Describe your unit testing approach for each of the three features.

In developing the software features, the unit testing approach was carefully aligned with the software requirements to ensure robust functionality and efficiency. For instance, when managing unique IDs through hash maps, the unit tests were designed to ensure that each ID was distinct and quickly accessible, which was crucial for meeting the requirement of efficient data management. This alignment was evident in the use of hash maps, which provided constant time operations for insertion, searching, and deletion, directly supporting the software's need for speed and reliability.

Additionally, a mix of dynamic and static testing was used to find errors at various phases of development. Static testing verified the software's behavior in real-world circumstances, while dynamic testing helped find problems early on, such as logical mistakes and syntax errors, without running the code. By using two different approaches, it was made sure that the software fulfilled both the specifications and the expectations for practical functionality.

The effectiveness of these unit tests was confirmed by the high coverage percentages, which indicated that a significant portion of the code was thoroughly tested. High coverage means that more lines of code were executed during testing, reducing the likelihood of untested bugs and ensuring that the software was both reliable and ready for deployment.

Describe your experience writing the JUnit tests.

Writing the JUnit tests was a critical part of ensuring both the technical soundness and efficiency of the code. To make sure the code was technically sound, I focused mainly on consistency and correctness throughout the testing process.

By using a hash table, I could avoid the linear time complexity associated with array lists, especially when searching for duplicates or performing deletions. This approach ensured that the code could handle large datasets quickly and effectively, meeting the performance requirements of the software.

Testing Techniques

In this project, I employed both **static testing** and **dynamic testing** techniques. Static testing involved analyzing the code without executing it, focusing on identifying syntax errors, logical flaws, and adherence to coding standards. Dynamic testing, on the other hand, involved executing the code to observe its behavior in real-world scenarios. Dynamic testing was somewhat limited due to the project not having an associated UI, but nonetheless we can still confirm that objects of are the correct type when the program is run.

These techniques have different practical uses depending on the project. Static testing is invaluable in the early stages of development when ensuring code quality and adherence to standards. Dynamic testing is essential for validating functionality and performance in real-world scenarios. Integration testing is crucial when multiple components need to work together, and system testing is key before releasing a product to ensure it meets all requirements and performs well in a production environment.

Mindset

In approaching this project, I adopted a mindset of caution and thoroughness. As a software tester, I recognized the importance of understanding the complexity and interrelationships of the code I was testing. This was crucial because a small change in one part of the code could have unforeseen consequences in another, potentially introducing bugs or inefficiencies. For example, when refactoring older code to align with new implementations, I carefully reviewed how changes in data structures would affect overall system performance and functionality. This cautious approach helped me prevent introducing errors that could compromise the software’s reliability.

To limit bias in my review of the code, I made a conscious effort to approach each test as objectively as possible. I reviewed my own code critically, as if it had been written by someone else, which helped me identify issues that I might have overlooked if I were too close to the code. This mindset is important because developers can sometimes become biased, believing their code is flawless, which can lead to missed defects. For instance, I used code reviews and peer feedback to gain different perspectives on my work, which helped me identify potential issues that I hadn’t considered.