but ordering will occur by an entry's attribute **`key`**. You will need to declare your key type to be comparable. Use the Week 2 Symbol Table as example code or read the Java documentation on how to define generic types.
2. Create a constructor which initializes the attribute **`A`** (array) from a provided list. Make sure you use the actual heap array starting with index 1 so it aligns with our textbook. Copy the methods **`left`**, **`right`**, and **`parent`** from class code.
3. Write a method **`void heapify(int index)`** which creates a min-heap recursively. Recall that keys cannot be compared with < or > if the type is not known at compile-time, so use a comparator as we did in week 2 (where **`left<index`** translates to **`this.comparator.compare(…)<0`**).
4. Create a list of key-value data from the string **`"Rowan CS-Dept"`** (length 13) by turning each character into the **`Data`** pair of its **`int`** and **`char`**. Create a min-heap from this list and print the keys of the resulting heap in order of the indices. Test and debug your methods. Provide a screenshot of this run.

Each of the problems 1 – 4 will be graded according to listed rubric for a total of 16 points.

Part 2 (Python BST)

In class we have seen implementations of several methods for a binary search tree data structure both in Java and Python. For this assignment, you will implement four methods as outlined below in **Python**. The main method will be `successor` which is to find the successor of an existing key.

0. Use the code from class / Canvas as a starting point or start yourself from scratch to have a class `BST` with key-value nodes. Note that you will need an implementation with parent pointers in order to implement `successor`.
1. Implement a ***private*** minimum method (named `__minimum(self, startNode)`) based on the pseudocode Minimum from our slides and text. This method will return the ***node*** with minimum key relative to node `startNode` (not necessarily `root`) if it exists, and `None` otherwise.
2. Implement a public minimum method (named `minimum(self)`) based on the pseudocode Minimum from our slides and the method `__minimum()` you created. This method will return the minimum ***key*** starting from `root` if it exists, and `None` otherwise. Adjust your code so that the final answer is a key and not a node.
3. Implement a private search method (`__search(self, startNode, key)`) which recursively finds the node `node` with `node.key==key`. Use the code for `__get(self, startNode, key)` as a blueprint, but make sure to return a node and not a value.
4. Implement a public successor method (`successor(self, key)`) which utilize `__search(self, startNode, key)`) to find the node `node` with matching key. Then operate on nodes as our pseudo code does from class does:
   a. If the key cannot be found, raise a key error.
   b. If the node has a right child, return the minimum key from the right child's subtree.
   c. If the node does not have a right child, traverse the tree in a while loop until you find an ancestor such that the original node is in its left subtree.
   d. If there is no successor (because the key was the maximum key) return `None`.
5. Test and debug your methods. Provide test runs in form of a main (driver) file in which you create appropriate variables, fill the tree with data from the ***provided JSON file***, and run some cases to show your methods work. In particular, show runs for the successor key of S06E12, S05E05, and S06E16. Do not forget to upload `main.py` and `bst.py` and provide screenshots of the test runs.

Each of the problems 1 – 5 will be graded according to the following rubric for a total of 20 points.

SCORE	**4**	**3**	**2**	**1**	**0**
SKILL LEVEL	Response gives evidence of a complete understanding of the problem; is fully developed; is clearly communicated.	Response gives the evidence of a clear understanding of the problem but contains minor errors or is not fully communicated.	Response gives evidence of a reasonable approach but indicates gaps in conceptual understanding. Explanations are incomplete, vague, or muddled.	Response gives some evidence of problem understanding but contains major math or reasoning errors.	No response or response is completely incorrect or irrelevant.