EE 422C Pair Programming Exercise

Purpose

In this lab exercise, you will participate in pair programming, a software development practice where two engineers work jointly to implement a single software product. Advocates of pair programming believe that the methodology trades off a small decrease in programmer productivity for a large improvement in software quality (and therefore lower maintenance costs for the software). In this lab, you will have an opportunity to experience pair programming first hand, and you can begin to form your expert opinion.

Background - How do you pair program?

Both engineers sit side-by-side at a terminal, and while one engineer is typing, the other reviews the code and makes suggestions for improvements. Both engineers are actively participating in the coding process at all times. Communication between the pair is vital to the process. It is also essential that the pair works as equals - pair programming is not a mentoring disicipline where one engineer works as a coach or an advisor, both engineers are equally invested in the work and share fully in the design, the implementation, and the validation of the software product.

The engineer seated at the keyboard is often called the *driver*, the other engineer is called the *navigator*. Each role is different. The driver has primary responsibility for the line-by-line correctness of what is being implemented while the navigator must track big picture design concerns and must also be on the lookout for coding errors or code quality issues (i.e., corner cases for testing). The two engineers will frequently switch roles, where the driver calls for the mouse/keyboard and begins realizing the joint work of the time in code, while the navigator takes on the role of reviewing, critiquing, and improving the product.

Pair programming can be difficult for some teams to learn and execute effectively. Personality issues can completely undermine the method, and so programmers must check their egos at the door and fully embrace the team concept.

Your Task

You will be creating a class that implements a subset of the java.util.Map interface. Maps are easy to build on top of any number of data structures (arrays, trees, hash tables), and have fairly modest design complexity. You should implement these methods: `clear`, `containsKey`, `containsValue`, `get`, `isEmpty`, `keySet`, `put`, `putAll`, `remove`, `size`, `values`, `entrySet`. You do not need to implement `equals` method.

You are asked to select a partner by yourself and submit your source code and test cases to Canvas once you finish the task.

Phase A

You will write the `PMap` class using an `ArrayList` for the underlying data structure. The `PEntry` will store the key and value for a single entry in the map. Performance of the functions (i.e., time complexity) is not a primary concern during this phase. The goal is simply to build the functions and confirm that they are (at least mostly) working. You need to write at least one test case per method via JUnit to test your implementation, for

example, to check the return of `put` and `remove`. Descriptions of the methods can be found in the Java documentation for java.util.Map.

You should rotate roles (driver/navigator) with your partner after Phase A.

Phase B - Extra exercise, for fun

You will re-implement the `PMap` class leveraging a data structure from the Java standard library. You can use any data structure you wish, although `HashSet` is an excellent choice and you are specifically prohibited from using any of the `Map` implementations (i.e., you can't use `HashMap`). During this phase, performance is a significant concern, and in addition to implementing each of the functions you must assess the time complexity of each of your functions. You need to write at least one test case per method via JUnit to test your implementation.

What to turn in

After finishing Phase A, one of you should zip your source code files and test cases and submit to Canvas. Create a README file and write down your team information:

```
Phase A
Driver <Name>, <UTEID>
Navigator <Name>, <UTEID>
```

If you did Phase B, make a similar entry in the README. Only one submission is required per team.

If you do not finish, finish what you can in each phase.